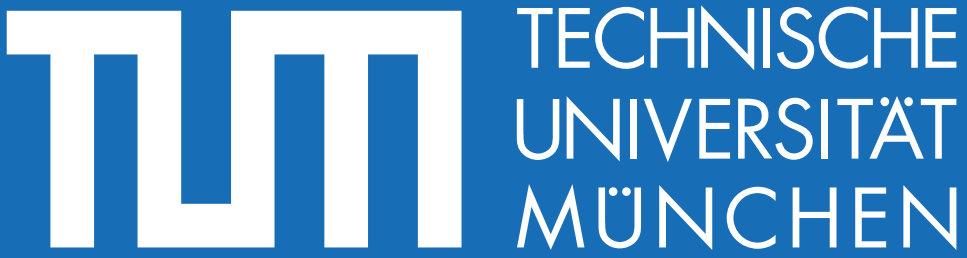
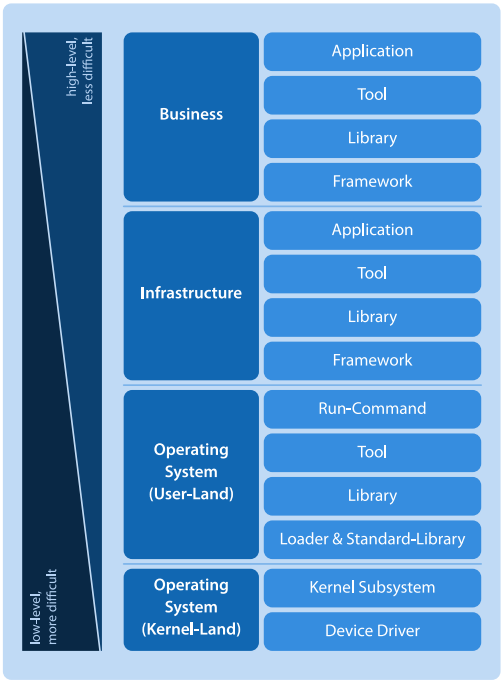
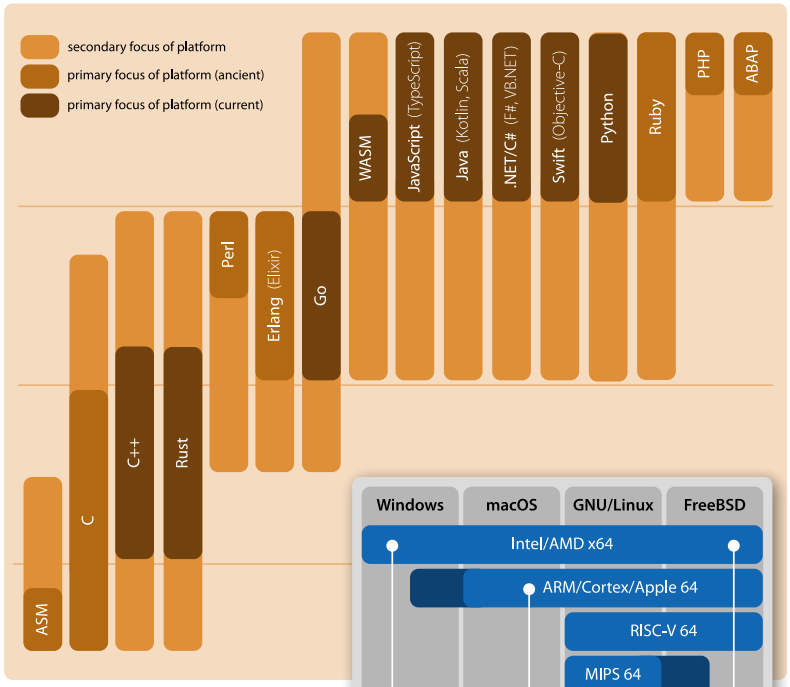


Software Engineering in Industrial Practice (SEIP)

Dr. Ralf S. Engelschall

Technology Platforms





Remember:
A *Technology Platform* is less about choosing a particular programming language and more about choosing a particular ecosystem for targeting a particular level of software!

Opinionated Recommendation (as of 2022):
Business: **Scala, Kotlin, TypeScript, AssemblyScript**
Infrastructure: **Go, Rust, Scala, Kotlin, TypeScript**
Operating System (UL): **Rust, Go**
Operating System (KL): **C, C++, Rust**

Typical Computing Devices (as of 2022):
Intel/AMD x64: **Personal Computer (PC)**
ARM/Cortex/Apple 64: **Raspberry PI, BeagleBone, ROCKSPro64, iMac**
RISC-V 64: **Beagle-V, HiFive Unmatched**
MIPS 64: **Compex WPJ344**

