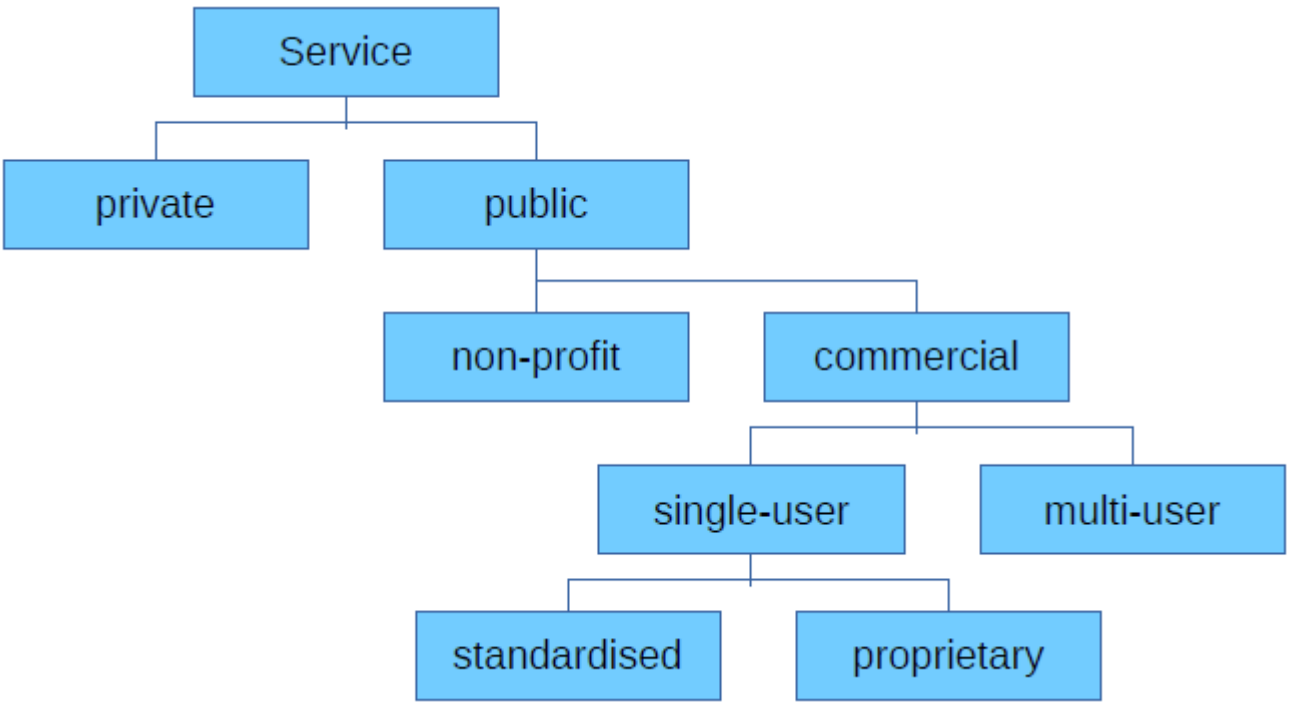
Summary

V01 - Prototyping Roadmap

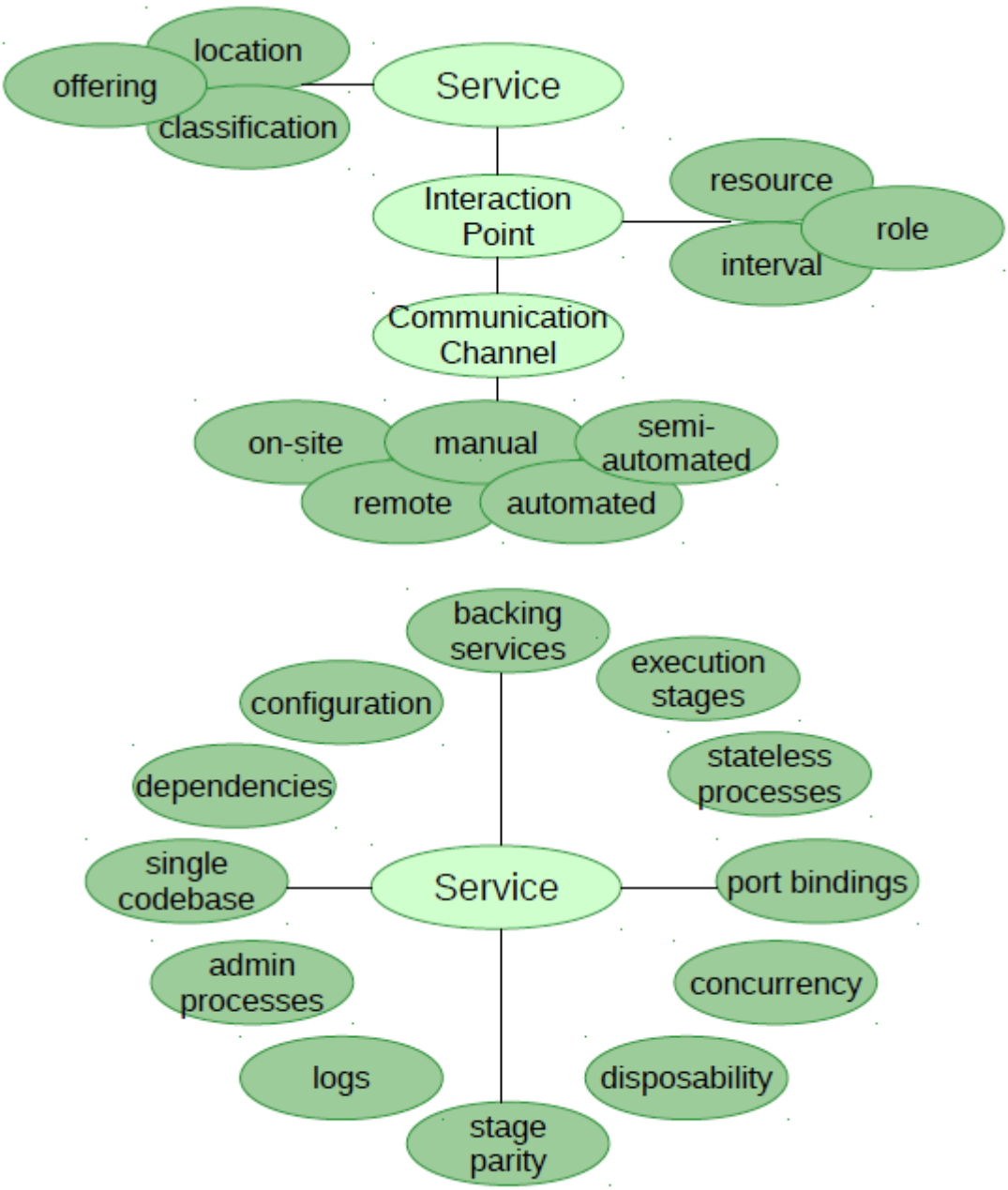
> Explore services & service ecosystems
>
> Know how the module is structured
>
> Understand prototyping goals



