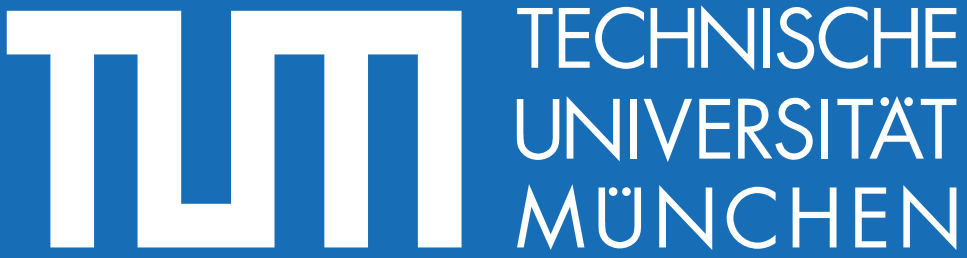


Software Engineering in der industriellen Praxis (SEIP)

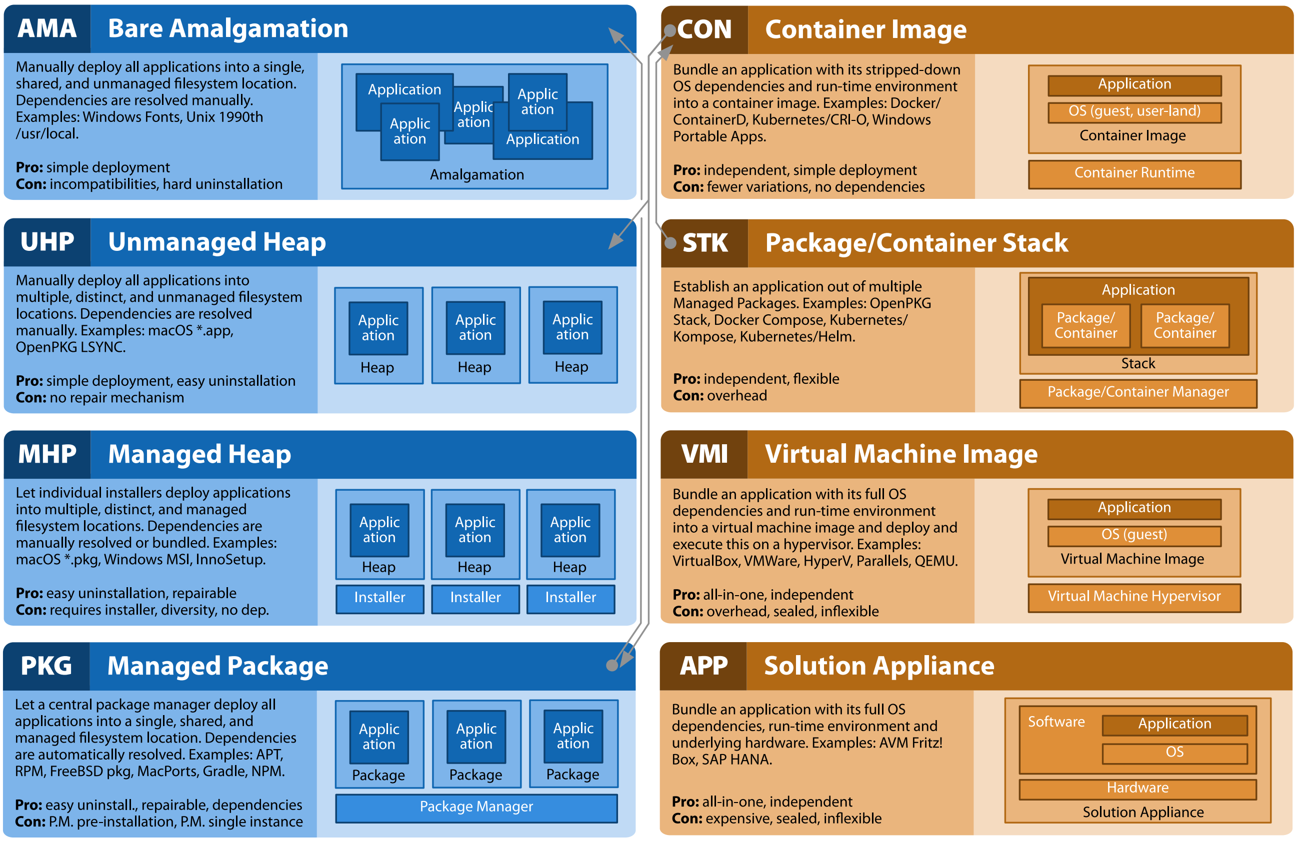
Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

Software Deployment

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 13.1



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).

