# Woche 1

* Definition einer Cloud nach NIST
  + Selbstbedienung/Selbstzuweisung: Das Bereitstellen neuer Ressourcen (z.B. CPU) passiert automatisch und ohne manuelle Interaktion ab
  + Weitgehender Netzwerkzugriff: Das System ist mit Standardmechanismen über das Netz verfügbar
  + Ressourcen-Pooling: Die Ressourcen werden vom Anbieter in Pools gebündelt. Der Anwender bezieht diese dann, weiss aber nichts von ihrer Gestaltung und Lokalität
  + Schnelle Elastizität: Services können schnell bezogen und auch wieder zurückgegeben werden
  + Gemessene Ressourcennutzung: Die Ressourcennutzung wird gemessen und überwacht
* Servicemodelle einer Cloud
  + Infrastructure as a Service (IaaS): Es werden nur die Hardwareressourcen zur Verfügung gestellt (CPU, RAM, Disk) und der Kunde muss für Betriebssystem und Software selber sorgen. Je nachdem wird das Bereitstellen eines Betriebssystems hinzugerechnet. Beispiele: OpenStack
  + Platform as a Service (PaaS): Es wird eine Infrastruktur und mindestens auch ein Betriebssystem zur Verfügung gestellt. Typischerweise kommen auch generische Dienste wie ein SQL Server oder Webserver dazu. Unklar ist, ob auch Lizenzen dazugehören. Beispiele: CloudFoundry, OpenShift
  + Software as a Service (SaaS): Weitgehendes Bereitstellen einer fertig vorkonfigurierten Softwarelösung. Beispiele: Office 365
* Aufbau einer typischen Virtualisierung
  + Traditionell: Alle Komponenten werden redundant ausgelegt (Doppelter Loadbalancer, doppelte Leitungen/Switches/Router und doppelte Datenbankserver). Hinzu müssen die Rechner je für eine Systemwartung und einen Ausfall gewappnet sein und die restlichen Systeme müssen immer die doppelte Last tragen können. Traditionell verwendet man spezialisierte, aber sehr teure Hardware
  + Modern: Man verwendet billige, gerade noch ausreichende Consumer Hardware und nimmt die Speicherressourcen wieder in den Server hinein. Mittels intelligentem Softwarealgorithmus werden die Daten dann auf mehreren Systemen abgelegt und nach einem Festplattenwechsel wieder abgeholt (Google Ansatz)
* Skalierungsarten
  + Vertikale Skalierung (Scale up): Einem System werden mehr/bessere Komponenten hinzugefügt, was schnell sehr teuer werden kann und auch an gewisse Grenzen stösst. Hinzu kommt, das gewisse Komponenten wie z.B. Storage ab einer gewissen Grösse wieder neue Probleme mit sich bringen (Replikationszeit einer 10 TB Festplatte)
  + Horizontale Skalierung: (Scale out): Es werden zusätzliche Systeme hinzugefügt, um die Aufgaben zu verteilen oder den Service mehrfach anzubieten
* Typischer Loadbalancer: Verteilt Last auf mehrere Endgeräte. Eigenschaften:
  + Verteilungsstrategie: Nach welcher Strategie werden neu ankommende Verbindungen verteilt. Typische Strategien sind Round Robin (via DNS oder direkt) oder Load Distribution
  + Scheduling Disziplinen: Nach welchem Grundsatz werden Backends zugeteilt. Typische Strategien sind hier beispielsweise FIFO oder Earliest Deadline First
  + Persistenz: Wie werden eingehende Verbindungen von bekannten Clients identifiziert, damit sie immer an dasselbe Backend geleitet werden (Stickyness)? Das kann nach Protokollkriterien (z.B. Senderaddresse) oder nach Inhalt (z.B. Cookie) erfolgen. Bei verschlüsselten Verbindungen muss der Loadbalancer je nachdem die verschlüsselte Verbindung terminieren und nach hinten mit Zertifikaten weiterarbeiten
  + Monitoring: Wie werden die Backends überwacht? Über einen simplen Ping, nimmt er TCP Verbindungen auf einem bestimmten Port an oder über eine vollständige Prüfung respektive Statusabfrage

