



**Software Engineering
in der industriellen Praxis
(SEIP)**

Dr. Ralf S. Engelschall

Declarative Languages

Express the **target state** and let the machine figure out the steps.

<h4>Markup Languages</h4> <p>Write text intermixed with markup information.</p> <pre>foo bar baz quux</pre> <p>Examples: Wiki, Markdown, AsciiDoc, SGML, HTML, TeX, R(un)joff, reStructuredText, RTF</p>	<h4>Configuration Languages</h4> <p>Express complex textual configurations.</p> <pre>foo bar quux { baz; quux id 7; baz }</pre> <p>Examples: INI, XML, SXML, JSON, YAML, TOML, HCL</p>
<h4>Rule Languages</h4> <p>Express logic and semantics through complex rules.</p> <pre>foo(x, y) <- bar(x, y, z) AND x < 42 AND z >= 10</pre> <p>Examples: SQRL, Datalog/RuleML, OWL/SWRL, RIF</p>	<h4>Constraint Languages</h4> <p>Find solutions for complex constraints.</p> <pre>foo @ bar(X, Y), baz(X, Y, _) ==> quux.</pre> <p>Examples: MiniZinc, CHR, OCL, Rego, Z3.</p>
<h4>Query Languages</h4> <p>Retrieve information through paths and expressions.</p> <pre>// foo / bar [@baz == "xxx" && @quux > 10]</pre> <p>Examples: Glob, RegExp, CSS Selector, XPath, YARA, GraphQL, SQL, SPARQL, Cypher, GQL, ASTq</p>	<h4>Validation Languages</h4> <p>Parse and validate complex textual information.</p> <pre>foo ::= "bar(#" (? [0-9a-fA-F]{2})+ "</pre> <p>Examples: RegExp, Ducky, BNF, PEG, RELAX NG</p>

solution approach: automatically, non-obvious; execution control: automatically, pre-defined; performance optimization: automatically, pre-defined

Imperative Languages

Express the **steps** how the machine has to reach the target state.

<h4>Shell Languages</h4> <p>Automate execution of system commands.</p> <pre>foo -x 2>&1 bar -y --quux <(cat *,cf)</pre> <p>Examples: Korn-Shell, Bourne-Shell, Bash, C-Shell, Batch-Script, PowerShell, AppleScript, DCL</p>	<h4>Programming Languages</h4> <p>Execute complex algorithmic steps.</p> <pre>for (let i = 0; i < 10; i++) foo(i, 42)</pre> <p>Examples: JavaScript, TypeScript, Scala, Kotlin, Java, C#, C/C++, Rust, Go, Python, Perl, Ruby, Lua</p>
<h4>Text-Processing Languages</h4> <p>Manipulate texts through transformations.</p> <pre>/^foo/,/bar.*baz/ s/quux\([0-9]*\)/foo\1/g</pre> <p>Examples: ed, ex, sed, AWK, TXR, XSLT, JSLT</p>	<h4>Macro Languages</h4> <p>Pre-process texts with macros.</p> <pre>define('foo', `bar\$1baz`) foo(quux)bar</pre> <p>Examples: m4, GPP, CPP, Zoem, ProMac</p>
<h4>Expression Languages</h4> <p>Expand path, arithmetic, and boolean expressions.</p> <pre>{{ foo,bar[*],baz[42] ,quux + 1 }}</pre> <p>Examples: JQ, YQ, MozJEXL, MathML, JUEL, SpEL</p>	<h4>Template Languages</h4> <p>Expand complex text fragments.</p> <pre>{% for k, v in items %} {{k}}: {{v}}{% endfor %}</pre> <p>Examples: JQ, Nunjucks, Handlebars, Mustache, Jinja, Jsonnet</p>

solution approach: manually, obvious; execution control: manually, fine-grained; performance optimization: manually, fine-grained; Examples: essential recommended alternative

