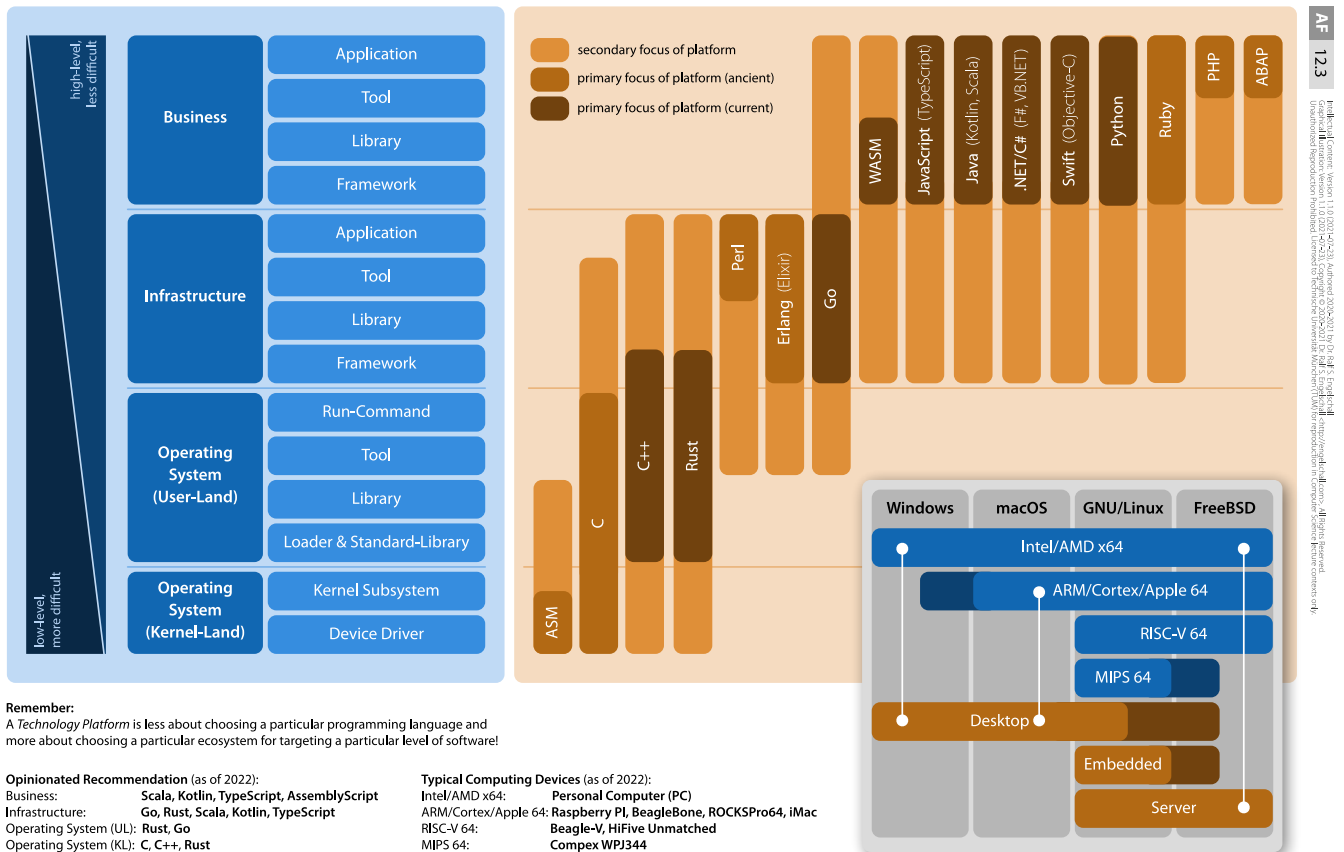




Software Engineering in Industrial Practice (SEIP)