- technical characteristics of a service are: visibility, interaction, effect, encapsulation, interoperability
- a web service typicall uses HTTP (REST), URL/URI, and JSON/YAML/XML/plain text/binary documents/streams
- cloud service: on-demand, pay-per-use, elasticity, robustness
- microservice: light-weight, single-function, tangible & portable implementation
- prototyping is about the Pareto principle (80/20) and focusses on features (less on security, conformance etc.)

V02 - Services

> Understand what services are and how they are used
>
> Know how to design, describe and develop services
>
> Use service description and engineering tools



- interfaces can be black, gray, or white boxes and consists of description + interaction (both are structured/declarative formats using YAML or similar as base + domain-specific language (DSL))
- description: describes intent, protocol, endpoints, messages, data types...
  - functional and non-functional properties
  - behavioural and compositional semantics
  - requirements, constraints, limitations
  - access mechanisms
- interaction: network protocol with invocation semantics, API, adheres to description

- visibility implies discoverability (broadcast, registry etc.)

Simple RAML Example

taken from https://atom.io/packages/api-workbench

```
#%RAML 0.8
title: XKCD
baseUri: http://xkcd.com
schemas:
  - comic: !include schemas/comic-schema.json
/{comicId}/info.0.json:
  uriParameters:
    comicId:
      type: number
  get:
    description: |
      Fetch comics and metadata  by comic id.
    responses:
      200:
        body:
          application/json:
            schema: comic
            example: !include examples/comic-example.json
/info.0.json:
  get:
    description: |
      Fetch current comic and metadata.
    responses:
      200:
        body:
          application/json:
            schema: comic
            example: !include examples/comic-example.json
documentation:
  - title: Headline
    content: !include docs/headline.md
```

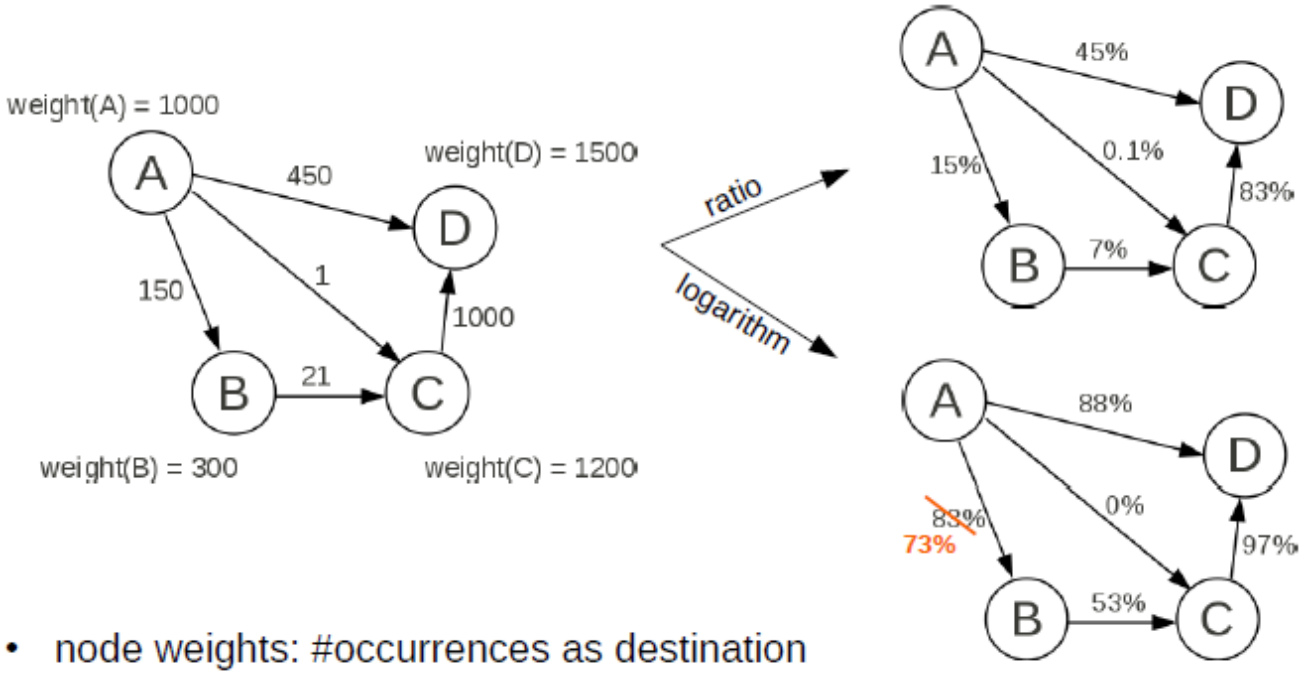
- description to implementation is top-down (code generation) and bottom-up (description generation) which leads to a roundtripping combined approach

