Whitepaper | Juni 2021

DIGITALE ÖKOSYSTEME

Technische Exzellenz von digitalen Plattformen

PAWLIK Digital

Blicken wir einmal zehn Jahre zurück. Im Jahre 2011 waren unter den zehn wertvollsten Unternehmen weltweit hauptsächlich Ölfirmen. Eine Schlagzeile der FAZ lautete damals: „Apple überholt Exxon Mobile als wertvollstes Unternehmen der Welt“[1]. Schon zu dieser Zeit gab es erste Anzeichen, dass Technologie- und Plattformunternehmen die Zukunft gehört.

Heute, im Jahr 2021, sind sagenhafte acht der zehn wertvollsten Unternehmen Plattformunternehmen.[2] Es gibt mittlerweile sogar neben Dow Jones und Nasdaq einen Plattform-Index, der Jahr für Jahr immer weiter ansteigt.

Doch wie definiert sich eine digitale Plattform?

Welchen technischen Herausforderungen stehen digitale Plattformen in digitalen Ökosystemen gegenüber?

Und wie schaffen es digitale Plattformen diese zu bewältigen?

[1] *FAZ, 2011*
[2] *Finanzen 100 (Heinweis: gemessen an der Marktkapitalisierung)*

Titelbild: (c)istock_metamorworks

WAS SIND DIGITALE PLATTFORMEN IN DIGITALEN ÖKOSYSTEMEN?

Digitale Plattformen (DP) sind digitale, kundenzentrierte Netzwerke, die eine direkte Interaktion von unterschiedlichen Nutzergruppen (z. B. Anbieter und Nachfrager) ermöglichen.

Beispielsweise können auf der Plattform eBay Käufer gleichzeitig als Verkäufer agieren und auf direktem Wege ihre angebotene Ware an weitere Kunden verkaufen, die wiederum auch eigene Produkte auf der Plattform verkaufen können.

Der Erfolg einer jeden Plattform wird dabei insbesondere von der Nutzerpartizipation (Netzwerkeffekte) bestimmt. Dies bedeutet: Je mehr Akteure auf einer DP aktiv sind, desto attraktiver ist diese für alle Teilnehmer des Ökosystems. Daher ist ein hochrangiges Ziel, eine möglichst hohe Anzahl an Plattformanwendern aufzubauen, um gleichzeitig den Wert der DP in einem digitalen Ökosystem zu steigern.

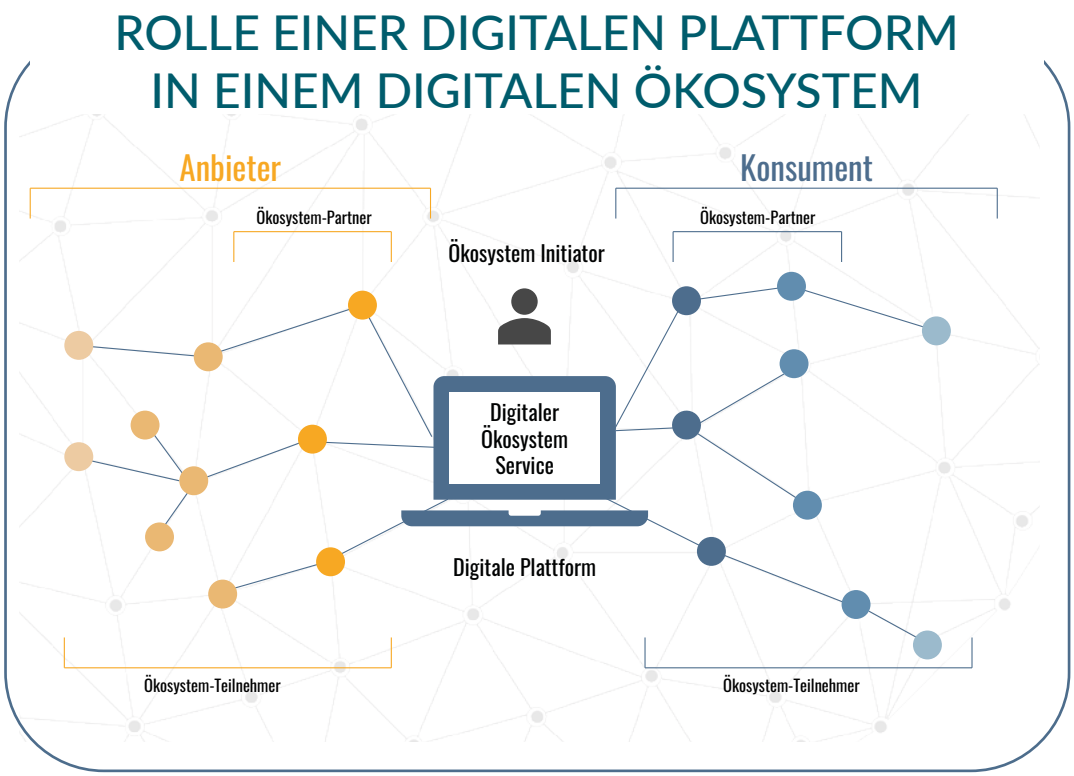
BEISPIELE FÜR UNTERSCHIEDLICHE TYPEN VON DIGITALEN PLATTFORMEN

Quelle: Tias, 2019

Digitale Ökosysteme gehen dabei noch einen Schritt weiter. Sie umfassen nicht nur eine DP, über die Waren, Dienstleistungen oder Daten ausgetauscht werden, sondern alle Software- und Hardwarelösungen sowie Akteure (u. a. Anwender oder Partner), die mit der DP im Zusammenhang stehen. Der wirtschaftliche Vorteil entsteht dadurch, dass die unterschiedlichen Parteien über ein Medium (der Plattform) verbunden werden.

Amazon nutzt beispielsweise den eigenen Online-Marktplatz dazu, eigene Produkte wie Echo-Dots, Kindle-Tablets, -Apps oder Lebensmittellieferungen ihren Kunden anzubieten und diese damit näher an den Konzern zu binden, um das Imperium durch zusätzliche Ertragskanäle zu erweitern. Sogar die kanalübergreifende Erweiterung um physische Präsenz in Form von Supermärkten, konnte Amazon mittlerweile bewältigen.[3]

[3] *Handelsblatt, 2021*



Quelle: TME Research 2021 - Darstellung in Anlehung an Frauenhofer IESE

Der Betreiber eines digitalen Ökosystems ist gleichzeitig **Ökosystem Initiator** und Besitzer der zentralen DP, die wiederum einen **digitalen Ökosystem Service** (z. B. Kommunikationsdienst bei WhatsApp, Vermittlung von Unterkünften bei AirBnb) anbietet. Der Ökosystem Service stiftet den eigentlichen Nutzen, der die **Ökosystem-Teilnehmer** hauptsächlich zur Partizipation an einem Ökosystem motiviert. Die Ökosystem Services werden in der Regel über eine Web- oder Mobile-Applikation zur Verfügung gestellt. Darüber hinaus können DPs direkte Kooperationen mit **Ökosystem-Partnern** eingehen, die entweder zu einer direkten Erweiterung des Ökosystems durch die Bereitstellung eigener Services beisteuern, oder die Services der DP auf eine andere (meist kommerzielle) Art und Weise nutzen.

Solche Partnerschaften werden selbstverständlich mit vorab definierten Servicevereinbarung geschlossen.

In klassischen digitalen Ökosystemen werden auf einer DP die **Anbieter** und **Konsumenten** eines Service vereint.

Ein Beispiel dafür ist die Plattform AirBnb, die Kurzzeitvermieter und -Mieter von Unterkünften auf ihrer DP vereint.

Allerdings gibt es auch noch die Konstellation, dass alle Teilnehmer die gleiche Teilnehmergruppe darstellen wie z. B. Nutzer des WhatsApp-Kommunikationsdienstes.

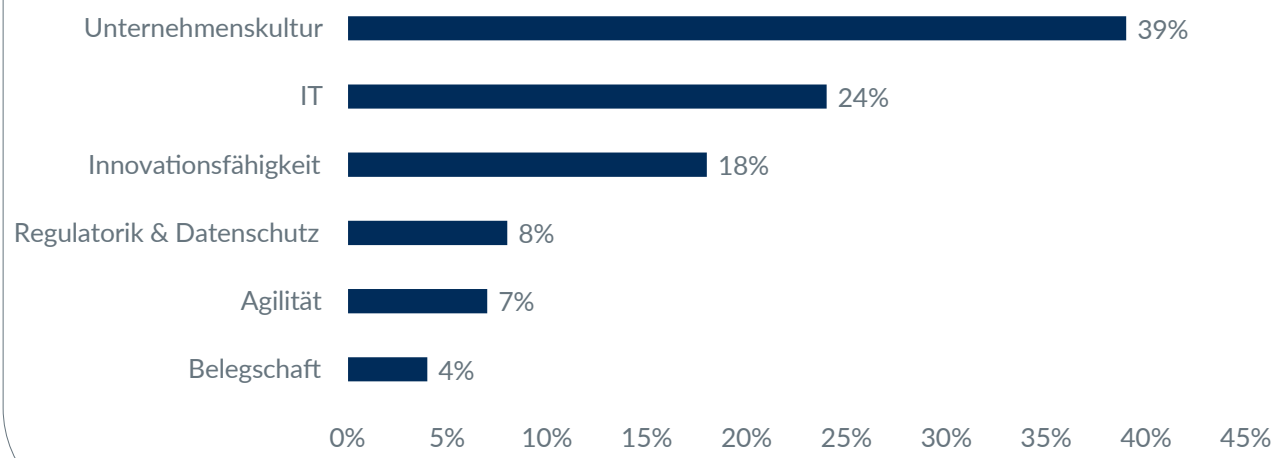
In eher selteneren Fällen gibt es mehr als zwei Teilnehmergruppen, wie beispielsweise in dem Ökosystem der Lieferdienst-Plattform Deliveroo, bei der selbstständige Lieferanten, Restaurants und Kunden vereint werden.