# Woche 2

* Begrifflichkeiten in einem Cluster
  + Als Host wird ein einzelner Server bezeichnet, welcher über virtuelle Maschinen, die sogenannten Guests, verfügt.
  + Das Betriebssystem des Hosts wird als Host OS, das der Guests als Guest OS bezeichnet
  + Speicher und CPU sind hostspezifisch und können nicht mit anderen Hosts in einem Cluster geteilt werden
  + Bei Containern wird nicht nur die Hardware virtualisiert, sondern auch das Betriebssystem
* Sinn des Ressourcen-Pooling in einem virtualisierten Cluster
  + Erhöhen der Verfügbarkeit von Gästen
    - Fault Tolerance: Ein Guest läuft auf zwei unabhängigen Hosts
    - High Availability: Beim Ausfall eines Hosts werden seine VMs auf einem anderen Host neu gestartet
  + Pooling der Resourcen, namentlich Diskkapazität: Durch das Deduplizieren von doppelten Daten (z.B. ein Grossteil eines Linuxsystems) kann Speicherplatz und Migrationszeit eingespart werden
* Sinn der Virtualisierung ist die Kostenersparnis, da man keine einzelnen Rechner mehr kaufen muss
* Typische Speichersysteme eines Clusters
  + Traditionelles repliziertes SAN (Storage Attached Network): Blockdevice, welches über ein dediziertes Netzwerk angeschlossen ist (z.B. Fiber Channel)
  + Repliziertes NAS (Network Attached Storage): Billige Consumerlösung, welche via Protokolle wie SMB, NFS oder iSCSI ein Speicher zur Verfügung stellt
  + Hardcorelösung mit Storage im Server à la Google: Zusätzliche Softwarelösung zum Duplizieren und Replizieren mit anderen Systemen notwendig
* Klassifizierung von Hypervisoren
  + Type 1 oder Bare Metal Hypervisor: Der Hypervisor läuft direkt auf der Hardware und wird nicht durch ein Gastbetriebssystem verwaltet. Beispiele: VMware ESXi oder XEN
  + Typ 2 Hypervisor: Der Hypervisor läuft auf einem Gastbetriebssystem und verwendet dessen Ressourcen. Beispiele: Oracle VirtualBox oder Parallels
  + Container (Eigentlich streng genommen kein Hypervisor): Die virtuellen Maschinen, sprich Container, verwenden das Betriebssystem des Host und bilden einen kleinen Teil davon virtualisiert nach. Beispiele: Docker, Linux LXC und Canonical LXD
* Auswirkungen der Virtualisierung auf Applikationen
  + Ressourcen sind unterschiedlich verfügbar
  + Die Latenzzeit und Transferleistung von Ressourcen kann abhängig von der Auslastung des Hosts oder der Verfügbarkeit seiner Komponenten sein (z.B. verursacht ein replizierendes SAN einen Einbruch des I/O Throughput)

