# Exam objective

You are required to cover both **theoretical aspects** (terminology, definitions, classifications, relations to other topics) as well as **operational aspects** (practical usage, processes, coding aspects) of the topics below.

I therefore expect you to prepare and bring **examples** of to the exam for presentation and discussion. This may typically be from your mandatory exercises but you may also give examples from your work or open source systems if you wish.

You may structure your presentation as theory first and then show how the theory is used in the example, or example first where theoretical aspects are presented as they are needed to explain the example. Either will do fine, but doing only one (only treat theory, or only present an example) will be considered a major flaw in the presentation.

# 1. Performance Engineering: Queue Theory and Performance Factors.

* Hvad er performance?
  + Kvantificerbare krav. Inklusiv måleteknik og belastning.
* Repsonse time, throughput, capacity, utilization (example CPU).
* Overvej performance tidligt (og vægtet I forhold til prioritering og andre krav)
* Kø-teori (A.K. Erlang)
  + Tegn exponentiel kurve
  + Flere konfiguration: 1 kø 1 server, flere køer, en server, … flere køer og servere i serie
  + Eksempler flere køer (international/DK), , én kasse (DSB). Én kø, flere kasser (posthus), …
  + α: arrival process, β: Service time distribution, m: Number of servers - sotcastic
  + α/β/m/K/P/D – M/M/1 (memory, memory, 1 server)
  + W = S \* U / (1 – U), W = wait time, U = Utilization, S = Service time – CPU = 70-8% max
  + Multi-kø og service system – bedst når balanceret.
  + Amdahl (performance-gain pga. øgede ressourcer har et max)
  + Gustafson (Ja, men de ekstra ressourcer kan så lave andre ting) – eksempel ansatte på en opgave.
* Performance factors (det der påvirker performance for det givne system)
  + HW: processor, memory, disk I/O, netværk I/O
  + OS: konfiguration (porte, page file, …, andre services der bruger ressourcer)
  + Application server: framework, f.eks. Java, .NET, … (OS abstraction) – garbage Collection
  + Application: designet af selve programmet
  + Database: valg, configuration, størrelse, design
  + Netværk: fragmentering, sikkerhed, teknologi, …

## 2. Performance Engineering: Testing and Performance Patterns.

* Hvad er performance?
  + Kvantificerbare krav. Inklusiv måleteknik og belastning.
* Repsonse time, throughput, capacity, utilization (example CPU).
* Overvej performance tidligt (og vægtet I forhold til prioritering og andre krav)
* Test
  + Stress, Load, Spike, Udholdenhed (endurance), Solskin (Single-Thread test)
  + Mocks, stubs, injector
  + Tools: JMeter, jvisualvm, LoadRunner, …
  + Selve test påvirker resutat (log filer, …, test program bruger CPU)
  + Identify Test environment, Identify Performance criteria, Plan and design test, Configure test environment, implement test, execute test, analyze report retest.
  + Hvad testes? (netværks-kø så kan service stubbes med en målt service time).
* Performance arkitektur
  + Coupling and cohesion (loose and high) – this is for scaling.
  + Communication (not chatty interfaces)
  + Concurrency (locks and shared resources)
  + Resource management (memory and threads)
  + Caching
  + State management (scaling (free session early or use session less)
  + Data structures and algorithms (list vs array vs hashset, … + limit conversion (XML <-> internal))
  + Be careful of anti-patterns:
    - “just throw more HW at it”
    - Many temporary objects
    - Chatty interfaces
    - Having to look many different places to gather info.
    - Error cascading
    - …
  + Husk 80/20 regel
  + Se efter anti-patterns
  + Diagnose: Ændre kun én ting ad gangen
  + Husk at performance tuning ikke forbliver. Når systemet eller dets load ændres skal der tunes igen

# 3. Cloud Computing and Virtualization.

* Cloud
  + Utilization (VI har ikke brug for fuld capacity 24 timer I døgnet)
  + Scaling
  + On-demand, rapid elasticity
  + Ressource pooling
* Virtulization
  + Moderne VM ligeså effektiv som ingen VM
  + Isolation i tilfælde af fejl
  + Backup, sharing
  + Deling af ressources
  + Legacy OS
  + Process VM vs. System VM (Process most common, other is an “OS” in itself)
  + SaaS
    - Software as a service (PBS, Krak)
  + Paas
    - Platform as a service (Google App engine, e-comerce)
  + IaaS
    - Infrastructure as a service (hosted server)
  + Amazon EC2
  + Forhindringer
    - Dependency on third party (Cloud provider)
    - Data Lock-in
    - Data confidentiality (and physical location)
    - Data transfer bottlenecks and cost (send disks via UPS)
    - Performance unpredictability
    - Fixing bugs and debugging (time)
    - Licensing (e.g. Windows server – only IaaS)
    - Reputation fate sharing
  + Supervisor mode/Privileged mode (kernel)
  + Mode switching er dyrt
  + Emulation
    - Langsomt
  + Trap-and-emulate
    - Trap betyder at fange kernel instruktionen i et IRQ, eksekverer den separat og returnerer
    - Hurtigere end emulering, da kun privileged instruktioner er langsommere
    - X86 supportere ikke trap på alle privileged instructions
  + Binary Translation
    - Translate instruction and send translated instruction directly to CPU
  + Future: make CPU directly designed for VM (VM can run in Ring -1 (ultra-privaleged)