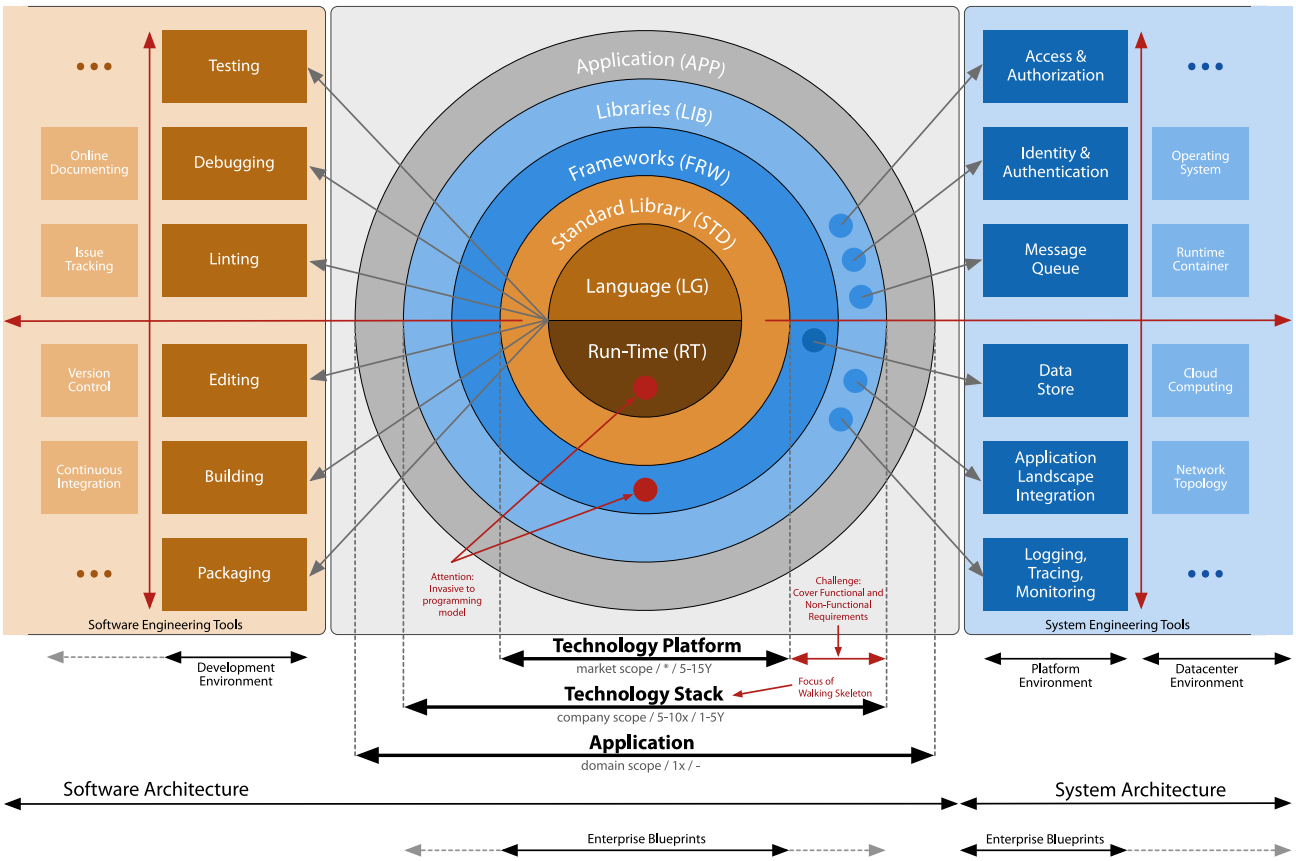
Es gibt verschiedene Ebenen von Software, von “low-level” und schwieriger (auf der Ebene des Betriebssystems) bis “high-level” und weniger schwierig (auf der Ebene von Business-Logik).

Bei einer Technologie-Plattform geht es weniger um die Auswahl einer bestimmten Programmiersprache, und mehr um die Auswahl eines bestimmten Ökosystems, um auf eine bestimmte Art von Software abzuzielen!

Fragen

- Ist die Technologie-Plattform **Node.js** geeignet, um auf der Ebene des Betriebssystems ein Kernel-Subsystem zu implementieren?

Technology Stack



Eine **Technology Platform** besteht aus einer **Language**, einer optionalen **Run-Time**-Umgebung und einer **Standard Library**. Darauf aufsetzend, erweitern **Frameworks** und **Libraries** dies zu einem **Technology Stack**, in dem mit ihnen vor allem die Voraussetzungen zum Erzielen der funktionalen und nicht-funktionalen Anforderungen in der **Application** geschaffen werden.

Es ist zu beachten, daß die **Run-Time** und die **Frameworks** üblicherweise extrem “invasiv” für das Programmiermodell sind und somit fast nie nachträglich ausgewechselt werden können. Deshalb wird beim sogenannten **Walking Skeleton** (deutsch “Technischer Durchstich”) der Fokus vor allem auf den zu definierenden und zu integrierenden **Technology Stack** gelegt.

Während die **Application** einen fachlichen Geltungsbereich besitzt und nur ein Mal implementiert wird, wird ein konkreter **Technology Stack** üblicherweise von einer Firma definiert und dann mehrfach über einen Zeitraum von ein paar Jahren wiederverwendet.

Die unterliegende konkrete **Technology Platform** dagegen wird von einem Dritten für den Markt realisiert, wird beliebig oft wiederverwendet und muss für einen recht langen Zeitraum bestehen.

Große Firmen legen deshalb üblicherweise die zu nutzenden **Technology Platforms** und **Technology Stacks** in ihren **Enterprise Blueprints** stringent fest.

Bei den **Software Engineering Tools** sollte man bei der Definition eines Technology Stacks auch die Werkzeuge für **Testing**, **Debugging**, **Linting**, **Editing**, **Building** und **Packaging** des **Development Environments** mit berücksichtigen, denn diese sind üblicherweise direkt vom konkreten Technology Stack abhängig.

