



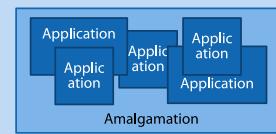
# Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

## 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.

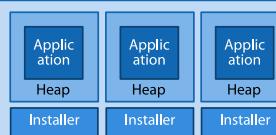
**Pro:** simple deployment, easy uninstallation  
**Con:** no repair mechanism



## 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.

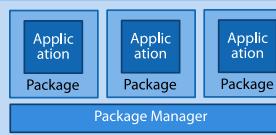
**Pro:** easy uninstallation, repairable  
**Con:** requires installer, diversity, no dep.



## 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 uninstall, 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.

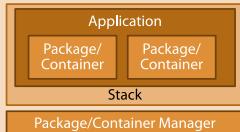
**Pro:** independent, simple deployment  
**Con:** fewer variations, no dependencies



## STK Package/Container Stack

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

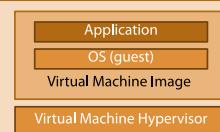
**Pro:** independent, flexible  
**Con:** overhead



## 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.

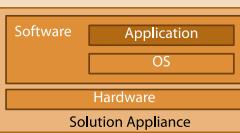
**Pro:** all-in-one, independent  
**Con:** overhead, sealed, inflexible



## 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.

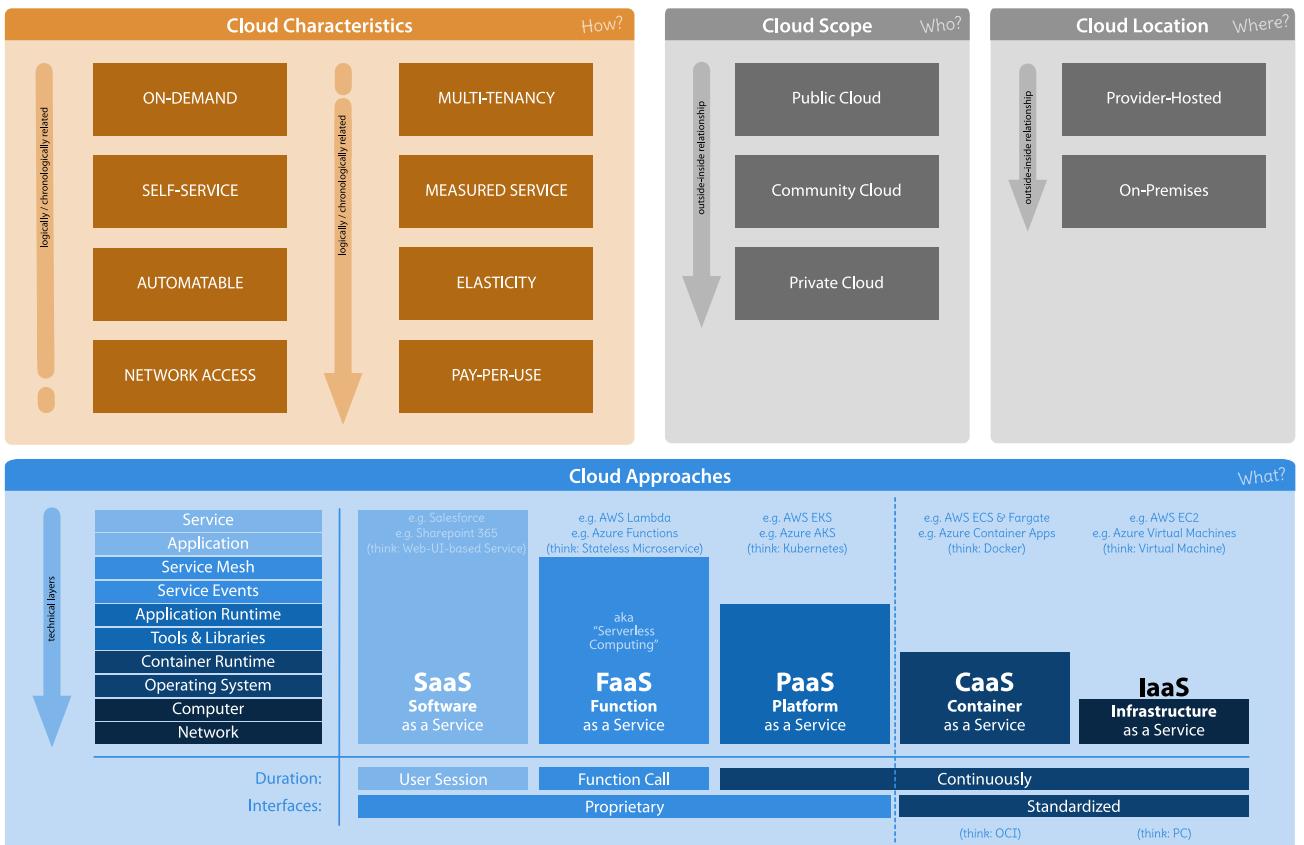
Um flexibler zu sein, kann man die **Managed Packages** oder **Container Images** sehr klein halten und stattdessen eine Application über einen ganzen **Package/Container Stack** definieren (z.B. Docker Compose).