Dr. Ralf S. Engelschall

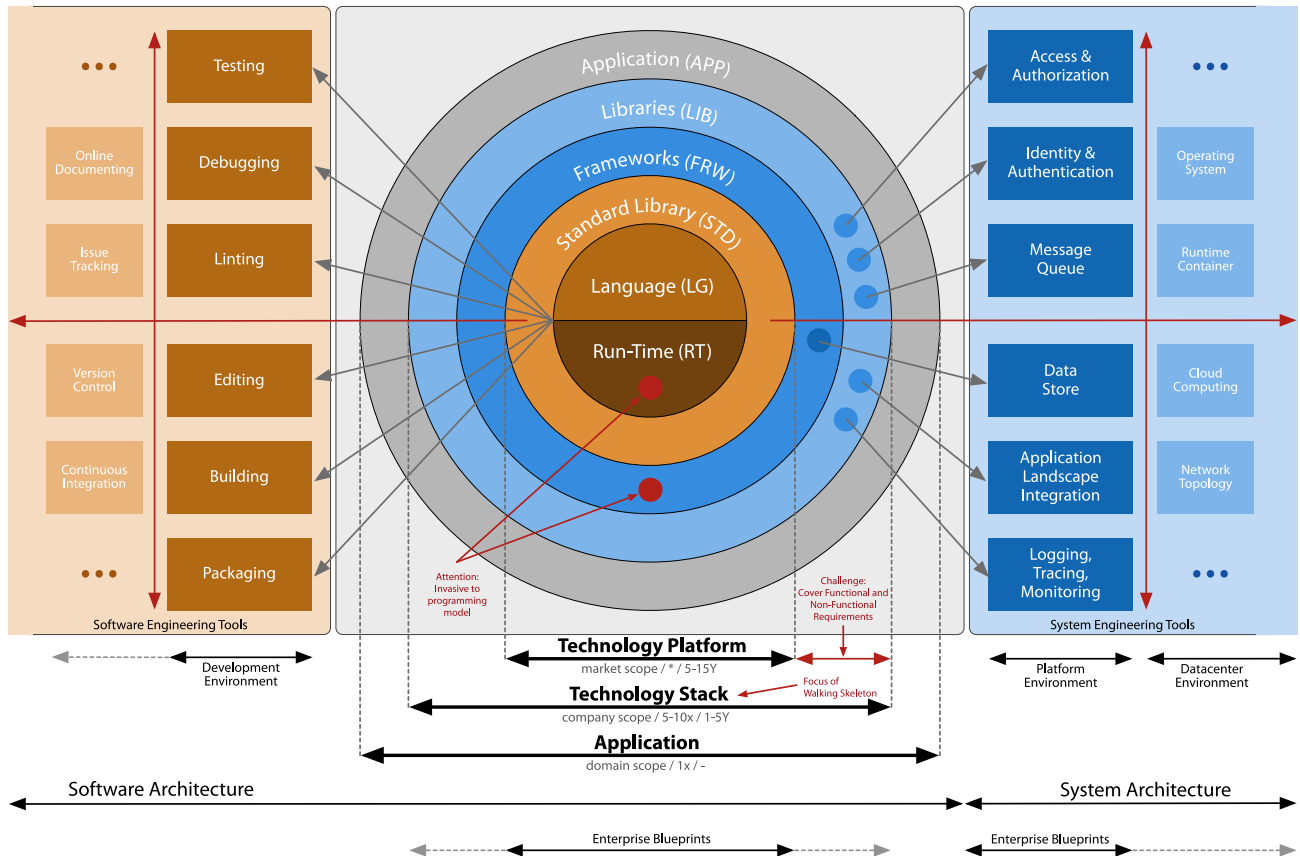


There are various levels of software, ranging from low-level and more difficult (operating system related) to high-level and less difficult (business-logic related).

A Technology Platform is less about choosing a particular programming language and more about choosing a particular ecosystem for targeting a particular kind of software!

Questions

- ❓ Is the Technology-Platform **Node.js** suitable to implement a Kernel-Subsystem at the level of the operating system?



A **Technology Platform** consists of a **Language**, an optional **Run-Time** environment and a **Standard Library**. On top of this, **Frameworks** and **Libraries** extend this to a **Technology Stack**, in which with them especially all the prerequisites for the functional and non-functional requirements in the **Application** are achieved.