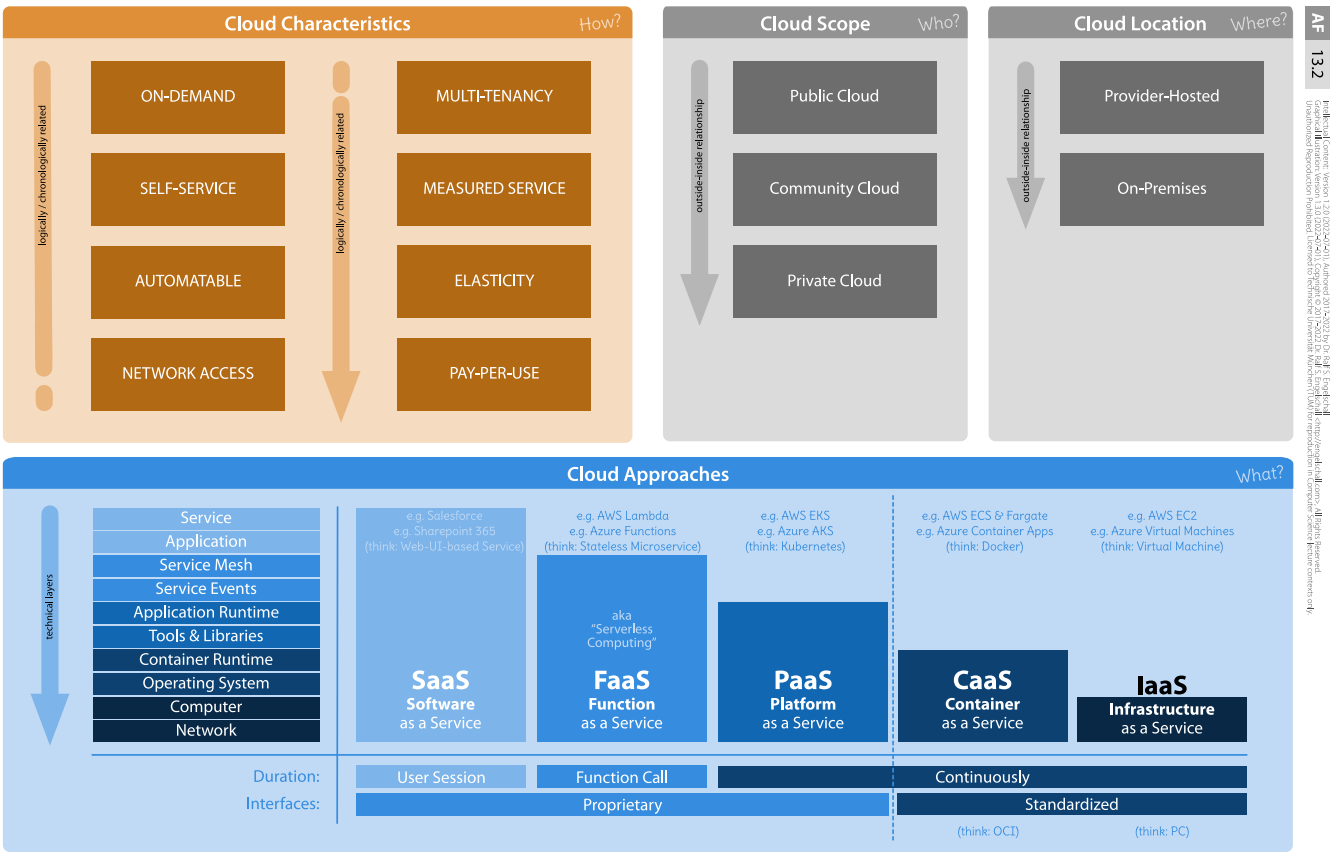
If one needs more shielding, a **Virtual Machine Image** offers itself. Here the application is bundled with all its dependencies and the complete operating system and is installed on a virtual machine (e.g. ORACLE VirtualBox). As the maximum expansion level, the application can be installed as a **Solution 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 Resources



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.