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 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.

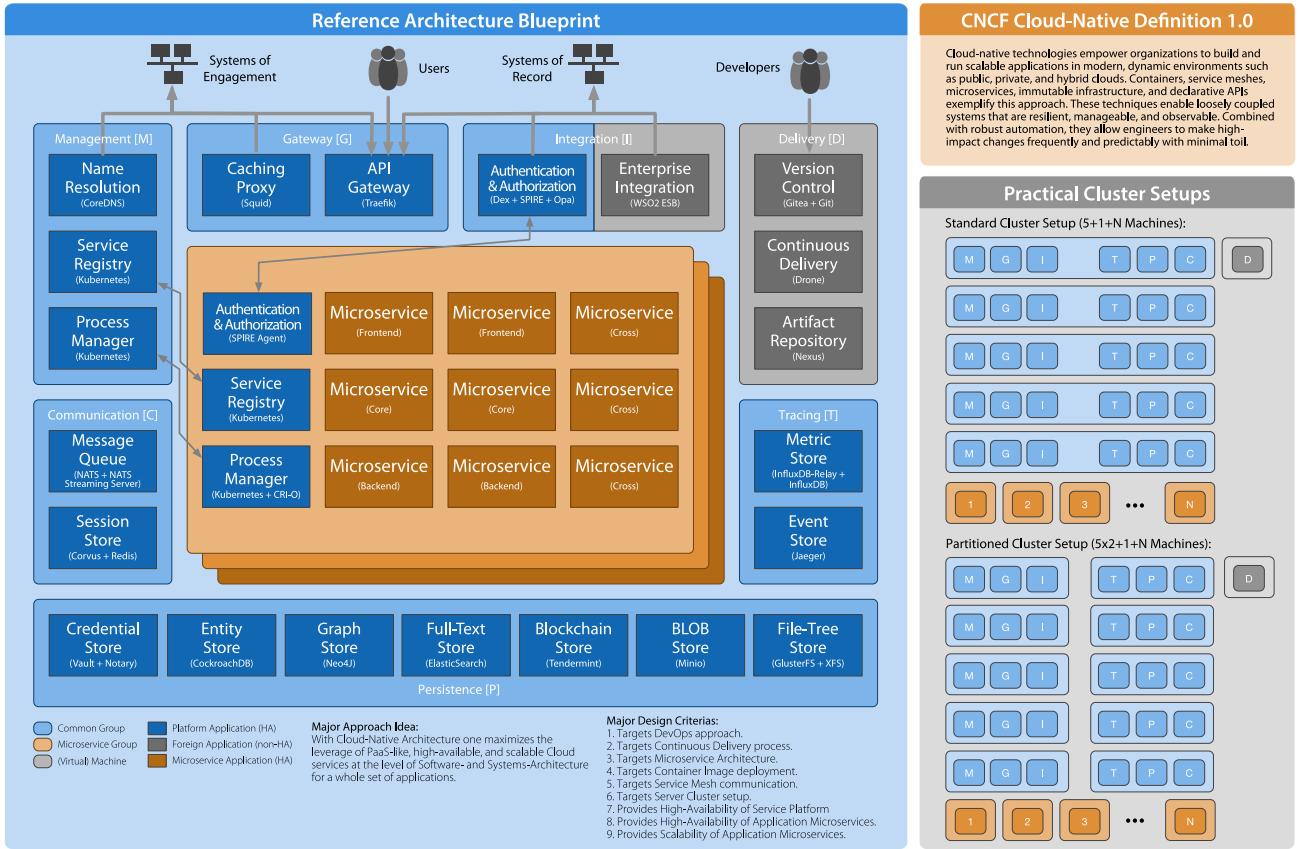
Bei **Platform as a Service (PaaS)** werden zusätzlich umgebende **Tools & Libraries** und eine **Application Run-Time** bereitgestellt; bei **Function as a Service (FaaS)** werden zusätzlich externe **Service Events** und ein **Service Mesh** bereitgestellt und bei **Software as a Service (SaaS)** werden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