It has to be noted that the **Run-Time** and the **Frameworks** are usually extremely “invasive” to the programming model and thus can almost never be replaced afterwards. Therefore, with the so-called **Walking Skeleton**, the focus is mainly on the **Technology Stack** to be defined and integrated.

While the **Application** has a functional scope and is implemented only once, a particular **Technology Stack** is usually defined by a company and then reused several times over a period of a few years.

The underlying particular **Technology Platform**, on the other hand, is implemented by a third party for the market, is reused as often as required and must exist for quite a long period of time.






















Large companies, therefore, usually stringently define the **Technology Platforms** and **Technology Stacks** in their **Enterprise Blueprints**.

For the **Software Engineering Tools** one should take into account the tools for **Testing**, **Debugging**, **Linting**, **Editing**, **Building** and **Packaging** of the **Development Environment**, because these are usually directly dependent on the particular **Technology Stack**.

The situation is similar for the **System Engineering Tools** of the **Platform Environment**: these require at least the associated **Libraries** in the **Technology Stack**, in order to be addressed during the run-time of the **Application**.

Questions

- ❓ What are the three components of a **Technology Platform**?
- ❓ Which two additional components make up a **Technology Stack**, compared to the **Technology Platform**?
- ❓ Which two components of a **Technology Stack** are most “invasive” to the programming model?

IT Interface Theme Style Reset, Shape, Color, Gradient, Shadow, Font, Icon Bootstrap TypoPRO, FontAwesome, Normalize 	18 Interface Internationalization Text Internationalization (I18N). VueJS vue-18next, 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, Breadcrumbs, Pagination, Badge, Alert, Panel, Modal, Table, Scrollbar, Carousel Bootstrap Select2, SlickGrid, ... 	DC Data Conversion Value Formatting, Value Parsing, Localization (L10N). VueJS Moment, Numeral, Accounting, ... \$1,234.56 2016-01-01 	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) <div>{{foo}}</div> var bar = "foo" 	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) <pre>data_username: string, first_name: string, last_name: string, data_password: string, login_required: boolean, login_event_name: string, login_bar: string</pre> 	BM Business Model Entity, Field, Relationship, Universally Unique Identifiers (UUID) (none) DataModelJS, Pure-UUID 
I 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 #/foo/123 root / foo / 123 	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 <pre>query { foo { id: 123 } { name: withBar: id, name, bar { id: 1 id, title } } }</pre> 
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 

AF	12.5
----	------

Intellectual Content: Version 1.15 (2020-03-24). Authored 2015-2020 by Dr. Ralf S. Engstschall
 Illustration: Version 1.23 (2020-04-24). Copyright © 2015–2020 Dr. Ralf S. Engstschall
<http://engstschall.com/>. All Rights Reserved.

To define a Technology Stack for a **Rich Client**, 21 **Aspects** have to be considered. Each aspect is covered by at least one **Framework** or **Library**. In practice, each aspect is usually covered by one Framework and zero or more Libraries. The goal always is: to achieve the greatest possible coverage of the aspects with a minimum number of Frameworks and Libraries.

It is advisable to use Open Source Software (OSS) for both Frameworks and Libraries and, if possible, to not own custom implementations, as the effort usually is not in proportion to the benefit. Because all aspects are technical — and not functional — aspects of a user interface.

In the case of a Thin-Client Architecture (instead of a Rich-Client Architecture), a few aspects like **Client Networking** and **Environment Detection** are omitted. All other aspects are still valid, even if, in the case of a Thin-Client Architecture, the frontend (and thus the aspects of the user interface) of the application runs on the server.

Two important aspects deal with the data model: the **Business Model** is a data model that comes directly from the server and is exactly the same in slicing and granularity than the business data model of the server. Its data is synchronized with a **Presentation Model**, which in slicing and granularity is more like the (more technical) data model of the user interface (especially via the aspects **Interface Mask** and **Data Binding**).

Questions

- ❓ How does one ensure in a **Rich-Client Architecture** that the numerous technical **Aspects** of a user interface are addressed?
- ❓ Which two **Aspects** of a **Rich-Client Architecture** hold the data model and take care of the fact that the data supplied by the server cannot be used directly in the user interface?

