



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.

Markup Languages

Write text intermixed with
markup information.

```
foo <em>bar <strong>baz  
</strong></em> quux
```

Examples:
Wiki, **Markdown**, AsciiDoc, SGML, **HTML**,
TeX, R(un)off, reStructuredText, RTF

Configuration Languages

Express complex textual
configurations.

```
foo bar quux { baz;  
quux id 7; baz }
```

Examples:
INI, XML, SXML, JSON,
YAML, TOML, HCL

Rule Languages

Express logic and semantic
through complex rules.

```
foo(x, y) <- bar(x, y, z)  
AND x < 42 AND z >= 10
```

Examples:
SQL, Datalog/RuleML,
OWL/SWRL, RIF

Constraint Languages

Find solutions for
complex constraints.

```
foo @ bar(X, Y),  
baz(X, Y, _) => quux.
```

Examples:
MiniZinc, CHR,
OCL, Rego, Z3.

Query Languages

Retrieve information through
paths and expressions.

```
// foo / bar [ @baz ==  
"xxx" && @quux > 10 ]
```

Examples:
Glob, **RegExp**, **CSS Selector**, XPath, YARA,
GraphQL, SQL, SPARQL, Cypher, GQL, ASTq

Validation Languages

Parse and validate complex
textual information.

```
foo ::= "bar(#" (?)  
[0-9a-fA-F]{2})+ "
```

Examples:
RegExp, Ducky, BNF,
PEG, RELAX NG

solution approach:
execution control:
performance optimization:

automatically, non-obvious
automatically, pre-defined
automatically, pre-defined

Imperative Languages

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

Shell Languages

Automate execution of
system commands.

```
foo -x 2>&1 | bar -y  
--quux <(cat *,cf)
```

Examples:
Korn-Shell, Bourne-Shell, **Bash**, C-Shell,
Batch-Script, **PowerShell**, AppleScript, DCL

Text-Processing Languages

Manipulate texts through
transformations.

```
/^foo/,/bar.*baz/  
s/quux\([0-9]*\)/foo\1/g
```

Examples:
ed, ex, **sed**, AWK,
TXR, XSLT, JSLT

Expression Languages

Expand path, arithmetic, and
boolean expressions.

```
{{ foo,bar[*],baz[42]  
,quux + 1 }}
```

Examples:
JQ, **YQ**, MozJEXL, MathML,
JUEL, SpEL

Programming Languages

Execute complex
algorithmic steps.

```
for (let i = 0; i < 10;  
i++) foo(i, 42)
```

Examples:
JavaScript, TypeScript, Scala, Kotlin, Java,
C#, C/C++, Rust, Go, Python, Perl, Ruby, Lua

Macro Languages

Pre-process texts with
macros.

```
define('foo', `bar$1baz`)  
foo(quux)bar
```

Examples:
m4, GPP, CPP,
Zeom, ProMac

Template Languages

Expand complex
text fragments.

```
{% for k, v in items %}  
{k}: {v}{% endfor %}
```

Examples:
Pug, **Nunjucks**, Handlebars,
Mustache, Jinja, Jsonnet

solution approach:
execution control:
performance optimization:

manually, obvious
manually, fine-grained
manually, fine-grained

Examples:
essential
recommended
alternative

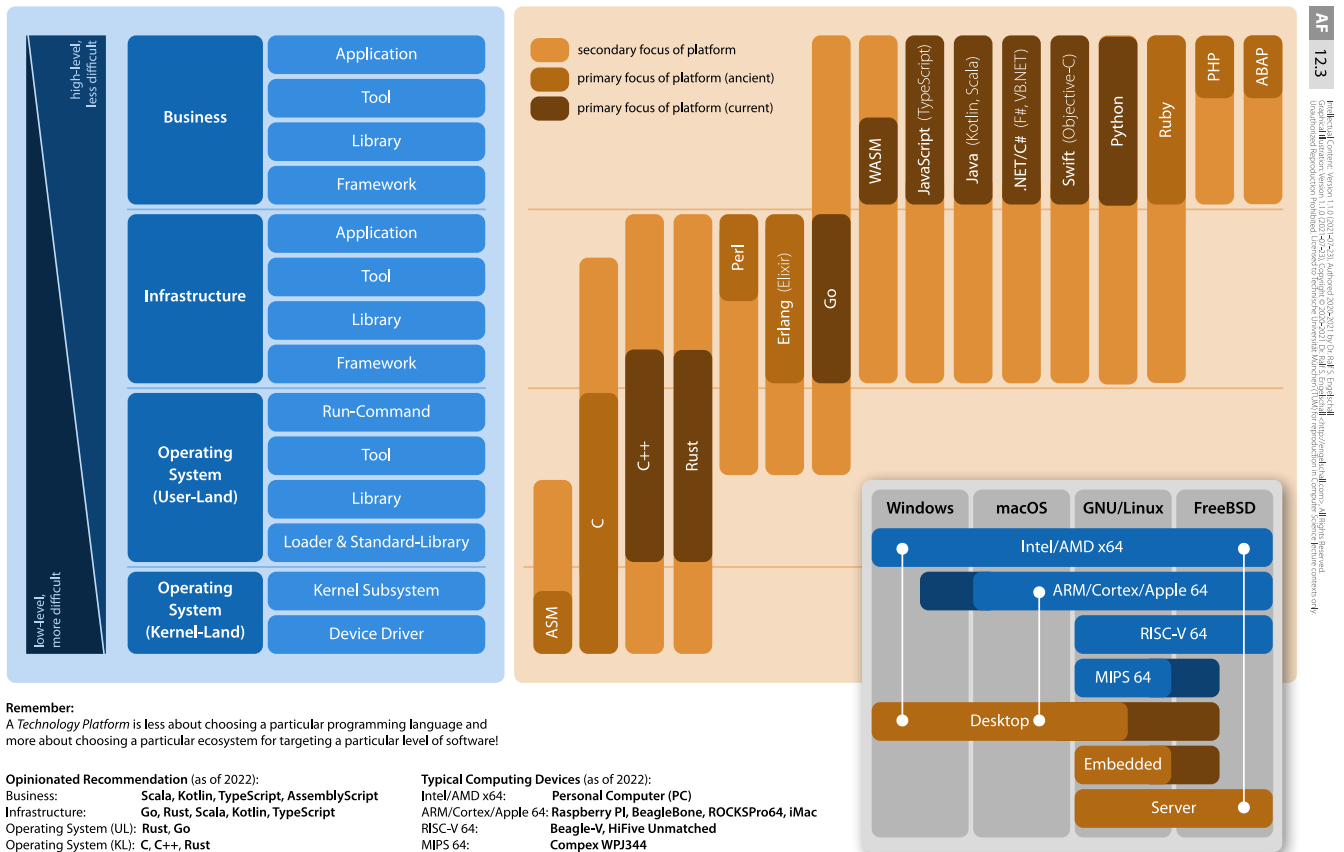
Es gibt unzählige formale Sprachen (**Formal Languages**). In einem konkreten Technology Stack kommen üblicherweise fast ein Dutzend solcher Sprachen gleichzeitig zum Einsatz. Der Architekt muss diese deshalb sehr sorgfältig auswählen.

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?