Ähnlich ist es bei den **System Engineering Tools** des **Platform Environments**: diese benötigen im **Technology Stack** mindestens zugehörige **Libraries**, um zu Laufzeit der **Application** angesprochen werden zu können.

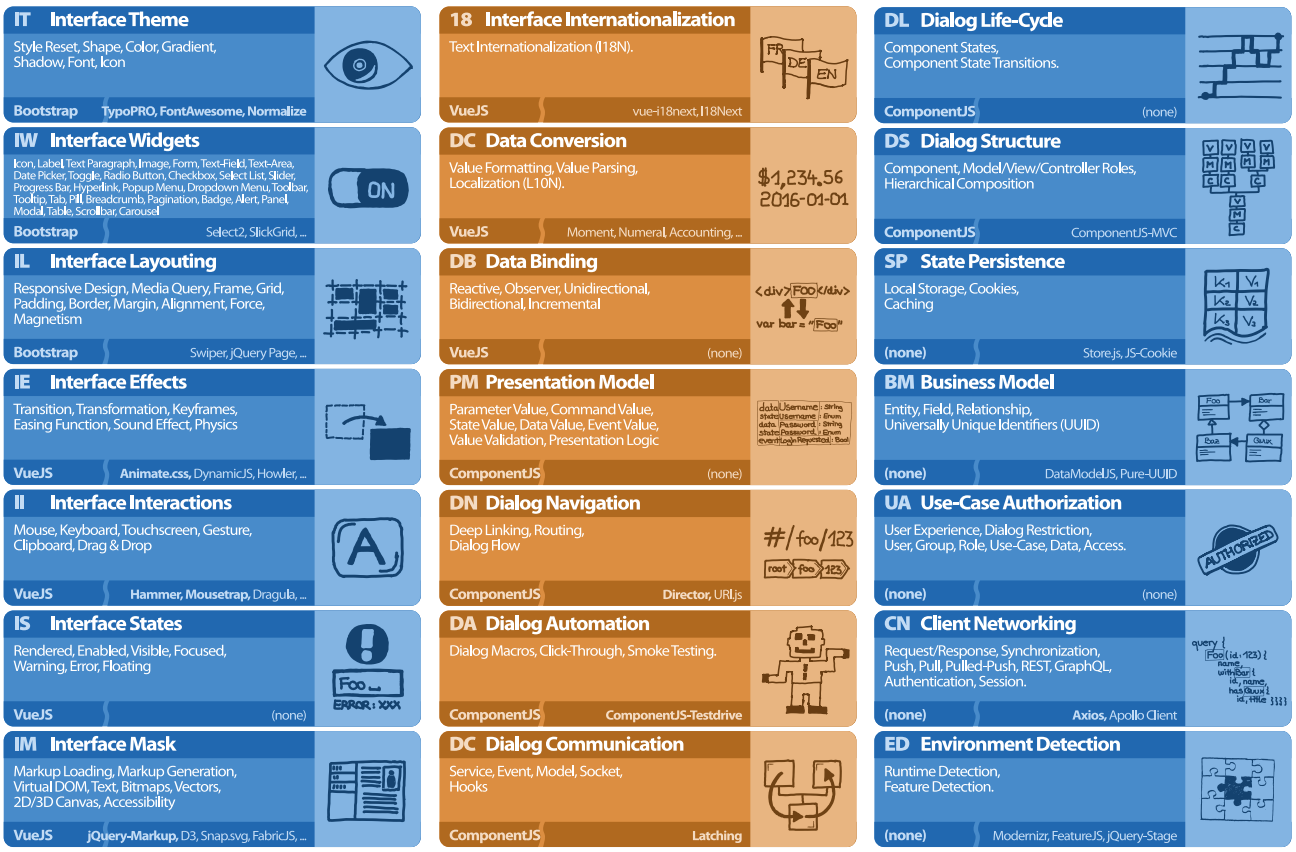
Fragen

- Aus welchen drei Bestandteilen besteht eine **Technology Platform**?
- Aus welchen zwei zusätzlichen Bestandteilen besteht ein **Technology Stack** gegenüber der **Technology Platform**?
- Welche zwei Bestandteile eines **Technology Stack** sind am “invasivstem” für das Programmiermodell?

ARCHITECTURE FUNDAMENTALS

Rich-Client Aspects

TECHNISCHE UNIVERSITÄT MÜNCHEN



AF 12.5

Um einen Technology Stack für einen **Rich-Client** zu definieren, müssen 21 **Aspects** berücksichtigt werden. Jeder Aspect wird dabei mit mindestens einem **Framework** oder einer **Library** abgedeckt. In der Praxis wird üblicherweise jeder Aspect von einem Framework und null oder mehreren Libraries abgedeckt. Das Ziel ist immer: mit einer minimalen Anzahl an Frameworks und Libraries eine möglichst große Abdeckung der Aspects zu erzielen.

Es ist ratsam für sowohl Frameworks als auch Libraries Open Source Software (OSS) zu verwenden und nach Möglichkeit keinerlei Eigenimplementierungen, da üblicherweise sonst der Aufwand nicht im Verhältnis zum Nutzen steht. Denn bei allen Aspects handelt es sich um technische — und nicht fachliche — Aspects einer Benutzeroberfläche.