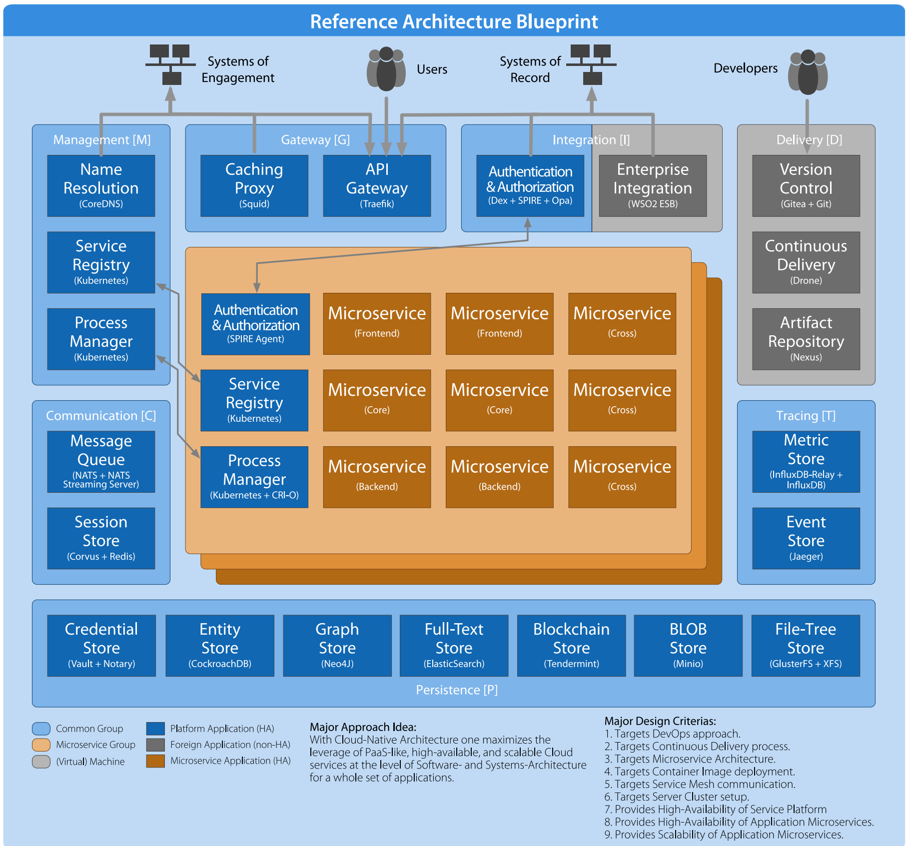
Questions

- 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?

ARCHITECTURE FUNDAMENTALS

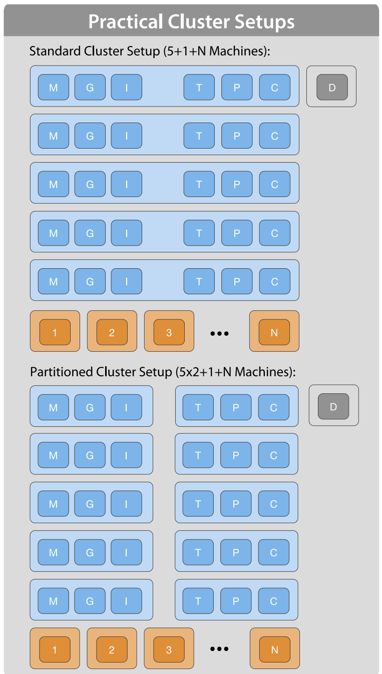
Cloud-Native Architecture

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.