# Woche 3

* Gemäss IEEE 829 müssen für einen [reproduzierbaren] Testplan folgende Punkte dokumentiert werden
  + Test plan identifier
  + Introduction
  + Test items
  + Features to be tested
  + Features not to be tested
  + Approach
  + Item pass/fail criteria
  + Suspension criteria and resumption criteria
  + Test deliverables
  + Testing tasks
  + Environmental needs
  + Respsonsibilities
  + Staffing and training needs
  + Schedule
  + Risk and contingencies
  + Approvals
* Dies kann wie folgt umgesetzt werden
  + Master Test Plan (MTP): Eigentlicher Testplan nach IEEE 829
  + Level Test Plan (LTP): Mehrere, genauer definierten Tasks zum Testen nach IEEE 829
  + Level Test Log (LTL): Ergebnisse der LTPs
  + Level Test Report (LTR) Interpretierung des LTL im Kontext des MTP
  + Master Test Report (MTR) Interpretation aller LTR im Kontext des MTP
* Aufbau JMeter
  + Test Plan: Hier können gewisse Punkte des IEEE 829 umgesetzt werden (aber eben nicht alle!)
  + Workbench: Hier werden die Testresultate festgehalten
* Überlegungen zum Testinhalt
  + Was möchten sie testen: Welche Leistungen (und nicht Funktionen) möchte man testen. Mögliche Werte sind maximale Durchsatzrate, durchschnittliche Antwortzeit etc.
  + Wie möchten sie testen: Welche Vorbedingungen brauchen sie? Sind am Anfang einer Messung Cache- oder sonstige Effekte zu beobachten? Sind diese für sie interessant oder nicht?
  + Welche Aussage erwarten sie von den Resultaten: Welche Leistungsdaten möchten sie ausgewiesen haben? Latenzzeiten? Maximale Benutzerzahl? Anzahl der Anfragen pro Sekunde?
* Überlegungen zur Testumgebung
  + Welche Umgebung steht zur Verfügung: Ist die Testumgebung überhaupt geeignet um Tests durchzuführe? Wie müsste eine ideale Umgebung aussehen und welche Falschaussagen der Tests erwarten sie aufgrund des realen Setups?
  + Wie können sie auf der vorliegenden Umgebung sinnvoll testen: Wie müssen sie Testfälle und Kennzahlen wählen, damit sinnvolle Resultate erzielt werden
  + Generell: Identische Kopie der Produktivumgebung vs. Lokales Testen vs. Cloud-Testing 🡪 Einpendeln der Requests auf einen Deltawert + Kommunikation innerhalb eines Clusters, welches ein anderes Cluster testet
  + Generell: Testresultate eines guten Performancetests bilden nie die Realität ab. Sie erlauben uns nur die Realität realistisch abzuschätzen

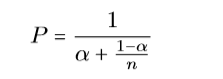
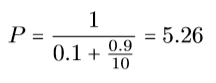
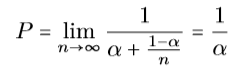
# Woche 4

