Auswirkungen einer serviceorientierten Architektur auf den Entwicklungs- und Auslieferungsprozess

Fabian Klimpel[1], Lukas Epple[2], Raphael Sack[3]

Abstract: Das Konzept der serviceorientierten Architektur (SOA) existiert schon seit den 1990er Jahren und wird zunehmend in größeren Softwaresystemen eingesetzt. Während diese Architektur viele Vorteile mit sich bringt, bietet die Umsetzung dieser auch viele Herausforderungen. Ziel dieser Arbeit ist es die Vor- und Nachteile der serviceorientierte Architektur zu dokumentieren und Unterschiede zu anderen konventionellen Software-Architekturen, insbesondere hinsichtlich der Entwicklung und Auslieferung aufzuzeigen. Dabei soll vor allem der Kontext der agilen Softwareentwicklung betrachtet werden. Für einen qualitativen Vergleich wurde eine umfassende Literaturrecherche durchgeführt. Die Recherchen zeigten, dass insbesondere die Qualität von umfangreichen und komplexen Softwaresysteme vom Einsatz der serviceorientierten Architektur und deren Auswirkungen auf den Entwicklungs- und Auslieferungsprozess profitieren kann.

Keywords: SOA; serviceorientierte Architektur; Software-Engineering; Software Architektur

1 Einleitung

Bereits seit den frühen 1990er-Jahren werden Konzepte für die Aufteilung von Anwendungen in einzelne Dienste (engl. *Services*) angewendet, um Verantwortlichkeiten innerhalb einer Anwendung entkoppeln und somit verteilen zu können. Zunehmend wurden diese Konzepte anschließend auch in Unternehmen für die Entwicklung umfangreicher Systeme übernommen. Große Systeme konnten somit in unterschiedliche getrennte Dienste aufgeteilt und auf mehrere Rechner verteilt werden. Dies ermöglichte die Entwicklung von Systemen, welche die Ressourcenkapazität eines einzigen Rechners überstiegen. Die isolierten Services waren zudem überschaubarer und konnten leichter gewartet werden. Ebenfalls konnten diese unabhängig von dem restlichen System weiterentwickelt werden.
Neben den Möglichkeiten bei der Entwicklung entstanden ebenfalls weitere Möglichkeiten für die Auslieferung und insbesondere für Aktualisierung und Skalierung von Systemen.

1.1 Motivation

Dieses Paper ist im Rahmen der Vorlesung *Advanced Software-Engineering* an der Dualen Hochschule Baden-Württemberg Stuttgart, Campus Horb entstanden. Im Rahmen dieser

[1] DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20021@hb.dhbw-stuttgart.de
[2] DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20009@hb.dhbw-stuttgart.de
[3] DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20029@hb.dhbw-stuttgart.de

Arbeit wird untersucht, welche Auswirkungen durch die Verwendung einer serviceorientierten Architektur (SOA) hinsichtlich des Entwicklungs- und Auslieferungsprozesses von Software resultieren.
Dabei wird zunächst der Begriff der SOA geklärt und definiert. Anschließend wird die geschichtliche Entwicklung bezüglich der Verwendung unterschiedlicher Softwarearchitekturen genauer untersucht. Zusätzlich wird beleuchtet, wie eine SOA realisiert werden kann. Im Anschluss daran werden die Auswirkungen der Architektur auf den Entwicklungs- und Auslieferungsprozess dargestellt.

1.2 Einführung und Definition

Für den Begriff der SOA existiert keine einheitliche, allgemein anerkannte Definition. Es handelt sich hierbei um keine konkrete Technologie, sondern um ein Konzept, welches über viele Jahre gewachsen ist und weiterentwickelt wurde. Da SOA auf unterschiedliche Bereiche angewendet werden kann, existieren viele unterschiedliche Definitionen, welche jeweils verschiedene Aspekte des Konzepts beleuchten.
Im Folgenden soll nun zunächst der Begriff des Services eingeführt werden. Anschließend sollen unterschiedliche Definitionen von SOA untersucht werden. Schließlich soll eine eigene Definition formuliert werden, die im Rahmen dieser Arbeit verwendet wird.

Ein Service ist ein eigenständiges Softwaremodul, welches zumeist die Funktionalität oder den Ablauf eines konkreten Geschäftsprozesses oder Vorgangs abbildet [Ra05]. Dabei kapselt ein Service meist mehrere zusammengehörige, feingranulare Komponenten und stellt eine präzise Schnittstelle für den Zugriff auf die Funktionalität bereit. Wichtig ist hierbei, dass der Service eine geschlossene Komponente darstellt, welche keine weiteren Abhängigkeiten nach außen besitzt [SHM08]. Dies gilt sowohl gegenüber verwendeten Bibliotheken innerhalb des Services als auch gegenüber anderen Services in einem System. Darüber hinaus sollte ein Service eine positionsunabhängige Adressierungsmöglichkeit anbieten, über welche die Schnittstelle nach außen nutzbar gemacht wird. Üblicherweise werden hierfür Netzwerkprotokolle verwendet. Die positionsunabhängige Kommunikation über standardisierte Netzwerkprotokolle ermöglicht die lose Kopplung mit anderen Services oder Konsumenten [ADM06]. Diese können also über die Schnittstelle die Funktionalität des Services nutzen, ohne dessen Position oder den verwendeten Technologiestack kennen zu müssen. Somit kann aus mehreren lose gekoppelten Services ein modulares Gesamtsystem zusammengesetzt werden, welches sehr einfach verteilt und skaliert werden kann [SHM08].

Eine Architektur für solche Systeme, welche auf einzelnen isolierten und verteilbaren Services basieren, wird *serviceorientierte Architektur* genannt [SHM08]. Dabei existieren in der Literatur unterschiedliche Ansichten, wie genau diese definiert wird. Es existieren viele verschiedene Definitionen aus unterschiedlichen Perspektiven. Diese beleuchten unterschiedliche Aspekte und sind somit zwar korrekt, aber selten einheitlich oder vollständig.
In einem Buch von Thomas Erl [Er09a] wird SOA als offene, agile und zusammensetzbare

Architektur, bestehend aus autonomen Web-Services beschrieben. Die Services sollen dabei eigenständig, wiederverwendbar und herstellerübergreifend interoperabel sein.
In einem Artikel aus dem *CrossTalk* Magazin der US Air Force [G 07] hingegen wird SOA nicht als Architektur, sondern aus architektonisches Muster beschrieben. Aus diesem Architekturmuster können laut den Autoren unbegrenzt viele konkrete Architekturen abgeleitet werden. In einem weiteren Buch von Erl [Er09b] wird SOA als Erweiterung der Objektorientierung beschrieben. Es werden zentrale Konzepte der Objektorientierung, wie beispielsweise Abstraktion, Kapselung und Wiederverwendbarkeit angewendet und durch stärkere Kapselung und unabhängige Kommunikation über Netzwerkprotokolle erweitert. Aus Software, bestehend aus interagierenden Objekten, wird Software aus interagierenden Services. Diese können außerdem mit heterogenen Technologien umgesetzt werden und über Systemgrenzen hinweg interagieren. Arnon Rotem-Gal-Oz beschreibt SOA in seinem Buch [Ro12] als Architekturstil für die Erstellung interaktiver Systeme, bestehend aus lose gekoppelten, grob-granularen und autonomen Services. Jeder Service definiert bestimmte Funktionalität beziehungsweise ein bestimmtes Verhalten und spezifiziert den Zugriff darauf über eine Schnittstelle, den sogenannten *Kontrakt*. Dieser Kontrakt wird über einen adressierbaren Endpunkt nach außen freigegeben. Er beinhaltet Nachrichten zur Benutzung der Funktionalität/des Verhaltens von externen Konsumenten.
Die Granularität der Services ist dabei mit Bedacht zu wählen. Sind die Services zu feingranular, wird das System extrem komplex und es muss mit einem riesigem Overhead umgegangen werden. Wählt man eine zu grobe Granularität, handelt es sich bei dem System mehr um ein Konglomerat mehrerer Monolithen [St21].
Folgende Definition soll die zentralen Punkte einer SOA zusammenfassen und wird im Rahmen dieser Arbeit verwendet.