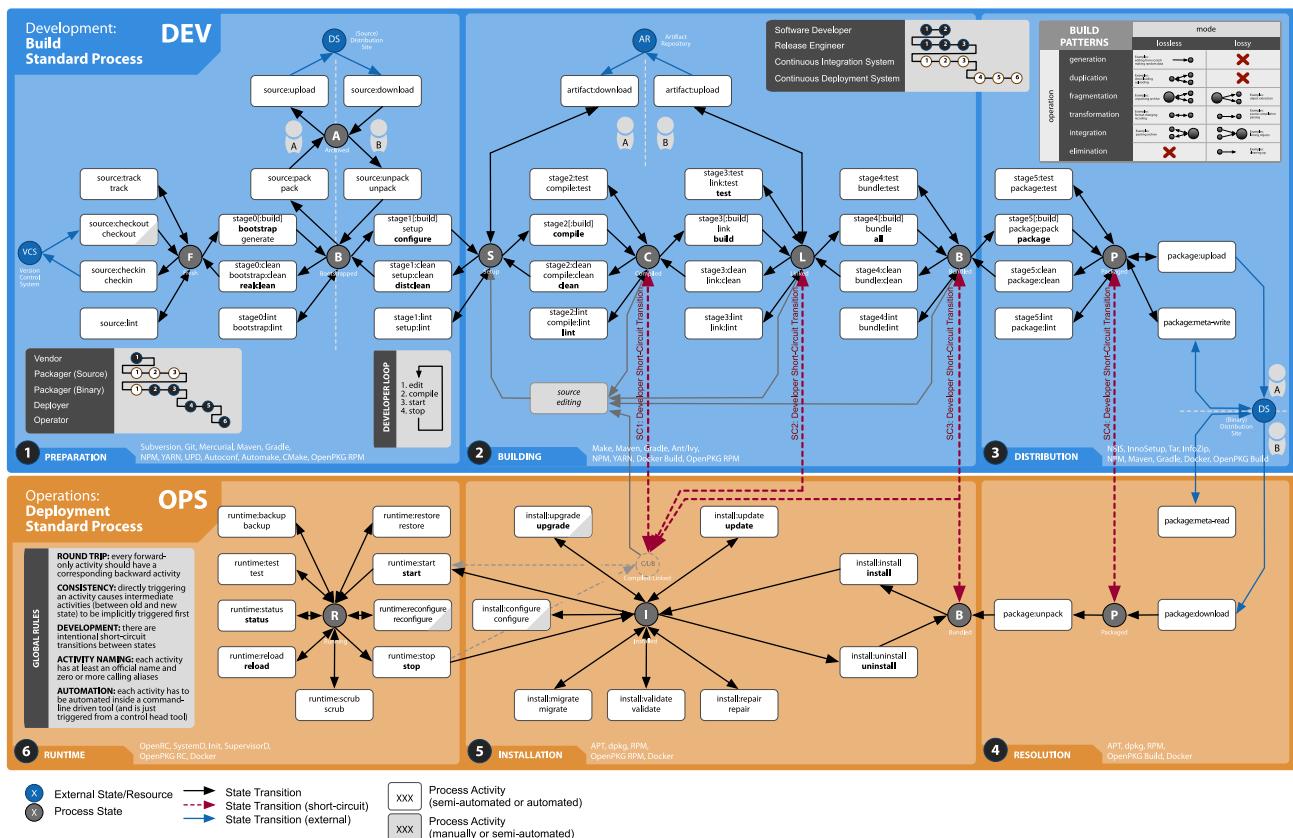
In der Praxis bedeutet dies im Idealfall die Kombination aus einem agilen **DevOps** Vorgehen, einem durchgängigen **Continuous Delivery** Prozess, einer flexiblen **Microservice** Software Architecture, dem Einsatz eines stabilen **Container Image** basierten Software Deployments, der Nutzung eines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



Bei der Assembly Process Architecture wird eine **DevOps Pipeline** genutzt, um eine Version eines Software-Produkts automatisiert von den Quellen im **Version Control System** bis zur laufenden Instanz im Betrieb zu bringen.