Neben dem eigentlichen Ökosystem-Service kann die Nutzung von Daten und anderen Services ebenfalls ein wesentlicher Grund für die Teilnahme und direkte Anbindung an die Schnittstellen einer DP sein. Dabei ist es üblich, dass mehrere Ökosystempartner gegenseitige Services und Daten miteinander austauschen, um Synergieeffekte zu schaffen.

Die Basis für die Funktionsfähigkeit und eine erfolgreiche Einbindung von DPs in digitale Ökosysteme stellt das Operationsmodell mit den verschiedenen technischen Komponenten dar. Unserer Umfrage aus dem Whitepaper ***„Digitale Ökosysteme – das größte Hindernis ist die Unternehmenskultur“*** zufolge, ist neben der Unternehmenskultur die IT eine der größten Hürden bei dem Aufbau von digitalen Ökosystemen.

UMFRAGE DER TME

Wo sehen Sie die Haupthemmnisse beim Aufbau eines Digitalen Ökosystems in Ihrer Bank?



200 Teilnehmerinnen und Teilnehmer, 2018

Unter der **technischen Exzellenz** von DPs werden nicht nur die klassischen, technischen IT-Ressourcen wie z. B. Software oder Hardware verstanden, sondern ebenso die damit einhergehenden Prozesse und Menschen, über die ein Mehrwert generiert wird.

Unsere Umfrage ergab ferner, dass neben der Unternehmenskultur und der IT, auch die Innovationsfähigkeit und Agilität, d. h. eine schnelle Anpassung an äußere Marktgegebenheiten, eine große Herausforderung für DPs darstellen. Alle oben genannten Dimensionen sind demnach potenzielle Hemmnisse zum Aufbau einer technischen Exzellenz von DPs, da sie einen direkten oder indirekten Einfluss auf die Prozesse, Menschen und Systeme haben, die für eine technische Exzellenz verantwortlich sind.

Aus diesem Grund haben wir uns das Ziel gesetzt, aufzuzeigen, welche technischen Voraussetzungen DPs in digitalen Ökosystemen erfüllen sollten. In diesem Whitepaper wird daher näher auf die Aspekte des technischen Designs und der Arbeitsprozesse eingegangen, die essenziell beim Aufbau und Operationalisierung von DPs sind.

INHALTE

1

APIs ALS GRUNDVORAUSSETZUNG FÜR DIE EXISTENZ VON DIGITALEN ÖKOSYSTEMEN

Technische Herausforderungen an digitale Plattformen

Unternehmen, die eine DP aufbauen oder betreiben möchten, können mehreren technischen Herausforderungen gegenüberstehen, um ihre Services anbieten zu können. Dabei steht im Mittelpunkt die **Service-Verfügbarkeit**. Funktionen und Produkte von DPs müssen zuverlässig, möglichst ohne Unterbrechung, selbst bei Service-Anpassungen oder unter erschwerten Bedingungen (z. B. erhöhten Aufrufzahlen), nutzbar sein. Darüber hinaus sollten digitale Plattformen einfach und schnell erweiterbar sein und somit Flexibilität und Skalierbarkeit mitbringen.

Um diese Herausforderungen meistern zu können, benötigen DPs ein anpassungsfähiges, skalierbares und zuverlässiges technisches Design sowie die dazu passenden Arbeitsprozesse und -methoden für die Entwicklung und Auslieferungen ihrer Services. DPs müssen demnach eine ausgereifte technische Exzellenz aufweisen. In diesem Kontext greifen insbesondere die Themen **API-Management, Microservice-Architektur und DevOps**. Was sich hinter diesen Begriffen verbirgt und warum sie essenziell für den Aufbau und Betrieb von DPs sind, wird im weiteren Verlauf dieses Whitepapers näher erläutert.

Als erste technische Hürde muss eine DP für die allgemeine Verfügbarkeit ihrer Services sorgen. Dabei muss zunächst das technische Grundkonzept betrachten werden, das die Existenz von digitalen Ökosystemen und die Interaktion der Ökosystem-Teilnehmer überhaupt ermöglicht. Im Vordergrund der technischen Komponente jeder modernen DP stehen ihre Schnittstellen (APIs), die die wichtigste Rolle in der Verbindung von Akteuren eines digitalen Ökosystems spielen.

API steht für *Application Programming Interface* und bezeichnet im Allgemeinen eine Programmierschnittstelle, die eine Interaktion zwischen verschiedenen Anwendungen realisieren kann. Genauer gesagt, ermöglichen APIs einen harmonisierten und einheitlichen Zugriff auf die Daten einer Anwendung und stellen ferner die Services und Funktionalitäten einer Anwendung zur Verfügung. Funktionen oder Services, die über APIs bereitgestellt werden können, sind beispielsweise der Abruf von Wettervorhersagen über eine App auf dem Smartphone, die Authentifizierung eines Benutzerkontos auf einer Webseite oder die Integration eines Zahlungsanbieters zum Abschluss eines Online-Einkaufs. APIs stellen also den Schlüssel dar, der den Ökosystem-Teilnehmern die Nutzung einer DP sowie die Interaktion der Teilnehmer in einem Ökosystem ermöglicht.

Im Wesentlichen gibt es drei unterschiedliche Typen von APIs, an denen sich auch die verschiedenen Einsatzmöglichkeiten orientieren.

PRIVATE API

Private APIs sind ausschließlich für den Daten- und Serviceaustausch innerhalb einer Organisation geeignet. Typische Anwendungsbeispiele hierfür sind die Integration einer neuen Unternehmenssoftware oder die Kommunikation und Kollaboration zwischen voneinander losgelösten Anwendungen.

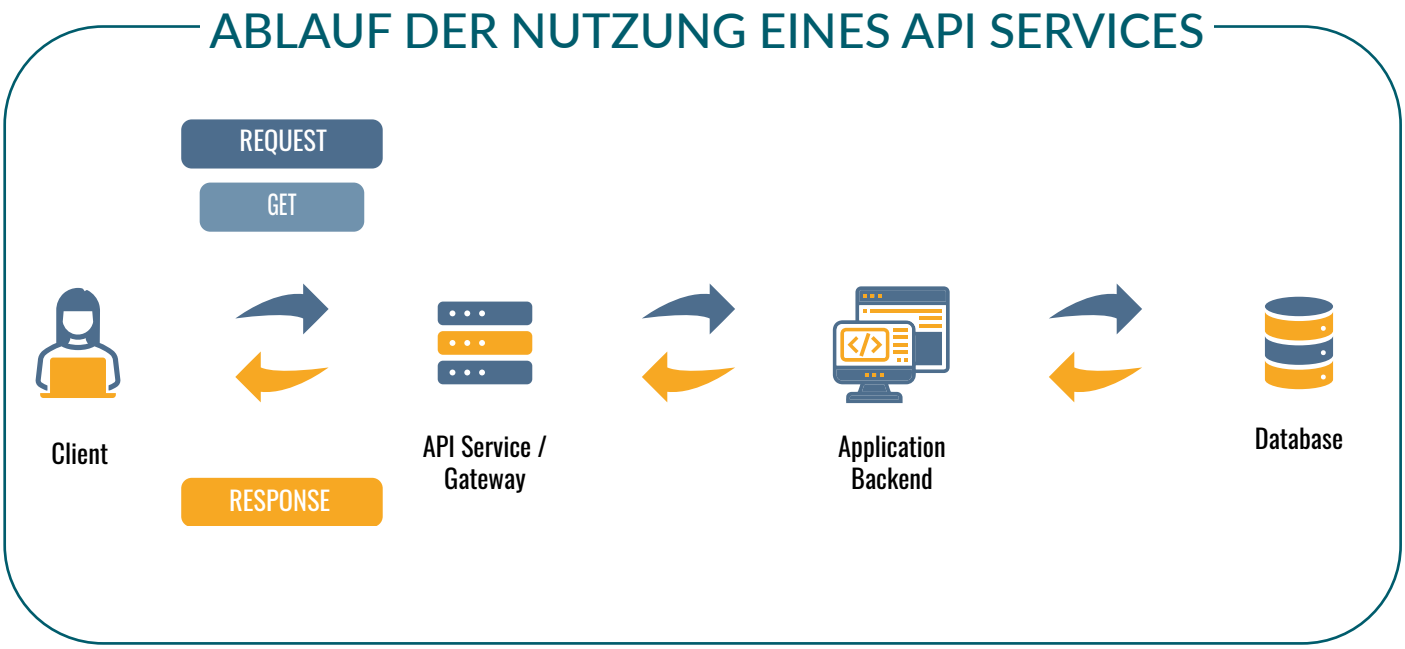
PARTNER API

Partner APIs befähigen DPs dazu, dass sie ihre Services und Daten über Schnittstellen für ausgewählte Partnersysteme zugänglich machen. Klassische Anwendungsfälle sind die Softwareintegration zwischen zwei Plattformen sowie die kommerzialisierte und unternehmensübergreifende Nutzung bestimmter Services.