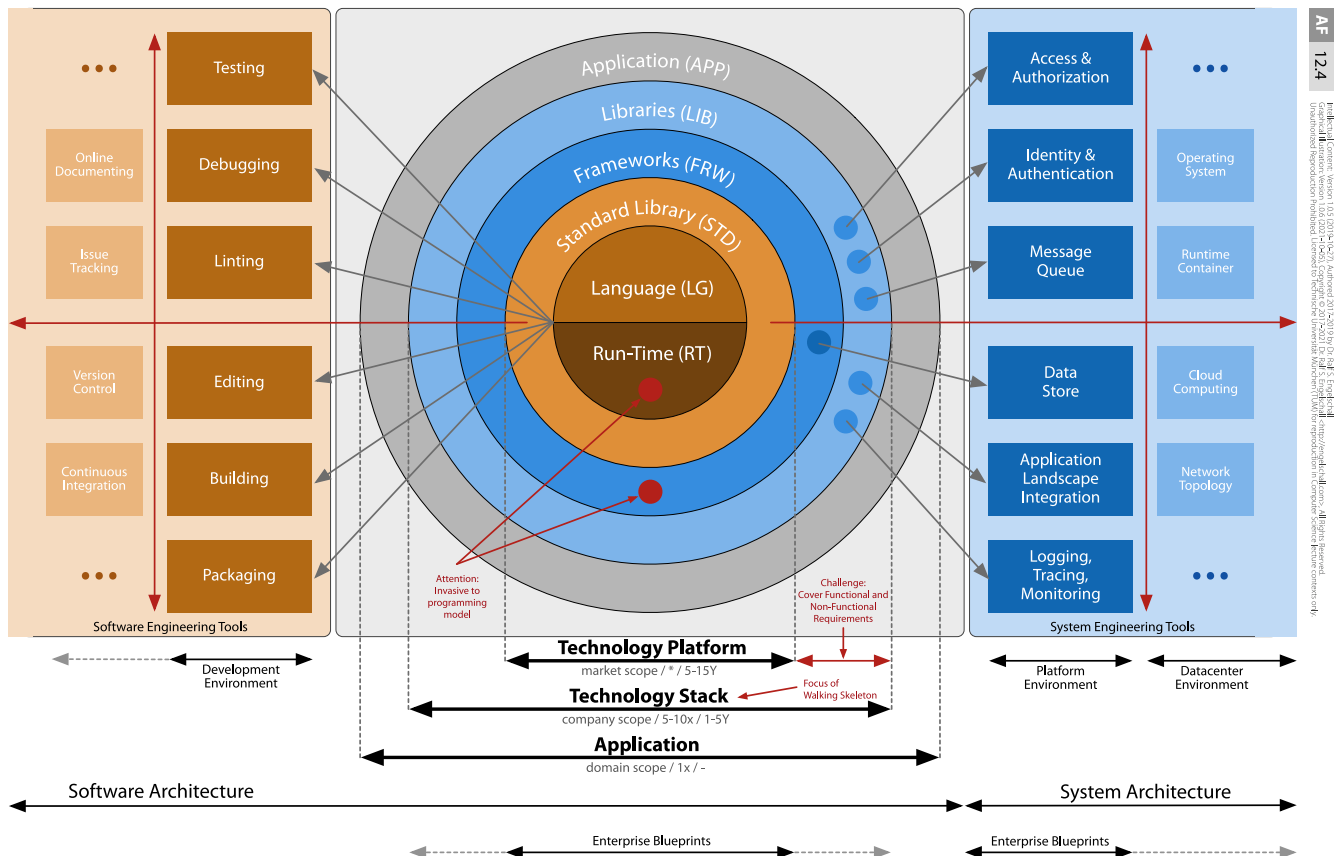


Es gibt verschiedene Ebenen von Software, von "low-level" und schwieriger (auf der Ebene des Betriebssystems) bis "high-level" und weniger schwierig (auf der Ebene von Business-Logik).

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), VueJS 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?