V03 - Composition

> Understand why complex applications consist of and/or bind to multiple services
>
> Know dependency analysis and composition techniques
>
> Use a microservice management platform

- component is a piece of e.g. software and a composition consists of several components, some equal, some with distinct function
- service composition is mainly driven by reusability and extensibility
- dependencies have a hidden cost (B goes down, A too), can bine fixed or late binding, inter- or intra-service
- dependency graphs are directed, a/cyclic (latter: bad), un/weighted
- A depends on B if

    - declared as such
    - A invokes B (traceable)
    - logic of A depends on state within B (not traceable)
- Peddycord's technique
    - passive network monitoring and analysis
    - logarithm-based ranking scheme
    - frequency inference



- node weights: #occurrences as destination
- edge weights: #nested occurrences
- confidence as ratio-based ranking: weight(A→B) / weight(A)
- confidence as logarithm-based ranking: $\log_{\text{weight(A)}}\text{weight(A}\rightarrow\text{B)}$

- dependency resolution can be immediate, interactive (complementing immediate res.), or using a SAT solver
- composite services:
    - offer a single service interface
    - distribute requests to multiple services within the composition (in parallel, serially, or more complex routing)
    - require knowledge of dependencies (internally or by the caller)
- advantages for composite services are improved QoS (e.g. higher availability) and QoE (e.g. flexibility to switch)
- techniques/patterns for composite services:

	Orchestration	Choreography	Bundling	Multiplexing
Definition	coordinated arrangement of service invocations; may be executable as another service	global interaction protocol between autonomous service partners	multiple services offered/used in a bundle	multiple services used in parallel handling partial requests
Potential benefits	creation of value-added services by re-using others	no central point of control; declarative messaging behavior	cheaper, less administrative overhead	flexible redundancy schemes, “survival of the fittest“
Potential risks	issues with dependency services (unavailable, faulty)	difficult decentralized enactment; little industry acceptance	less flexibility for exchanging single service	more administrative overhead, higher cost due to candidates

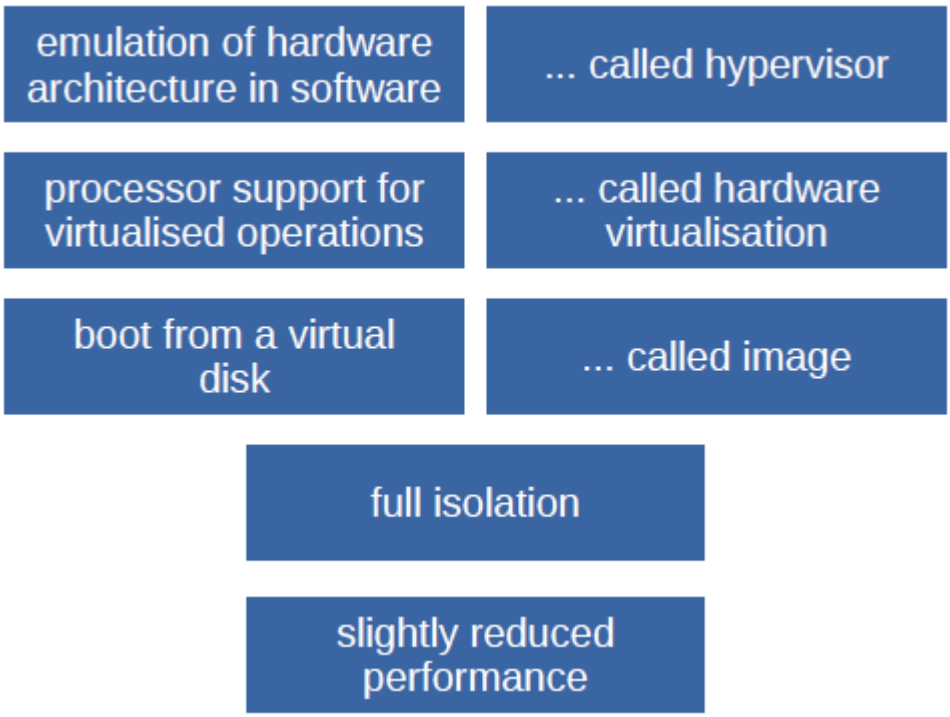
(Orchestration: e.g. AWS CloudFormation)