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.

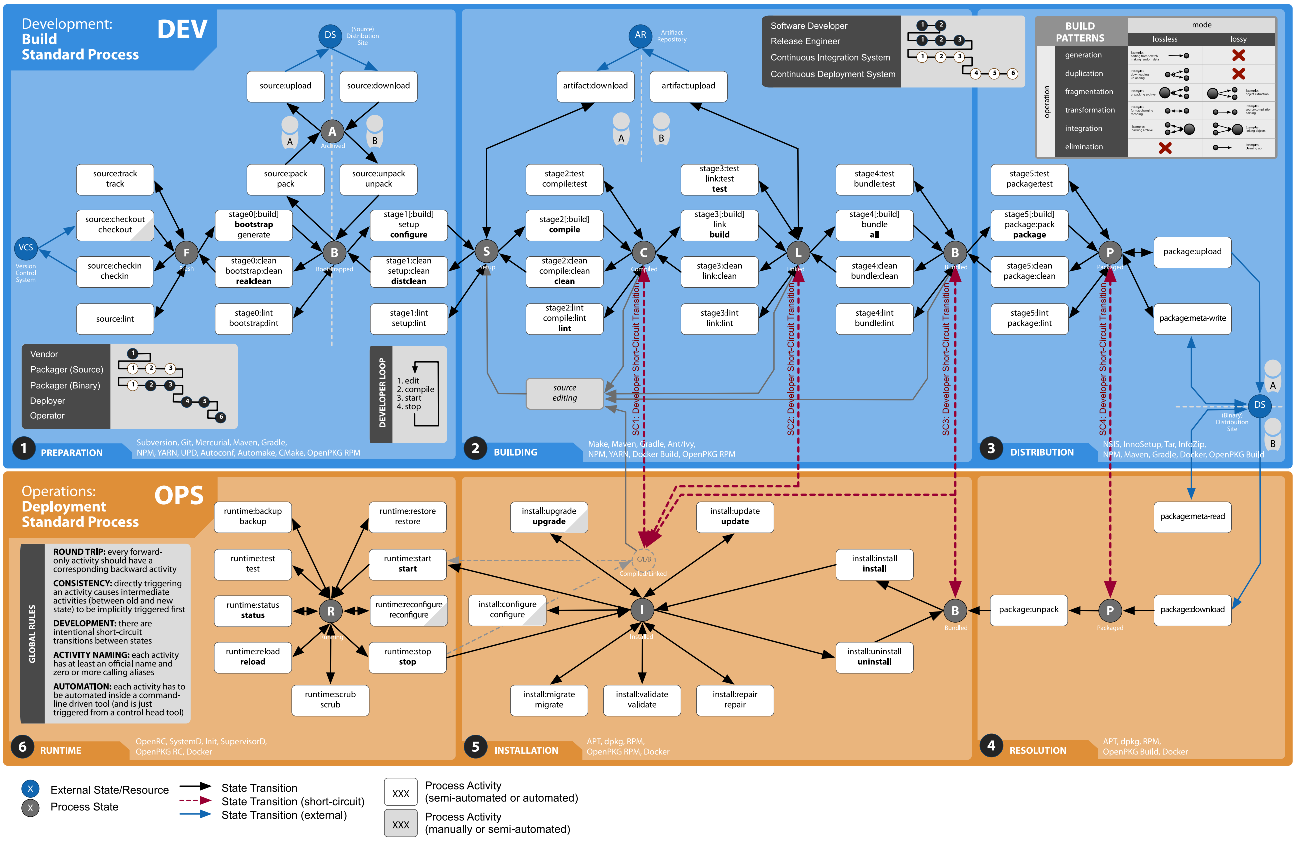
Questions

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

ARCHITECTURE FUNDAMENTALS

Assembly Process Architecture

TECHNISCHE UNIVERSITÄT MÜNCHEN



In the **Assembly Process Architecture**, a **DevOps Pipeline** is used to automatically transition a version of a software product from the sources in the **Version Control System** to the running instance in operation.