ÖFFENTLICHE API

Öffentliche APIs können für alle Teilnehmer eines digitalen Ökosystems zugänglich sein und treten meistens in Form von Web-APIs auf, die in der Regel direkt durch Endanwender konsumierbar sind.

Ein vereinfachter Ablauf der Verwendung einer API ist in der nachfolgenden Grafik veranschaulicht.



Quelle: TME Research, 2021

Die Besucherin einer Webseite ruft über ihren Browser (Client) den Service einer DP auf. Eine gängige Service-Abfrage ist beispielsweise die Anzeige der Verfügbarkeit eines bestimmten Artikels in einem Onlineshop. Die Webseitenbesucherin startet die Abfrage (Request) des API Services bzw. API Gateway[4] über einen GET-Befehl (GET = Abfrage von Informationen). Dieser Befehl wird an die entsprechende Anwendung weitergeleitet, die für die Bereitstellung der benötigten Daten verantwortlich ist. Die Information über die Verfügbarkeit des Artikels bekommt die Webseitenbesucherin über eine Antwort (Response), welche in lesbarer Form auf der Webseite angezeigt wird. Der gesamte Service-Aufruf findet im Hintergrund statt und ist daher nicht direkt wahrnehmbar für die Webseitenbesucherin. Weitere Informationen über die Rolle von APIs in digitalen Ökosystemen und die genauere Funktionsweise von APIs können Sie in dem PAWLIK Digital Whitepaper *„API-Struktur ein Schlüssel zum Erfolg?“* finden.

[4] *API Gateway sitzt zwischen einem Client und den Backend-Services und koordiniert die API-Aufrufe und -Antworten. Das bedeutet, dass das API Gateway Aufrufbefehle (GET, POST, PUT, PATCH, DELETE, OPTIONS) entgegennimmt, diese an die entsprechenden Backend Services weiterleitet und dafür sorgt, dass die Antwort an den Client übermittelt wird. Darüber hinaus bieten API Gateways weitere Vorteile wie z. B. Authentifizierung, Durchsatzratenbegrenzung sowie Analyse- und Überwachungszwecke.*

PAWLIK DIGITAL EXPERTENTIPP 1: SPEZIFIKATION VON APIs

Es ist empfehlenswert, alle Arten von APIs, ob private, partnerspezifische oder öffentliche, mit einer API-Spezifikation in einem gängigen Format (z. B. Swagger) zu erstellen und diese Spezifikation auf einem API-Portal den Teilnehmern des Ökosystems verfügbar zu machen. Darüber hinaus kann das API-Portal gleichzeitig als Testumgebung für die Nutzung von APIs fungieren.

Es existieren Implementierungsansätze (*Code-First*), mit denen alle Versionen einer API automatisiert in einer API-Spezifikation dokumentiert sowie die API-Portale entsprechend kontinuierlich erweitert werden. Im Gegensatz zu dem sogenannten *API-First*-Ansatz, bei dem die API-Spezifikation vor der Implementierung entworfen wird, können Anbieter von APIs über den *Code-First*-Ansatz üblicherweise Zeit für die API-Dokumentation sparen und sicherstellen, dass die API-Spezifikation stets auf dem aktuellen Stand ist.

Auch eine Kombination der *API-First*- und *Code-First*-Ansätze ist denkbar, indem eine vorläufige API-Spezifikation für die Entwicklung entworfen wird und die endgültige API-Spezifikation automatisch im API-Portal generiert wird, sobald die Entwicklung abgeschlossen ist.

2

SERVICE-ÄNDERUNGEN RICHTIG BEWÄLTIGEN DURCH EIN API-LEBENSZYKLUSMANAGEMENT

DPs können ihre APIs für Partner oder die allgemeine Öffentlichkeit zugänglich machen, um ihre Services zu kommerzialisieren oder das allgemeine Serviceangebot ihrer Plattform auszubauen. Doch die öffentliche Verfügbarkeit von DP Services über APIs bringt auch eine große Verantwortung mit sich. Denn Partnersysteme, die sich über eine sogenannte Partner API anbinden, begeben sich in eine Abhängigkeit von der Verfügbarkeit dieser Services. Sollte eine API nicht aufrufbar sein, können geschäfts- und reputationsschädigende Konsequenzen folgen – sowohl für den Anbieter als auch für den Nutzer einer API. Wenn beispielsweise der Bezahlservice eines Onlineshops nicht aufrufbar ist, kann ein Kaufvorgang nicht abgeschossen werden und der Onlineshop verliert möglicherweise einen Kunden. Wenn man diese Situation auf mehrere tausende potenzielle Käufer hochskaliert, dann entgehen dem Onlineshop nicht nur zahlreiche Umsätze, sondern die DP kann darüber hinaus Einbußen in Reputation und Kundenzufriedenheit erleiden. Daher ist es wichtig, dass APIs trotz der Service-Anpassungen verfügbar sind sowie entsprechende Änderungen öffentlich kenntlich gemacht werden.

Insbesondere bei kommerzialisierten Partner APIs ist es wichtig, dass Anbieter von APIs über sogenannte Service-Vereinbarungen (SLAs) garantieren, eine gewisse Service-Kontinuität sicherzustellen. SLAs definieren dabei konkret, welche Service-Versprechen über die Nutzbarkeit und Verfügbarkeit einer API bestehen. API-Anbieter treffen dabei üblicherweise bestimmte Vorkehrungen, um die Vereinbarungen der SLAs einhalten zu können. Unter diesen Vorkehrungen zählen unter anderem: die Bekanntmachung und Kommunikation der APIs, die Kenntlichmachung der Änderungen sowie die konzeptgerechte Versionierung und Abschreibung der APIs.

Die Versionierung für APIs ist sehr wichtig für die Erhaltung der Service-Kontinuität einer DP. Wenn größere Änderungen (sogenannte *breaking changes*) an einer API durchgeführt werden, kann es passieren, dass Partner die API nicht mehr aufrufen und somit den Service nicht mehr nutzen können. Daher ist es notwendig, dass DPs *breaking changes* über eine entsprechende Versionierung kenntlich machen. Ein bewährter Ansatz ist die Verwendung einer dreistelligen Versionsnummer, anhand der die Kritikalität einer Änderung identifiziert werden kann.

EXEMPLARISCHE KONVENTIONEN ZUR API-VERSIONIERUNG

Quelle: TME Research, 2021

Die erste Zahl gibt an, ob es sich bei einer Änderung um einen *breaking change* handelt. Eine solche Anpassung kann z. B. das Hinzufügen eines Abfrageparameters sein. Ohne die Berücksichtigung dieser neuen Variablen fehlen obligatorische Daten und der Aufruf der API ist nicht mehr möglich. Änderungen an der zweiten Stelle der Versionsnummer betreffen hingegen kleinere Anpassungen, wie z. B. das Hinzufügen einer neuen nicht obligatorischen Funktionalität. Diese Änderung führt im Gegensatz zu *breaking changes* nicht zu der Gefahr, dass eine API anschließend nicht mehr aufrufbar ist (Protokollbruch). Wenn ein Partner diese Anpassung nicht berücksichtigt, kann die neue Funktionalität lediglich nicht verwendet werden. Die dritte Stelle der Versionsnummer kennzeichnet ausschließlich eine Änderung, die keinen größeren Einfluss auf die eigentliche Nutzung eines Service hat (z. B. Fehlerbehebungen).

Mit einer Analogie lässt sich die allgemeine Funktionsweise von API sowie der Versionierung leichter verstehen. Nehmen Sie an, die API ist eine Steckdose nach dem europäischen Steckdosenstandard und der Service-Austausch stellt den Stromfluss dar. Sobald der Elektriker die Steckdose um eine USB-Lademöglichkeit erweitert, können Elektronikgeräte, die an der Steckdose angeschlossen sind, ohne Anpassungen weitergenutzt werden. In diesem Fall handelt es sich um eine Änderung der Minor-Version durch das Hinzufügen einer zusätzlichen Funktionalität, welche die aktuelle Nutzungsweise nicht beeinflusst. Nun entscheidet sich der Elektriker, die Steckdose vollständig durch eine US-amerikanische Steckdose auszutauschen. Die Elektronikgeräte, die für den europäischen Standard ausgelegt sind, können somit ohne eine entsprechende Anpassung (z. B. einen Adapter) keinen Strom mehr über die neue Steckdose erhalten. In diesem Fall handelt es sich um eine Änderung der Major-Version.

Partner, die sich an eine DP über eine API anbinden, können die Version einer API in der Regel in der URL, die für den Aufruf der API verantwortlich ist sowie in der API-Spezifikation erkennen. Meistens wird dabei die Konvention verwendet, dass nur die Major-Versionsnummer angezeigt wird, z. B. v1, v2, v3. Eine weitere Möglichkeit, einer API die Versionsnummer mitzuteilen, ist die Nutzung des sogenannten Header-Parameters des HTTP(S)-Aufrufs, der in der URL nicht sichtbar ist.