Im Falle einer Thin-Client Architecture (statt einer Rich-Client Architecture) fallen ein paar Aspects wie **Client Networking** und **Environment Detection** weg. Alle anderen Aspects sind aber weiterhin gültig, auch wenn im Fall einer Thin-Client Architecture das Frontend (und damit die Aspekte der Benutzerschnittstelle) der Application auf dem Server läuft.

Zwei wichtige Aspects behandeln das Datenmodell: das **Business Model** ist ein Datenmodell, welches direkt vom Server kommt und vom Schnitt und der Granularität exakt dem fachlichen Datenmodell des Servers entspricht. Dessen Daten werden mit einem **Presentation Model** synchronisiert, welches vom Schnitt und der Granularität exakt dem (eher technischen) Datenmodel der Benutzeroberfläche (vor allem über die Aspects **Interface Mask** und **Data Binding**) entspricht.

Fragen

- Wie sorgt man in einer **Rich-Client Architecture** dafür, daß die zahlreichen technischen **Aspects** einer Benutzeroberfläche adressiert werden?
- Welche zwei **Aspects** einer **Rich-Client Architecture** halten das Datenmodell und kümmern sich um die Tatsache, daß man die fachlichen Daten, wie sie vom Server geliefert werden, nicht direkt in der Benutzeroberfläche verwenden kann?

ARCHITECTURE FUNDAMENTALS

Thin-Server Aspects

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 12.6

Um einen Technology Stack für einen **(Thin-)Server** zu definieren, müssen 21 **Aspects** berücksichtigt werden. Jeder Aspect wird dabei mit mindestens einem **Framework** oder einer **Library** abgedeckt. In der Praxis wird üblicherweise jeder Aspect von einem Framework und null oder mehreren Libraries abgedeckt. Das Ziel ist immer: mit einer minimalen Anzahl an Frameworks und Libraries eine möglichst große Abdeckung der Aspects zu erzielen.

Es ist ratsam für sowohl Frameworks als auch Libraries Open Source Software (OSS) zu verwenden und nach Möglichkeit keinerlei Eigenimplementierungen, da üblicherweise sonst der Aufwand nicht im Verhältnis zum Nutzen steht. Denn bei allen Aspects handelt es sich um technische — und nicht fachliche — Aspects eines Servers.

Es ist zu beachten, daß ein Server üblicherweise nicht nur den Aspect **Server Networking** (für die Anbindung der Rich-Clients) besitzt, sondern auch den Aspect **Client Networking**, um selbst andere Server abfragen zu können.

Außerdem ist zu beachten, daß vor allem zwei wichtige Aspects den Themenkomplex Security adressieren: der Aspect **User Authentication** identifiziert und authentifiziert den Benutzer (“Ist der Benutzer derjenige?”). Der Aspect **Role Authorization** dagegen überprüft vor allen fachlichen Vorgängen, ob der authentifizierte Benutzer aufgrund seiner Rolle(n) auch wirklich autorisiert ist, die Vorgänge auszulösen (“Darf der Benutzer das?”).