* Testtypen
  + Stresstest: Testen bei welche, Punt eine Applikation zusammenbricht und welcher Art der Zusammenbruch ist (Applikation ist nicht mehr verfügbar, Applikation wird unzuverlässig, Applikationsdaten werden inkonsistent etc.). Ferner ist es wichtig zu wissen, wie die Recovery vonstattengeht (Automatische Recovery nach Abklingen der Last, manuelle Recovery notwendig, Recovery mit Datenverlust, Recovery nur in kannibalisierter Form). Stresstests finden nie oder nur in Ausnahmefällen auf produktiven Umgebungen statt
  + Performancetest: Performancetests analysieren das Verhalten einer Applikation unter einer definierten Last. Die Definition der Last ist dabei häufig willkürlich gewählt und soll einer möglichst durchschnittlichen Benutzung entsprechen. Performancetests können in Ausnahmefällen auf einer produktiven Umgebung stattfinden (Beispielsweise während eines Wartungsfensters)
  + Leistungstests/Bottleneck-Analysen: Leistungstests werden zum Optimieren von Teilfunktionen verwendet. Zwar wird auch bei einem Leistungstest eine definierte Last auf das zu testende System gegeben. Anders als beim Lasttest liegt der Fokus hier aber nicht auf der Infrastruktur als Ganzes, sondern im Finden ganz spezifischer Limitierungen des Systems (Bottlenecks), die es zu optimieren gilt.
* Lastgenerator: Wenn die Systeme nicht in ihrer Leistung stabil sind, muss die erzeugte Last gemessen oder geregelt werden. Das ist bei Systemen mit mehreren unabhängigen Entitäten schwierig
  + Server gestützte Lastgeneratoren
  + Cloud gestützte Lastgeneratoren
  + Farm gestützte Lastgeneratoren
* Lastverhalten
  + Das Verhalten beim Erhöhen einer Last: Kurzfristig (Sekunden bis Stunden) können Effekte Mechanismen wie Buffering, Caching oder Parallelisierung das Resultat stark beeinflussen. So kann es beispielsweise zu Peaks (z.B wenn Strukturen initialisiert werden müssen), verzögerten Ramp-Ups (z.B. wenn ein Buffer die Last vorübergehend auffängt) oder gar Drop-Ins (z.B. wenn adaptiv Cachesizes festgelegt werden) kommen
  + Das Verhalten während der definierten Last: Dieses Verhalten ist typischerweise konstant, kann aber durch Effekte wie Logging oder Memoryleaks beeinträchtigt werden
  + Das Verhalten beim Herunterfahren der Last: In dieser Phase ist typischerweise die Erholung des Systems interessant. Wie schnell reduzieren sich Disk I/O oder Antwortzeiten (In Spezialfällen wie Herunterfahren von Concurrent Threads können diese sogar erhöht werden).
* Stressverhalten einer Applikation
  + Einzelne Aussetzer: Wie viele Aussetzer akzeptiert werden ist hierbei Definitionssache. Je nach Anwendung sind von einer Nulltoleranz bis hin zu mehreren Prozenten alle Werte zulässig
  + Totalausfall: Dieser Zustand ist am einfachsten zu detektieren
  + Datenverlust/Inkonsistenz: Dieser Zustand ist teilweise nur schwierig zu erkennen
  + Failover/Kannibalisierung von Ressourcen: Wenn ein simples Failover auf ein Reservesystem oder Verteilen der Last auf verbleibende Nodes erfolgt, ist zunächst das Einzelsystem mehr belastet (Die Clusterlast bleibt ja per Definition konstant). Je nachdem erzeugt aber der Failover zusätzliche Last, die durch das Kompensieren des Ausfalls entsteht (Bei einem Virtualisierungssystem kann das beispielsweise das Umverteilen von Guest-Nodes sein). Diese zusätzliche Last, die über extern generierte Last hinausgeht, bezeichnet man als Kannibalisierung. Das System verwendet zusätzlich Ressourcen, um im Fehlerzustand noch richtig funktionieren zu können.