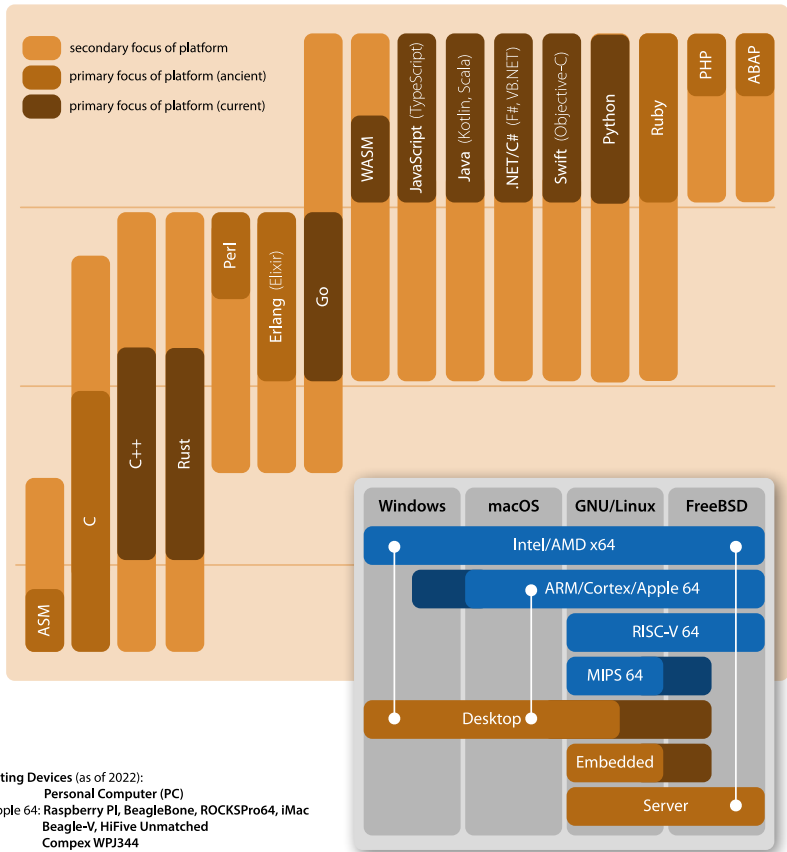
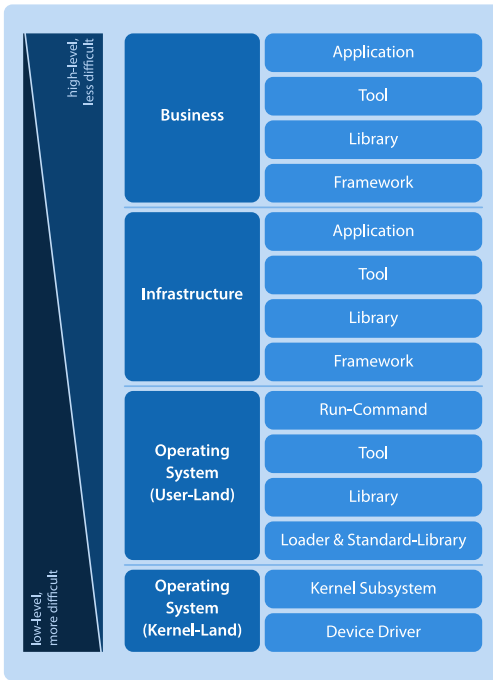
Es gibt unzählige formale Sprachen (**Formal Languages**). In einem konkreten Technology Stack kommen üblicherweise fast ein Dutzend solcher Sprachen gleichzeitig zum Einsatz. Der Architekt muss diese deshalb sehr sorgfältig auswählen.

Die formalen Sprachen lassen sich zunächst in deklarative Sprachen (**Declarative Languages**) und imperative Sprachen (**Imperative Languages**) aufteilen. Erstere drücken einen Ziel-Zustand aus ("WAS"), letztere drücken dagegen den Weg dorthin aus ("WIE").

Deklarative Ansätze sind imperativen Ansätzen meist zu bevorzugen, da sie es der Implementierung (und nicht dem Programmierer) überlassen, den optimalen Weg zu finden. Außerdem erlauben sie inkrementelle Herangehensweisen, bei denen der nächste Schritt durch die konkrete Differenz zwischen dem aktuellen Ist-Zustand und dem gewünschten Ziel-Zustand abgeleitet wird. Dies ist besonders in sehr dynamischen und fehleranfälligen Umgebungen beim Wiederaufsetzen entscheidend.

Fragen

? In welche zwei Klassen lassen sich formale Sprachen (**Formal Languages**) aufteilen?



AF 123
Public Domain, Version 1.1 (2023-04-23), Author(s) 2023-04-23, CC BY-SA, English, All rights reserved. Distribution and reproduction prohibited. Licensed to Technische Universität München (TUM) for reproduction in Computer Science lecture content only.