> **Definition:** SOA stellt ein technologieunabhängiges Architekturmuster für verteilbare Systeme, bestehend aus homogenen oder heterogenen, lose gekoppelten und über Nachrichten interagierenden Services dar. Ein Service stellt dabei eine autonome Softwarekomponente dar, die konkrete Funktionalität oder einen Geschäftsprozess kapselt und über eine klar definierte Schnittstelle positionsunabhängig bereitstellt.

Basis für diese Definition ist der überwiegende Konsens in der Literatur.
Um die Isolation eines Services und gleichzeitig eine klar definierte Interaktion mit diesem zu gewährleisten, werden bestimmte Komponenten benötigt. Diese müssen in einem System, welches die serviceorientierte Architektur realisiert, vorhanden sein. Neben den Services zählen dazu folgende Punkte. [Ro12]

Schnittstelle/Kontrakt

Die Schnittstelle definiert die Menge aller möglichen Nachrichten zur Interaktion mit

dem Service. Der Kontrakt eines Services ist also vergleichbar mit einem Interface in der Objektorientierung.

Endpunkt

Der Endpunkt stellt einen Unique Ressource Identifier (URI) dar, über welchen die Schnittstelle freigegeben wird. Über den Endpunkt kann der Service also gefunden und adressiert werden.

Nachrichten

Über Nachrichten kann mit einem Service interagiert werden. Ein Konsument kann eine Nachricht an einen Service-Endpunkt senden. Erhält der Service eine Nachricht, die einer Spezifikation in dessen Kontrakt entspricht, reagiert der Service auf diese.

Darüber hinaus gibt es noch weitere optionale Bestandteile, wie beispielsweise Policies, die den Zugriff auf Services regeln oder ein Serviceregister, in welchem alle verfügbaren Service-Endpunkte dokumentiert werden.
Die Verwendung einer SOA führt also zu konkreten Eigenschaften eines Systems. So ist beispielsweise die bereits genannte Verteilbarkeit ein zentraler Punkt [Ad10]. Diese resultiert aus der losen Kopplung und positionsunabhängigen Adressierbarkeit. Neben der Flexibilität hinsichtlich der Positionierung wird außerdem eine stärkere Heterogenität der einzelnen Bestandteile eines Systems ermöglicht [SHM08]. Durch die Verwendung von standardisierten und Programmiersprachen-unabhängigen Protokollen für den Nachrichtenaustausch werden Implementierungsdetails vollständig hinter dem Kontrakt verborgen. Diese Möglichkeit zur Heterogenität fördert außerdem die Wiederverwendbarkeit, da einzelne Services in den unterschiedlichsten Systemen völlig unabhängig des verwendeten Technologiestacks genutzt werden können [Ad10]. Die gewonnene Flexibilität steht größerem Overhead und steigender Komplexität gegenüber.

1.3 Anwendungsfälle

Eine serviceorientierte Architektur kann in vielen unterschiedlichen Bereichen eingesetzt werden. Jedoch gibt es Anwendungsfälle wofür eine SOA besser oder schlechter geeignet ist. Gut geeignet ist eine SOA zum Beispiel für komplexe Applikationen, bei denen es viele Komponenten mit klar trennbaren Funktionalitäten gibt. Ebenfalls bietet SOA sich gut für skalierbare Anwendungen, welche auf verschiedenen Umgebungen oder in einer Cloud laufen an. Für kleine Applikationen mit nur wenigen Funktionalitäten bietet sich SOA weniger an, da dabei der Mehraufwand für die Entwicklung und Auslieferung zu groß ist. Statische Applikationen, welche keine Verbindung zu einem Backend benötigen, profitieren ebenfalls nicht von der Verwendung einer SOA.

Spezifischere Anwendungsfälle für SOA sind in der Finanzindustrie für die Integration verschiedener Banking-Systemen, in der Logistik für die Verbindung zwischen Transport-

und Lagersysteme oder in der Telekommunikationsindustrie für die Integration zwischen Netzwerk- und Dienstsystemen. Neben den zuvor genannten Anwendungsfälle gibt es noch viele verschiedene Fälle in denen SOA in der Praxis eingesetzt werden kann. [He07]

2 Geschichtliche Entwicklung

Um die Notwendigkeit der serviceorientierten Architektur nachvollziehen zu können, lohnt es sich die Vergangenheit - vor SOA - anzusehen. In diesem Kapitel wird die Entwicklung von Software-Architekturen vom Monolithen bis hin zur Serviceorientiertheit betrachtet.

2.1 Monolithische Architektur

Als monolithisch werden Objekte bezeichnet, die „aus einem Stück bestehen[.]; zusammenhängend und fugenlos [sind]“ [Du22]. Im Kontext des Software-Engineerings sind damit jegliche Anwendungen gemeint, deren Module nicht unabhängig voneinander ausgeführt werden können und deren Funktionalitäten in einer Applikation gekapselt sind [PMA19].

Während den Anfängen des Software-Engineerings und bevor das Fachgebiet der Software-Architektur existiert hatte, existierten fast ausschließlich Monolithen [KOS06].

Ende der 1960er Jahre gab es die ersten Bemerkungen, Software nicht nur als „amorphous lump of program“ [KOS06][S.24] anzusehen, sondern gezielt die Architektur eines Software-Systems zu designen.

Monolithen haben noch heute ihre Daseinsberechtigung. Gerade kleine stand-alone Anwendungen werden häufig von einem Service dargestellt. Der große Vorteil dabei ist das einfache Testen [PMA19]. Einzelne Dienste einer Anwendung müssen nicht mit Integrationstests auf verschiedenen Status abgebildet werden, da es nur einen Dienst zum Testen gibt. Auch das Deployment ist simpel, da es nur aus einer ausführbaren Komponente besteht.

Durch die „fehlende“ Architektur wird zunächst Zeit gewonnen, da weniger geplant werden muss. Für kleine Projekte oder Prototypen ist das ideal. Gerade bei Letzterem können durch den erstmaligen Aufbau im Monolith die Komplexität und einzelne Komponenten erforscht werden [PMA19]. Aber bei großen Projekten geht schnell der Überblick verloren und die Entwicklung kommt ins Stocken [PMA19].

Die Entwicklung einer monolithischen Anwendung bedeutet eine enge Bindung zur genutzten Technologie [PMA19]. Falls der Support verwendeter Drittanbieter-Software ausläuft, oder neue Schwachstellen gefunden werden, ist es nur schwer möglich Änderungen vorzunehmen. Und gleichzeitig: Egal wie groß oder klein eine Anpassung ist, die gesamte Anwendung muss immer neu getestet, gebaut und verteilt werden. Falls ein Laufzeitfehler auftritt, hat dies den Absturz der gesamten Anwendung zur Folge [PMA19]. Und obwohl solch ein

Bug durch eine kleine Modifikation gefixt werden kann. Manchmal ist ein neuer Build zu aufwändig und es wird auf mehrere Änderungen gewartet. Darunter leidet die Qualität der Software. Auch lässt sich die Applikation nicht beliebig skalieren. Durch zum Beispiel Load Balancer kann die gesamte Anwendung horizontal skalieren, nicht jedoch einzelne Komponenten.