Einhergehend mit der Versionierung kann eine API verschiedene Phasen des Lebenszyklus durchlaufen und dabei stufenweise bis hin zur vollständigen Außerbetriebsetzung abgeschrieben werden. Die gängigsten Phasen der API-Versionen sind:

AKTUELLE API (CURRENT)

Aktuelle und vollständige API-Version, welche zur Nutzung empfohlen wird.

VERALTETE API (DEPRECATED)

Eine API-Version, die bereits von einer neueren Version abgelöst wurde. Diese Art von API wird nur noch eine gewisse Zeit unterstützt und es erfolgen keine Neuerungen mehr, sondern nur noch Fehlerbehebungen. Außerdem kann einer neuen Anwendung der Zugriff auf veraltete APIs verweigert werden. Wenn der Zugriff auf eine veraltete API vom API-Anbieter nicht verweigert wird, dann wird in der Regel eine Abschaltungsbenachrichtigung (*deprecation notification*) als Teil der Antwort (im Response-Header) mitgeliefert. Nach einer gewissen Zeit (z. B. sechs Monate), wird eine veraltete API als ausgeschieden oder stillgelegt deklariert.

AUSGESCHIEDENE API (RETIRED)

Eine API-Version ist ausgeschieden, wenn sie nicht mehr unterstützt wird. Anwendungen, die ausgeschiedene APIs verwenden, müssen auf eine aktive API-Version umsteigen. Dennoch sind ausgeschiedene APIs in der Produktion vorübergehend noch verfügbar. Nach einer gewissen Zeit (z. B. drei Monate), wird eine ausgeschiedene API stillgelegt.

STILLGELEGTE API (DECOMMISSIONED)

Eine API-Version, die nicht mehr in der Produktion verfügbar ist.

Neben den oben beschriebenen Versionsphasen, die in absteigender Reihenfolge der endgültigen Außerbetriebsetzung sukzessive näherkommen, gibt es eine weitere Form einer API-Version – die sogenannte zukünftige Version (*future*). Diese kündigt die baldige Einführung einer neuen API-Version an, welche bereits als Beta-Testversion verfügbar ist. Diese Version stellt noch eine instabile Form einer API dar. Diese Art der Versionierung eignet sich gut für Anbieter von APIs, um baldige API-Versionen vor dem tatsächlichen Release ordentlich testen zu können. Darüber hinaus ist es wichtig festzulegen und bekanntzugeben, wie lange sich die API-Version in der jeweiligen Phase befindet bzw. wann sie ihren Status ändern wird.

Änderungen an APIs sollten stets an Partner kommuniziert werden. Am besten eignet sich die Bereitstellung einer API-Spezifikation samt der ausführlichen Änderungshistorie in einem API-Portal. Darüber hinaus sollten API-Anbieter ihren Partnern eine Abonnement-Möglichkeit (z. B. E-Mail-Newsletter) bereitstellen, so dass sie über jegliche Neuerungen und Anpassungen über den gesamten Lebenszyklus einer API-Version informiert bleiben.

PAWLIK DIGITAL EXPERTENTIPP 2: ÄNDERUNGSVERWALTUNG VON APIs

Bei Änderungen der Signaturen (verschiedene Informationen, die relevant für den API-Aufruf sind, z. B. Input-Parameter) von APIs sollte entweder die Major-Version (z. B. von v1.5 auf v2.0) oder die Minor-Version (z. B. von v1.5 auf v1.6) erhöht werden. Letzteres betrifft nur das Hinzufügen von neuen, nicht obligatorischen Aufrufparametern in der Signatur. Ältere API-Versionen sollten nach Möglichkeit für das Ende der Lebensdauer vorbereitet werden und entsprechend als *Deprecated* (veraltet) kenntlich gemacht werden. Das Außerbetriebsetzungsdatum kann je nach API-Endpoints entsprechend den Geschäftsanforderungen und/oder Support-Intervallen definiert werden. Neue, bisher noch nicht-existierende API-Endpoints (z. B. URL für den Aufruf) können die jeweils letzte aktuell genutzte Versionsnummer zugewiesen bekommen.

SUCCESS STORY 1: SOLARISBANK AG

Die Solarisbank AG ist ein deutsches Kreditinstitut, das eine Banking-as-a-Service-Plattform anbietet. Unternehmen, die zwar keine Banklizenz haben, aber dennoch Finanzprodukte und -dienstleistungen anbieten wollen, können sich an die APIs der Solarisbank anbinden, um die notwendigen Services (z. B. Digital Banking, Debitkarten, Konsumentenkredit) zur Verfügung stellen zu können.[5] Um den kontinuierlichen Service anbieten zu können, bietet die Solarisbank eine umfangreiche API-Dokumentation[6] an, die sich auf die technischen API-Spezifikationen konzentriert und das Ziel verfolgt, alle Informationen zu liefern, die für die reibungslose Einrichtung und Nutzung der Produkte und Services benötigt werden. Dazu gehört u. a. die Beschreibung der API-Versionierung. Darüber hinaus stellt die Solarisbank eine Status-Informationsseite[7] zur Verfügung, auf welcher die Verfügbarkeit aller APIs sowie vergangene Störungen inklusive der einzelnen Behebungsschritte kommuniziert werden.

[5] *Solarisbank https://www.solarisbank.com/de/about/*
[6] *Solarisbank https://docs.solarisbank.com/*
[7] *Solarisbank https://status.solarisbank.de/*

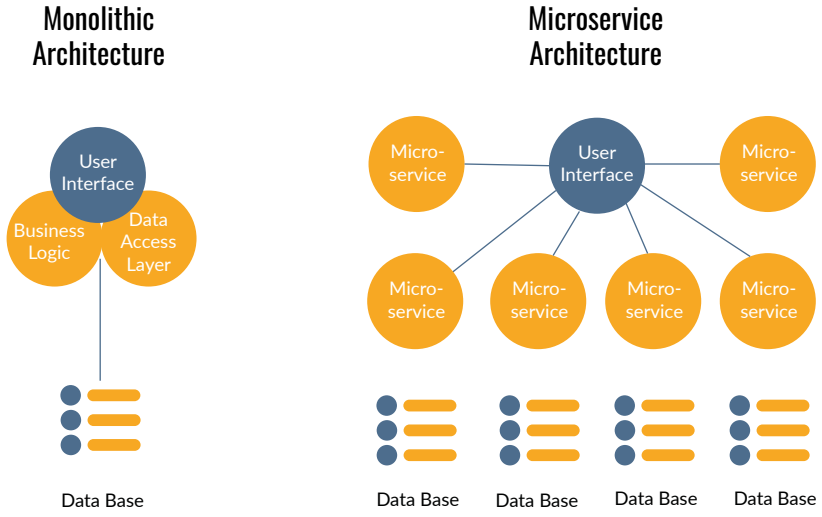
3

SKALIERUNG DER ARCHITEKTUR EINER DIGITALEN PLATTFORM DURCH MICROSERVICES

Neben dem Aspekt, dass DPs möglichst beständige APIs mit vorhersehbaren Versionsänderungen über das richtige Lebenszyklusmanagement ihrer APIs anstreben sollten, müssen sie die Verfügbarkeit ihrer Services auch unter erschwerten Bedingungen gewährleisten können. Diese Fähigkeit nennt sich **Skalierbarkeit**. Um als DP skalierbar zu sein, wird eine entsprechende zugrundliegende Architektur benötigt, die auch unter hoher Last leistungsfähig bleibt bzw. schnell und einfach erweitert werden kann. In diesem Kontext sind **Microservices-Architekturen** ein bewährtes Architekturmuster, welches das Pendant zu traditionellen monolithischen Architekturen darstellt.

Der Unterschied zwischen Microservices- und monolithischen Architekturen besteht im Wesentlichen darin, dass ein Monolith eine Vielzahl von unterschiedlichen Funktionalitäten in einem einzigen, fest verbundenen sowie homogenen System bündelt, welches in der Regel aus einer einzigen, gemeinsamen Codebasis besteht. Microservices-Architekturen besitzen hingegen **mehrere, verteilte Service-Komponenten**, die nach unterschiedlichen Funktionalitäten aufgeteilt sind und jeweils eine eigene Codebasis aufweisen. Dadurch können die einzelnen Microservices getrennt voneinander schnell und flexibel entwickelt, integriert, erweitert und gewartet werden. Dies ermöglicht eine **effizientere und einfachere Skalierung** von Microservices-Architekturen, da im Gegensatz zu Monolithen nicht die Gesamtarchitektur, sondern nur einzelne, von der hohen Auslastung betroffene Microservices skaliert werden können.

UNTERSCHIEDLICHER AUFBAU VON MONOLITHISCHEN UND MICROSERVICE-ARCHITEKTUREN[8]



Quelle: TME Research, 2021 Darstellung in Anlehnung an: divante, 2020

[8] *Hinweis zur Abbildung: Zur Abstraktion der Microservices-Architektur wird in der Abbildung kein API Gateway zwischen User Interface (UI) und Microservices berücksichtigt. Ein API Gateway ist allerdings ein bewährtes Architekturmuster, das dazu dient, einzelne Services mit einer Schnittstelle nach außen zu abstrahieren.*

Der Ansatz für die Leistungsskalierung von Microservices durch die Replikation der Serviceinstanzen, meist basierend auf zusätzlicher Hardware (physische Server oder virtuelle Maschinen bzw. Container), nennt sich **horizontale Skalierung** (Scale Out). Diese Art der Skalierung ist üblicherweise unbegrenzt möglich. Die **vertikale Skalierung** (Scale Up) wird hingegen durch die Erhöhung der Rechenleistung bzw. der Speicherkapazität der technischen Infrastruktur (z. B. Server mit leistungsstärkerem Prozessor oder größerem Arbeitsspeicher bzw. Speichermedium) durchgeführt und ist daher nur in einem begrenzten Maße möglich.