ED Environment Detection Detect the run-time environment, like underlying operating system, execution platform, network topology, feature toggles, etc.  Node process, syspath	SN Server Networking Listen to network sockets, accept connections and manage request/response and message communication.  HAPI hapi-plugin-websocket, ws	CN Client Networking Provide mechanisms to connect to peers over the network and perform request/response and/or publish/subscribe communication.  (none) Axios, MQTT.js, ws
AP Argument Parsing Parse options and arguments of the Command-Line Interface (CLI) to bootstrap application parameters.  (none) yargs	PI Peer Information Determine unique identification and add-on information about the client peer.  HAPI hapi-plugin-peer, geopip	TS Task Scheduling Schedule and execute recurring tasks independent of regular I/O operations.  (none) node-scheduler
CP Configuration Parsing Load and parse directives from configuration file to bootstrap application parameters.  (none) js-YAML	SH Session Handling Manage secured per-connection sessions to keep state between communication requests and/or client sessions.  HAPI YAR	ET Execution Tracing Provide mechanisms for tracing the execution by logging event and measurement information at certain points of interest.  Microkernel Winston
PD Process Daemonizing Detach from the startup terminal and host process in order to run fully independently.  (none) daemonize2	UA User Authentication Determine and validate the unique identity of the user communicating over the current network connection.  HAPI JWT, Passport	DA Database Access Map in-memory domain entities onto data store dependent persistent data structure.  Sequelize GraphQL-Tools-Sequelize
PM Process Management (Pre-)fork child processes and/or threads of execution and monitor and control them during the life-cycle of the application.  (none) cluster, nodeemon	RV Request Validation Validate the syntactical and semantical compliance of the requests and sanitize the requests.  HAPI Joi, DuckyJS	DC Database Connectivity Locally or remotely connect the database access layer to the underlying data store.  Sequelize sqlite3.pg
CM Component Management Structure the code into components, instantiate them under run-time and manage them in a stateful component life-cycle.  Microkernel (none)	RP Request Processing Process the request by dispatching execution according to the provided request and determined context information.  HAPI GraphQL.js	DS Database Schema Create, update or downgrade the data schema inside the underlying data store.  Sequelize (none)
CC Component Communication Provide inter-component communication mechanisms like events, hooks, registry, etc.  Microkernel Latching	RA Role Authorization Determine whether the role of the current user is allowed to execute the current request.  (none) GraphQL-Tools-Sequelize	DB Database Bootstrapping Create, update or downgrade both mandatory bootstrapping and optional domain-specific data inside the underlying data store.  Sequelize ini

AF	12.6
----	------

Intellectual Content: Version 1.0.6 (2020-10-17), Authored 2017-2020 by Dr. Raj S. Englehall
 Illustration: Version 1.0.3 (2018-06-25), Copyright © 2017-2018 Dr. Raj S. Englehall <<http://englehall.com>>. All rights reserved.

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"?

The diagram illustrates eight different software deployment models, each with a unique structure and set of characteristics:

- AMA Bare Amalgamation:** Applications are manually deployed to a single, shared filesystem location. Dependencies are resolved manually. Examples include Windows Fonts and Unix 1990th /usr/local.
 - Pro:** simple deployment
 - Con:** incompatibilities, hard uninstallation
- CON Container Image:** An application with stripped-down OS dependencies and run-time environment is bundled into a container image. Examples include Docker/ContainerD, Kubernetes/CRI-O, Windows Portable Apps.
 - Pro:** independent, simple deployment
 - Con:** fewer variations, no dependencies
- UHP Unmanaged Heap:** Applications are manually deployed to multiple, distinct, and unmanaged filesystem locations. Dependencies are resolved manually. Examples include macOS *.app and OpenPKG LSYNC.
 - Pro:** simple deployment, easy uninstallation
 - Con:** no repair mechanism
- STK Package/Container Stack:** An application is established out of multiple Managed Packages. Examples include OpenPKG Stack, Docker Compose, Kubernetes/Kompose, and Kubernetes/Helm.
 - Pro:** independent, flexible
 - Con:** overhead
- MHP Managed Heap:** Individual installers deploy applications into multiple, distinct, and managed filesystem locations. Dependencies are manually resolved or bundled. Examples include macOS *.pkg, Windows MSI, and InnoSetup.
 - Pro:** easy uninstallation, repairable
 - Con:** requires installer, diversity, no dep.
- VMI Virtual Machine Image:** An application with its full OS dependencies and run-time environment is bundled into a virtual machine image and deployed and executed through a hypervisor. Examples include VirtualBox, VMWare, HyperV, Parallels, and QEMU.
 - Pro:** all-in-one, independent
 - Con:** overhead, sealed, inflexible
- PKG Managed Package:** A central package manager deploys all applications into a single, shared, and managed filesystem location. Dependencies are automatically resolved. Examples include APT, RPM, FreeBSD pkg, MacPorts, Gradle, and NPM.
 - Pro:** easy uninstall., repairable, dependencies
 - Con:** P.M. pre-installation, P.M. single instance
- APP Solution Appliance:** An application with its full OS dependencies, run-time environment, and underlying hardware is bundled. Examples include AVM Fritz! Box and 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.