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, geoip		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, nodemon		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	

To define a Technology Stack for a **(Thin-)Server**, 21 **Aspects** have to be considered. Each Aspect is covered by at least one **Framework** or **Library**. In practice, each Aspect is usually covered by one Framework and zero or more Libraries. The goal always is: to achieve the greatest possible coverage of the Aspects with a minimum number of Frameworks and Libraries.

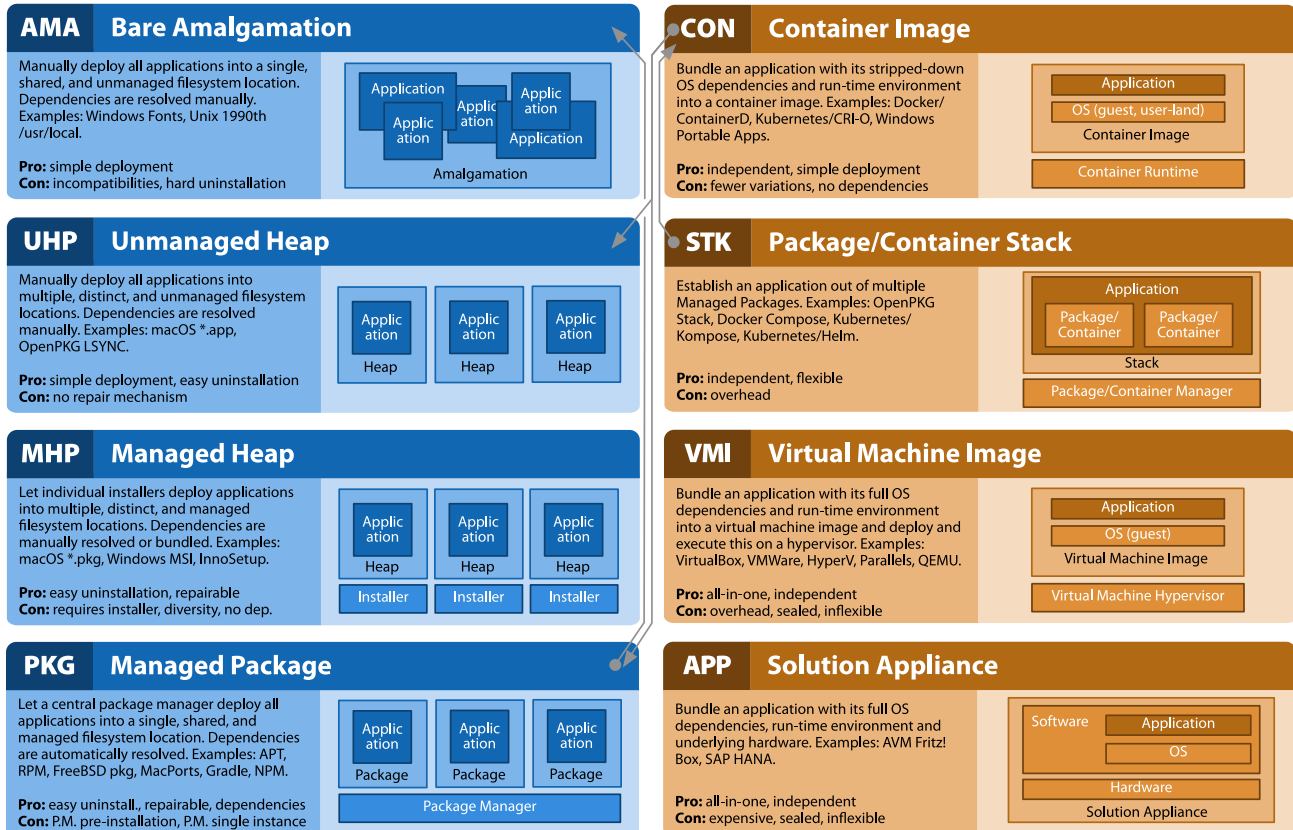
It is advisable to use Open Source Software (OSS) for both Frameworks and Libraries and, if possible, to no own custom implementations, as the effort usually is not in proportion to the benefit. Because all Aspects are technical — and not functional — Aspects of a user interface.