# 4. Architectural Evaluation: ATAM and aSQA.

* ATAM
  + 3 parts, 9 steps
  + Aranger møde datoer og deltagere, projektleder, stakeholders, arkitekt, …
  + Afhold møde med de tekniske specialister (arkitekt, projektleder, arkitekturspecialister)
  + Renskriv præciseret arkitektur fra møde og send ud
  + Afhold møde inklusiv stakeholders
  + Renskriv
  + Afhold opsummeringsmøde
  + Anvender Utilty tree (root -> QA -> group -> scenario incl. importance and difficulty (H,M)
  + QAS – Quality Attribute Scenario
  + QAW – Quality Attribute workshop (ligger før). ATAM er efter Arkitektur design
* aSQA
  + Light weight – udføres ved hver sprint
  + 1. Define metrics, 2: Define levels, 3:Define Components, 4:Evaluation, 5:Analysis, 6:Define Initiatives, 7: Imporve, Start from 4 again.
  + Uses “Excel”
  + Gruppere efter komponeneter ikke scenarier (ulempe ifølge Systematic og måske grund til de ikke bruger den ret meget mere), e.g. Terminal, Database, App Server.
  + 1 – 5 (1 uacceptabelt, 3 OK, 5 perfekt)
  + 4.1: Target level (t)
  + 4.2: Current level (c)
  + 4.3: Health (h) = 5 – max(0, (target – current)
  + 4.4: Importance (i)
  + 4.5: Focus (f) = ((6 – health) \* importance/5)
  + 4.1 + 4.2 -> 4.3 + 4.4 -> 4.5. Fokuser på de høje tal.
  + Svært at gruppere efter komponenter.

# 5. Architectural Reconstruction.

* Manglende eller forældet documentation
* Arkitekturbeslutninger og patterns kan ikke direkte ses I koden.
* Hvorfor?
  + Evaluering af arkitekturen mod et fremtidigt ønske
  + Refaktorering
  + Integration
  + En kompleks og dyr opgave, så det gøres ikke for sjov
* Symphony
  + Problem elicitation (hvorfor og hvad skal undersøges) -> problem statement
  + Concept determination (hvad har vi brug for af arkitektur infomationer) -> Source and Target viewpoints
  + Data Gathering (source kode og eksisterende documentation (som hypotese))
  + Knowledge inference (passer hypotesen – source -> target view)
  + Information Interpretation (visualisering, UML)
  + Manuel, Semi-automatisk, Automatisk
  + MagicDraw (UML fra source) – kun statisk view
  + Dynamisc view (component connector, run-time) – eksekvering
    - Instrumentering (JSeq)
  + Concept determination er interativ med forfining
  + Placer data i database så der kan queries på det, sorteres, opdateres og skjules uvæsentlige dele (manuel)

# 6. BigData and CAP Theorem.

* To punkter: Dataens størrelse og hastigheden hvormed data kan quiries.
* En database er delt en ressource,
* Traditionelle RDBMS er én instans.
* Flere queries => større HW (vertikal skalering). Dårlig performance => tabte kunder
* RDBMS sharding er en mulig løsning (fordeling af rækker på forskellige DB). Manuel process. F.eks. fordel Danske kunder og europæiske kunder og amerikanske kunder på forskellig DB.
* Store datamængder i RDBM er langsomt p.g.a. random read (sequentiel er hurtigere) samt disk access (det hele kan ikke være i RAM). Mindre parallel processering.
  + RDBMS har ingen intern ordering => random
  + Normalisering for at undgå duplikater
  + Ved at anvende normalisering og lægge data i logisk rækkefølge bliver ”ofte” look-up hurtigere, men footprint større.
* RDBMS er ACID
  + Atomicity (Alt eller intet)
  + Concistency (Før og efter transaction er DB consistent)
  + Isolation (transaction – sikre ingen andre ændre DB samtidig)
  + Durability (Transaktion kan ikke overskrives (uheld) efter commit)
* CAP
  + Consistency (handlingerne blev udført som én)
  + Availability (operationen er færdig indenfor forventet tid)
  + Partition Tolerence (Operationen returnere altid)
* Horizontal skalering (Partition Tolerance valgt)
  + RDBMS = consistency
  + NoSQL = Availability (eventual consistency)
  + Med ACID => multipel commit (availability skal multipliceres => mindre availability)
  + BASE
    - Basically available
    - Soft state
    - Eventual consistency
    - Der kan være inconsistens mellem det skrevne og det læste

# 7. NoSQL databases and MapReduce.

* NoSQL = Not only SQL
* Horizontal scaling (mange servere, replikering og denormalisering, simpelt interface, not ACID, parallel processering da data er distribueret, dynamis records (RDBMS = fixed tables)
* Simple key-value lookup. Ingen Join
* Mange servere = store risk for failure
* Horizontal partitioning = fordeling af tabeller på forskellige servere.
* Key-value
  + Automatic sharding + replication.
  + Only look-up on primary key
  + Preferable if sufficient
  + MVCC – Multi verison concurrency control
    - Timestamp til at sikre nyeste replikering
* Document
  + Mongo (JSON)
  + Look-up inde I document
  + Atomic update eller ingen concurrency (best effort)
  + Automatic sharding
  + Master-slave replication
* Extensible records
  + Row-column tables split both ways
  + Memory-based with flush to disk
* MapReduce
  + Query på documenter men kan gøres distribueret
  + 2 (3) metoder
    - Map (emit the key,value pair of interrest – value may be document)
    - Reduce (loop the emitted keys with a value array). Fordel da et kan distribueres og ofte er data fordelt efter key på maskinerne.
      * Reduce kan kaldes flere gange, inklusiv med resultatet af et tidligere reduce – dette skal der tages højde for.
    - Finalize (kaldes én gang for hver key,value emitted)