Neben der flexiblen und einfachen Skalierbarkeit sind Microservices eigenständig, spezialisiert, agil, einfach bereitstellbar, technologisch unabhängig, wiederverwendbar sowie hochverfügbar. Trotz der zahlreichen Vorteile führen verteilte Microservices-Architekturen, die getrennt voneinander erweitert und angepasst werden können, zu einer zusätzlichen Herausforderung, die durch das **CAP-Theorem** beschrieben werden kann. Das CAP-Theorem besagt, dass in einem verteilten System die Eigenschaften **Konsistenz** (von replizierten Daten), **Verfügbarkeit** (= Antwortzeit von Systemen) und **Partitionstoleranz** (= Ausfalltoleranz von Rechnernetzen, wenn einzelne Komponenten ausfallen) nicht im gleichen Maße sichergestellt werden können. Lediglich zwei der Eigenschaften können bedient werden.

PAWLIK DIGITAL EXPERTENTIPP 3: AUFSETZEN VON MICROSERVICES

Microservices werden üblicherweise mit Hilfe von Technologien zur Containerisierung (z. B. Docker, AKS, Kubernetes) bereitgestellt, die ressourcensparend und effizient die Skalierung und damit die Service-Verfügbarkeit ermöglichen. In einigen Anwendungsszenarien können Microservices auf dem sogenannten Serverless Computing aufsetzen, durch die Anwendung von PaaS-Kapazitäten großer Cloud-Provider. Diese stellen die Orchestrierung der Service-Skalierung mit höchsten SLA-Anforderungen sicher (z. B. Azure Functions, Amazon Lambda).

Neben dem Vorteil der Skalierung, kann bei Microservices-Architekturen in der Regel das Risiko und die Orchestrierung von Code-Änderungen reduziert werden. Dies liegt darin begründet, dass nicht die gesamte Codebasis bzw. das Datenmodell des Systems (wie bei einem Monolithen) angepasst werden muss, sondern nur der Code der Services, die geändert werden müssen. Allerdings ist die Voraussetzung hierfür, dass die Änderungen innerhalb des Kontextes eines Microservices stattfinden und nicht über die Kontextgrenzen hinaus einen Einfluss auf andere Microservices haben. In diesem Zusammenhang können DPs auf eine zusätzliche Komplexitätskomponente stoßen, die bei der Einführung von Microservices-Architekturen einhergeht – und zwar die Zuteilung von lose gekoppelten Microservices auf einzelne, fachliche Unternehmensfunktionen. Bei dieser Herausforderung, den **richtigen fachlichen Schnitt** der gesamten Funktionalität zu finden, können DPs durch das sogenannte **Domain-Driven-Design (DDD)** unterstützt werden.

Domain-Driven-Design für den richtigen fachlichen Schnitt der Architektur in Domänen

DDD ist ein Ansatz für die Modellierung von komplexen Architekturen.[9] Dabei ist es essenziell, dass bei der Entwicklung von Anwendungen der Fokus nicht auf der Technologie liegt, sondern auf der **Fachlichkeit** der Geschäftslogik, die durch die entsprechenden Anwendungen unterstützt werden soll. Durch den Einsatz von DDD kann die Anwendungslogik in klar abgegrenzte fachliche Kontexte geschnitten werden – die sogenannten **Domänen**. Die aus dem DDD resultierende Abgrenzung der Funktionen in Domänen dient als Ausgangslage für die Implementierung von Microservices, da sich Microservices auf die Realisierung bestimmter Geschäftsfunktionen konzentrieren sollten. Innerhalb der durch Domänen abgegrenzten Microservices ist es möglich, unterschiedliche Technologie-Stacks (z. B. Programmiersprachen, Datenbanken, Betriebssysteme) zu verwenden, was wiederum einen effizienten und richtigen Einsatz von arbeitsunterstützenden Werkzeugen sicherstellt. Dadurch kann innerhalb jedes Microservices unabhängig und entsprechend der strategischen Ausrichtung der Domäne gearbeitet werden.

[9] *Eric Evans, 2003*

4

KOLLABORATION UND KOMMUNIKATION ZWISCHEN MICROSERVICES

Private APIs erlauben die Kommunikation und Zusammenarbeit zwischen verschiedenen Services in verteilten Microservices-Architekturen.

Durch die Aufteilung von Anwendungen in losgelöste Microservices ist es notwendig, einen effizienten und standardisierten Weg für die Kollaboration und Kommunikationen zwischen den einzelnen Domänen zu etablieren. **Private APIs** ermöglichen, dass einzelne Microservices einzelne Funktionen von anderen Anwendungen abrufen können. Dadurch können einzelne Microservices schlanker gehalten werden, da sie benötigte Funktionen nicht im eigenen Code festhalten müssen, sondern bei Bedarf von anderen Microservices anfordern können.

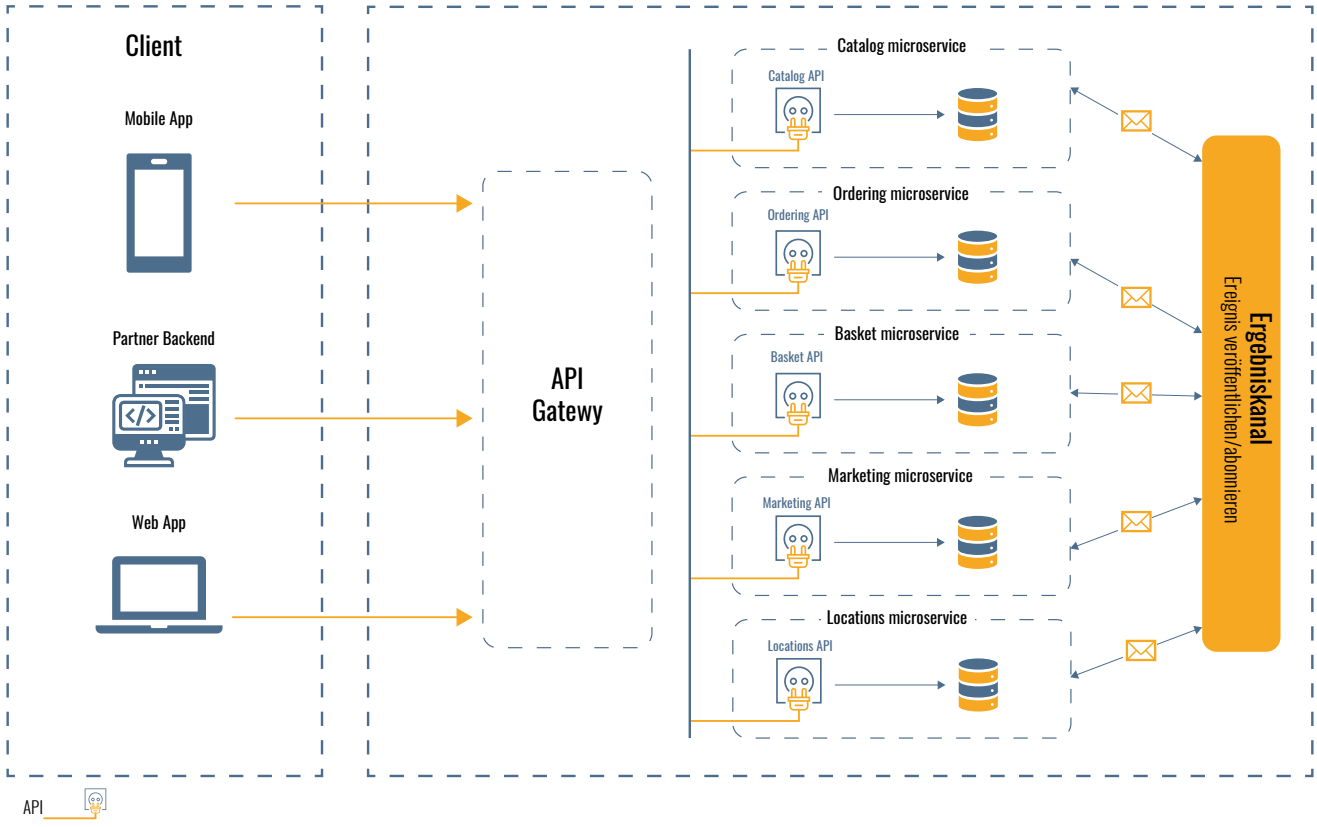
Bei der Event-Driven-Architecture wird das Zusammenspiel von Microservices über Ereignissen gesteuert

Neben den anfragegesteuerten APIs eignet sich das Prinzip der **ereignisgesteuerten Architektur** (Event-Driven-Architecture, kurz: EDA) für die Interaktionen von systeminternen Microservices. Wie der Name schon sagt, wird bei der EDA das Zusammenspiel von Microservices über Ereignisse (*Events*) gesteuert und nicht wie bei APIs über Anfragen. Ein Ereignis bezeichnet die Veränderung eines Systemzustands oder ein besonderes Geschehen innerhalb eines Systems. Beispielsweise kann das Hinzufügen eines Artikels in den Warenkorb eines Onlineshops als Ereignis bezeichnet werden. Ein Ereignis muss aber nicht zwingend aktiv durch die Aktion eines Anwenders ausgelöst werden, sondern auch automatisch durch bestimmte Veränderungen (z. B. Sensordaten).