- microservices: single-function-oriented services which scale elastically and operate resiliently backed by a portable implementation; typical tasks:
  - deployment and management of microservice representations (i.e. containers)
  - partial upgrades without downtimes, honouring dependencies
  - canary testing
  - monitoring, migration, scaling, … without downtimes
  - rapid prototyping

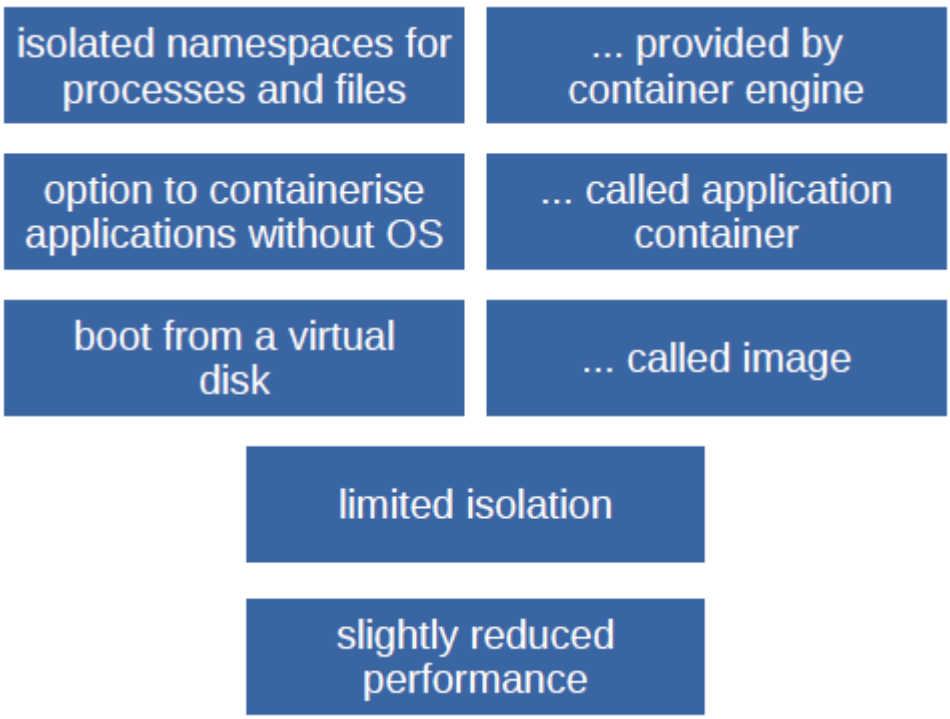
V04 - Service Technologies

> Understand five distinct service execution technologies
>
> Know how to package, deploy and run service implementations
>
> Use an API modelling tool to complement the implementation with a description

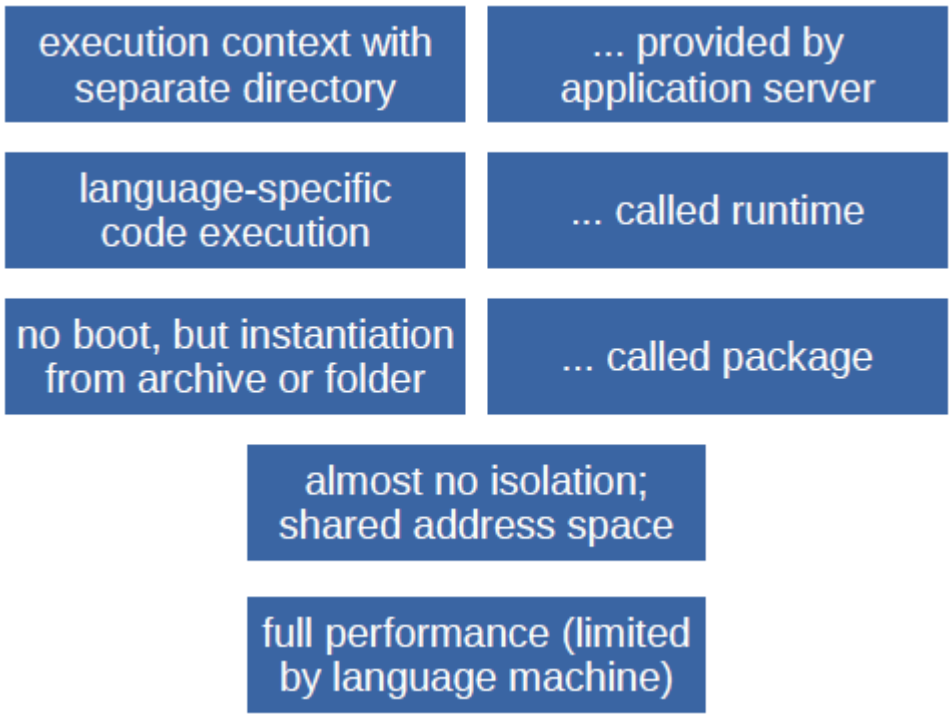
Virtual machine (e.g. AWS EC2)