Der Dev(evelopment)-Anteil der **DevOps Pipeline**, der sog. **Build Standard Process**, wird üblicherweise über **Continuous Integration (CI)** automatisiert. Der Op(eration)s-Anteil der **DevOps Pipeline**, der sog. **Deployment Standard Process**, wird üblicherweise über **Continuous Deployment (CD)** automatisiert. Alle Aktivitäten der **DevOps Pipeline** werden über spezialisierte Build- und Deployment-Werkzeuge automatisiert und diese Aktivitäten in einem CI/CD-System nach jeder Änderung im **Version Control System** automatisch ausgeführt.

Der **Assembly Process** sollte insbesondere einen sinnvollen **Round Trip** ermöglichen, indem der Prozess als Zustandsautomat verstanden wird und zu allen (Vorwärts-)Aktivitäten sinnvolle zugehörige Rückwärts-Aktivitäten existieren. Wenn von außen ein bestimmter Ziel-Zustand gefordert wird, werden implizit alle Zwischen-Aktivitäten zwischen dem Quell- und dem Ziel-Zustand ausgeführt.

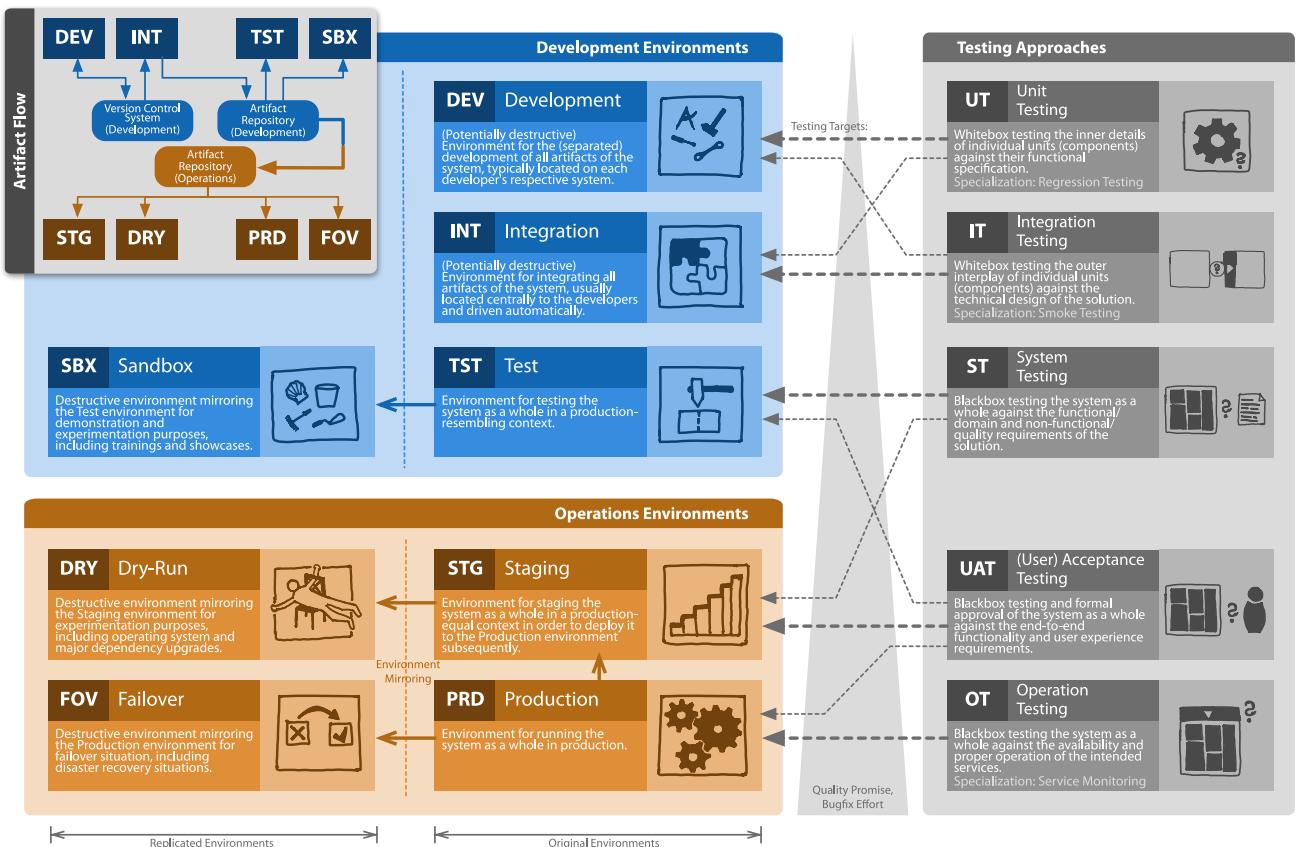
Für die Produktivität bzw. die möglichst effiziente "Developer Loop" in der Software Entwicklung sollten bis zu vier **Short-Circuit Transitions** unterstützt werden, bei dem eine bewusste "Abkürzung" vom **Build Standard Process** zum **Deployment Standard Process** vorgenommen werden kann.