In dem Konzept der EDA existieren **Produzenten** und **Konsumenten** von Ereignissen – beide Parteien können Microservices darstellen. Der Produzent erkennt ein Ereignis und stellt es über einen sogenannten **Ereigniskanal** (z. B. Event Bus) zur Verfügung. Konsumenten, die ein bestimmtes Ereignis abonniert haben, werden darüber informiert, sobald dieses eintritt. Wenn eine Kundin eines Onlineshops beispielsweise einen Artikel in den Warenkorb hinzufügt, dann könnte ein Ereignis ausgelöst werden. Der Warenkorb-Service erhält eine direkte Anweisung über seine API (anfragegesteuert), verändert die in seiner Domäne verankerte Datenbasis und leitet die Information über den veränderten Zustand des Warenkorbs an den Ereigniskanal weiter. Ein anderer Service, z. B. der Marketing-Service, wird unmittelbar über dieses Ereignis benachrichtigt und kann daraufhin seine eigene Geschäftslogik ausführen – z. B. der Kundin andere Produkte in der gleichen Produktkategorie vorschlagen.

Somit wird sichergestellt, dass die Interaktion zwischen den beiden internen Microservices lose durch den Ereigniskanal hergestellt ist. Das Lösen von direkten Abhängigkeiten zwischen Microservices dient der einfacheren Skalierung, besseren Wartbarkeit bzw. Veränderbarkeit und reduziertem Regressionsrisiko. Zusätzlich wird dadurch ein technischer und organisatorischer Ablauf geschaffen, damit die im CAP-Theorem beschriebenen Grundsätze für jeden Microservice gemäß den jeweiligen Anforderungen an Datenkonsistenz und Verfügbarkeit sichergestellt werden. Systeme, die EDA anwenden, können somit Entscheidungen in Echtzeit treffen und flexibler auf Veränderungen reagieren.

MICROSERVICES-ARCHITEKTUR UND DAS ZUSAMMENSPIEL DER MICROSERVICES ÜBER APIs UND EVENTS



Quelle: TME Research, 2021 Darstellung in Anlehnung an: Microsoft 2020

Das Architekturbild stellt oberflächlich das Zusammenspiel von APIs, Microservices und Ereignissen dar. Die Anfrage des *Client* wird über das API Gateway auf die entsprechenden Microservices weitergeleitet. Die Antwort erfolgt entgegengesetzt. Die Microservices können Ereignisse, die beispielsweise über den *Client* ausgelöst wurden, über den Ereigniskanal miteinander austauschen.

5

SCHNELLE UND SICHERE ENTWICKLUNG UND AUSLIEFERUNG DURCH DEVOPS

Im Zuge der Separierung der Geschäftsdomänen in Microservices wird die Arbeitsweise in isolierten und autonomen Teams ermöglicht. Diese Teams agieren in kleineren, klar abgegrenzten Kontexten eigenverantwortlich. Dies befähigt sie dazu unabhängig und effizient voneinander zu arbeiten, wodurch sich die Zykluszeiten für die Entwicklung und die Auslieferung von Services und Funktionen verkürzen. Die benötigten Denkweisen, Praktiken und Tools werden durch das sogenannte **DevOps** definiert.

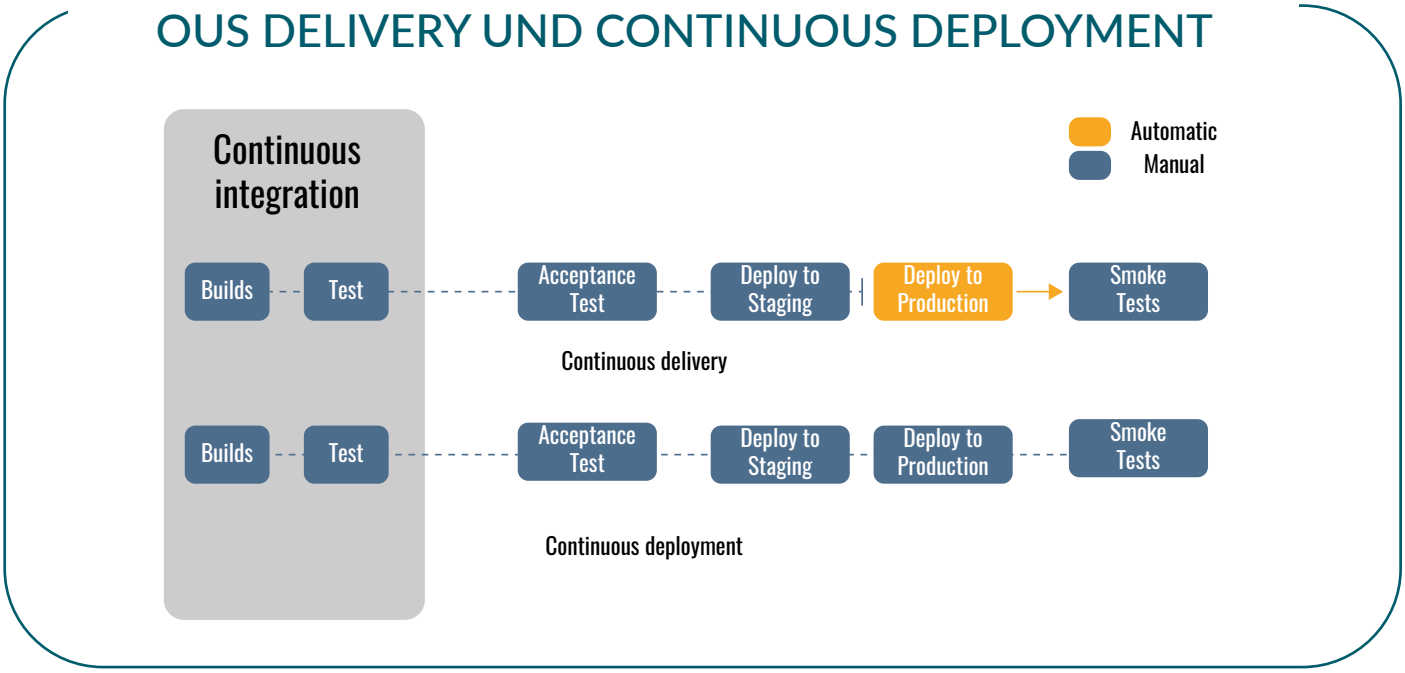
DevOps setzt sich aus den beiden englischen Begriffen *Development* (Entwicklung) und *Operations* (Betrieb) zusammen und beschreibt einen Ansatz aus der **agilen Softwareentwicklung**. Der Kerngedanke von DevOps ist, dass Entwickler und Betriebsverantwortliche (z. B. Systemadministratoren) eng in einem kleinen Team zusammenarbeiten und somit gemeinsam für den gesamten Anwendungslebenszyklus, von der Planung, Entwicklung, Qualitätssicherung bis zur Bereitstellung neuer Features verantwortlich sind.

Um effiziente Arbeitsprozesse innerhalb der Teams gewährleisten zu können, bedient sich DevOps an bewährten Praktiken zur **Automatisierung** sämtlicher Auslieferungsprozesse wie z. B. Testing. Dabei stellt die Code-Erstellung einen der wenigen manuellen Schritte dar. Ein möglichst hoher Automatisierungsgrad wird durch die Verwendung von verschiedenen Automatisierungstools ermöglicht. Durch den automatischen Ablauf der Prozesse werden für die jeweiligen Lebenszyklusphasen keine Expertenteams benötigt. Somit kann der Code durch Entwickler eigenständig bis in die Produktion mit nur wenigen *clicks* gebracht werden. Dies führt zu einer erheblichen Geschwindigkeitssteigerung und Flexibilität in der Bereitstellung neuer Funktionen und der Behebung von Fehlern in der Produktion.

Continuous Integration und **Continuous Delivery** (kurz: CI/CD) sind DevOps Methoden, die der Automatisierung von der Build-Erstellung[10] bis zum Deployment unterstützen. Dabei unterstützt Continuous Integration (CI) die Entwickler bei der regelmäßigen Zusammenführung von neuem oder geändertem Code in einem zentralen, gemeinsam genutzten Code-Repository. Nach jedem Build-Vorgang wird der Code automatisiert durch einfache **Unit Tests** (= Testen der funktionalen Einzelteile) getestet, so dass Fehler schneller identifiziert und behoben werden können. Ohne CI müssten Entwickler ihre jeweiligen Codeanpassungen manuell zusammenführen und hinsichtlich der Integrität in der jeweiligen Codebasis testen. Dies würde zu einem erheblichen zeitlichen Mehraufwand und einer hohen Komplexität führen. Eine Erweiterung zur CI ist die Continuous Delivery, die dafür sorgt, dass der Code in einer Testumgebung bereitgestellt und durch weitere Testverfahren (z. B. Integrations-, Performance- und Regressionstests) ausreichend geprüft wird, bevor die Auslieferung in die Produktionsumgebung erfolgt. Dadurch werden Entwickler bei der gründlichen Qualitätssicherung unterstützt. Eine zusätzliche Erweiterung von Continuous Delivery ist das **Continuous Deployment**. Die Abgrenzung zwischen den beiden Methoden ist, dass bei der Continuous Delivery eine manuelle Genehmigung vor der Bereitstellung des neuen Codes erfolgen muss. Bei dem Continuous Deployment wird der Code automatisch bis in die Produktionsumgebung bewegt, ohne dass eine manuelle Freigabe notwendig ist.