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

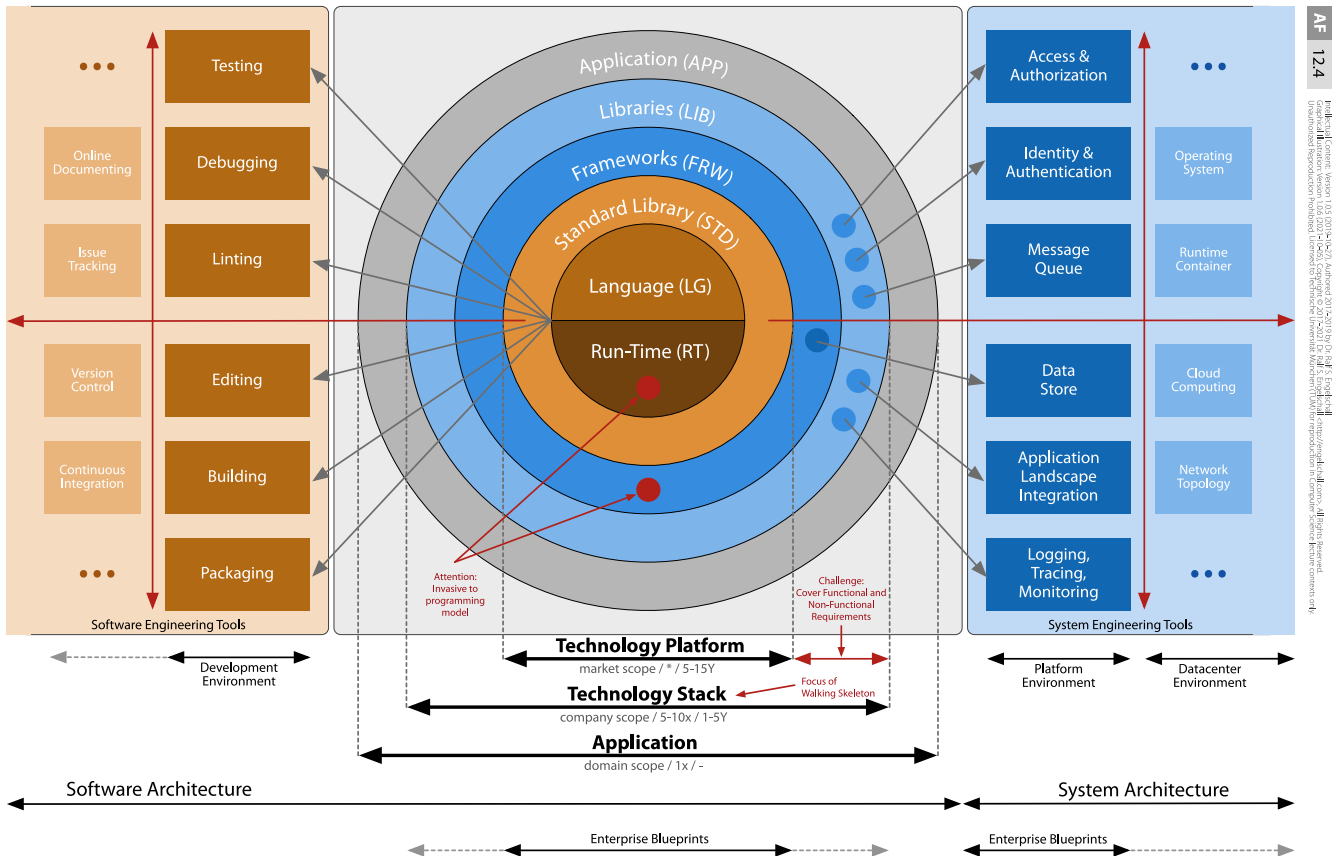
Typical Computing Devices (as of 2022):
 Intel/AMD x64: Personal Computer (PC)
 ARM/Cortex/Apple 64: Raspberry Pi, BeagleBone, ROCKSPro64, iMac
 RISC-V 64: Beagle-V, HiFive Unmatched
 MIPS 64: Compex WPJ344

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 abzielen!

Fragen

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



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 "invasivsten" für das Programmiermodell?