It is to be noted that a server usually does not only have the Aspect **Server Networking** (for the connection of the Rich Clients), but also the Aspect **Client Networking**, in order to be able to query other servers.

In addition, it is to be noted that, above all, two important Aspects address security issues: the Aspect **User Authentication** identifies and authenticates the user ("Is the user the one?"). The Aspect **Role Authorization**, on the other hand, before all business processes, checks whether the authenticated user really is authorized to initiate the processes due to his role(s) ("Is the user allowed to do this?").

Questions

- Why does a **(Thin-)Server** usually have, besides the obvious aspect **Server Networking**, also the aspect **Client Networking**?
- Which Aspect of a **(Thin-)Server** takes care of the Question "Is the user the one"?
- Which aspect of a **(Thin-)Server** takes care of the question "Is the user allowed to do this"?



During **Software Deployment**, an **Application** is installed on a file system for execution. With the **Bare Amalgamation**, the files are copied into a central directory (e.g., Windows C:\Windows\system32). This is easy to realize but makes the clean removal later on very hard.

With **Unmanaged Heap**, each application is copied into a separate directory (e.g., macOS *.app). This is very easy to realize and also allows easy removal. But one still has no repair possibilities. With **Managed Heap**, an own installer is required for each application, among other things, to get repair possibilities (e.g., Windows MSI).

With **Managed Package**, a central Package Manager is used, which standardizes the administration (e.g., DPKG/APT or RPM). It also allows the resolving of dependencies. If, on the other hand, one wants to make the application more independent of the operating system and install it as a shielded unit, the **Container Image** deployment offers itself (e.g., Docker). This is where the application is bundled together with all its dependencies and a part of the operating system.

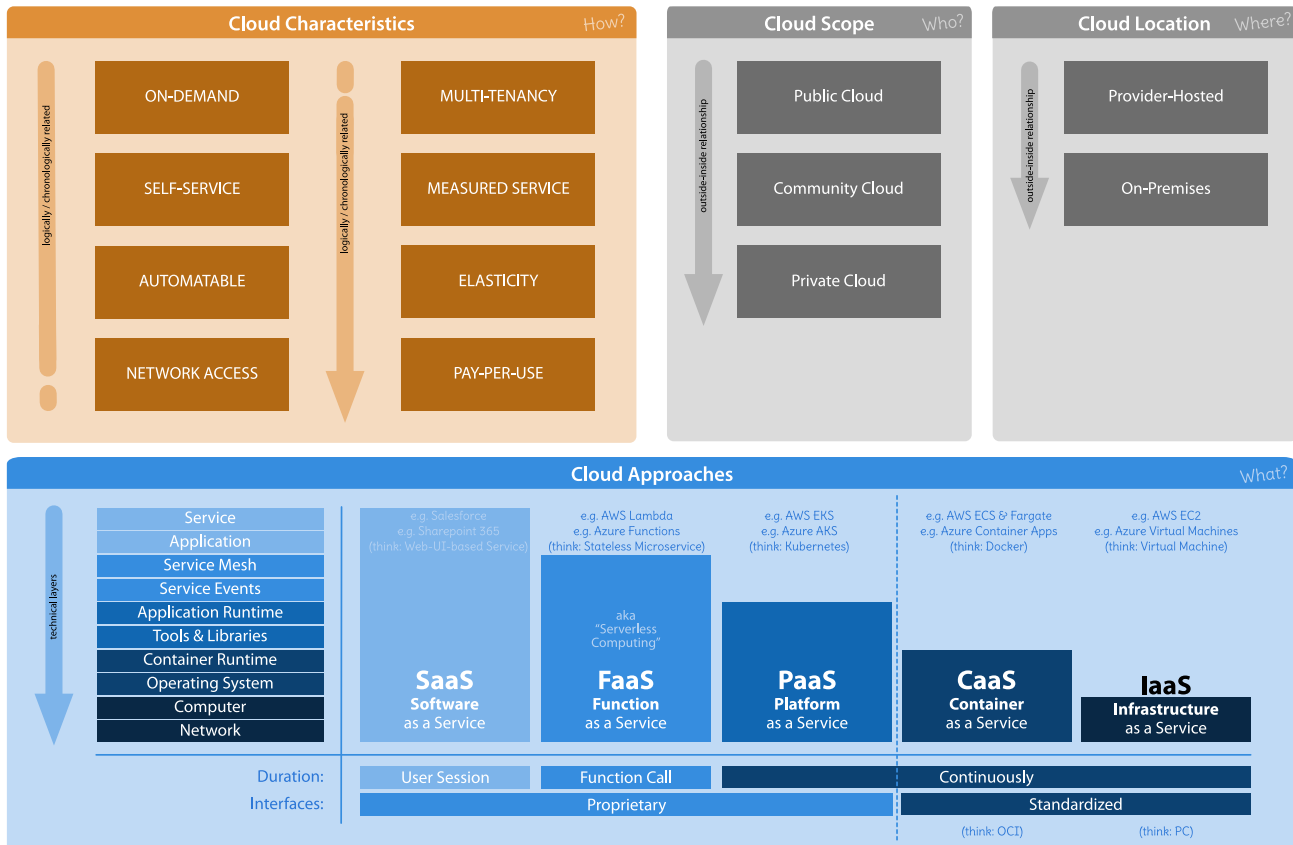
To be more flexible, one can keep the **Managed Packages** or **Container Images** very small and instead define an application through an entire **Package/Container Stack** (e.g., Docker Compose).