# Woche 5

* Funktionen eines Caches
  + Optimiert Zugriffe einer Einheit auf ein Medium. Werte müssen in einem Cache viel schneller abgelegt und gelesen werden können als im Original
  + Optimiert Lese- und je nachdem auch Schreibzugriffe von/auf ein Medium. Zielt im besten Fall auf zirkuläre und zusammengefasste Zugriffe ab
  + Je grösser der Cache, desto länger due Suchdauer 🡪 Ein Cache kann also nicht beliebig gross sein und wird oft als Hashstruktur/Addressstruktur implementiert
* Cachetypen
  + Lese-Cache: Enthält Datenblöcke für bereits erfolgte Leseoperationen. Kann auch spezielle Funktionen wie ein Prefetch für ein sinnvolles Vorladen beinhalten
  + Schreib-Cache: Konsolidiert die Schreiboperationen um Leistungen zu sparen. Entweder durch das Zusammenfassen mehrerer gleicher Schreiboperationen auf den gleichen Block oder das Zusammenfassen von mehreren Schreiboperationen auf hintereinanderliegende Datenblöcke. Je nachdem kann auch der Zeitpunkt der Schreiboperation beeinflusst werden
* Unterschied Buffer und Cache: Unterschied Buffer vs. Cache: Ein Buffer ist eine Warteschlange in den Daten zur Verarbeitung zwischengespeichert werden. Ein Cache beinhaltet hingegen Daten, die in der Zukunft wieder wichtig sein werden könnten und die man dann nicht nochmals neu beschaffen möchte
* Aufbau eines Caches
  + Limitierte Grösse
  + Enthält nur Datenobjekte vom gleichen Typ
  + Optimiert die Leistung
* Daumenregel für Caches: Ein klassischer Lesecache lohnt sich typischerweise, wenn zwischen gecachtem Medium und verarbeitender Einheit einen Geschwindigkeitsunterschied von Faktor 10 besteht und der Cachespeicher nicht mehr als Faktor 3 langsamer ist als die verarbeitende Einheit
* Wichtige Kennzahlen und Begriffe
  + Ein Zugriff auf ein durch ein Cache verwaltetes Medium nennt man Cache Access – unabhängig davon, ob der gecachte Wert oder der vom Medium verwendet wird
  + Ein Zugriff, welcher vom Medium geholt werden muss, nennt man Cache Miss. Die Cache Miss Ratio = Cache Miss / Cache Access
  + Ein Zugriff, welcher vom Cache geholt werden kann, nennt man Cache Hit. Die Cache Hit Ratio = Cache Hit / Cache Access
  + Ein Cache mit einer hohen Hit Ratio wird als hot bezeichnet, einer mit einer hohen Miss Ratio als cold
  + Ein noch nicht auf das Medium zurückgeschriebener Block wird als dirty bezeichnet
* Cachestrategie mit dem Bélády Algorithmus: Der effizienteste Caching-Algorithmus ist derjenige, bei dem immer das Element aus einem Lesecache entfernt wird, welches am weitesten in der Zukunft wieder gebraucht wird
  + Typischerweise gibt es aber keine Funktion, welche Aussagen über den Gebrauch in der Zukunft machen kann. Man optimiert also auf den Algorithmus hinaus, kann ihn aber nie erfüllen
* Cachestrategie First In First Out (FIFO): Beim First In First Out Cache wird bei Bedarf immer das Element entfernt, welches sich am längsten im Cache befindet. Er lässt sich als einfachen Ringbuffer ohne besondere Anforderungen realisieren 🡪 CircularFifoBuffer aus Apache Commons
* Cachestrategie Least Recently Used (LRU): Beim Least Recently Used Cache wird bei Bedarf immer das Element entfernt, welches am längsten nicht zu einem Cache Hit geführt hat.
* Cachestrategie Most Recently Used (MRU): Beim Most Recently Used Cache wied bei Bedarf immer das Element entfern, welches am jüngsten zu einem Cache Hit geführt hat
* Cachestrategie Random Replacement (RR): Beim Random Replacement Cache wird bei Bedarf immer ein zufälliges Element entfernt
* Cachestrategie bei Schreibcaches: Write through vs. Write back
  + Write through: Es findet kein Caching von Schreibzugriffen statt. Jeder Schreibzugriff gilt erst als vollzogen, wenn er im Medium gespeichert wurde
  + Write back: Der Wert wird nur in den gesch geschrieben (dirty) und zu einem späteren, undefinierten Zeitpunkt zurück auf die Disk geschrieben
* Cachestrategie bei Lesecaches: Prefetch und Speculative Prefetch
  + Prefetch Cache: Der typische Prefetch Cache ist beispielsweise der Spurcache einer Festplatte. Bei einem Zugriff auf einen Block einer Spur wird immer die gesamte Spur gecached weil die Zugriffszeit auf eine Spur (10 ms) ungleich höher ist als die Lesezeit für eine Spur (0.5 ms). Ein Prefetch Cache von dieser einfachen Art belastet das darunterliegende Datensystem nur sehr begrenzt
  + Speculative Prefetch Cache: Beim Speculative Prefetch Cache geht es darum, die nächsten Zugriffe auf das Medium zu erahnen (Im Sinne von berechenbarer Wahrscheinlichkeit) und entsprechende Datenblöcke vorausschauend vom Medium zu beschaffen. Für Vorhersagen können Hinweise in den Daten oder Hinweise im Zugriff auf die Daten verwendet werden
* Verteilte Caches
  + Caches welche verteilt sind (zum Beispiel 1st-Level-Caches von CPU-Cores oder ihr Anwendungscache) haben ein wichtiges Problem: Das Cache-Kohärenz-Problem (Cache Coherence Problem).
  + Ein Cache kann einen Datenblock eines Mediums cachen, während dem eine zweite Verarbeitungseinheit auf eben diesen Datenblock schreibt. Wenn die erste Verarbeitungseinheit diesen Datenblock anfordert, erhält sie von ihrem Cache nur den gecachten und mittlerweile ungültigen Block.