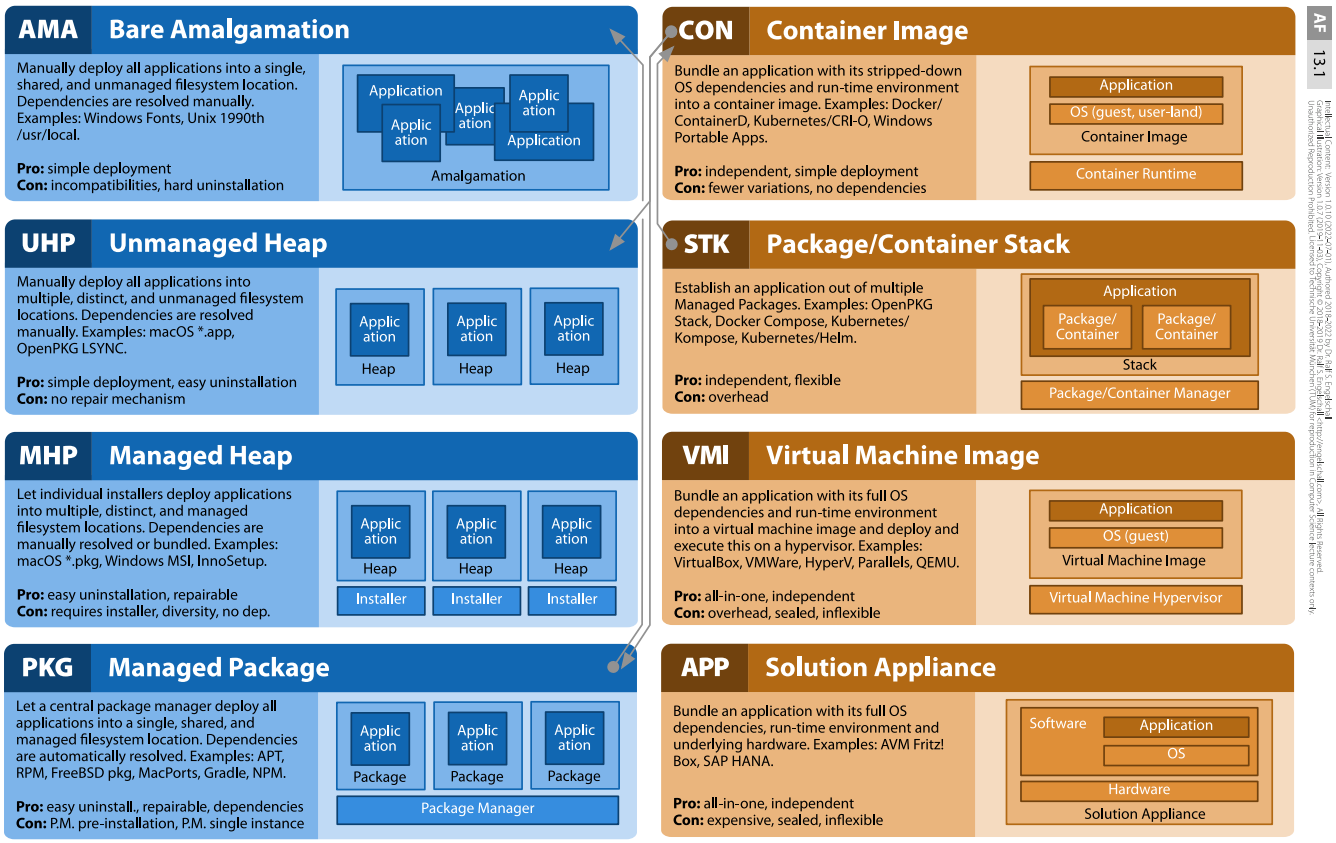
Fragen

- Wieso besitzt ein **(Thin-)Server** üblicherweise neben dem offensichtlichen Aspect **Server Networking** auch den Aspect **Client Networking**?
- Welcher Aspect eines **(Thin-)Server** kümmert sich um die Frage “Ist der Benutzer derjenige”?
- Welcher Aspect eines **(Thin-)Server** kümmert sich um die Frage “Darf der Benutzer das”?

ARCHITECTURE FUNDAMENTALS

Software Deployment

TECHNISCHE UNIVERSITÄT MÜNCHEN



Beim **Software Deployment** wird eine **Application** für die Ausführung auf einem Filesystem installiert. Bei der **Bare Amalgamation** werden die Dateien in ein zentrales Verzeichnis kopiert (z.B. Windows `C:\Windows\system32`). Das ist einfach zu realisieren, erschwert aber später das saubere Entfernen.

Beim **Unmanaged Heap** wird jede Application in ein eigenes Verzeichnis kopiert (z.B. macOS `*.app`). Das ist sehr einfach zu realisieren und erlaubt auch ein leichtes Entfernen. Man hat aber noch keinerlei Reparatur-Möglichkeiten. Beim **Managed Heap** wird dagegen ein eigener Installer pro Application verwendet, um u.a. Reparatur-Möglichkeiten zu erhalten (z.B. Windows MSI).

Beim **Managed Package** wird ein zentraler Package Manager eingesetzt, was die Verwaltung vereinheitlicht (z.B. DPKG/APT oder RPM). Er erlaubt auch das Auflösen von Abhängigkeiten. Will man dagegen die Application unabhängiger vom Betriebssystem und als abgeschirmte Einheit installieren, so bietet sich das **Container Image** Deployment an (z.B. Docker). Hier wird die Application zusammen mit allen ihren Abhängigkeiten und einem Teil des Betriebssystems gebündelt.

Um flexibler zu sein, kann man die **Managed Packages** oder **Container Images** sehr klein halten und stattdessen eine Application über einen ganzen **Package/Container Stack** definieren (z.B. Docker Compose).