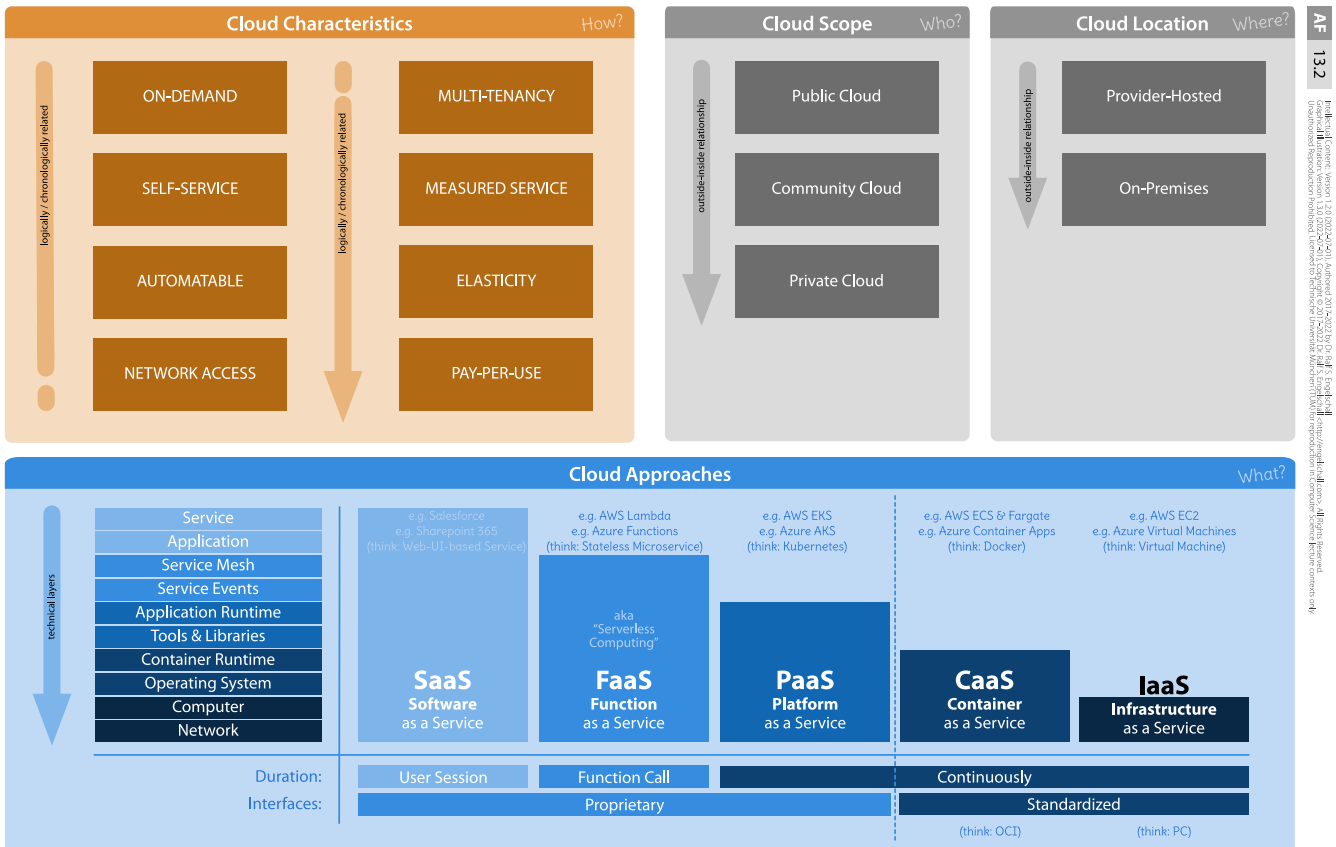
Um flexibler zu sein, kann man die **Managed Packages** oder **Container Images** sehr klein halten und stattdessen eine Application über einen ganzen **Package/Container Stack** definieren (z.B. Docker Compose).