IT Interface Theme Style Reset, Shape, Color, Gradient, Shadow, Font, Icon Bootstrap TypoPRO, FontAwesome, Normalize		18 Interface Internationalization Text Internationalization (I18N), vue-i18next, I18Next		DL Dialog Life-Cycle Component States, Component State Transitions. ComponentJS (none)	
IW Interface Widgets Icon, Label, Text, Paragraph, Image, Form, Text-Field, Text-Area, Date Picker, Toggle, Radio Button, Checkbox, Select List, Slider, Progress Bar, Hyperlink, Popup Menu, Dropdown Menu, Toolbar, Tooltip, Tab, Pill, Breadcrumb, Pagination, Badge, Alert, Panel, Modal, Table, Scrollbar, Carousel Bootstrap Select2, SlickGrid, ...		DC Data Conversion Value Formatting, Value Parsing, Localization (L10N), VueJS Moment, Numeral, Accounting, ...		DS Dialog Structure Component, Model/View/Controller Roles, Hierarchical Composition ComponentJS ComponentJS-MVC	
IL Interface Layouting Responsive Design, Media Query, Frame, Grid, Padding, Border, Margin, Alignment, Force, Magnetism Bootstrap Swiper, jQuery Page, ...		DB Data Binding Reactive, Observer, Unidirectional, Bidirectional, Incremental VueJS (none)		SP State Persistence Local Storage, Cookies, Caching (none) Store.js, JS-Cookie	
IE Interface Effects Transition, Transformation, Keyframes, Easing Function, Sound Effect, Physics VueJS Animate.css, DynamicJS, Howler, ...		PM Presentation Model Parameter Value, Command Value, State Value, Data Value, Event Value, Value Validation, Presentation Logic ComponentJS (none)		BM Business Model Entity, Field, Relationship, Universally Unique Identifiers (UUID) (none) DataModelJS, Pure-UUID	
II Interface Interactions Mouse, Keyboard, Touchscreen, Gesture, Clipboard, Drag & Drop VueJS Hammer, Mousetrap, Dragula, ...		DN Dialog Navigation Deep Linking, Routing, Dialog Flow ComponentJS Director, URL.js		UA Use-Case Authorization User Experience, Dialog Restriction, User, Group, Role, Use-Case, Data, Access. (none) (none)	
IS Interface States Rendered, Enabled, Visible, Focused, Warning, Error, Floating VueJS (none)		DA Dialog Automation Dialog Macros, Click-Through, Smoke Testing. ComponentJS ComponentJS-Testdrive		CN Client Networking Request/Response, Synchronization, Push, Pull, Pulled-Push, REST, GraphQL, Authentication, Session. (none) Axios, Apollo Client	
IM Interface Mask Markup Loading, Markup Generation, Virtual DOM, Text, Bitmaps, Vectors, 2D/3D Canvas, Accessibility VueJS jQuery-Markup, D3, Snap.svg, FabricJS, ...		DC Dialog Communication Service, Event, Model, Socket, Hooks ComponentJS Latching		ED Environment Detection Runtime Detection, Feature Detection. (none) Modernizr, FeatureJS, jQuery-Stage	

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) Datenmodell 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?

AMA Bare Amalgamation

Manually deploy all applications into a single, shared, and unmanaged filesystem location. Dependencies are resolved manually. Examples: Windows Fonts, Unix 1990th /usr/local.

Pro: simple deployment
Con: incompatibilities, hard uninstallation

UHP Unmanaged Heap

Manually deploy all applications into multiple, distinct, and unmanaged filesystem locations. Dependencies are resolved manually. Examples: macOS *.app, OpenPKG LSYNC.

Pro: simple deployment, easy uninstallation
Con: no repair mechanism

MHP Managed Heap

Let individual installers deploy applications into multiple, distinct, and managed filesystem locations. Dependencies are manually resolved or bundled. Examples: macOS *.pkg, Windows MSI, InnoSetup.

Pro: easy uninstallation, repairable
Con: requires installer, diversity, no dep.

PKG Managed Package

Let a central package manager deploy all applications into a single, shared, and managed filesystem location. Dependencies are automatically resolved. Examples: APT, RPM, FreeBSD pkg, MacPorts, Gradle, NPM.

Pro: easy uninst., repairable, dependencies
Con: P.M. pre-installation, P.M. single instance

CON Container Image

Bundle an application with its stripped-down OS dependencies and run-time environment into a container image. Examples: Docker/ContainerD, Kubernetes/CRI-O, Windows Portable Apps.

Pro: independent, simple deployment
Con: fewer variations, no dependencies

STK Package/Container Stack

Establish an application out of multiple Managed Packages. Examples: OpenPKG Stack, Docker Compose, Kubernetes/Kompose, Kubernetes/Helm.