Die **Assembly Process Architecture** nutzt vier unterschiedliche Arten von externen Speicherorten: das **Version Control System** speichert die "nackten" Quell-Dateien, die **(Source) Distribution Site** speichert die für den Build-Prozess vorbereitete "Source Distribution" der Quell-Dateien, das **Artifact Repository** speichert wiederverwendete Build-Artefakte (vor allem Bibliotheken) und die **(Binary) Distribution Site** speichert die für das Deployment vorgesehene "Binary Distribution" des Produkts.

Die ersten drei Speicherorte dienen vor allem der Übergabe von Daten zwischen unterschiedlichen Personen. Letzterer dient vor allem der Übergabe von Daten zwischen Dev(evelopment) und Op(eration)s.

## Fragen

- Wieso sollten in der **Assembly Process Architecture** bis zu vier sogenannte **Short-Circuit Transitions** zur Abkürzung vom **Build Standard Process** zum **Deployment Standard Process** unterstützt werden?
- Welche vier Arten von externen Speicherorten kennt die **Assembly Process Architecture**?



In der Praxis unterscheidet man üblicherweise 4 **Development Environments**, 4 **Operations Environments** und 5 zugehörige **Testing Approaches**.

Die **Development Environments** sind **Development** (z.B. der Rechner des Entwicklers), **Integration** (z.B. ein zentraler Server mit Continuous Integration (CI) System), **Test** (z.B. ein zentraler Server, der dem **Production** Environment ähnelt, aber keine Kopie davon ist) und ggf. **Sandbox** (z.B. ein Server, der als Kopie von **Test** für Schulungen und Show-Cases zur Verfügung steht).