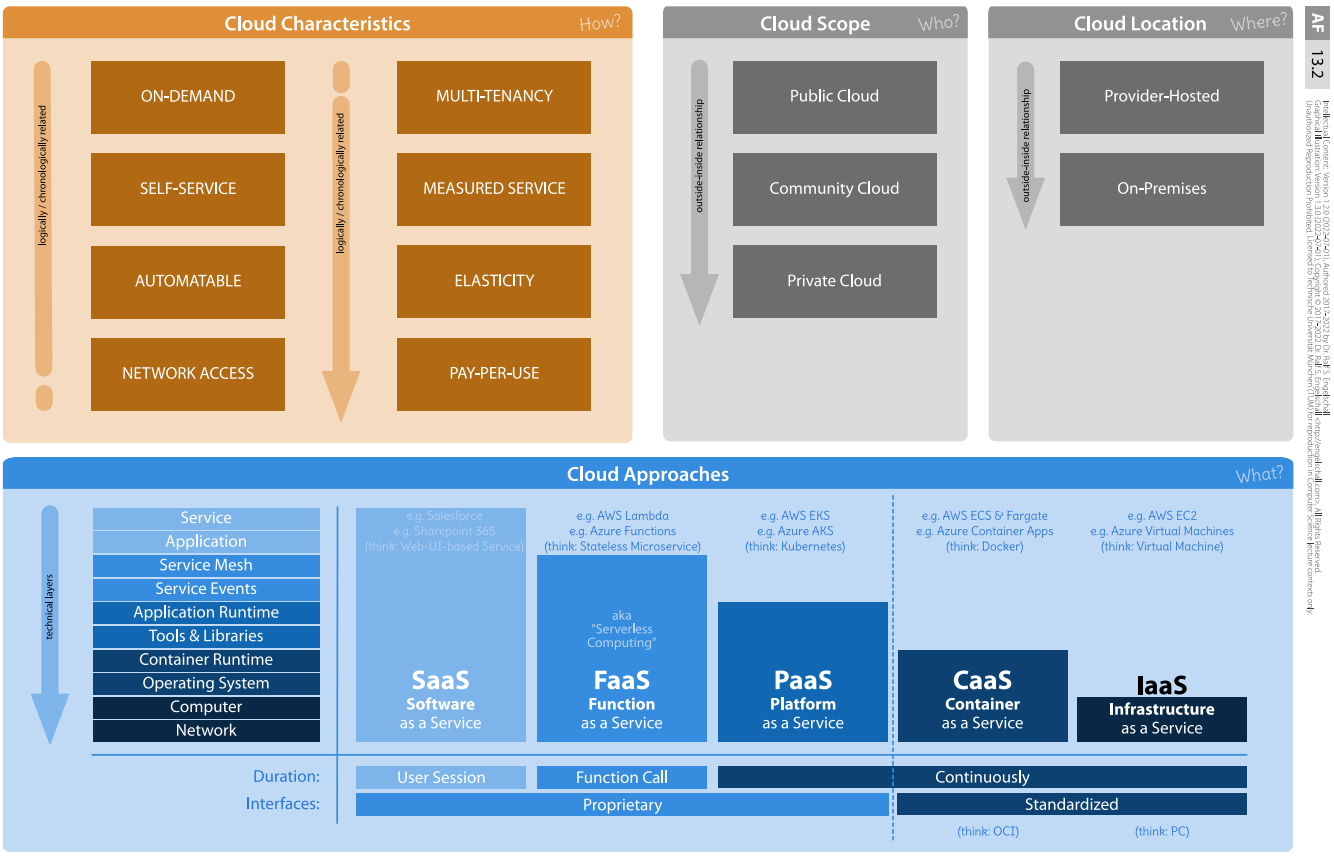
Benötigt man mehr Abschirmung, bietet sich ein **Virtual Machine Image** an. Hier wird die Application mit allen ihren Abhängigkeiten und dem kompletten zugehörigen Betriebssystem gebündelt und auf einer virtuellen Maschine installiert (z.B. ORACLE VirtualBox). Als maximale Ausbaustufe kann die Anwendung auch als **Solution Appliance** installiert werden, bei dem die Application, ihre Abhängigkeiten, das zugehörige Betriebssystem und die unterliegende Hardware zu einer Gesamtlösung gebündelt werden (z.B. SAP HANA).

In der Praxis kommen die verschiedenen Ansätze vorallem in Kombination vor. Ein **Container Stack** besteht aus **Container Images**. Diese wiederum werden dadurch gebaut, daß Abhängigkeiten über **Managed Packages** und die Anwendung selbst als **Unmanaged Heap** in dem Container installiert wird. Die **Managed Packages** werden davor bei ihrer Paketierung über einen **Bare Amalgamation** Schritt erzeugt.

Fragen

- Bei welcher Art des **Software Deployments** wird die Application mit allen ihren Abhängigkeiten und einem Teil des Betriebssystems gebündelt installiert?

Cloud Computing Resources



Cloud Computing hat vier wesentliche Dimensionen. Die erste Dimension **Cloud Characteristics** (“How?”) beschreibt die acht Eigenschaften, wie eine Ressourcen-Bereitstellung passieren muss, damit die Bereitstellung als **Cloud Computing** gilt: **On-Demand**, **Self-Service**, **Automatable**, **Network-Access**, **Multi-Tenancy**, **Measured Service**, **Elasticity** (aka Scalability) und **Per-Per-Use**.

Mit diesen Eigenschaften gibt es in der zweiten Dimension verschiedene **Cloud Approaches** (“What?”), welche angeben, was bereitgestellt wird: bei **Infrastructure as a Service (IaaS)** wird nur **Network** und ein **Computer** bereitgestellt, also üblicherweise eine virtuelle Maschine. Bei **Container as a Service (CaaS)** werden zusätzlich ein (Host) **Operating System** und eine **Container Run-Time** bereitgestellt.

Bei **Platform as a Service (PaaS)** werden zusätzlich umgebende **Tools & Libraries** und eine **Application Run-Time** bereitgestellt; bei **Function as a Service (FaaS)** werden zusätzlich externe **Service Events** und ein **Service Mesh** bereitgestellt und bei **Software as a Service (SaaS)** werden zusätzlich die **Application** und ihr (fachlicher) **Service** bereitgestellt.

Die dritte Dimension **Cloud Scope** (“Who?”) besagt, für wen die Ressourcen bereitgestellt werden: **Public Cloud** für öffentliches Cloud Computing, **Community Cloud** für Cloud Computing einer geschlossenen Gruppe von Organisationen und **Private Cloud** für Cloud Computing einer einzelnen Organisation.

Die vierte Dimension **Cloud Location** (“Where?”) besagt schließlich, wo die Ressourcen physikalisch bereitgestellt werden: **Provider-Hosted** bedeutet bei einem externen Anbieter, **On-Premises** bedeutet lokal bei der nutzenden Organisation.