Pro: independent, flexible
Con: overhead

VMI Virtual Machine Image

Bundle an application with its full OS dependencies and run-time environment into a virtual machine image and deploy and execute this on a hypervisor. Examples: VirtualBox, VMWare, HyperV, Parallels, QEMU.

Pro: all-in-one, independent
Con: overhead, sealed, inflexible

APP Solution Appliance

Bundle an application with its full OS dependencies, run-time environment and underlying hardware. Examples: AVM Fritz! Box, SAP HANA.

Pro: all-in-one, independent
Con: expensive, sealed, inflexible

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.

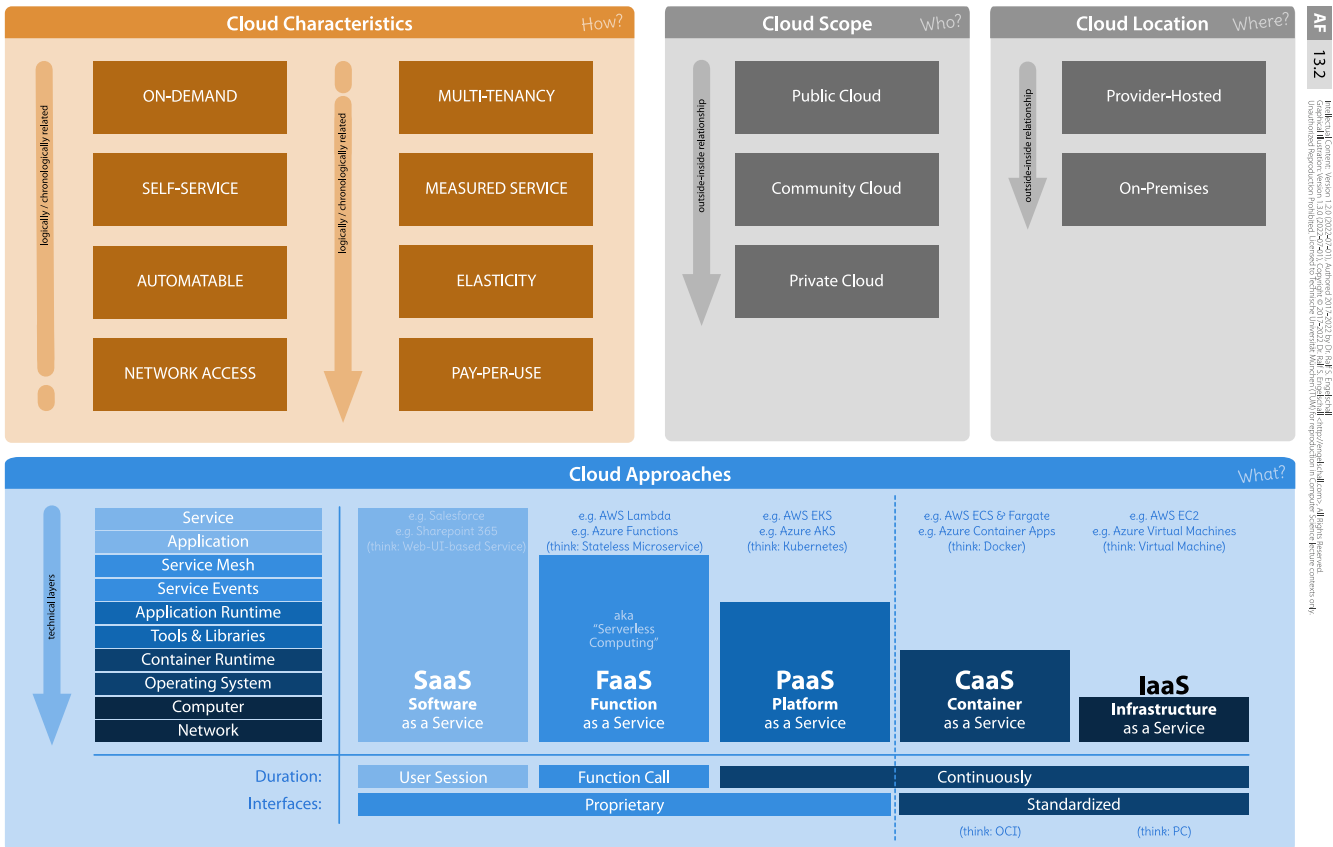
Um flexibler zu sein, kann man die **Managed Packages** oder **Container Images** sehr klein halten und stattdessen eine Application über einen ganzen **Package/Container Stack** definieren (z.B. Docker Compose).