The Dev(evelopment) part of the **DevOps Pipeline**, the so-called **Build Standard Process**, is usually automated via **Continuous Integration (CI)**. The Op(eration)s part of the **DevOps Pipeline**, the so-called **Deployment Standard Process**, is usually automated via **Continuous Integration (CI)**. **Deployment Standard Process**, is usually automated via **Continuous Deployment (CD)**. All activities of the **DevOps Pipeline** are automated via specialized build and deployment tools and these activities are automatically executed in a CI/CD system after each change in the **Version Control System**.

In particular, the **Assembly Process** should enable a meaningful **Round Trip**, in that the process is understood as a state machine and for all (forward) activities there are meaningful associated backward activities are existing. If a certain target state is externally requested, all intermediate activities between the source and the target state are implicitly executed.

For productivity or the most efficient "Developer Loop" possible in software development, up to four **Short Circuit Transitions** are supported, with which a deliberate "Shortcut" from the **Build Standard Process** to the **Deployment Standard Process** can be made.

The **Assembly Process Architecture** makes use of four different types of external storage locations: the **Version Control System** stores the "bare" source files, the **(Source) Distribution Site** stores the "source distribution" of the source files prepared for the build process, the **Artifact Repository** stores reused build artifacts (especially libraries) and the **(Binary) Distribution Site** stores the "binary distribution" of the product intended for deployment.

The first three locations are mainly used for passing data between different people. The last one is mainly used for passing data between Dev(elopment) and Op(eration)s.