Zusammenfassend: Monolithen sind gut für kleine Projekte, oder erste Proof of Concepts. Falls eine Software jedoch nachhaltig, mehrere Jahre lang zuverlässig betreut und weiterentwickelt werden soll, steigt die Komplexität dieser drastisch mit der Zeit. Neue Teammitglieder müssen sich teilweise in die komplette Codebase einarbeiten, um Änderungen vorzunehmen. Es kann auch dazu kommen, dass keine neuen Arbeitskräfte gefunden werden die sich mit 20 Jahre alter Software-Technologie auseinandersetzen wollen und somit ist eine Neuentwicklung früher oder später unumgehbar.

Wenn Architekturen - und damit der Aufbau von Software - verglichen werden, dann wird dies mithilfe von „fundamentalen Grundprinzipien“ getan [Fr]. Im Folgenden werden diese Punkte aufgezählt um zu zeigen, gegen welche Grundprinzipien die monolithische Architektur verstößt. In den darauf folgenden Kapiteln werden Architekturen gezeigt die iterativ verschiedene Probleme lösen, welche letztendlich einen historischen Verlauf zu SOA aufzeigen soll:

- **Abstraktion**: „Die essenziellen Eigenschaften eines Objekts, die es von allen anderen Arten von Objekten unterscheidet und somit klar definierte konzeptionelle Grenzen in Bezug auf die Perspektive des Betrachters setzt“ [Bo93]. In einem Monolith kann es keine echte Abstraktion geben, da die Software nur aus einem Objekt - sich selbst - besteht. Natürlich existieren auf einem niedrigeren Level eine Abstraktion, aber nicht in der Architektur selbst.
- **Kapselung**: Durch die Verkapselung verschiedenster Abstraktionen können diese gruppiert und auseinander gehalten werden. Dies fördert die Änderbarkeit und Wiederverwendbarkeit [Fr]. Die Kapselung, zum Beispiel von Klassen einer Programmiersprache kann auch für Monolithen existieren, aber nicht in einer Form, die eine Wiederverwendbarkeit anstrebt.
- **Modularisierung**: Hierbei geht es um die sinnvolle Zerlegung eines Softwaresystems und dessen Gruppierung in Subsysteme und Komponenten. Module dienen als physische Container für Funktionalitäten oder Verantwortlichkeiten einer Anwendung. [Fr]. Wie bei vielen dieser Prinzipien lässt die monolithische Architektur Modularisierung zu einem gewissen Grad zu, aber nur auf einem niedrigen Level.
- **Trennung von Verantwortlichkeiten**: Unterschiedliche Verantwortlichkeiten innerhalb eines Softwaresystems sollten voneinander getrennt werden.
- **Kopplung und Kohäsion**: Die Kopplung ist das Maß für die Stärke der Assoziation zwischen Modulen. Eine Starke Kopplung verkompliziert das System [Fr]. Kohäsion

misst den Grad der Konnektivität zwischen den Funktionen und Elementen eines einzelnen Moduls.

- **Suffizienz, Vollständigkeit und Primitivität**: Suffizienz meint, dass eine Komponente alle notwendigen Merkmale einer Abstraktion erfasst und eine sinnvolle und effiziente Interaktion ermöglicht. Vollständigkeit heißt, dass alle relevanten Merkmale erfasst werden. Mit Primitivität ist gemeint, dass jede Operation, die eine Komponente ausführen kann, einfach implementiert werden kann [Fr].
- **Single Point of Reference**: Jede Entität innerhalb eines Softwaresystems sollte nur einmal definiert werden. Dadurch entstehen keine inkonsistenten Zustände.

2.2 Schichtenarchitektur

Die Schichtenarchitektur war eines der ersten Architektur-Pattern mit welchem versucht wurde die Probleme einer monolithischen Architektur zu lösen [SM09] und ist bis heute wahrscheinlich einer der am häufigsten angewendeten Software-Architekturen (vor allem im Web).

Ein Programm wird dabei in *n* Schichten aufgeteilt, jede Schicht ist ein Modul der Software und kommuniziert über Protokolle mit anderen Schichten. Die Aufteilung alleine löst schon fast alle oben genannten Probleme. Die Schichten bilden meist eine Hierarchie ab, wobei die Kommunikation nur strikt durch gewisse Schichten geschieht.

Vor allem lassen sich Verantwortlichkeiten sehr gut damit trennen. Als Beispiel, kann das 3-Schichten Modell aus typischen Web-Anwendungen betrachtet werden[4]:

1. **Präsentationsschicht**: das Frontend als grafische Benutzeroberfläche im Browser.
2. **Anwendungsschicht**: die eigentliche Funktionalität der Software abgeschottet von Anwendenden.
3. **Datenschicht**: eine Datenbank, auf der die Anwendungsdaten verwaltet werden.

Bei der Modellierung können die Schichten einzeln betrachtet werden, um Schnittstellen zu definieren. Entwicklerteams können daraufhin gleichzeitig an Front- und Backend arbeiten und sind dabei technologisch unabhängig voneinander. Und im Laufe des Lebenszyklus der Anwendung können einzelne Schichten ausgetauscht werden, um neue Technologie einzusetzen. Weitere Vorteile sind auch, dass sich die Geschäftslogik direkt im Code befindet und dass geheime Informationen nicht öffentlich für Benutzer zugänglich sind, sondern sich auf einer unzugänglichen Schicht befinden.

[4] Auch wenn dies ein prominentes Beispiel ist, besteht eine Schichtenarchitektur nicht immer aus 3 Schichten

Für große Projekte mit größeren Teams ist es leichter Software in Schichten zu schreiben. Einzelne Teams können sich dabei auf eine bestimmte Schicht spezialisieren, um effizienter zu sein.

Die Verteilung von Software bringt auch Nachteile mit sich, bzw. alle Vorteile der monolithischen sind die Nachteile von dieser Architektur:

- Das Testen ist wesentlich aufwändiger. Unit-Tests sind auf den einzelnen Schichten unverändert, aber bei der Integration aller Schichten ist es schwer alle Testfälle abzudecken bzw. die Tests überhaupt zu schreiben.
- Die Installation von Software ist wesentlich aufwändiger, da mehrere Schichten meist über das Internet kommunizieren müssen.
- Der Planungsaufwand ist höher und vor allem kleine Projekte könnten unnötig Zeit an der Trennung der Schichten verlieren. Macht sich aber in der späteren Betreuung des Codes bezahlt.

Zusammengefasst lässt sich sagen: Die Schichtenarchitektur teilt **eine** Software in **verschiedene** Schichten auf. Dabei muss stets auf die Balance zwischen Kopplung und Kohäsion geachtet werden. Die Schichten können dabei verschiedenste Technologien implementieren, solange diese mithilfe wohl definierter Protokolle kommunizieren können. Die Schichten können in der Entwicklung anderer Software wiederverwendet werden, was die Entwicklung in Zukunft erleichtert. Aber die Architektur zeigt ebenfalls Probleme auf. Die Schichten trennen zwar zu einem gewissen Grad die Verantwortlichkeiten der Software, aber für große Projekte ist die statische Hierarchie der Schichten nicht fördernd. Oftmals werden bei einer 3-Schichten Architektur auch nur 2 große Monolithen entwickelt (mit einer Datenschicht) [Fr].

2.3 Serviceorientierte Architektur

Die Herausforderung der Schichtenarchitektur kann gelöst werden, indem die „Schichten“ granularer werden. Und anstatt diese in einer festen Hierarchie anzuordnen - wo zum Beispiel Schicht n nur mit Schicht $n + 1$ kommuniziert - gibt es eine liberalere Kommunikation, die nicht mehr statisch vorgegeben sein muss. Anstatt, dass eine Schicht für die gesamte Geschäftslogik einer Anwendung zuständig ist, werden weitere Verantwortlichkeiten innerhalb dieser in Services aufgeteilt. Dadurch wird eine losere Kopplung und eine erhöhte Kohäsion erreicht.