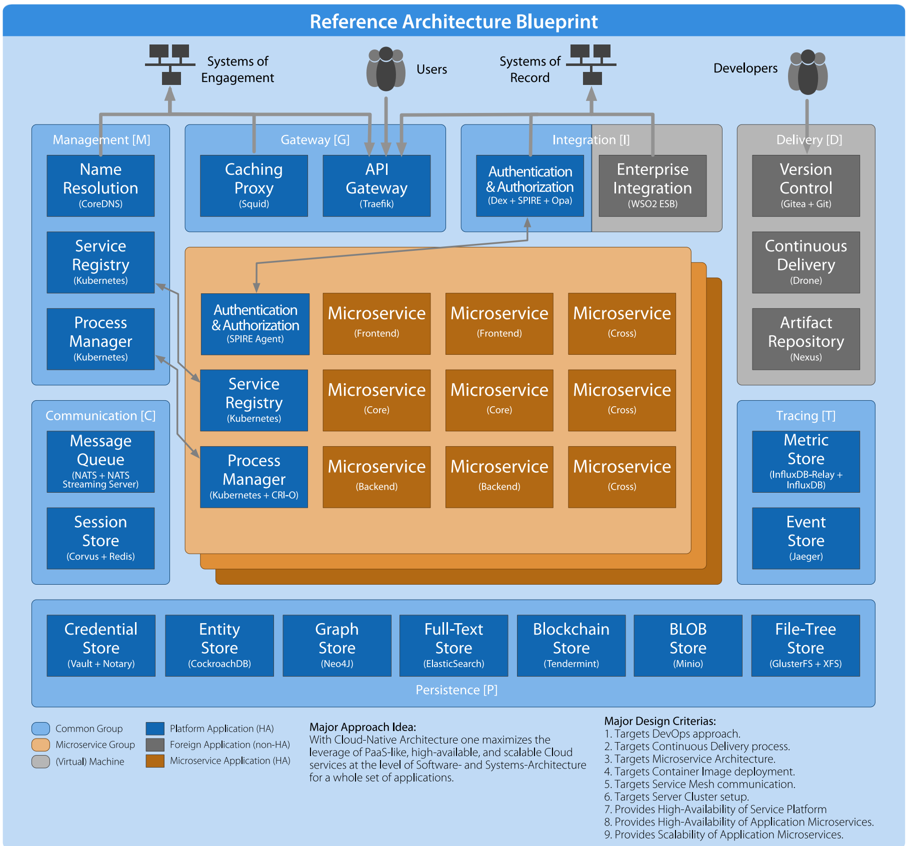
Fragen

- Zählen sie mindestens 5 der 8 **Cloud Characteristics** auf, die eine Resourcen-Bereitstellung erfüllen muss, damit es als **Cloud Computing** gilt!
- Bei welchem **Cloud Approach** wird nur **Network** und **Computer** bereitgestellt?

Cloud-Native Architecture

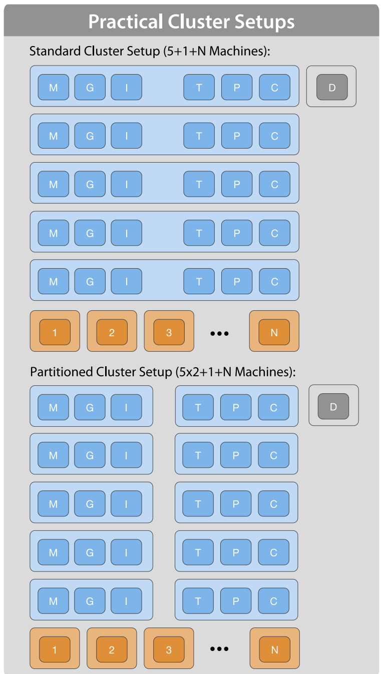
ARCHITECTURE FUNDAMENTALS

TECHNISCHE UNIVERSITÄT MÜNCHEN



CNCF Cloud-Native Definition 1.0

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.



Bei der **Cloud-Native Architecture** werden Anwendungen so entwickelt, installiert und betrieben, daß die Vorteile des **Cloud Computing** maximal genutzt werden und insbesondere alle Infrastruktur-Dienste von einer zentralen **Service Platform** übernommen werden.

In der Praxis bedeutet dies im Idealfall die Kombination aus einem agilen **DevOps** Vorgehen, einem durchgängigen **Continuous Delivery** Prozess, einer flexiblen **Microservice** Software Architecture, dem Einsatz eines stabilen **Container Image** basierten Software Deployments, der Nutzung eines **Service Meshes** zur internen Microservice-Kommunikation und eines **Server Clusters** zur Skalierung der Microservices.