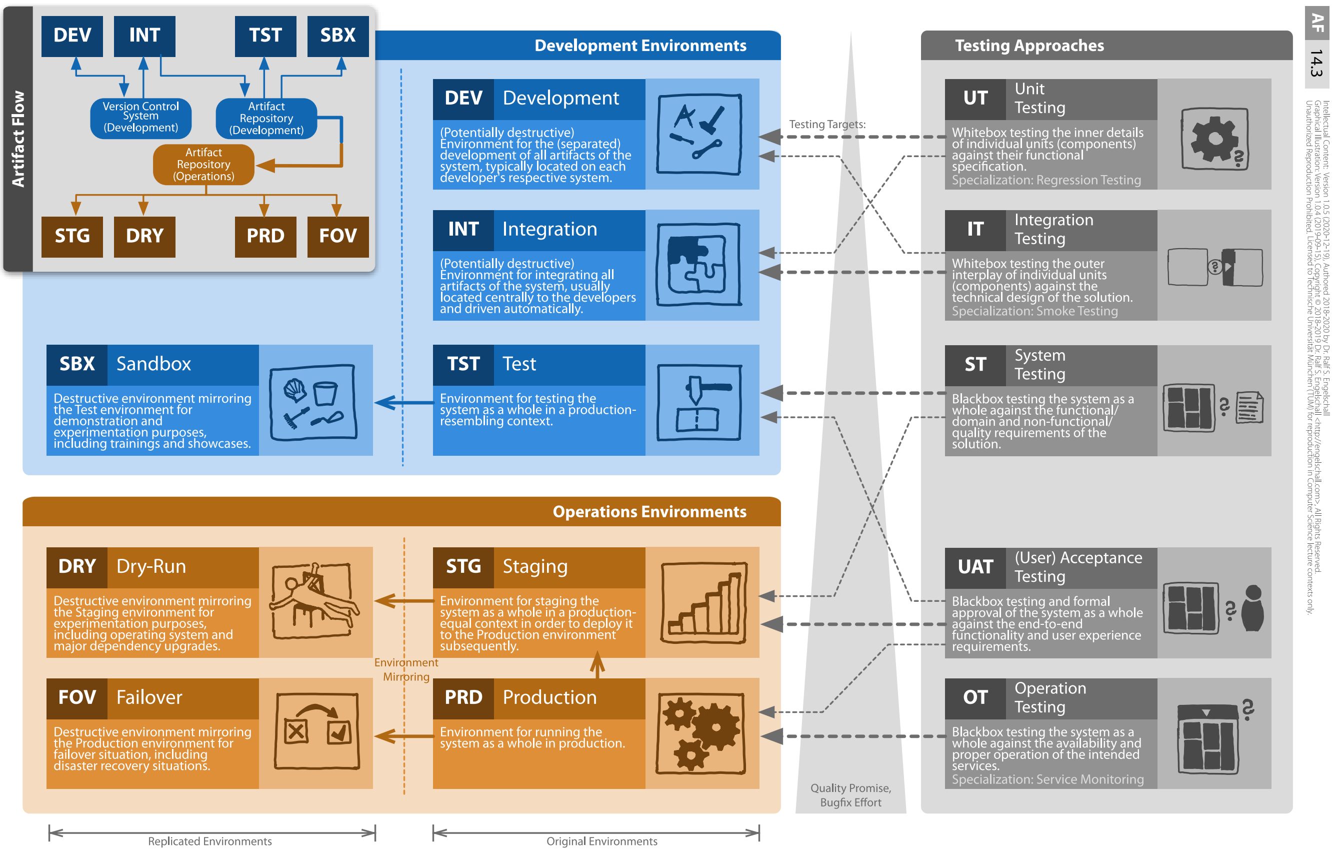
Questions

- Why should in the **Assembly Process Architecture** up to four so-called **Short-Circuit Transitions** be supported to shortcut from the **Build Standard Process** to the **Deployment Standard Process**?
- What are the four **types of external storage locations** supported by the **Assembly Process Architecture**?

ARCHITECTURE FUNDAMENTALS

Environments & Quality Assurance

TECHNISCHE UNIVERSITÄT MÜNCHEN



In practice, a distinction is usually made between 4 **Development Environments**, 4 **Operations Environments** and 5 associated **Testing Approaches**.

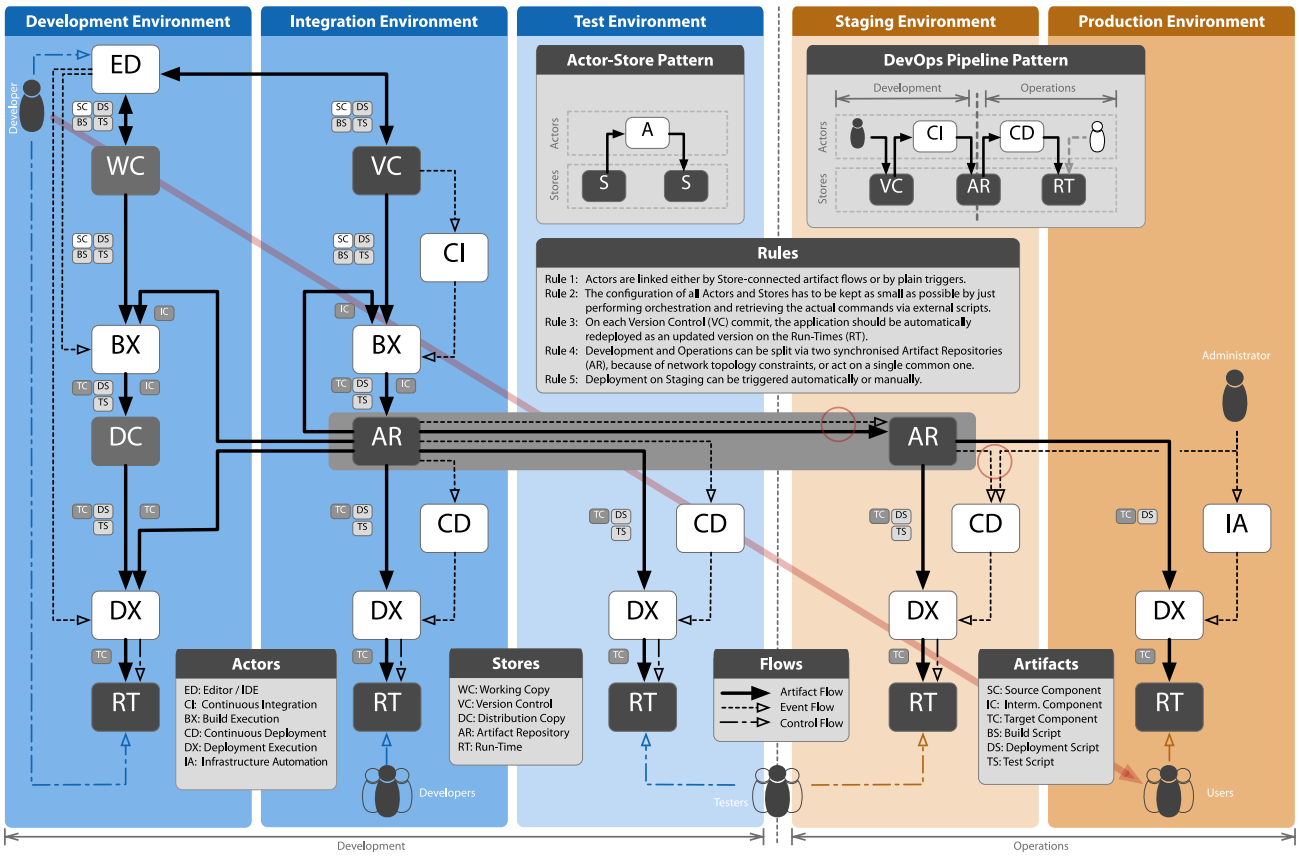
The **Development Environments** are **Development** (e.g., the computer of the developer), **Integration** (e.g., a central server with a Continuous Integration (CI) system), **Test** (e.g., a central server that resembles the **Production** environment, but is not a copy of it) and possibly **Sandbox** (e.g., a server that is a copy of **Test** for training and show-cases).