[10]*Der Build-Prozess beschreibt die Programmkompilierung, d. h. die Übersetzung von Quellcodes einer Programmiersprache in eine anwendbare Form, die von einem Computer gelesen und als Programm ausgeführt werden kann.*

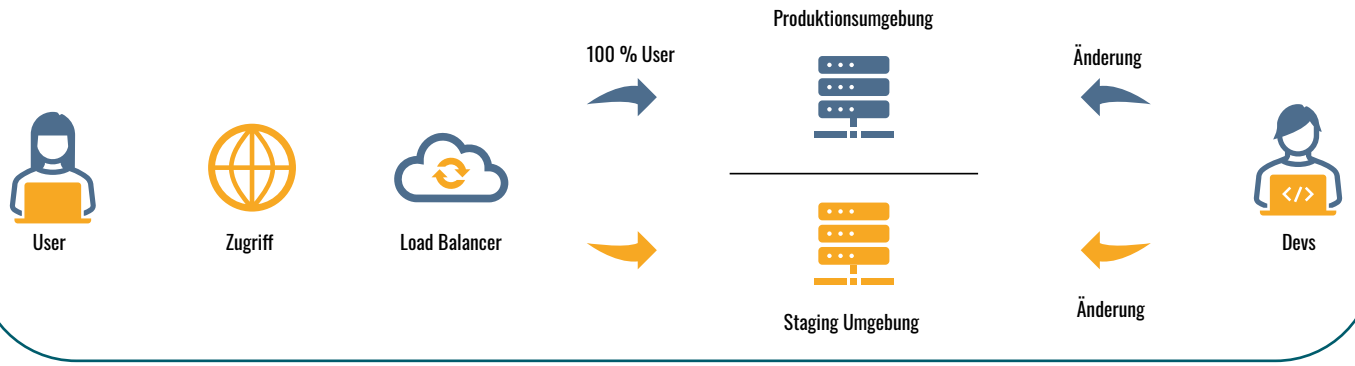
ABLAUF VON CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY UND CONTINUOUS DEPLOYMENT



Quelle: TME Research, 2021 Darstellung in Anlehnung an: MT AG, 2020

Im Kontext der sicheren Auslieferung und Bereitstellung von Code-Änderungen sowie der Sicherheitsstellung der Service-Verfügbarkeit bedient sich der DevOps-Ansatz an Deployment-Methoden wie dem **Blue-Green-Deployment**. Bei dem Blue-Green-Deployment gibt es mindestens zwei Server-Umgebungen – die **Produktions- und Staging-Umgebung**. Die Staging-Umgebung ist eine Instanz, die möglichst identisch zur Produktionsumgebung sein sollte. Sobald eine Änderung des Codes auf dem Produktionsserver durchgeführt wird, werden alle Anwender auf die Staging-Umgebung weitergeleitet, so dass die Services ohne Unterbrechung weiter genutzt werden können. In der Zwischenzeit erfolgt das Deployment der Code-Änderungen in der Produktionsumgebung. Sobald die Anpassungen durchgeführt sind, werden alle Anwender von der Staging- zurück auf die Produktionsumgebung umgeleitet.

BLUE-GREEN-DEPLOYMENT



Quelle: TME Research, 2021 Darstellung in Anlehnung an: Norbert Eder

SUCCESS STORY 2: MICROSERVICES UND DEVOPS BEI AMAZON

Eine Success-Story für die erfolgreiche Implementierung von Microservices-Architekturen und Anwendung von einem auf DevOps basierenden Continuous Delivery Modell ist Amazon. Der Onlinehandel-Gigant hat durch diese Ansätze 75 % weniger Ausfälle, 90 % weniger Ausfallminuten und schafft es auf über 7.000 Deployments pro Tag, was im Durchschnitt ein Deployment alle 11,6 Sekunden bedeutet.[11] Auch die Qualität der ausgelieferten Codes hat sich deutlich verbessert, so dass nur jedes tausendste Deployment fehlerbehaftet ist. Im Falle eines Fehlers in der Produktion kann dieser sehr schnell durch entsprechende Rollbackmechanismen behoben werden.

Neben CI/CD bedient sich DevOps an weiteren Methoden wie der **Versionskontrolle, Infrastruktur als Code und Continuous Monitoring**.

Versionskontrolle: Die Verwaltung von Code in verschiedenen Versionen wird Versionskontrolle bezeichnet. Durch die Aufzeichnung des Änderungsverlaufes kann die Überprüfung und Wiederherstellung des Codes vereinfacht werden. Die Anwendung der Versionskontrolle wird häufig durch eine kollaborative Versionsverwaltungssoftware unterstützt, die es ermöglicht, mehrere Entwickler gleichzeitig am Code arbeiten zu lassen.

Infrastruktur als Code: Infrastruktur in Form von Code beschreibt ein Konzept für eine programmierbare Infrastruktur, welches bei der Verwaltung von Entwicklungs-, Test- und Produktionsumgebungen zum Einsatz kommt. Dabei wird die Infrastruktur, z. B. Rechenleistung, Speicherkapazität und Netzwerk, basierend auf Code zur Verfügung gestellt und konfiguriert.

Continuous Monitoring: Fortlaufende Überwachung liefert Informationen über den Zustand eines Systems in Echtzeit. Unterstützt wird das Monitoring durch die Erfassung von Meta- und Telemetriedaten (z. B. Ereignisdaten, Protokolle) sowie die Definition von Warnungen bei der Abweichung von bestimmten Standardwerten. Ein Continuous Monitoring ist vor allem im Kontext der Automatisierung wichtig, um die Kontrolle über Prozesse zu behalten.

Insgesamt unterstützt der DevOps-Ansatz dabei, eine hohe Qualität der gelieferten Services sowie eine höhere Verfügbarkeit zu gewährleisten und somit die Einhaltung der Service-Vereinbarungen sicherzustellen. Darüber hinaus wird die generelle Auslieferung von neuen Funktionen und Services beschleunigt, wodurch DPs schneller auf neue Kundenbedürfnisse durch innovative Lösungen reagieren können.

EXPERTENTIPP 4: TESTUMGEBUNGEN FÜR PARTNER

Um agile Entwicklungszyklen bei Partnern zu unterstützen, sollten die Testumgebungen stets zur Verfügung stehen. Dies gilt insbesondere für die Durchführung von Regressions- und Schnittstellentests, die in der Regel automatisiert bei jedem Build-Prozess während der Continuous Integration durchgeführt werden.

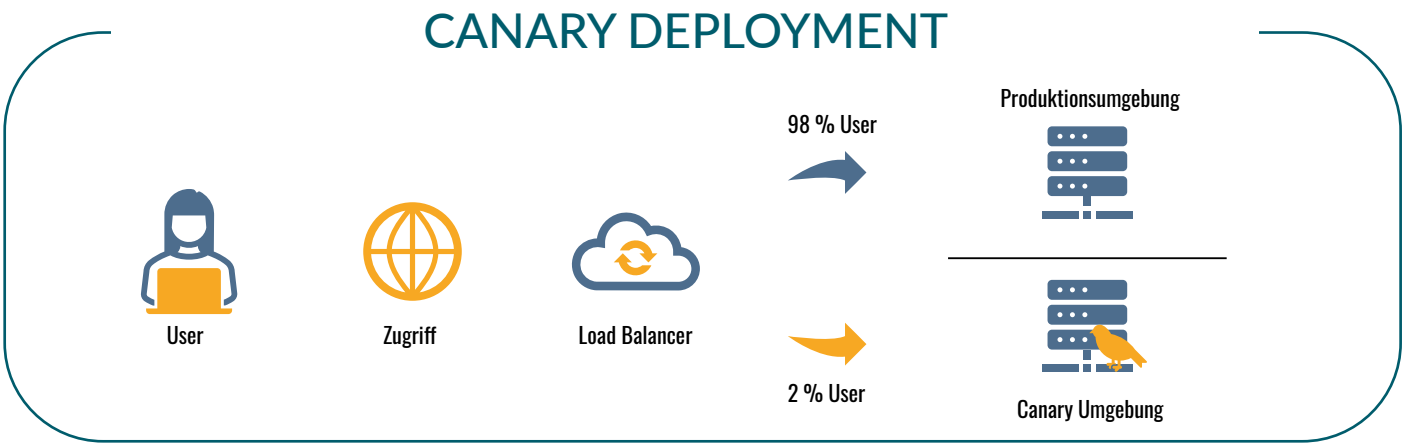
[11] *Comupterwoche, 2015*

6

AUSTESTEN INNOVATIVER FUNKTIONALITÄTEN DURCH CANARY DEPLOYMENTS

Durch Microservices und DevOps können DPs zusätzlich ihre Experimentierfreudigkeit und Innovativität fördern. Teams, die ihre Domäne am besten kennen, können innovative Ideen schnell (durch Automatisierung) und mit minimiertem Risiko (ausreichendes Testing, zuverlässiges Monitoring) austesten. Letzteres ist besonders wichtig, da Innovationen meistens mit einer gewissen Unsicherheit verbunden sind.