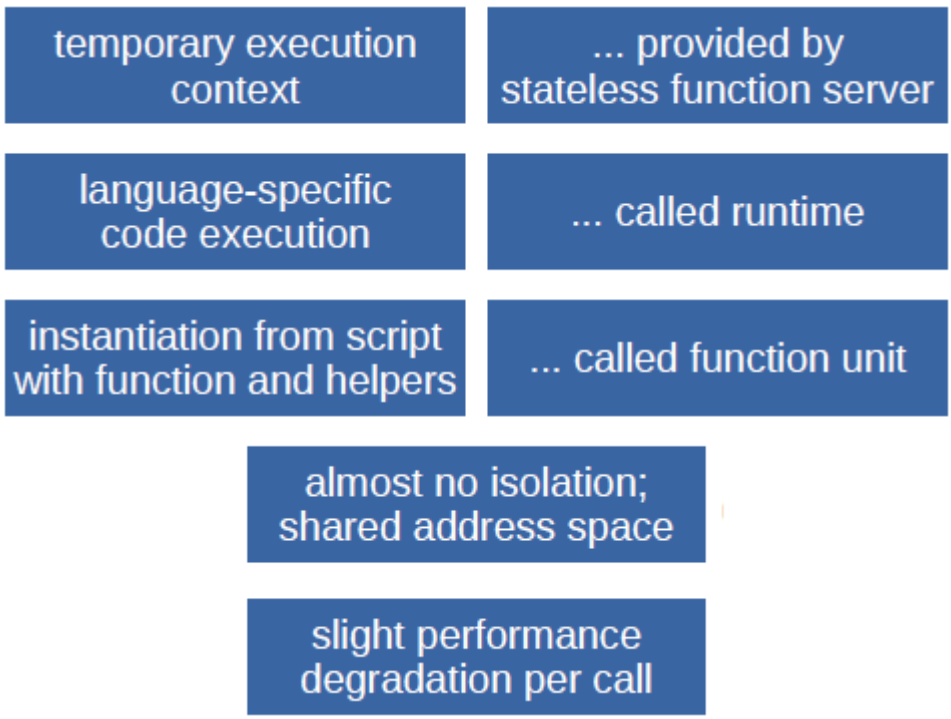
Container (e.g. AWS ECS)



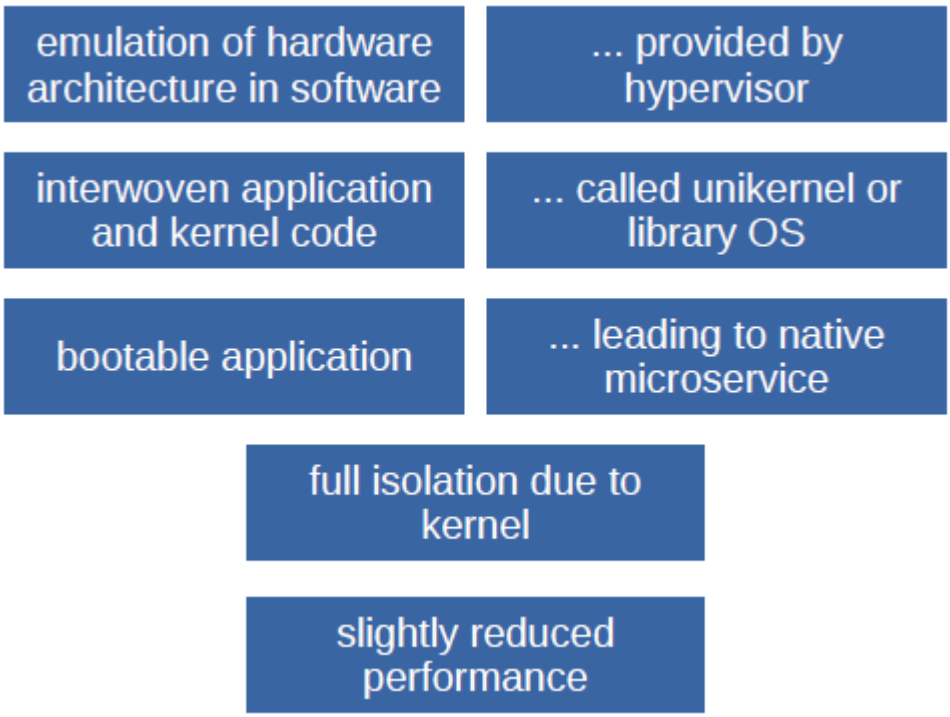
Application package (e.g. Heroku, OpenShift)



Function (e.g. OpenWhisk, AWS Lambda)



Unikernel (works on any IaaS)



Commonalities and Differences:

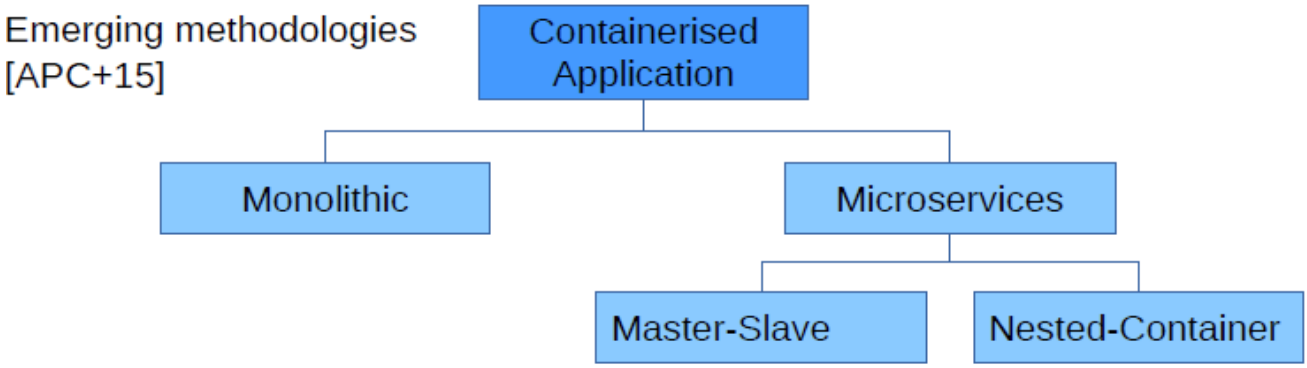
- VM, C, UK: bootable
- AP, F: executed on language virtual machine
- UK, F: instantiation in milliseconds
- AP, C, VM: instantiation in seconds to minutes
- AP, C: application-level, easy wrapping
- F, VM, UK: specialised development techniques
- F, C: common for cloud-native applications
- VM, AP: common for rather monolithic applications
- UK: not yet common for anything

V05 - Containers

Container Technology	Background	Booting	Namespacing	Integration
LXC	vendor-neutral open source	directories	kernel	frontend
Rocket	vendor open source	AppC/Docker images, OCI	Rocket launcher	boot manager
Docker	vendor open source	Docker images, OCI	Docker daemon	composition

- Dockerfile "commands": `FROM, LABEL, COPY, ADD, RUN, CMD, EXPOSE, WORKDIR`
- Docker can map ports, volumes
- Docker Compose describes:

- a set of microservices, implemented as Docker containers
- build and run configuration
- dependencies
- instance details: replicas, restart policies, placement, networking, volumes



- Containers in VMs: simple but limited use, more complexity, single points of failures

V06 - Functions

> Combine knowledge on Function-as-a-Service in greater detail
>
> Know about the state of technology and research challenges
>
> Apply your knowledge with a “FaaSification“ research prototype

- in FaaS, a function is the elementary unit, app/bundle the complex unit
- input is application-specific; includes context
- processing & output are application specific
- usually constrained code size, memory, timeout etc.
- pricing based on requests (calls) and/or load (memory)
- code is either designed for FaaS or (automatically) transformed
- automatic transformation use static (-> source code patching) and/or (=hybrid) dynamic code analysis (-> binary patching)

Transformation rules

- entry points
  - no transformation of main function
- functions definitions
  - adapt to FaaS conventions: parameters, return value
  - scan recursively for function calls
  - export as function unit including dependencies
- function calls
  - if internal, rewire
  - if input/output, replace
  - otherwise, leave unchanged
- monads
  - functional programming with side effects (i.e. input/output as side channel)

- FaaSification: process of automated decomposition of software application into a set of deployed and readily composed function-level services

V07 - Monetization

> Capture the mechanisms for services as utilities
>
> Understand how to keep service consumption under control
>
> Know how monitoring and API management works
>
> Use a rating-charging-billing framework to charge for your application

- maximize API potential: directories/portals, developer sites (e.g. docs), management tools/frameworks/gateways (differentiate customers -> plans; monitoring etc.)
- Kong: logical centralization; several microservices (traffic procy, admin API, DB services, dashboards), API registration, API use
- API Management aaS, e.g. AWS API Gateway
- monitoring: quantified statements about system condition -> information at your fingertips
- passive monitoring (small overheads) vs active (probes; possibly non-idempotent functions)
- for services: monitor application, stack, dependency services -> CPU, utilization etc.
- e.g. ELK (Elasticsearch, Logstash, Kibana)
- Metering: monitoring & collection of metrics through (software) meters
  - leads to time series with characteristics such as sampling frequency/precision/accuracy, windows etc.
  - can be used for forecasting, detecting patterns, and anomaly detection
- monetize APIs:
  - data collection (linked 3rd party services and aggregation+correlation of user data)
  - product adoption (custom integrations)
  - developer usage: charge per call / fixed quotas / overage pricing
- rating: assign rates to usage data records (from metering)

- charging: process pricing model to create charge records
- billing: aggregate charge records to payable amounts on bills

V08 - Scalability

> Understand horizontal and vertical scaling
>
> Know how to design a scalable application
>
> Explain the influence of distributed designs on scalability
>
> Develop your first scalable distributed service

- service scalability: potential to be adapted in order to accommodate varying demand
  - Trigger: manual scaling vs. autoscaling
  - Properties:
    - Elasticity (time): how elastic? what is the degree of adaptation to varying demand? (= to which degree is potential used?)
    - Boundedness (space): how bounded? what are upper/lower limits?
    - Transformativity (space/behaviour): how transformative? is self-similarity of system guaranteed, or are there any bottlenecks?
- for VMs/containers: vertical/up through hypervisor or horizontal/out (replication/re-instantiation; one (processes/threads) or multiple hosts (distributed over the network))
- auto-scaling: on-demand provisioning by using vertical and horizontal scaling; can be re- and proactive; reactive can be app-agnostic and -specific, proactive only -specific
- reactive: risk SLA violation; proactive: SLA maintained
- autoregression: $y_t = c + \varepsilon_t + \sum_{i=1}^{p} a_i y_{t-i}$ where ε_j are terms, $a_i y_{t-1}$ is the rolling average
- moving average: $y_t = c + \sum_{j=0}^{q} b_j \varepsilon_{t-j}$, where ε_{t-j} are terms, b_j are damping factors
- prediction pipeline: predictor -> resource planner -> action plan generator