Dieses Ziel hat gerade für große Softwareunternehmen einen immensen Vorteil. Anstatt einzelne Services nur in der Entwicklung wiederzuverwenden, können diese jetzt zur Laufzeit wiederverwendet werden. Die Verantwortlichkeiten sind so granular, dass sich diese in anderen Projekten unverändert wiederverwenden kann.

Langfristig bilden sich nur geringe Kosten durch den hohen Grad der Wiederverwendbarkeit der Dienste, aber bei diesem Grad der Granularität stößt die serviceorientierte Architektur an einen großen Overhead der auch Probleme mit sich bringt [So22]:

- Die Analyse, Konzeption und Implementierung ist initial wesentlich höher als bei anderen Architekturen.
- Die Wiederverwendbarkeit kann auch ein Problem darstellen, da dies nur eingeschränkt und nur auf sehr langer Zeit gesehen möglich ist. Das erfordert eine noch genauere Planung für die Zukunft.
- Die Granularität erhöht die Anzahl der Schnittstellen, deren Änderungen oftmals zu Kompatibilitätsproblemen führen können (=erhöhte Komplexität).
- In der Regeln hat eine SOA-Anwendung eine schlechtere Performance und höheres Datenvolumen, da die Dienstkommunikation höhere Latenzen darstellt.
- Die Bereitstellung und Konfiguration der Bindung zwischen den Diensten ist häufig komplex und Themen wie die Authentifizierung/Autorisierung sind Herausforderungen die mit hohen Kosten verbunden sind.
- Entwicklerteams müssen über ein breiteres Wissen verfügen um SOA wirklich anwenden zu können.
- Die Kostenverwaltung der Entwicklung einer zentralen und Abteilungsübergreifenden SOA ist eine politische Herausforderung und muss von Mitarbeitenden unterstützt werden.

Die Information, die aus diesem Kapitel herausstechen sollte, ist die Tatsache, dass sich SOA erst bezahlbar macht, nachdem die gesamte Architektur steht. Bis zu diesem Punkt sind immense Aufwände seitens des Unternehmens notwendig. Bei kleinen bzw. neuen Softwareunternehmen ist der Systemaufbau mit SOA meist zu teuer und würde sich nicht lohnen, bzw. bei einer Gründung ist es schwer in die Zukunft zu blicken, um die richtigen Services herauszusuchen. Für bestehende Unternehmen heißt eine SOA, die Neuentwicklung der Softwarelandschaft. Ein Transformationsprojekt hin zu SOA kann Jahre dauern und ist ein großes für die Zukunft. Die abgeschlossene Transformation macht sich jedoch bezahlt.

Interessant zu sehen: bei diesen Architekturen gibt es einen Trade-off zwischen Komplexität in der Planung und der Komplexität in der Pflege/Entwicklung. Die Gesamtkosten eines Projektes sollten unabhängig von der Wahl einer Architektur sein, jedoch werden die Kosten und Aufwände anders über den Projektlebenszyklus verteilt.

Monolithen haben wenig initiale Aufwände, aber die spätere Pflege ist mit einem hohen Kostenaufwand verbunden. SOA hat immense Anfangskosten, welche nach dem Aufbau der Architektur wieder fallen. Und die Schichtenarchitektur ist eine Balance dazwischen. Monolithische Projekte können ohne viel Aufwand angefangen werden und falls diese ohne Abschluss scheitern, so wurde nur der minimale Aufwand erreicht. Im Gegensatz dazu kostet ein gescheitertes SOA-Projekt wesentlich mehr Geld und Aufwände, da das meiste davon in der Planung (am Anfang) aufgebracht wird.

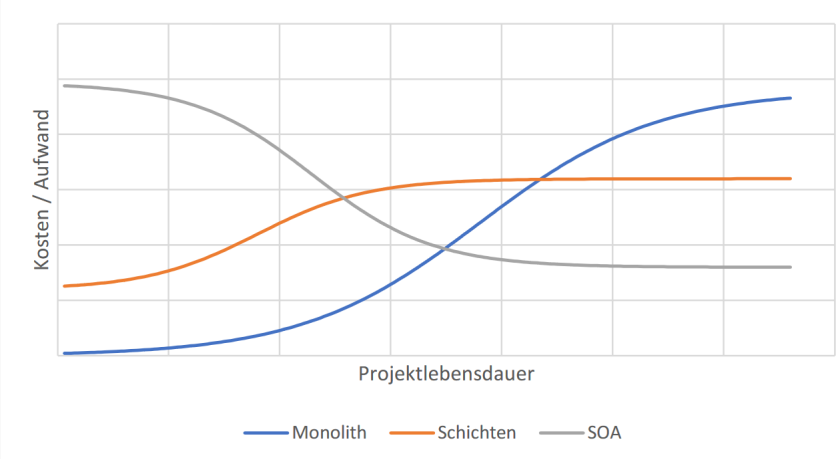


Abb. 1: Relativer Kosten-/Aufwand-Vergleich der vorgestellten Architekturen

Trotzdem kann sich der Umstieg zu einer serviceorientierten Architektur lohnen. Im Nachfolgendem wird tiefer darauf eingegangen wie sich SOA auf den Entwicklungs- und Auslieferungsprozess in der agilen Softwareentwicklung auswirkt.

3 Realisierung einer SOA

Es gibt viele unterschiedliche Wege eine SOA in der Praxis zu realisieren. Eine der verbreitetsten Möglichkeiten eine serviceorientierte Architektur zu realisieren ist mit Web-Services. Weitere Technologien zur Implementierung von SOA ist der Komponentendienst COM+ von Microsoft, Java 2 Platform Enterprise Edition (J2EE) oder die CORBA-Spezifikation. Im Folgenden werden jedoch nur Web-Services betrachtet, da dies der verbreitetste Weg ist eine SOA zu implementieren.

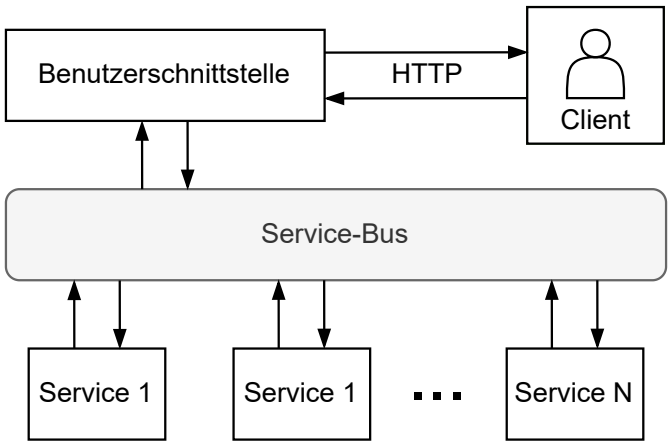


Abb. 2: SOA Aufbau

Wichtig zu sagen ist noch, dass es nicht nur einen richtigen Weg gibt, eine SOA zu implementieren. Es gibt meist viele verschiedene Wege, welche zum gewünschten Ziel führen können. In der Regel müssen mehrere Services implementiert werden, welche gemeinsam über einen Service-Bus miteinander kommunizieren können. Je nach expliziter Implementierung ist der Service-Bus dabei eine globale Instanz über die gesamte Applikation hinweg, oder eine Abstraktion für direkte Verbindungen zwischen verschiedenen Services. In Abbildung 2 wird ein beispielhafter Aufbau einer serviceorientierten Architektur dargestellt. Der Service-Bus ist dabei eine Abstraktion der Schnittstellen und an ihn angebunden sind verschiedene Service-Komponenten. Einer der Services ist dabei die Benutzerschnittstelle, mit der die Benutzer zum Beispiel über ein Frontend interagieren können.