If one needs more shielding, a **Virtual Machine Image** offers itself. Here the application is bundled with all its dependencies and the complete operating system and is installed on a virtual machine (e.g. ORACLE VirtualBox). As the maximum expansion level, the application can be installed as a **Solution Appliance**, where the application, its dependencies, the associated operating system, and the underlying hardware are bundled into one total solution (e.g., SAP HANA).

In practice, the various approaches occur mainly in combined form. A **Container Stack** consists of **Container Images**. These, in turn, are built by installing dependencies via **Managed Packages**, and the application itself as an **Unmanaged Heap**, into the container. The **Managed Packages**, beforehand during packaging, are created with **Bare Amalgamation** steps.

Questions

- ❓ Which type of **Software Deployment** bundles and installs an application with all its dependencies and part of the operating system?



Cloud Computing has four essential dimensions. The first dimension **Cloud Characteristics** ("How?") describes the eight characteristics of how a resource provisioning must happen in order for the provisioning to be considered as **Cloud Computing**: **On-Demand**, **Self-Service**, **Automatable**, **Network-Access**, **Multi-Tenancy**, **Measured Service**, **Elasticity** (aka Scalability), and **Per-Per-Use**.

With these characteristics, in the second dimension, there are various **Cloud Approaches** ("What?"), which specify what is provided: for **Infrastructure as a Service (IaaS)**, only **Network** and a **Computer** is provided, usually a virtual machine. With **Container as a Service (CaaS)** additionally a (Host) **Operating System** and a **Container Run-Time** are provided.