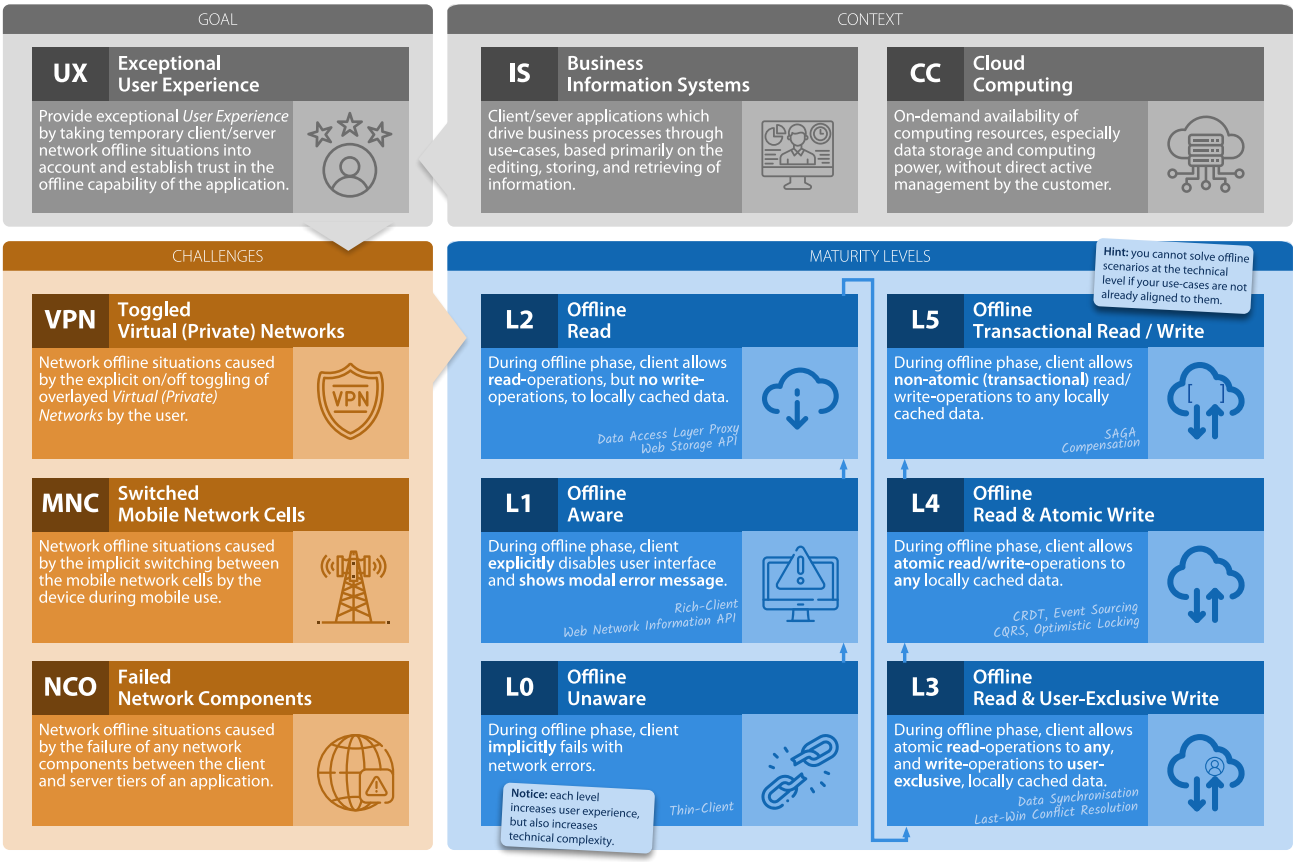
Die **Service Platform** ist in die 7 Service-Bereiche **Management**, **Gateway**, **Integration**, **Tracing**, **Persistence**, **Communication** plus **Delivery** unterteilt, welche üblicherweise ausfallsicher auf 5+1 oder alternativ in teilweise partitionierter Form auf 5x2+1 Maschinen installiert werden. Die Microservices der Application werden auf der **Service Platform** auf getrennten Maschinen installiert.

Bei einer **Cloud-Native Architecture** kommt es darauf an, **High-Availability** und **Scalability** für sowohl die Services der Platform, als auch für die Microservices der Anwendung zu erzielen.

Fragen

- Auf welchen zwei wesentlichen Aspekten basiert die **Cloud-Native Architecture**?
- Was bietet die **Cloud-Native Architecture** den **Microservices** der Anwendung?

Offline Capability



Offline-Fähigkeiten können für eine außergewöhnliche User Experience von Client/Server-basierten betrieblichen Informationssystemen im Kontext von Cloud Computing entscheidend sein. Die Herausforderungen für die zeitweiligen Netzwerk-Offline-Situationen beinhalten umgeschaltete Virtual Private Networks (VPN), gewechselte mobile Netzwerkzellen und ausgefallene Netzwerkkomponenten.

In Offline-Szenarien kann eine Anwendung während einer Offline-Phase eine bestimmte Reifegradstufe unterstützen: Bei **Offline Unaware** schlägt der Client implizit mit Netzwerkfehlern fehl; Bei **Offline Aware** deaktiviert der Client explizit die Benutzeroberfläche und zeigt eine modale Fehlermeldung an; Bei **Offline Read** erlaubt der Client Leseoperationen, aber keine Schreiboperationen, auf lokal zwischengespeicherte Daten; Bei **Offline Read & User-Exclusive Write** erlaubt der Client atomare Leseoperationen auf beliebige, und Schreiboperationen auf benutzerexklusive, lokal zwischengespeicherte Daten; Bei **Offline Lesen & Atomares Schreiben** erlaubt der Client atomare Lese-/Schreiboperationen auf beliebige lokal zwischengespeicherte Daten; Bei **Offline Transactional Read / Write** erlaubt der Client nicht-atomare (transaktionale) Lese-/Schreibvorgänge auf beliebige lokal zwischengespeicherte Daten.

Fragen

- Wieso kann eine Offline-Fähigkeit von Anwendungen im Kontext von Cloud Computing entscheidend sein?