Service-Bus

Mit der wichtigste Bestandteil eines Services ist seine Schnittstelle. Für die Realisierung der Schnittstelle kommen zwei verschiedene Designprinzipen infrage. Entweder ein nachrichtenorientiertes Design oder ein hypermedia-gesteuertes Design.
Bei dem nachrichtenorientierten Design kommunizieren die Services über einen Service-Bus. Über diesen können die einzelnen Komponenten Nachrichten austauschen. In der Regel wird dabei ein Message-Broker oder eine direkte Kommunikation der Services über TCP/IP verwendet. Für die Kommunikation zum Konsumenten wird in der Regel eine HTTP-API bereitgestellt.
Die zweite Möglichkeit ist die hypermedia-gesteuerte Implementierung. Im Gegensatz zur nachrichtenorientierten Implementierung werden bei der hypermedia-gesteuerten Implementierung nicht nur Daten, sondern auch Inhalte mit Beschreibungen möglicher Aktionen ausgetauscht. Dabei wird zum Beispiel direkt HTML Quelltext mit einer Form zurückgegeben. Dies ist vorteilhaft, wenn der Konsument über eine Webseite auf einen Service zugreifen will. [Na16]

Für die eigentliche Implementierung von dem Service-Bus gibt es viele Möglichkeiten. Ein Service-Bus kann beispielsweise als separate Software-Komponente implementiert werden. Der Service-Bus kann dabei von Grund auf implementiert werden oder es kann ein bestehender Message-Broker wie zum Beispiel RabbitMQ verwendet werden. Dabei kann der Service-Bus Nachrichten empfangen und über Routing an den richtigen Service weiterleiten. Der Vorteil daran ist, dass der Service-Bus dabei beliebig skaliert werden kann. Eine weitere Möglichkeit wäre eine in die Services integrierte Middleware, welche die benötigten Schnittstellen zur Kommunikation unter den Services zur Verfügung stellt. Dabei werden üblicherweise Web-Protokolle wie SOAP oder REST eingesetzt. [He07]

Service-Komponente

Eine Service-Komponente ist eine Software-Entität, welche als eigenständige, unabhängige und in sich geschlossene Einheit mit einem klaren Zweck besteht. Die Komponente ist dabei eine unabhängige Einheit von einer Funktionalität mit standardisierten Schnittstellen zur Kommunikation mit anderen Komponenten in einer Anwendung. In der Regel hat die Service-Komponente eine eindeutige Funktion, welche unabhängig von anderen Service-Komponenten der Anwendung ist.

Die eigentliche Implementierung des Services kann in einer beliebigen Programmiersprache geschehen. Es können auch unterschiedliche Services in unterschiedlichen Programmiersprachen implementiert werden. Dabei kann individuell für die Funktionalität eines jeweiligen Services eine optimale Programmiersprache verwendet werden. Wichtig ist dabei lediglich, dass die Schnittstellen korrekt angesprochen werden.

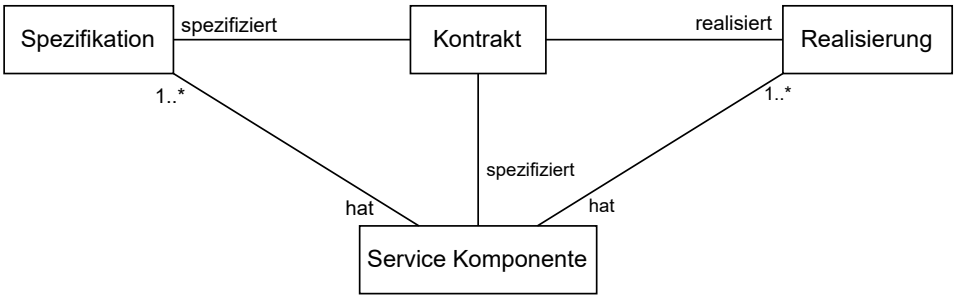


Abb. 3: Spezifikation und Realisation einer Service-Komponente [SDS04]

Die Basiselemente für die Planung und Umsetzung einer Service-Komponente sind die Spezifikation, der Kontrakt und die Realisierung. Die Zusammenhänge der Basiselemente sind in Abbildung 3 dargestellt. Im Kontrakt sind dabei die Regeln und Anforderungen der Service-Komponente spezifiziert. Also der Funktionsumfang, welcher der Service bereitstellen soll. Dieser Kontrakt wird in der Spezifikation weiter spezifiziert. In der Spezifikation wird die Art und die Verwendung der Schnittstellen definiert. Darunter zählen zum Beispiel die verwendeten Protokolle und Standards für die Kommunikation

und die Information welche Funktionen von dem Service bereitgestellt werden. Wurde alles korrekt spezifiziert kann der Service-Kontrakt implementiert werden. Für die Benutzung des Services ist dessen Implementierung nicht von Bedeutung, solange der Kontrakt des Services voll erfüllt wurde. [SDS04]

3.1 Konzepte und Technologien

In der serviceorientierten Architektur gibt es keine Vorschriften wie genau etwas zu implementieren ist und welche Technologie dabei verwendet werden soll. Jedoch gibt es einige Technologien und Konzepte, welche sich in dem Bereich der SOA etabliert haben. Die Wahl der explizit zu verwendeten Technologien ist dabei von den Anforderungen der Anwendung abhängig. Bei der Planung und dem Design sollten die Technologien jedoch genau spezifiziert werden, welche anschließend zur Implementierung verwendet werden.

Die verbreitetste Technologie zur Umsetzung von SOA sind Web-Services. Web-Services kommunizieren über standardisierte Schnittstellen in einem Netzwerk miteinander. Dabei funktioniert die Kommunikation über Netzwerkprotokolle wie zum Beispiel SOAP oder REST. [He07]

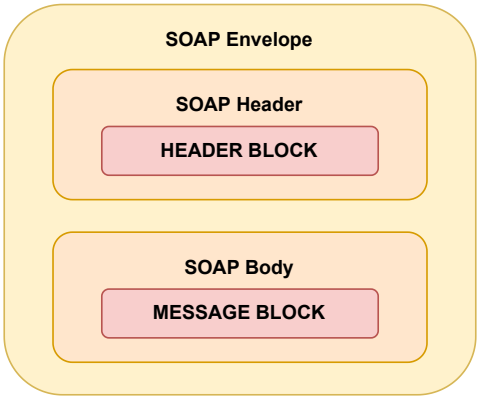


Abb. 4: SOAP Aufbau [Gi22]

Das Simple Object Access Protocol (SOAP) benutzt in der Regel das HTTP-Protokoll zur Datenübertragung. Dabei werden die Daten in Form von XML gesendet. SOAP ist unabhängig von der verwendeten Programmiersprache und wird meist in verteilten Systemen und somit auch in SOA verwendet. Der Aufbau einer SOAP-Komponente ist in Abbildung 4 dargestellt. Außen befindet sich der SOAP Envelope, welcher alle Daten enthält und das XML-Dokument als SOAP-Nachricht identifiziert. In dem SOAP Envelope befindet sich ein Header und ein Body. Der Header beinhaltet dabei zusätzliche Informationen über die Nachricht. Dies sind zum Beispiel zusätzliche Daten zur Authentifizierung. Im Body steht

der eigentliche Inhalt der Nachricht. SOAP kann sowohl für Anfragen an einen Service, als auch für dessen Antwort verwendet werden. [Gi22]