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 vor allem 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.

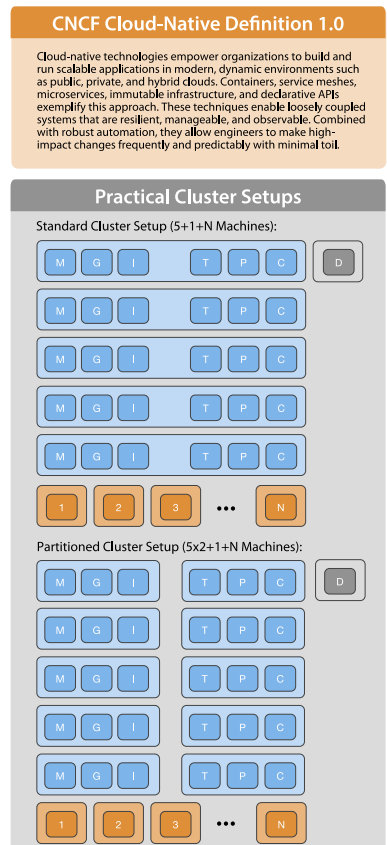
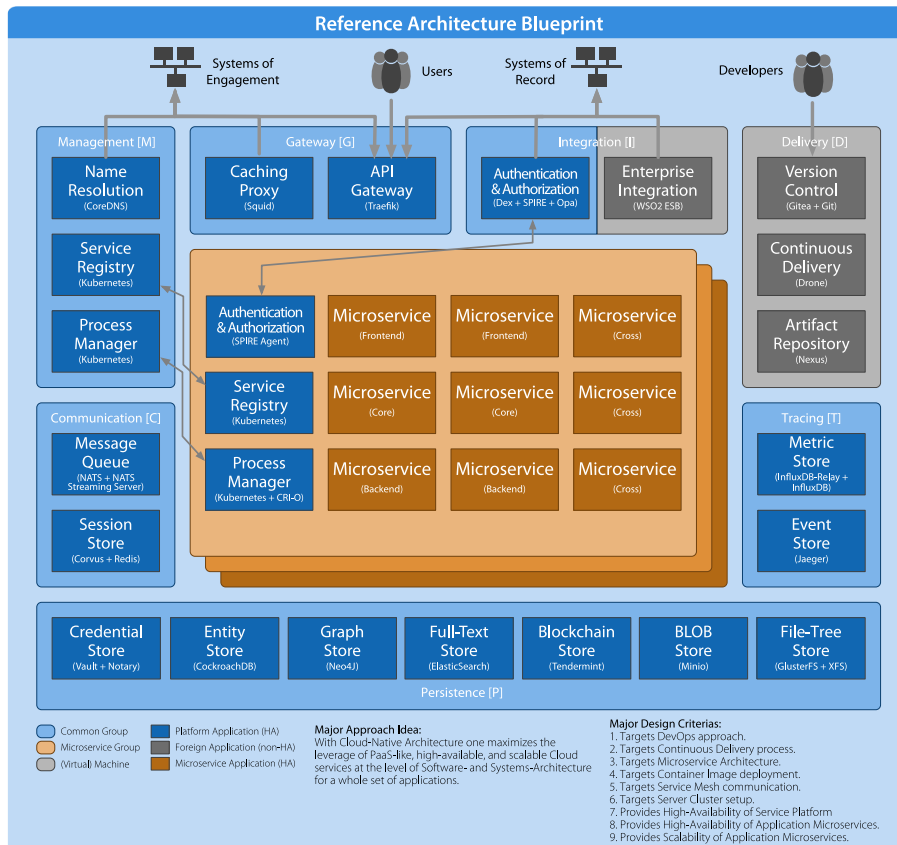
Bei **Platform as a Service (PaaS)** werden zusätzlich umgebende **Tools & Libraries** und eine **Application Run-Time** bereitgestellt; bei **Function as a Service (FaaS)** werden zusätzlich externe **Service Events** und ein **Service Mesh** bereitgestellt und bei **Software as a Service (SaaS)** werden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