# Woche 6

* Performance Management in einer Cloud – relevante Eckdaten
  + Beim Performance Management werden relevante Eckdaten, sogenannte Key Performance Indicator
  + Im Cloudbereich könnten dies beispielsweise die Kosten oder aber die durchschnittliche/mediane/n.te Quartil Antwortzeit sein. Diese Eckdaten werden stetig erfasst und überwach. Beim Unter- oder Überschreiten von Schwellwerten müssen Massnahmen getroffen werden
  + Schlechte Kenndaten wären zum Beispiel Stress- oder Lastverhalten. Auch Verfügbarkeit ist twar als KPI für eine Applikation gut, aber schlecht direkt zu steuern. Hochvolatile Kenndaten (CPU Last, kurzzeitige Run-Queue-Length) sind ebenfalls schlecht geeignet
* Relevante KPI messen und Schwellwerte definieren
  + Relevante Eckdaten werden permanent erhoben. Normalerweise werden zwei Schwellenwerte mit einer ausreichend grossen Hysterese definiert. Beim Unterschreiten der Nominallast unter den unteren Hysteresewert werden Leistungsfaktoren aus der Applikation entfernt (z.B Frontendserver). Und beim Überschreiten welche hinzugefügt
  + Eine zu schmale Hysterese führt zu einem hohen Aufwand bei der Provisionierung und Deprovisionierung von Teilsystemen
  + Normalerweise ist nur eine komplette Deprovisionierung respektive Provisionierung von Teilsystemen sinnvoll. Auch deaktivierte/heruntergefahrene Systemen verwenden Systemressourcen (z.B. Festplattenspeicher) und müssen gepflegt (z.B. Patching) werden. Deshalb wird im Optimalfall immer ein automatisches, vollständiges Provisionieren des kompletten Teilsystems angestrebt
  + Wegen Limitierungen beim maximalen RAM und der maximalen Corezahl (CPU) wird hier praktisch ausnahmslos Scale-Out (horizontal) statt Scale-In (vertikal) betrieben
* Amdahls Gesetz
  + Amdahl macht die Beobachtung, dass ein Teil der ausführbaren Tasks eines Systems Parallelisierbar sind während andere nur sequentiell abgewickelt werden können. In unserem Fall sind dies beispielsweise das Loadbalancing-Modul und die Storage (Schreiben ist nur exklusiv möglich). Wir bezeichnen den sequentiellen Teil mit α und den Parallelisierbaren Anteil mit 1−α. Mit l bezeichnen wir die einzelnen, gleichstarken Leistungseinheiten eines Systems. Wenn P die Gesamtleistung eines Systems in Relation zu einer einzelnen Verarbeitungseinheit darstellen soll, gilt nach Amdahl:  
    
  + Das heisst, dass eine einzelne Leistungseinheit unabhängig von Faktor immer 1 ist. Bei Alpha = 0.1 und n = 10 gilt bereits:   
    
  + Es wird also bereits die Hälfte der intuitiv erwarteten Leistung verheizt. Es ist auch einfach aus der Formel ersichtlich, dass im vorliegenden Fall nie mehr als die zehnfache Leistung erbracht werden kann:  
    