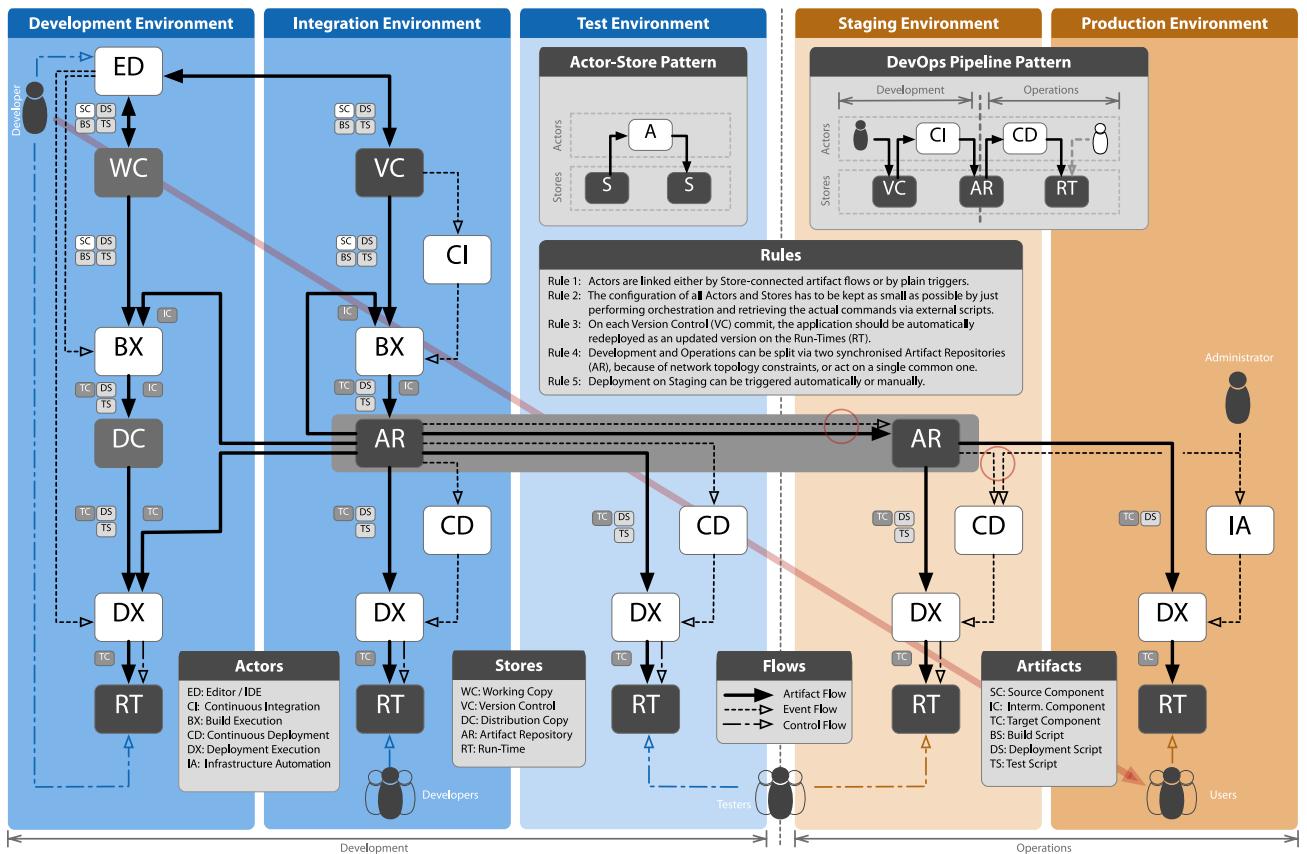
Die **Operations Environments** sind **Staging** (eine Kopie von **Production**), **Dry-Run** (eine 1:1 Kopie von **Staging**), **Production** (die reguläre Produktionsumgebung) und **Failover** (eine 1:1 Kopie von **Production**).

Die **Testing Approaches** sind **Unit Testing** für die Funktionalität von Komponenten auf dem **Development** (und ggf. **Integration**) Environment, **Integration Testing** für das Zusammenspiel von Komponenten auf **Integration** (und ggf. **Development**) Environment, **System Testing** für die funktionalen und nicht-funktionalen Eigenschaften des Gesamtsystems auf dem **Test** (und ggf. **Staging**) Environment, **(User) Acceptance Testing** für die "End-To-End" Funktionalität des Gesamtsystems auf **Staging** (oder ggf. **Production**) Environment und **Operation Testing** für die Verfügbarkeit des Gesamtsystems auf dem **Production** Environment.

Als Übergabepunkte der Artefakte zwischen den 8 Environments dienen ein **Version Control System** und ein **Artifact Repository** auf Seite der **Development Environments** und ein zugehöriges **Artifact Repository** auf Seite der **Operations Environments**.

## Fragen

- ❓ Wie heißt die Kopie des **Production** Environments, auf dem u.a. **(User) Acceptance Testing** durchgeführt wird?



Um einen DevOps-Ansatz auch Werkzeug-seitig zu unterstützen, bietet sich eine **DevOps Toolchain** an. Diese basiert im Kern auf einem Muster, bei dem ein **Actor** zwischen jeweils zwei **Stores** agiert, indem er ein oder mehrere **Artifacts** als Eingabe von einem **Store** liest, diese verarbeitet und als Ausgabe ein oder mehrere **Artifacts** in einen anderen **Store** schreibt. Zusätzlich können **Actors** über **Events** angestoßen werden oder direkt von verschiedenen Personenkreisen durch Interaktionen gesteuert werden.