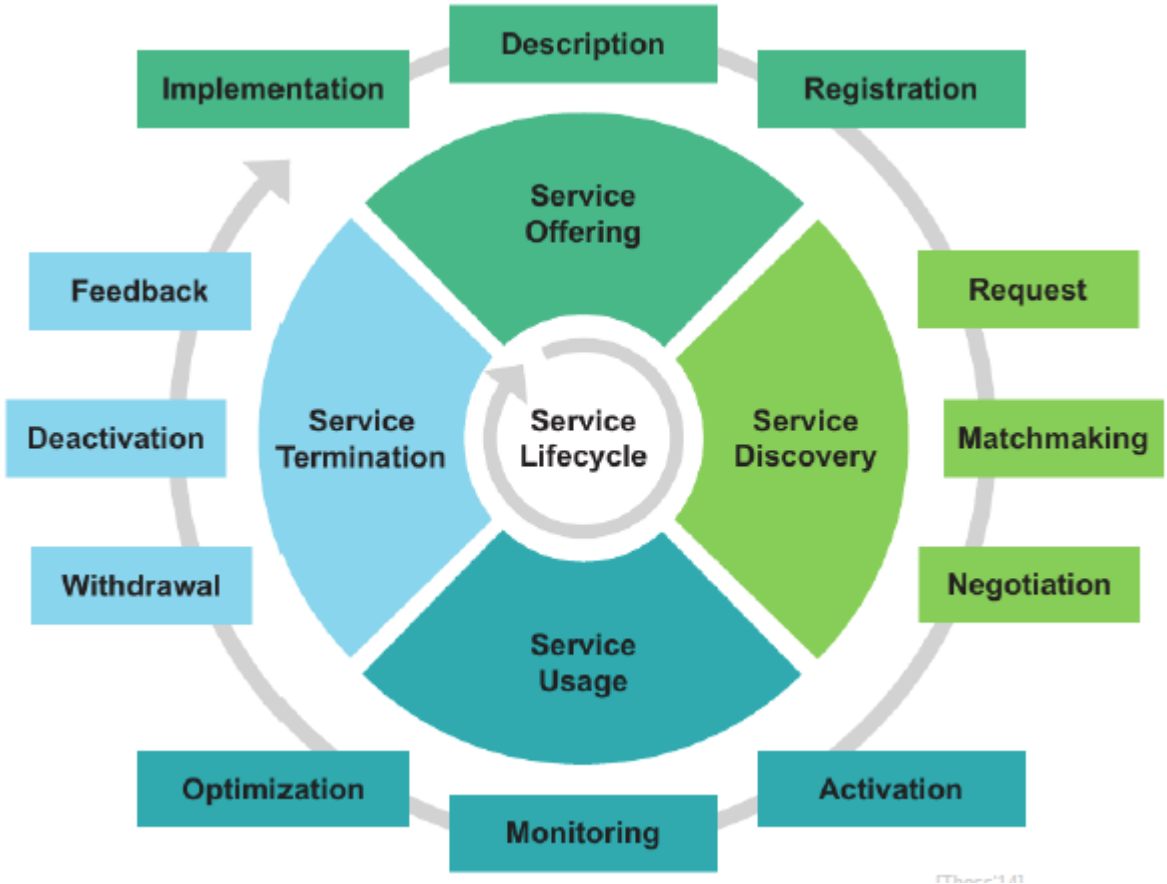
	multi-tier design	service-oriented design	cloud-native application design
description	3: front-/backend, DB; 4: client, delivery+cache, aggregation/logic, services+data	uniform description + communication, encapsulation, re-use; discovery; flexible dynamic bindings	main target properties: scalability (↑) & resilience (↑); refines service orientation with microservices; exploits capabilities of cloud environments
advantages	simple, monolithic; established protocols; sufficient flexibility for complementary blocks	consequent evolution of complex software design; viable basis for rapid prototyping of services	closest to ideal scalability: elastic, hardly bounded, hardly transformative; matched by contemporary technologies, e.g. containers, functions; conversion of legacy applications possible
disadvantages	inequal loads and bottlenecks; only applicable to isolated applications	waning support for classical SOA technologies; no guidance for how to design, build, run services	technological immaturity and volatility; emerging tools and languages, incompatibilities, insufficiencies

- coordination != cooperation != communication
- design for failure incl. coordinator faults and attacks ("byzantine faults")
- types of majorities: none, relative ($n > m, \forall n, m \in \mathbb{N}$), absolute ($n > \frac{1}{2}N$), byzantine ($n > \frac{2}{3}N$), consensus ($n = N$)
- paxos consensus requires absolute majority of working processes ($N = 2F + 1$ where F are the number of faulty, non-malicious processes)
  - roles: voters, learners, proposers, leader
  - depends on leader election (successful iff one leader)
  - depends on redundancy (ignores failures of redundant participants)
- Swedish leader election: based on rounds - elimination of candidates
  - each round: flip a coin -> winners proceed into next round; if no winner, all play again
  - last remaining candidate is leader
- Raft: uses log replication; random timeout for follows; absolute majority of votes; heartbeats define election

V09 - Quality

> Understand how to express quality levels.
>
> Know established quality models.