Der Aufbau in Abbildung 4 wird wie folgt im XML-Format geschrieben:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/">
    <soap:Header>
        HEADER BLOCK
    </soap:Header>
    <soap:Body>
        MESSAGE BLOCK
    </soap:Body>
</soap:Envelope>
```

Um einen Service und vor allem dessen Schnittstellen zu beschreiben kann die Web Services Description Language (WSDL) verwendet werden.
WSDL ist ein Standardformat um einen Web-Service zu beschreiben. Die Beschreibung des Services wird in dem XML-Format in der WSDL-Datei definiert. Neben einer Beschreibung der Funktionalität des Services sind auch alle Informationen, um mit einem dem Web-Service zu kommunizieren, darin enthalten. [He07]

Häufig wird WSDL in Verbindung mit SOAP verwendet. Dabei kann der Client, welcher den Service anfragt, die WSDL Datei einlesen, um somit die verfügbaren Funktionen des Services abzurufen. Nun kann er die in WSDL definierten verfügbaren Funktionen über das SOAP-Protokoll verwenden. [He07]

Für die Umgebung, in der die Services ausgeführt werden, bietet sich eine skalierbare Cloud-Infrastruktur an, bei der eine schnelle Bereitstellung an Rechenkapazitäten und ein automatisches Deployment der Services realisiert werden kann.

3.2 Sicherheitskritische Aspekte der SOA

Eine serviceorientierte Architektur ist ohne IT-Sicherheit nicht denkbar. Neben klassischen, bei jeder IT-Lösung, auftretenden Aufgaben gibt es auch - durch den verteilten Ansatz von SOA - spezielle Anforderungen an das System [ZM09].

Die einzelnen Aufgaben müssen dabei nicht neu erfunden werden, sondern haben lediglich besondere Ausprägungen in der Anwendung von SOA. In allen Fällen muss sich mit folgenden Problemen befasst werden [ZM09]:

- Identitätsverwaltung
- Authentisierung

- Autorisierung (=Access Control)
- und die verschlüsselte Kommunikation zwischen den Diensten.

Nachfolgend wird beschrieben, wieso diese Aufgaben gerade in SOA kritische Herausforderungen sind und wie sie gelöst werden.

Access Control: In einer SOA müssen nicht nur viele Benutzer, sondern auch Dienste authentifiziert werden. Die Verwaltung dieser Identitäten, insbesondere über Organisationsgrenzen hinweg ist eine Notwendigkeit [ZM09].

Bei SOA existieren - anders als bei monolithischen Architekturen - viele potenzielle Schwachstellen an jedem Service. Wird eine davon ausgenutzt, so ist zwar nur ein kleiner Teil der Anwendung kompromittiert, aber die Sicherheit des Gesamten ist nicht mehr vorhanden.

Jeder einzelne Service sollte ausreichend mit geschützt sein, damit nur autorisierte Identitäten Zugriff auf eine Ressource haben. Gleichzeitig sollten Benutzer sich nicht bei jedem Service einzeln anmelden, da es nicht benutzerfreundlich ist.

Dies kann erreicht werden, indem bestehende Sicherheits-Architekturen oder Standards in SOA als weitere Services integriert werden. Der Hauptzweck hinter all diesen Standards ist die Orchestrierung der Authentifizierung zwischen mehreren Diensten.

Die dabei am häufigsten verwendeten Standards sind:

- **XACML**: Dieser Standard besteht aus mehreren Services (Points), welche anhand von Policies Anfragen einer Entität bewerten und gegebenenfalls weiterleiten. „XACML bleibt die einzige standardisierte Methode zur dynamischen Durchsetzung von Autorisierungen, indem Zugriffskontrollen von Anwendungen und Datenbanken ausgelagert und Geschäftsrichtlinien verwendet werden.“ [Ax22]

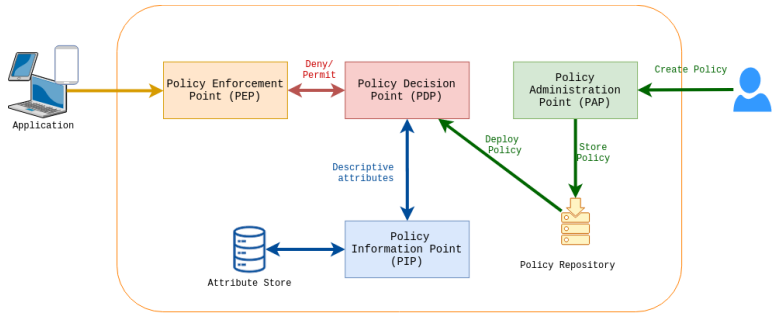


Abb. 5: XACML Flow-Diagramm

- **SAML2**: Hierbei exisitiert ein Identity Provider, an den jeder Service „weiterleitet“, um zentral an einer Stelle die Autorisierung zu überprüfen. [Wi06]

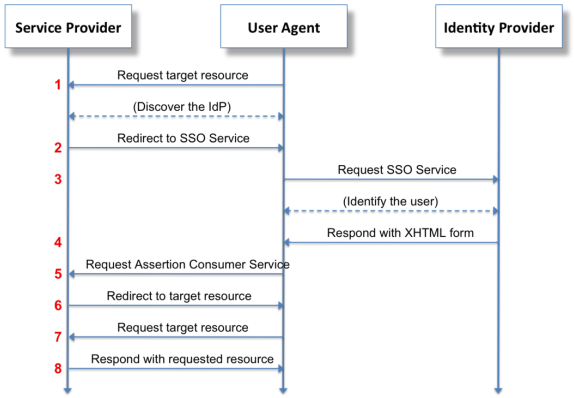


Abb. 6: SAML Sequenz-Diagramm

Die detaillierte Beschreibung dieser Standards würde den Rahmen dieser Arbeit übersteigen. Es existieren jedoch viele Ressourcen, um sich diese näher anzusehen.

Verschlüsselung: Neben den Diensten selbst, stellen auch die Kommunikationskanäle dazwischen einen Angriffsvektor dar, da diese meist Netzwerkprotokolle verwenden. Um die Integrität, Vertraulichkeit und die Authentizität der Nachrichten zu erfüllen wird der WS-Security (Web-Service Security) Standard verwendet.

WS-Security baut auf SOAP auf und klärt:

- Wie SOAP-Nachrichten signiert werden um die Integrität sicherzustellen.
- Wie SOAP-Nachrichten verschlüsselt werden um Vertraulichkeit zu gewährleisten.
- Wie Sicherheitstokens an SOAP-Nachrichten angehängt werden können um die Identität des Absenders sicherzustellen.

Die WSS-Spezifikation gibt nicht explizit vor, welche Signaturformate, Verschlüsselungsalgorithmen oder Sicherheits-Token Modelle verwendet werden müssen. Somit kann dieser für die Zukunft geändert werden, falls bessere und neuere Modelle existieren.

Das große Problem bei der Kanalverschlüsselung in SOA ist die große Anzahl der Kanäle. Egal welche Sicherheitsmaßnahmen eingesetzt werden, sie werden immer die Performance beeinträchtigen.

WSS ist ca. 8 Mal langsamer als herkömmliches HTTPS [LF04], bietet jedoch sichere Funktionalitäten.

Ein großer Performanceverlust liegt in der Generierung eines neuen Schlüssels für jede geschickte Nachricht. Als Upgrade existiert WSS-Conversation (WSSC), welches gleichzeitige Sessions unterstützt, ohne einen Schlüssel neu definieren zu müssen (verdoppelt den durchschnittlichen Durchsatz) [LF04].

4 Auswirkungen auf den Entwicklungsprozess

Der Entwicklungsprozess beschreibt den Prozess von der Planung über das Design bis zur letztendlichen Implementierung einer Anwendung. Durch die serviceorientierte Architektur gibt es in einigen Bereichen des Entwicklungsprozesses größere oder kleinere Auswirkungen im Gegensatz zu herkömmlichen Architekturen wie zum Beispiel der monolithischen Architektur. Dabei gibt es bei SOA auch andere Schwerpunkte auf welche bei der Entwicklung zu achten sind. Da es nicht nur einen richtigen Weg gibt, eine SOA zu implementieren können die Schwerpunkte je nach spezifischer Implementierung etwas abweichen.

Der Software-Entwicklungsprozess hat sich über die letzten Jahrzehnte stark weiterentwickelt. Agile Softwareentwicklung wurde immer populärer und damit einhergehend auch die Entwicklung von verteilten Applikationen. Der serviceorientierte Ansatz für die Softwarearchitektur hat sich dabei zu einer wichtigen Alternative gegenüber der traditionellen Softwareentwicklung entwickelt. [HR10]

4.1 Feldstudie