Dieses Grundmuster wird nun zu einem **DevOps Pipeline Pattern** kombiniert, bei dem bei jedem "Commit" eines **Developer** in das **Version Control** (VC) System ein automatischer Übersetzungs- und Integrations-Prozess einer Anwendung in einem **Continuous Integration** (CI) System angestoßen wird. Dessen Ergebnisse werden in einem **Artifact Repository** (AR) zentral aufgesammelt, wodurch wiederum ein automatischer Installations-Prozess in einem **Continuous Deployment** (CD) System angestoßen wird. Dessen Ergebnis ist die installierte Anwendung auf einem **Run-Time** (RT) System, auf welches durch **Tester** und **User** zugegriffen werden kann.

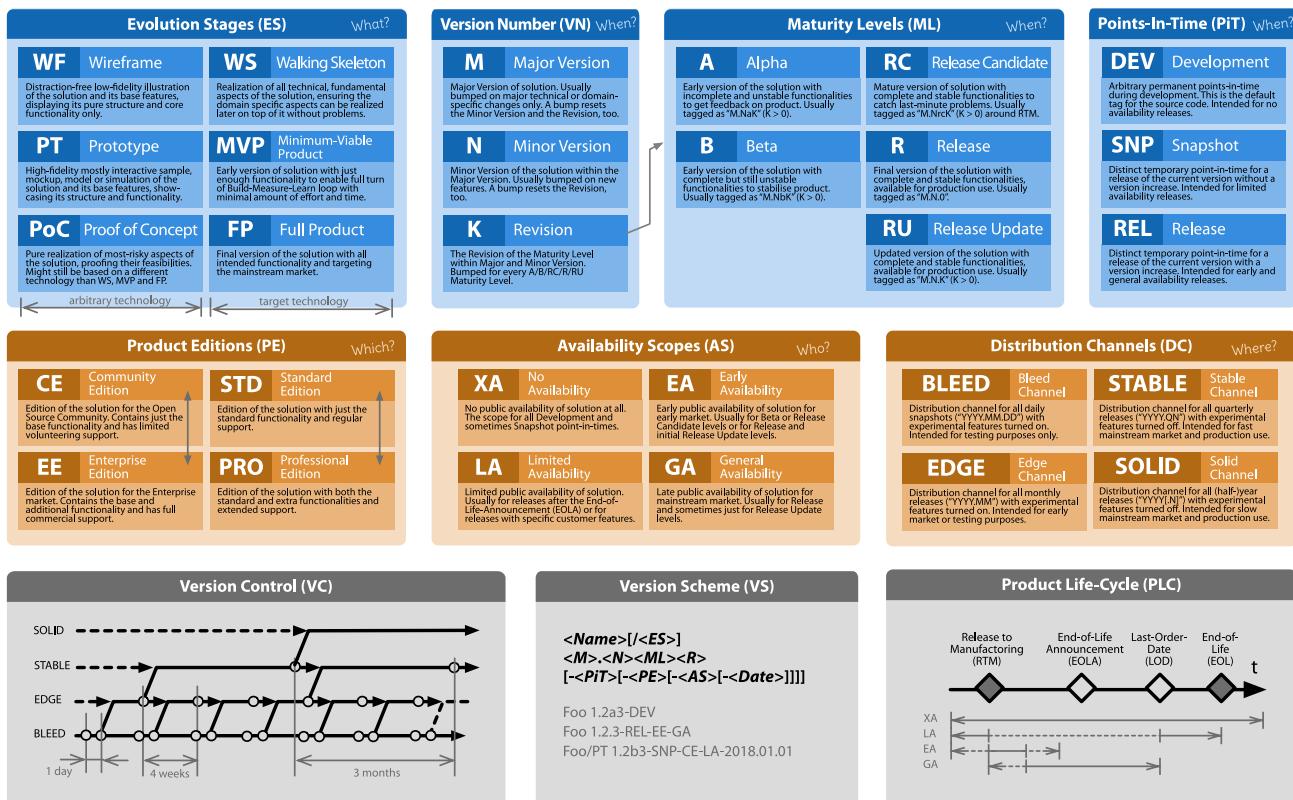
Logisch gehören die Systeme VC und CI zum Bereich **Development**, während die Systeme CD und RT zum Bereich **Operations** gehören. Das System AR wird dagegen von beiden Bereichen als gemeinsamer Übergabepunkt verwendet. Da in der Praxis die **DevOps Toolchain** nicht in einem einzigen **Environment** existiert, sondern üblicherweise auf die logisch (oder sogar physisch) getrennten **Environments Development, Integration, Test, Staging und Production** aufgeteilt ist, existieren einige Systeme mehrfach.