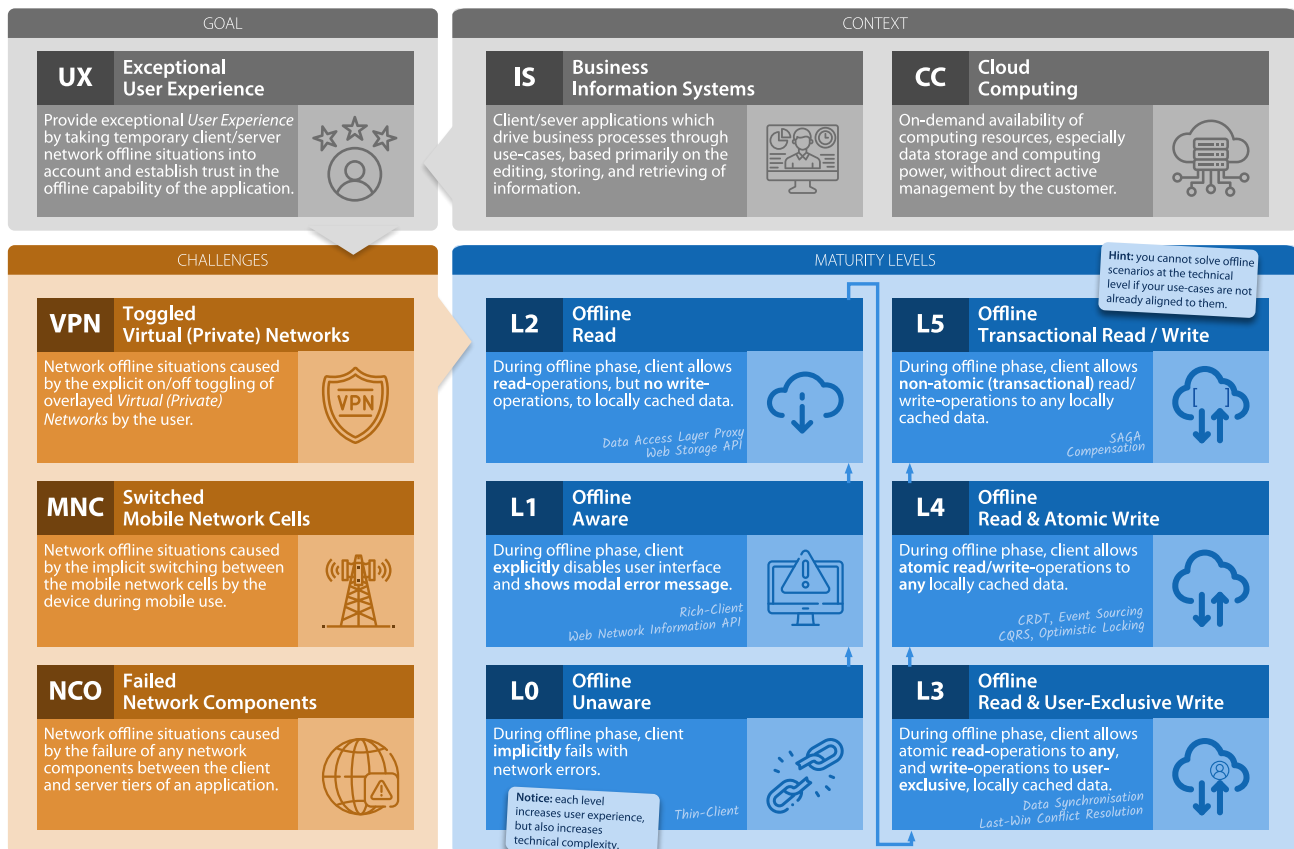
In der Praxis bedeutet dies im Idealfall die Kombination aus einem agilen **DevOps** Vorgehen, einem durchgängigen **Continuous Delivery** Prozess, einer flexiblen **Microservice** Software Architecture, dem Einsatz eines stabilen **Container Image** basierten Software Deployments, der Nutzung eines **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?



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?