Um die Veränderungen von SOA auf den Entwicklungsprozess genauer zu untersuchen, wurden im Jahr 2010 in einer Feldstudie Softwareentwickler und IT-Manager aus fünf verschiedenen Unternehmen zu diesem Thema befragt. Bei den gestellten Fragen ging es explizit um die Auswirkungen auf die Softwareentwicklung von SOA basierenden Web-Service. Eines der Unternehmen ist ein Consulting Unternehmen und die restlichen vier Unternehmen sind direkt in der Softwareentwicklung tätig. In jedem der Unternehmen wurde SOA bereits etabliert, oder es war zu dem Zeitpunkt der Umfrage dabei zu SOA umzustellen. Die einzelnen Aussagen der Interviewteilnehmer wurden analysiert, gegeneinander verglichen und zusammengefasst. Zur Validierung der Ergebnisse haben die Teilnehmer im Anschluss noch einmal über die Ergebnisse geschaut. Die Resultate wurden in fünf Phasen des Software-Entwicklungsprozesses eingeteilt. Darunter zählen die Planungsphase, Analysephase, Designphase, Implementierungsphase und die Testphase. [HR10]

Im Folgenden werden die aus der Feldstudie ermittelten Unterschiede von SOA zu herkömmlichen Architekturen, wie zum Beispiel dem Monolithen, erläutert.

Planungsphase

Die erste Phase ist die Planungsphase. In dieser Phase gibt es größere Veränderungen gegenüber der monolithischen Architektur. Die genaue Umgebung der Applikation ist in einer serviceorientierten Architektur in der Planungsphase noch sehr unbestimmt. Viele Aspekte sind mit SOA deutlich schwerer vorherzusagen und zu kontrollieren, vor allem wenn verschiedene Services von unterschiedlichen Bereichen eines Unternehmens oder sogar von anderen externen Unternehmen stammen. Vor allem in der Planungsphase ist es essenziell, dass effektive Kommunikationskanäle zwischen den verschiedenen Stakeholdern aufgebaut werden. Dies ist zwar auch bei anderen Architekturen nötig, doch für SOA ist es wesentlich wichtiger für einen zukünftigen Projekterfolg. Ein weiterer wichtiger Punkt ist es, Standards festzulegen, welche von den Services eingehalten werden müssen. Darunter fallen zum Beispiel Protokolle zur Kommunikation zwischen verschiedenen Services. An diese Standards muss sich bei der Designphase aller benötigten Services gehalten werden. [HR10]

Analysephase

In der Analysephase fallen die wenigsten Veränderungen im Vergleich zu herkömmlichen Architekturen an. Jedoch ist die Analysephase in SOA ein sehr wichtiger Bestandteil des Entwicklungsprozesses. Ein wichtiger Punkt ist dafür zu sorgen, dass alle verwendeten Datenmodelle und Schemata alle benötigten Daten zur Verfügung haben, da bei SOA im Vergleich zu herkömmlichen Architekturen eine globale Perspektive über alle Services hinweg benötigt wird. [HR10]

Designphase

In der Designphase gibt es wie in der Planungsphase wesentliche Unterschiede im Gegensatz zu herkömmlichen Architekturen. Bei SOA ist es sehr wichtig, dass bei der Designphase von Anfang an alle benötigten Schnittstellen definiert werden. Wenn zu einem späteren Zeitpunkt etwas an den Schnittstellen zwischen den Services geändert werden soll, ist mit einem erheblichen Mehraufwand zu rechnen. Wenn bei herkömmlichen Architekturen mit Schnittstellen gearbeitet wird, ist es ebenfalls wichtig diese schon zu Beginn zu definieren, jedoch sind die Auswirkungen bei Änderungen zu einem späteren Zeitpunkt bei vergleichsweise SOA deutlich verstärkt. Die allgemeine Entwicklung von Schnittstellen verändert sich mit SOA auch stark, da in die Services Standards wie zum Beispiel WSDL eingebunden werden müssen. [HR10]

Implementierungsphase

Auf die Implementierungsphase hat SOA die meisten Auswirkungen. Mit SOA können schnell neue und verbesserte Funktionalitäten implementiert werden, ohne mit anderen Geschäftsprozesse dabei in Konflikte zu geraten. Es ist somit einfacher neue Services bereitzustellen und somit hat SOA eine sehr positive Auswirkung auf die Implementierung. Das Verwalten und Management der einzelnen Services ist nun jedoch ein etwas kritischerer

Punkt, da jeder Service ein potenzieller „single point of failure“ einer Anwendung darstellt. Je nach Aufbau der Anwendung kann man dies zum Beispiel mit redundanten Services beschränken. Bei der Implementierung der einzelnen Services ist in der Implementierungsphase ebenfalls ein hohes Maß an Kommunikation und Koordination zwischen den einzelnen, in den Entwicklungsprozess involvierten Gruppen nötig. Durch unterschiedliche Gruppen besteht jedoch auch die Gefahr, dass Services mehrere unterschiedliche Kanäle zur Kommunikation benötigen. Deswegen ist es wichtig die zuvor spezifizierten Standards für SOA einzuhalten. [HR10]

Testphase

Ebenfalls hat SOA einen großen Einfluss auf die Testphase. Dabei muss vor allem mehr Fokus auf die Integrationstests gelegt werden. Bei den Tests sind zwei verschiedene Umgebungen zu beachten. Eine Umgebung, in welcher die Service-Entwickler möglichst einfach Test-Clients erstellen können, um die Services zu testen und eine weitere Umgebung für Client-Entwickler, welche die entwickelten Applikationen gegen Test-Services testen können. Tools für automatisierte Test-Prozesse spielen dabei ebenfalls eine große Rolle. Nicht nur für funktionale Fehler, sondern auch für die Sicherheit und Performance der Services. Durch die Komplexität der Umgebung sind automatisierte Integration-Tests im Gegensatz zu einer monolithischen Architektur wesentlich wichtiger. [HR10]

Ergebnis der Feldstudie

Die Feldstudie zeigt, dass durch SOA der Entwicklungsprozess teilweise stark angepasst werden muss, um ein effektives Arbeiten zu ermöglichen. Einige Bereiche des Entwicklungsprozesses sind dabei stärker betroffen als andere. Ein größeres Augenmerk muss bei SOA auf die Planungs- und Designphase gelegt werden, um die unterschiedlichen Services inklusive deren Schnittstellen korrekt zu definieren. Die Kommunikation zwischen den verschiedenen Teams hat dabei ebenfalls einen besonders großen Stellenwert. Eine weitere Auffälligkeit ist, dass Änderungen an der anfänglichen Planung mit sehr hohen Kosten bzw. Mehraufwand verbunden sind.

4.2 Agile Softwareentwicklung

Agile Softwareentwicklung hat in der Vergangenheit immer mehr an Popularität gewonnen. Damit einhergehend hat der serviceorientierte Ansatz für die Entwicklung immer mehr an Bedeutung gewonnen. Größere Unternehmen können nun mit vielen kleineren Entwicklungsteams unabhängig voneinander parallel an einem Projekt produktiv arbeiten. Somit ist zum Beispiel jedes Team für einen Service zuständig. Bei herkömmlichen monolithischen Ansätzen nimmt die Produktivität ab einer gewissen Teamgröße nicht mehr zu oder sogar ab, da sich die Teammitglieder dabei behindern oder in die Quere kommen. Ebenfalls können in einer Applikation unterschiedliche Programmiersprachen für die Services benutzt werden,