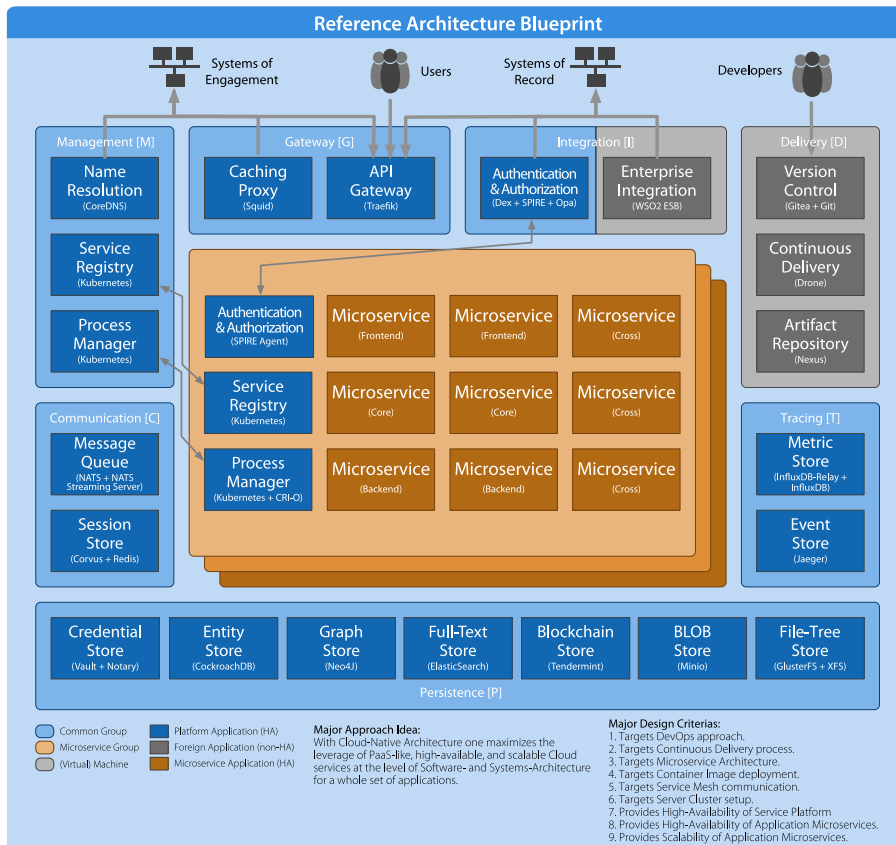
For **Platform as a Service (PaaS)**, additional surrounding **Tools & Libraries** and an **Application Run-Time** are provided; with **Function as a Service (FaaS)** additionally external **Service Events** and a **Service Mesh** and for **Software as a Service (SaaS)** the **Application** and the its (functional) **Service** are additionally provided.

The third dimension **Cloud Scope** ("Who?"), states for whom the resources are provided: **Public Cloud** for public Cloud Computing, **Community Cloud** for Cloud Computing of a closed group of organizations, and **Private Cloud** for Cloud Computing of a single organization.

Finally, the fourth dimension **Cloud Location** ("Where?"), states where the resources are physically provided: **Provider-Hosted** means at an external provider, **On-Premises** means locally at the using organization.

Questions

- ? List at least 5 of the 8 **Cloud Characteristics** that a resource provisioning must fulfill for it to be considered **Cloud Computing**!
- ? In which **Cloud Approach** is only **Network** and **Computer** provided?



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.

Practical Cluster Setups

Standard Cluster Setup (5+1+N Machines):



Partitioned Cluster Setup (5x2+1+N Machines):



In **Cloud-Native Architecture**, applications are developed, installed and operated in such a way that the advantages of **Cloud Computing** are maximized and, in particular, that all infrastructure services are provided by a central **Service Platform**.