The **Operations Environments** are **Staging** (a copy of **Production**), **Dry-Run** (a 1:1 copy of **Staging**), **Production** (the regular production environment) and **Failover** (a 1:1 copy of **Production**). **Production**).

The **Testing Approaches** are **Unit Testing** for the functionality of components on the **Development** (and possibly **Integration**) environment, **Integration Testing** for the interaction of components on **Integration** (and if applicable **Development**) environment, **System Testing** for the functional and non-functional properties of the overall system on **Test** (and **Staging** if applicable) environment, **(User) Acceptance Testing** for the “end-to-end” functionality of the overall system on **Staging** (or **Production** if applicable) environment and **Operation Testing** for the availability of the overall system on the overall system on the **Production** environment.

As transfer points for the artifacts between the 8 environments serve a **Version Control System** and an **Artifact Repository** on the side of the **Development Environments** and a corresponding **Artifact Repository** on the side of the **Operations Environments**.

DevOps Toolchain



To support a DevOps approach on the tool-side as well, a **DevOps Toolchain** is advised to be used. At its core, this is based on a pattern in which an **Actor** acts between two **Stores** in each case by taking one or more **Artifacts** as input from a **Store**, processes these and writing one or more **Artifacts** as an output to another **Store**. Additionally, **Actors** can be triggered by **Events** or can be controlled directly by different groups of people through interactions.

This basic pattern is now combined to a **DevOps Pipeline Pattern**, where every "Commit" of a **Developer** in a **Version Control** (VC) system triggers an automatic compilation and integration process of an application in a **Continuous Integration** (CI) system. The results of this process are stored in an **Artifact Repository** (AR), which in turn triggers an automatic installation process in a **Continuous Deployment** (CD) system. The result is the installed application on a **Run-Time** (RT) system, which can be accessed by **Testers** and **Users**.

Logically, the systems VC and CI belong to the area **Development**, while the CD and RT systems belong to the **Operations** area. The AR system, on the other hand, is used by both areas as a common transfer point. Since in practice, the **DevOps Toolchain** is not on a single **Environment**, but is usually distributed on the logically (or even physically) separated **Environments Development**, **Integration**, **Testing**, **Staging** and **Production**, some systems exist multiple times.

We distinguish the **Artifacts** between **Source Component** (`foo.java`), **Intermediate Component** (`foo.jar`) and **Target Component** (`foo.exe`) and, on the other hand, between **Build Script** (`foo.make`), **Deployment Script** (`foo.spec`) and **Test Script** (`foo-test.java`).