Bei den **Artifacts** wird einerseits zwischen **Source Component** (`foo.java`), **Intermediate Component** (`foo.jar`) und **Target Component** (`foo.exe`) und andererseits zwischen **Build Script** (`foo.make`), **Deployment Script** (`foo.spec`) und **Test Script** (`foo-test.java`) unterschieden.

Beachte absichtliche Sonderfälle: Das **Development Environment** ist unterschiedlich, da es eine schnelle "Edit-Build-Install-Start-Stop"-Schleife unterstützt. Das AR-System kann 1 oder 2 Mal existieren, abhängig davon, wie stark Development und Operations miteinander verzahnt sind. Das Deployment im **Production Environment** sollte üblicherweise manuell über ein **Infrastructure Automation** System angestoßen werden.

## Fragen

- ② Welche beiden Actor-Systeme steuern beim DevOps Pipeline Pattern den automatisierten Integrations- und Installations-Prozess?



Beim Software Release Management werden Releases von Software-Produkten über 7 Dimensionen gesteuert, welche über ein wohl-definiertes Version Schema das Release identifizieren.

Bei der Dimension **Evolution Stages** geht es um die Art und Ausbaustufe des Produkts. Man unterscheidet zwischen den (teilweise noch nicht auf der späteren Technologie basierenden) Vor-Stufen **Wireframe**, **Prototype** und **Proof of Concept** und den (auf der späteren Ziel-Technologie basierenden) produktionsreifen Stufen **Walking Skeleton**, **Minimum-Viable Product** und **Full Product**.

Bei der Dimension **Version Number** wird das Produkt über drei Zahlen identifiziert: **Major Version**, **Minor Version** und **Revision**. Die ersten beiden beziehen sich inhaltlich auf das Produkt. Letztere bezieht sich auf den jeweiligen **Maturity Level** des Produkts. Die Dimension **Maturity Levels** selbst legt die Reife des Produkts innerhalb **Major/Minor Version** fest: **Alpha** (a, unvollständig, instabil), **Beta** (b, vollständig, instabil), **Release Candidate** (rc, vollständig, stabil), **Release** und **Release Update**.

Die Dimension **Points-In-Time** sagt aus, ob das aktuelle Release den Stand **Development** besitzt (so wie das Produkt im VCS liegt), ob es sich um einen speziellen **Snapshot** handelt (z.B. für extrem zeitkritische Hotfixes oder Early Feedbacks) oder ob es sich um eine ganz normale **Release** handelt.

Die Dimension **Product Editions** legt die Ausgabe bzw. Variante des Produkts fest: hier wird üblicherweise **Community/Enterprise Edition** (mit dem Fokus auf den Zielmarkt) oder **Standard/Professional Edition** (mit dem Fokus auf den Funktionsumfang) unterschieden.

Die Dimension **Availability Scopes** legt fest, für wen das Release des Produkts verfügbar ist: **No Availability** (nur für den Hersteller intern), **Limited Availability** (für Spezialsituationen), **Early Availability** (für "Early Adopters" unter den Kunden) und **General Availability** (für alle Kunden). Üblicherweise werden die Availability Scopes für konkrete **Maturity Level** genutzt, es besteht aber kein notwendiger Zusammenhang.

Die Dimension **Distribution Channels** legt fest, über welche Kanäle ein Release verfügbar ist: **Bleed Channel** (für "Visionaries" und "Die Hards"), **Edge Channel** (für "Early Adopters"), **Stable Channel** und **Solid Channel** (für alle Kunden). In jedem Fall sollten die **Distribution Channels** direkt mit den Branches des VCS gekoppelt sein.

## Fragen

- ?
- Bei welchem **Maturity Level** im Software Release Management hat man eine noch unvollständige und instabile Funktionalität?
- Bei welchem **Maturity Level** im Software Release Management hat man eine bereits vollständige aber üblicherweise noch instabile Funktionalität?