> Understand the need to “design for change“ and “design for failure“, generalizable to “design for inherent quality“.
>
> Use simulation and emulation tools to see the effects of suitable designs without breaking anything.



- Service Quality Model:
  - simplified capturing of non-functional service characteristics
  - examples: performance, cost, compliance to standards
  - domain-specific models: e.g. e-government: transparency important aspect or e.g. e-learning: focus on knowledge and competencies
- example 1: runtime, transactions, configuration/cost, security
- example 2: performance, usability, reliability, responsibility, costs&conditions, data security, flexibility, customer service
- measurable quality: described metrics (non-/authorative sources); measured metrics (monitoring, attestation); estimated metrics (further sources, prediction etc.); static analysis (descriptions); dynamic analysis (measurements)
- metaquality: dynamic (re-)binding
- replica: repeated/redundant software component or data
  - benefit: faster response time (“survival of the fittest“ - individual use); improved safety (backup - joint use)
  - cost: increased storage requirements; increased computation and bandwidth depending on replication mode
- service replication can be partial/decomposed (microservices) or full/n-fold (horizontal scaling)
- data replication can be partial/erasures or full/n-fold
- data replication can be done using XOR erasure coding (c = a XOR b -> 50% redundancy with 2 significant and 1 redundant fragment), possibly in a GF(2) resulting in arbitrary redundancy

- replication tools for services are component descriptions or proxy servers (e.g. HAProxy)
- replication tools for data are incremental/delta backups, distributed versioned replication (e.g. Git) or erasure coding
- fault injection allows to find problems before they releases; either in a the live or simulated/emulated system
- fault injection levels and methods:
    - software level: fuzzing: random or crafted (invalid) data input to applications; on the API level, through files or databases, input device emulation; data modifications (e.g. user removal, file substitution)
    - system level: resource exhaustion: full disk, full memory, CPU overload, network flooding; limits exhaustion: max # of file descriptors, sockets, threads, files etc.; random process signalling and termination
    - service/network level: artificial slowdown; random port blocking; protocol mutation

V10 - Development

> Decide on how and where to spend your development effort
>
> Distinguish between general approaches and concrete tools
>
> Use online development, build, test and integration tools

- development for the cloud:
    - value proposition: separation of development and operation
        - long feedback cycles; mismatches; inconsistencies
    - value proposition: shortened time to first development cycle
        - always-online requirement; higher risk for vendor lock-in; development/operations separations of concerns lifted
- there are web-based SaaS IDEs
- service artefacts: service source software, service implementation, service description, service configuration, service contract template, composition description, client package

V11 - Application Patterns

> Identify patterns at a higher level compared to software engineering
>
> Know useful patterns for building cloud-native applications
>
> Enhance your software with more sophisticated patterns

- 1-node, 1-container patterns
    - upward: information exposure; metrics: health, queries per second; profiling: threads, stack, network statistics, configurations, logs
    - downward: lifecycle adherence; yields software components which can be deterministically controlled
    - containers: resource accounting and allocation; packaging, deployment, re-use; failure containment boundary
- 1-node, n-containers patterns (multiple containers, shared namespace, shared IP address, atomic scheduling)
    - sidecar / sidekick: "pick-up car" for log transport; "parcel car" for serving synchronized content; "ACS car" for repair; "toll car" for service discovery
    - ambassador: to proxy and represent; example: frontend, backend + external shards for backend
    - adapter: to normalize and present; example: frontend, monitoring adapter + external monitoring service

- m-nodes, n-containers patterns (multiple containers, distinct namespaces, distinct IP addresses, independent scheduling)
    - leader election: election microservice as re-usable container; standard API for voting and determining leader available to application containers
    - work queue: set of containers: one coordinator, some grabbers; connected to application-specific workers via standard API
    - scatter/gather: distribution of requests
- CNA (cloud-native application) technologies ingredients: application (microservices, support services) and environment (microservice management platform, distributed initialisation system, operating system)

V12 - Service Domains

> Understand the importance of services in various domains
>
> Know service technologies, protocols and tools for three particular domains
>
> Use diverse protocols and message exchange patterns in your application

- domain: context with specialized knowledge, processes and tools
- functional domains correspond to sectors of the economy
- service domains correspond to functional domains
- e.g. robot OS: messaging uses simple types and arrays/structs and uses both TCP and UDP; based on pub/sub topics as well as synchronous request/reply and asynchronous, monitorable, and interruptable actions
- e.g. mobile application: code offloading to e.g. server
- e.g. multimedia: long-running interactions (many internet protocols have limited session semantics) using XMPP or WebRTC

V13 - Rapid Prototyping

> Understand what rapid prototyping is
>
> Master complex service engineering projects with reduced effort
>
> Use one tool to realize your next service

- requirements on service development: speed, quality, alignment
- constrained optimization: minimum time with acceptable quality
- why rapid?
    - engineering progress: evolutionary rather than revolutionary
    - new technologies: quick feedback about feasibility
    - focus on doing, not talking
    - reduction of abstraction and cognitive load
- kinds: throwaways, evolutionary, incremental, extreme
- methodologies: blueprinting (design-first/top-down), method extraction (code-first/bottom-up), minimalism, simulation & emulation, staging
- minimalism: replace prod solutions with light-weight solutions (e.g. PostgreSQL by SQLite)
- fragmented tooling support; no complete lifecycle solution