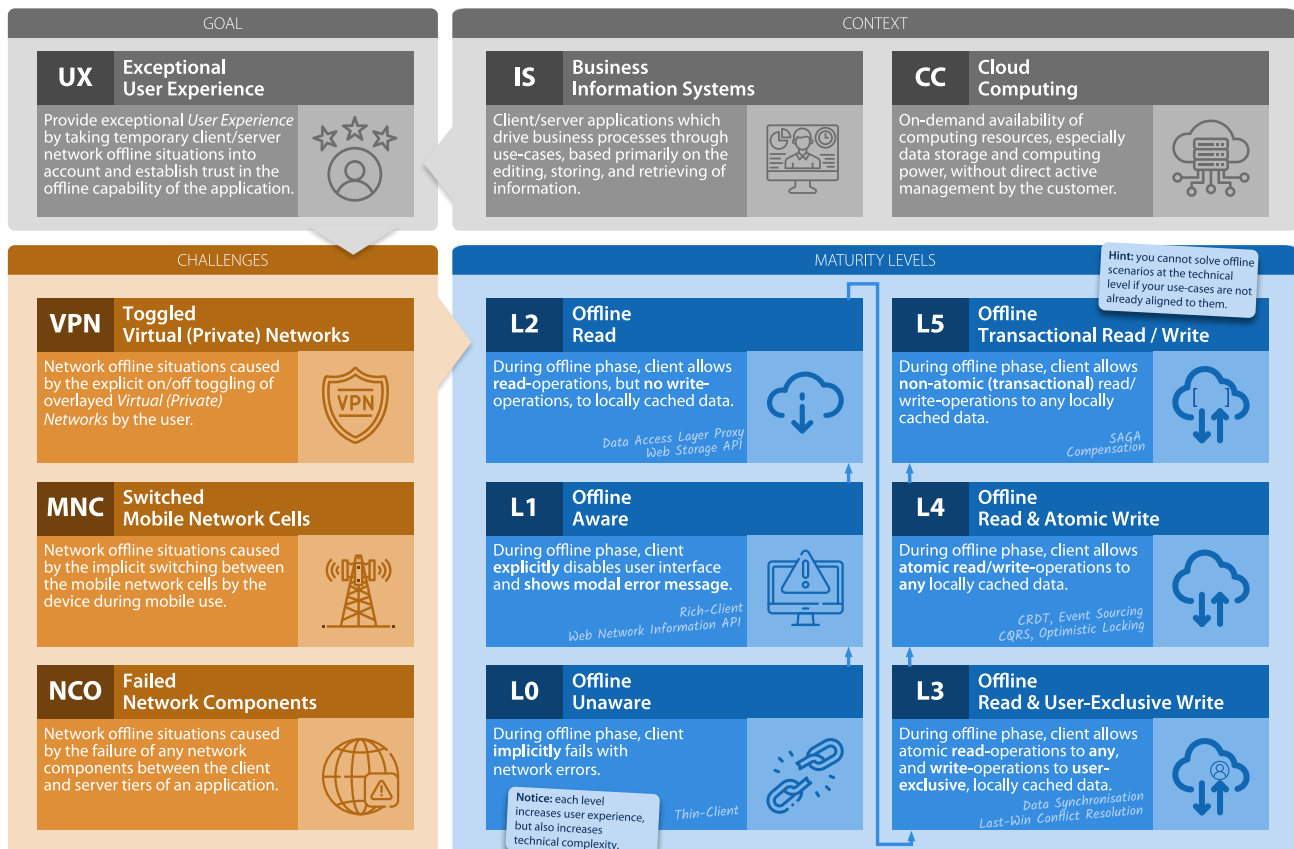
In practice, this ideally means the combination of an agile **DevOps** approach, an end-to-end **Continuous Delivery** process, a flexible **Microservice** software architecture, the use of a stable **Container Image** based software deployment, the use of a **Service Mesh** for internal Microservice communication, and the use of a **Server Cluster** for scaling the Microservices.

The **Service Platform** is divided into the 7 service areas **Management, Gateway, Integration, Tracing, Persistence, Communication** plus **Delivery**, which are usually partitioned in a failsafe 5+1 or alternatively in a partially partitioned form on 5x2+1 machines. The Microservices of the application are installed on the **Service Platform** on separate machines.

In a **Cloud-Native Architecture**, it comes down to achieving **High Availability** and **Scalability** for both the services of the platform as well as for the Microservices of the application.

Questions

- ❓ On which two essential aspects is the **Cloud-Native Architecture** based?
- ❓ What does the **Cloud-Native Architecture** offer to the **Microservices** of an application?



Offline capabilities can be essential for an exceptional User Experience of client/server-based Business Information Systems in the context of Cloud Computing. The challenges for the temporary network offline situations include toggled Virtual Private Networks (VPN), switched mobile network cells, and failed network components.

In offline scenarios, during an offline phase, an application can support a particular maturity level: At **Offline Unaware**, the client implicitly fails with network errors; At **Offline Aware**, the client explicitly disables user interface and shows modal error message; At **Offline Read**, the client allows read-operations, but no writeoperations, to locally cached data; At **Offline Read & User-Exclusive Write**, the client allows atomic read-operations to any, and write-operations to userexclusive, locally cached data; At **Offline Read & Atomic Write**, the client allows atomic read/write-operations to any locally cached data; At **Offline Transactional Read / Write**, the client allows non-atomic (transactional) read/ write-operations to any locally cached data.

Questions

- ?** Why can offline capability of applications be crucial in the context of Cloud Computing?