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 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.

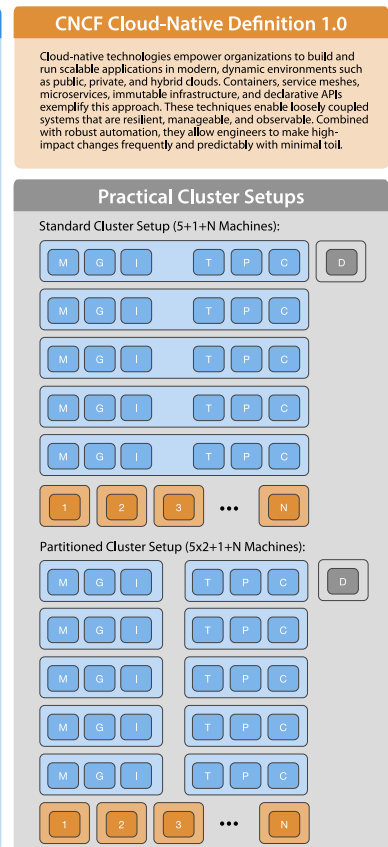
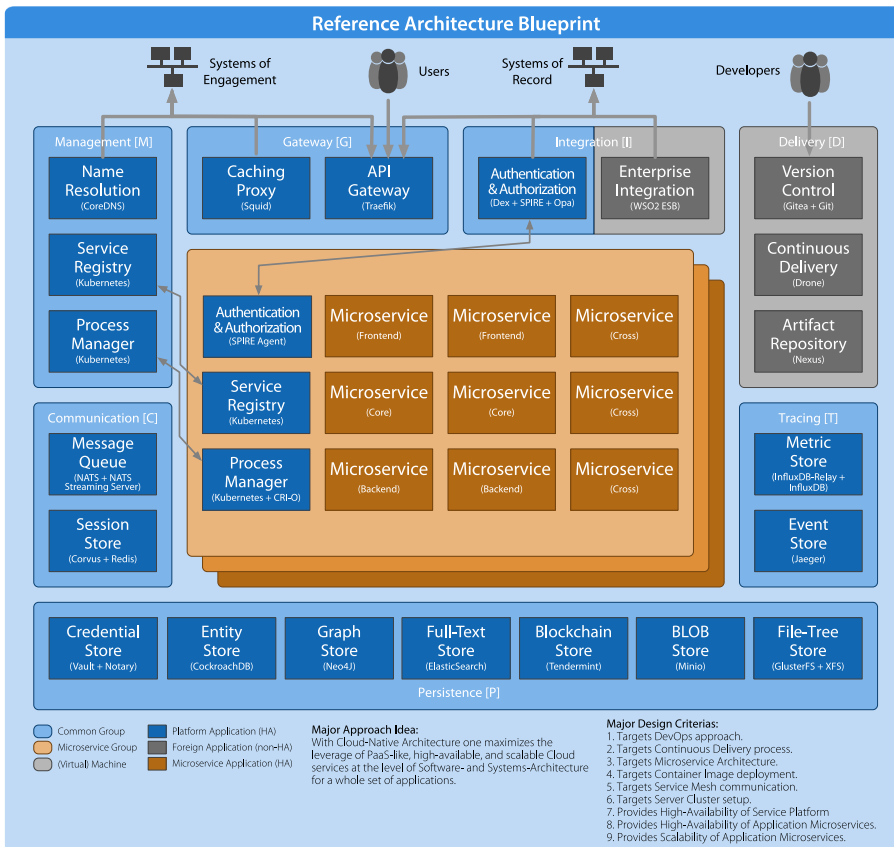
Bei **Platform as a Service (PaaS)** werden zusätzlich umgebende **Tools & Libraries** und eine **Application Run-Time** bereitgestellt; bei **Function as a Service (FaaS)** werden zusätzlich externe **Service Events** und ein **Service Mesh** bereitgestellt und bei **Software as a Service (SaaS)** werden 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 Ressourcen-Bereitstellung erfüllen muss, damit es als **Cloud Computing** gilt!
- ❓ Bei welchem **Cloud Approach** wird nur **Network** und **Computer** bereitgestellt?



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?