Note intentional special cases: The **Development Environment** is different because it allows a fast "edit-build-install-start-stop" loop. The AR system can exist 1 or 2 times, depending on how strongly interweaved Development and Operations are. Deployment in the **Production Environment** should be manually triggered via an **Infrastructure Automation** system.

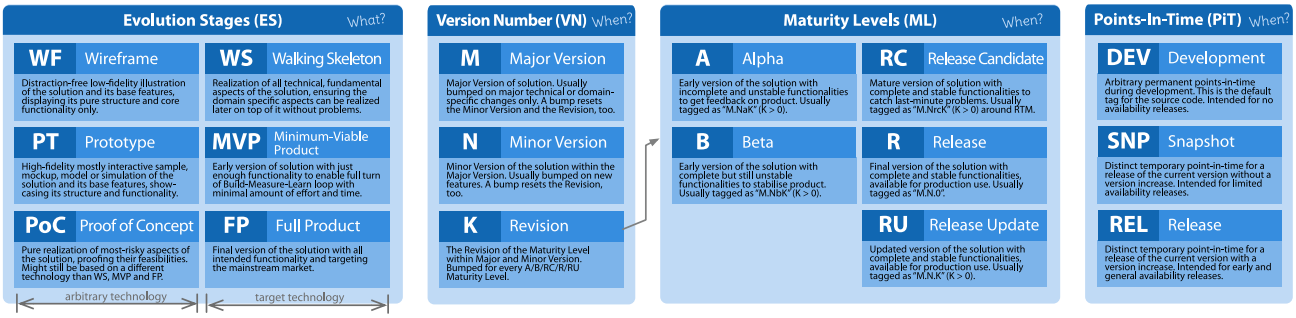
Questions

- Which two **Actor** systems control in the **DevOps Pipeline Pattern** the automated integration and installation process?

ARCHITECTURE FUNDAMENTALS

Software Release Management

TECHNISCHE UNIVERSITÄT MÜNCHEN



In **Software Release Management**, **Releases** of software products are controlled by 7 dimensions, which identify the release via a well-defined **Version Schema**.

The dimension **Evolution Stages** is about the type and development stage of the product. A distinction is made between the (partly not yet based on the later technology) pre-stages **Wireframe**, **Prototype** and **Proof of Concept** and the production-ready stages (which are based on the target technology) **Walking Skeleton**, **Minimum-Viable Product** and **Full Product**.

In the **Version Number** dimension, the product is identified by three numbers: **Major Version**, **Minor Version**, and **Revision**. The first two refer to the content of the product. The latter refers to the respective **Maturity Level** of the product. The **Maturity Levels** dimension itself defines the maturity of the product within the **Major/Minor Version**: **Alpha** (a, incomplete, unstable), **Beta** (b, complete, unstable), **Release Candidate** (rc, complete, stable), **Release** and **Release Update**.

The dimension **Points-In-Time** tells you whether the current release has the status **Development** (as the product is in the VCS), whether it is a special **Snapshot** (e.g., for extremely time-critical hotfixes or early feedbacks) or if it is a normal **Release**.

The dimension **Product Editions** defines the edition or variant of the product: usually **Community/Enterprise Edition** (with focus on the target market) or **Standard/Professional Edition** (with the focus on the range of functions).

The dimension **Availability Scopes** defines for whom the release of the product is available: **No Availability** (only for the manufacturer internally), **Limited Availability** (for special situations), **Early Availability** (for early adopters among customers) and **General Availability** (for all customers). Usually, the Availability Scopes are used for specific **Maturity Levels**, but there is no necessary connection.

The dimension **Distribution Channels** defines over which channels the release is available: **Bleed Channel** (for "Visionaries" and "Die Hards"), **Edge Channel** (for "Early Adopters"), **Stable Channel** and and **Solid Channel** (for all customers). In any case, the **Distribution Channels** should be directly linked to the branches of the VCS.

Questions

- At which **Maturity Level** in **Software Release Management** does one have an incomplete and unstable functionality?
- At what **Maturity Level** in **Software Release Management** does one have complete but usually still unstable functionality?