Daher eignen sich sogenannte **Canary Deployments**, um die Wirkung neuer Features auszutesten und somit das Risiko des Scheiterns einer Innovation reduzieren zu können. Bei Canary Deployments werden innovative Features nur für eine kleine Gruppe von Benutzern freigegeben. Dabei werden die Anwender über einen Load Balancer auf verschiedene Server-Umgebungen verteilt – entweder auf die Produktionsumgebung oder auf die Canary Umgebung. Beide Umgebungen sollten weitestgehend den gleichen Code aufweisen und sich nur durch die neuen, experimentellen Features unterscheiden, welche sich ausschließlich auf der Canary Umgebung befinden.



Quelle: TME Research, 2021

Mithilfe von Continuous Monitoring kann überprüft werden, ob ein Canary Deployment erfolgreich war oder nicht. Sollte bei einem Build-Prozess ein erhöhtes Fehleraufkommen auftreten, kann der Durchsatz, also die Anzahl der Anwender, die sich auf einer Canary Umgebung befinden, wieder auf 0 % reduziert werden. Bei einem fehlerfreien Deployment hingegen kann der Durchsatz auf 100 % erhöht werden. Analog zum Blue-Green-Deployment kann eine vollständige Umverteilung der gesamten Anwender von der Produktions- auf eine Alternativumgebung (Staging oder Canary) stattfinden. Darüber hinaus können DPs durch das Feedback der Anwender, welche die neuen Features auf der Canary Umgebung austesten konnten, den potenziellen Erfolg oder Misserfolg von neuen Funktionen besser abschätzen und entsprechend ihre Kapazitäten effizienter planen und aufteilen.

7

APIs, MICROSERVICES UND DEVOPS – DIE TREIBER DER TECHNISCHEN EXZELLENZ

In diesem Whitepaper werden die technischen Herausforderungen für den Aufbau und die Operationalisierung von digitalen Plattformen (DPs) beschrieben sowie bewährte Ansätze gegeben, um diese Herausforderungen über eine technische Exzellenz zu meistern.

Eine DP, die ihre Services über APIs öffentlich oder Partnern zugänglich macht, begibt sich durch Service-Vereinbarungen und ökonomische Ziele in die Verantwortung, ihre Services mit einer sehr hohen Verfügbarkeit anzubieten. Vor allem gilt es, die Verfügbarkeit selbst während der Anpassungen von Services ununterbrochen aufrechtzuhalten. Hierbei spielt ein ordentliches API-Lebenszyklusmanagement eine übergeordnete Rolle, welches definiert, wie verschiedene API-Versionen verwaltet und kommuniziert werden. Darüber hinaus sollten die Services von DPs auch unter erschwerten Bedingungen verfügbar sein, was eine gewisse Skalierbarkeit der technischen Architektur voraussetzt. Dafür eignen sich insbesondere moderne Microservices-Architekturen, bei denen einzelne Microservice-Komponenten losgelöst und unabhängig voneinander eine bestimmte fachliche Geschäftsfunktion erfüllen. Basierend auf dem Domain-Driven-Design können diese fachlichen Domänen geschaffen werden, die als Grundlage für den Einsatz von Microservices gelten. Die Interaktion zwischen den verteilten Services und den Domänen erfolgt über standardisierte und effiziente Kollaborations- und Kommunikationswege – und zwar über private APIs und Events. Damit autonome und isolierte Teams, die verantwortlich für die einzelnen Microservices sind, ihren Code schnell, effizient und sicher entwickeln und ausliefern können, empfiehlt sich ein DevOps-Ansatz. Dieser zeichnet sich insbesondere durch hochautomatisierte Prozesse sowie enge Zusammenarbeit aus Entwicklungs- und Betriebsverantwortlichen aus. Die Potentiale von Microservices-Architekturen und DevOps ermöglichen, dass DPs schnell, flexibel sowie mit hoher Qualität und Sicherheit auf die Veränderungen der Kundenbedürfnisse reagieren können. Begünstigt werden diese Eigenschaften durch experimentelle Vorgehensweise wie z. B. das Canary Deployment. Der richtige Umgang mit APIs, Microservices und DevOps ermöglicht DPs eine technische Exzellenz, mit der sie sich einen Wettbewerbsvorteil gegenüber ihren Konkurrenten verschaffen können.

Technical Excellence Assessment by PAWLIK Digital

Im Rahmen unserer Projekte im Bereich Digital Platforms konnten wir unsere Kunden bereits dabei unterstützen, eine Beurteilung ihrer technischen Exzellenz durchzuführen. Wir unterstützen auch Sie dabei, eine solche Bewertung durchzuführen sowie mit Ihnen eine Roadmap für die Erreichung des technischen Reifegrades aufzustellen, der Ihnen Wettbewerbsvorteile der agilen digitalen Ökosysteme sichert.

Gerne stellen wir Ihnen einen unserer CTO as a Service Experten zur Verfügung, um Sie bei den technischen Herausforderungen der digitalen Plattformisierung zu unterstützen. Bei Interesse kontaktieren Sie uns unter **info@pawlik-digital.com**.

AUTOREN UND ANSPRECHPARTNER:

STEFAN ROßBACH ist Mitgründer der PAWLIK Digital. Als Vorstand betreut er den Bereich Digitalisierung. Er ist Experte für Digitale Plattformen, Agile Transformation und Umsetzung digitaler Strategien. *srossbach@pawlik-digital.com*

ALEXANDER REZUN ist Consutling Manager bei PAWLIK Digital und IT-Experte mit langjähriger Erfahrung in den Themengebieten Software-Architektur, Business Intelligence und Cloud-Computing. *arezun@pawlik-digital.com*

ERNA SAKIC und **MARVIN ARNOLD**

QUELLENANGABEN:

Computerwoche, *Continuous Delivery in der Praxis*, 16.09.2015, https://www.computerwoche.de/a/continuous-delivery-in-der-praxis,3215540

Divante, *Monolithic architecture vs microservices*, 14.01.2020, https://www.divante.com/blog/monolithic-architecture-vs-microservices

Eric Evans, *Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison-Wesley*, 2003

FAZ, *Apple überholt Exxon Mobile*, 25.01.2011, https://www.faz.net/aktuell/wirtschaft/unternehmen/wertvollstes-unternehmen-der-welt-apple-ueberholt-exxon-mobil-11112818.html

Finanzen 100, *Die Top100 größten börsennotierten Unternehmen der Welt*, https://www.finanzen100.de/top100/die-grossten-borsennotierten-unternehmen-der-welt/

Frauenhofer IESE, *Security in Digitalen Ökosystemen: Die sichere Digitale Plattform ist nur die halbe Miete*, 03.09.2020, https://www.iese.fraunhofer.de/blog/security-in-digitalen-oekosystemen-die-sichere-digitale-plattform-ist-nur-die-halbe-miete/

Handelsblatt, *Amazon eröffnet ersten Laden ohne Kassen in Europa*, 04.03.2021, https://www.handelsblatt.com/unternehmen/handel-konsumgueter/einzelhandel-amazon-eroeffnet-ersten-laden-ohne-kassen-in-europa/26973790.html?ticket=ST-11408326-5Q0IDa1OqFOvzPWb4zkT-ap5

Microsoft, *Implementieren von API-Gateways mit Ocelot*, 02.03.2020, https://docs.microsoft.com/de-de/dotnet/architecture/microservices/multi-container-microservice-net-applications/implement-api-gateways-with-ocelot

MT AG - Dennis Hoffman, *CI/CD: Theorie meets Praxis*, 01.07.2020, https://www.mt-ag.com/blog/know-how/devops/ci-cd-theorie-meets-praxis/

Norbert Eder, *Blue Green Deployment*, https://www.norberteder.com/blue-green-deployment/

Norbert Eder, *Canary Deployment*, https://www.norberteder.com/canary-deployment/

Solarisbank, *About*, 2021 https://www.solarisbank.com/de/about/

Solarisbank, *Banking, Effortlessly Integrated into Your App*, 2021 https://docs.solarisbank.com/

Solarisbank, *Status of APIs*, 2021 https://status.solarisbank.de/

Tias, Digital Strategy: *Digital Platform Map*, 02.08.2019, https://www.tias.edu/en/item/digital-strategy-digital-platform-map/

IMPRESSUM

Pawlik Digital AG | Hamburger Allee 26-28 | 60486 Frankfurt am Main | Tel: +49 (0)69 7191 309 – 0 | Fax: +49 (0)69 7191 309 – 30
E-Mail: info@pawlik-digital.com | Internet: www.pawlik-digital.com | Gesetzlicher Vertretungsberechtigter: Vorstand: Stefan Roßbach, Thomas Deibert, Layla Dolfen Aufsichtsrat: Joachim Pawlik (Vorsitzender),
Marc Kulemann, Dr. Ralf Friedrichs | Sitz der Gesellschaft ist Frankfurt am Main |Registergericht: Amtsgericht Frankfurt am Main, HRB 99000
Inhaltlich Verantwortlicher gemäß § 55 Rundfunkstaatsvertrag: Pawlik Digital AG | Haftungshinweis: Trotz sorgfältiger inhaltlicher Kontrolle übernehmen wir keine Haftung für die Inhalte externer Links. Für den Inhalt der verlinkten Seiten sind ausschließlich deren Betreiber verantwortlich.