  + In der Realität ist es normalerweise sogar so, dass die Leistung bei zunehmenden Nodes einknickt. Dies aufgrund der Tatsache, dass der Management Overhead mit jeder zusätzlichen Node zunimmt. Das ist implizit auch in Amdahls Formel enthalten. Alpha ist nämlich bei einem realen System keine Konstante, sondern eine Funktion, die typischerweise von n abhängig ist, wobei sich n typischerweise exponentiell Einzug hält: z.B Alpha = f(n) = c + ((1 + Beta)^n -1))
* Automatisierte Scale-Out/Scale-In
  + Bei einem automatisierten Scale-out müssen bei einer n-Tier Architektur folgende Operationen automatisiert ablaufen können:
    - Deployment der BasisVM/Container inklusive der Installation und dem Patching aller Software und OS-Komponenten
    - Node-spezifische konfiguration
    - Clustering der Software intern (z.B. Replikation des gegenwärtigen Datenbestandes) inklusive Aufnahme der Node in den Cluster.
    - Starten aller Dienstleistungen auf der Node
    - Aufnahme der Node in externe Dienstleistungen (z.B. Monitoring)
    - Testen der Funktionsfähigkeit
    - Integration in den Cluster (Eventuell anpassen der Loadbalancer-Konfiguration)
  + Bei einem Scale-in analog:
    - Entfernen der Node aus dem Cluster
    - Entfernen der Node in den externen Dienstleistungen
    - Graceful shutdown der Dienstleistungen auf der Node
    - Entfernen des Software-Clusterings
    - Deprovisionierung aller Ressourcen
  + Aus Sicherheitsgründen werden solche Systeme normalerweise nicht automatisch gepatched. Normalerweise werden einfach die am längsten laufenden Nodes von Zeit zu Zeit durch neu provisionierte ersetzt (zuerst neues System provisionieren; anschliessend altes System de-provisionieren). Das sorgt für ein stabiles Verfahren beim Patching. Sorgt eine neue Software oder Systempatch für Probleme wird dies üblicherweise beim Systemtest des neuen Systems erkannt und das alte weiterhin laufen gelassen. Gleichzeitig muss natürlich eine Alarmierung des Supports erfolgen um das Problem zu beseitigen.
* Herausforderungen von Hardware Scaling
  + Damit eine Cloud Kosteneffektiv ist, muss auch die Hardware Skalieren. Dies gilt sowohl in der Anschaffung (es ist normalerweise nicht Kosteneffizient Enclosures (bei Blade-Servern) mit einem Befüllungsgrad unter 80−90% zu betreiben). Für Clouds werden sinnvollerweise nicht Server mit Redundanzen, sondern solche mit einem möglichst guten Preis/Leistungs-Verhältnis eingesetzt (häufig sogar Consumerware). Der Ausfall von Nodes wird typischerweise jederzeit in Kauf genommen. Entsprechend wird auch das Hardware-Scaling stark automatisiert. Typischerweise werden die Netzwerk- und Strom-Anschlüsse normiert angeschlossen, die Mac-Adressen erfasst und dann via einen Netzwerk-Boot-Prozess (PXE-Boot) das entsprechende System aufgesetzt.
  + Hardware sollte immer gut ausgelastet sein. Nicht benötigte Hardware sollte automatisiert heruntergefahren werden, um Kosten zu sparen. Dies wird normalerweise aber nicht gemacht, weil damit “nur” Strom gespart wird und das Wartungs-Personal sowie die erwarteten Folgeschäden durch häufiges auskühlen normalerweise teurer eingeschätzt werden. Ausserdem wird das normalerweise grösste Problem
  + Als “Light-Variante” des Herunterfahrens ist aber heute das aktivieren der Stromsparmassnahmen wieder in Mode gekommen. Diese Praxis war bei den Servern lange Zeit verpönt.