und somit auf die Anforderungen des Services oder auf die Kompetenzen des jeweiligen Entwicklungsteams angepasst werden.

Ein wichtiger Punkt, auf welchen in dem Entwicklungsprozess zu achten ist, ist die Service-Granularität. Die Service-Granularität beschreibt den Funktionsumfang eines einzelnen Services. Bei einer hohen Granularität gibt es somit viele Services mit jeweils sehr kleinem Funktionsumfang und bei einer geringen Granularität gibt es wenige Services mit einem größeren Funktionsumfang.
In der agilen Softwareentwicklung geht es um die Reduzierung der Größe und des Umfangs der Probleme, der Reduzierung der Zeit für die Implementierung und die Reduzierung der Zeit um Feedback zu enthalten. Dafür bieten sich eine hohe Service-Granularität und somit kleine Services mit sehr beschränkten Funktionalitäten an. Somit können kleine Services mit einem kleinen Funktionsumfang eigenständig entwickelt werden. Wie klein genau ein Service sein sollte, ist jedoch schwer zu pauschalisieren, geschweige denn zu messen. In der Praxis ist jedoch eine höhere Service-Granularität und somit mehrere auf jeweils einen einzelnen Funktionsbereich zugeschnittene Services vorzuziehen. [Na16]

4.3 Modularität und Wartbarkeit

Durch die Services ist ebenso ein höheres Level an Modularität in einer Applikation gegeben. Durch die serviceorientierte Architektur muss nur die Kommunikation unter den Services über die Schnittstellen fest definiert sein. Wie ein Service im inneren aufgebaut ist, ist dabei nebensächlich. Verschiedene Services können somit wiederverwendet werden um redundante Softwareentwicklungen vermeiden. Dabei kann nicht nur der Quelltext wiederverwendet werden, sondern teilweise auch ganze Software-Komponenten. Die entwickelten Service-Komponenten können dabei auch in anderen Anwendungen und Systemen eingesetzt werden, um deren Funktionalität zu erweitern. Durch die Wiederverwendbarkeit der Komponenten kann der Entwicklungsprozess langfristig beschleunigt und die Fehleranfälligkeit reduziert werden. Allerdings gibt es dabei andere Schwerpunkte, worauf geachtet werden muss. Bei größeren Änderungen an Services muss darauf geachtet werden, dass die gesamte Applikation mit allen Services noch funktioniert. Gegebenenfalls müssen dabei noch andere Services angepasst werden. Schwerwiegender wird das Problem, wenn der Service in mehreren Applikationen verwendet wird. Dies ist ein weiterer Grund für die Wichtigkeit der Planungsphase bei einer SOA.

Neben der Modularität bringt auch die Wartbarkeit aus der Sicht des Entwicklungsprozesses langfristig deutliche Vorteile. Durch die Aufteilung in kleine Services können ohne Beachtung von anderen Services, Updates oder Erweiterungen für einen Service implementiert werden. Späteres Refactoring wird dank simplen Services anstelle einer komplexen monolithischen Applikation ebenfalls deutlich vereinfacht. Bei Funktionsupdates können dabei auch weitere Services ohne Probleme implementiert werden. Bei komplexen monolithischen Applikationen ist bei Funktionsupdates mit einem deutlichen Mehraufwand zu rechnen. Dies trifft vor allem zu, wenn einer Applikation über die Zeit immer weitere

Funktionalitäten ohne ein größeres Refactoring hinzugefügt werden oder sich viele Altlasten in dieser befinden. [Na16]

5 Auswirkungen einer SOA auf den Auslieferungsprozess

Auslieferung von Software beschreibt den Prozess der Verteilung, Installation beziehungsweise Aktualisierung und Konfiguration von Software für den produktiven Einsatz. Üblicherweise werden hierbei Softwarepakete mit zugehörigen Nutzungsrechten vom Softwarehersteller an den Softwarebetreiber übertragen.
Dabei existieren völlig unterschiedliche Herangehensweisen für den Auslieferungsprozess. Je nach Art der auszuliefernden Software und dem späteren Einsatz kann entweder manuell oder automatisiert ausgeliefert werden. Dabei existieren unzählige Zwischenstufen mit unterschiedlichen Automatisierungsgraden. Des Weiteren kann Software beispielsweise entweder über ein Netzwerk oder über ein physisches Installationsmedium ausgeliefert werden. Auch die Art und Frequenz der Durchführung von Aktualisierungen kann sich erheblich unterscheiden. Darüber hinaus existieren viele weitere Möglichkeiten und Eigenschaften, in welchen sich der Auslieferungsprozess eines Softwaresystems unterscheiden kann.
Viele der Entscheidungen für die Gestaltung des Auslieferungsprozesses hängen weniger von der Architektur der Software ab als viel mehr von der Art des Systems und von den Wünschen des Kunden. Daher werden viele mögliche Eigenschaften des Auslieferungsprozesses in diesem Abschnitt nicht behandelt. Dennoch ergeben sich aus der Wahl der Softwarearchitektur unterschiedliche Möglichkeiten beziehungsweise Vor- und Nachteile, welche insbesondere bei der Wahl der Verteilungsstrategie ausschlaggebend sind. Einige Verteilungsstrategien werden im nächsten Abschnitt vorgestellt.

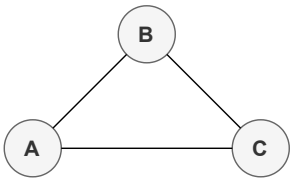
Eine wesentliche Eigenschaft von Services ist ihre Eigenständigkeit und Unabhängigkeit. Um diese Eigenschaften auch im Prozess der Auslieferung von serviceorientierten Systemen in größtmöglichem Umfang beibehalten zu können, werden Services üblicherweise in virtuellen Maschinen (VMs) gekapselt und ausgeliefert. VMs sind komplexe Softwaresysteme, die eine isolierte Umgebung auf einem Rechnersystem darstellen. Sie emulieren physische Maschinen und stellen eine Umgebung bereit, welche exakt wie die physische Maschine reagiert, unabhängig von dem tatsächlichen physischen System auf welchem die VM läuft. Der große Vorteil von Virtualisierung ist, dass jegliche Software innerhalb einer VM zunächst vollständig vom außenstehenden System isoliert ist. Zum einen ist ein Service somit von unerwünschtem Zugriff von außen geschützt, da Zugriffsmöglichkeiten zunächst explizit freigegeben werden müssen. Zum anderen ist die Software innerhalb der VM völlig unabhängig von dem physischen System, auf welchem diese läuft, sodass die Software mitsamt der VM auf einen beliebigen anderen virtualisierten Host portiert werden kann. Somit ist ein Service nicht nur positionsunabhängig adressierbar, sondern auch positionsunabhängig ausführbar.
Für Microservices werden außerdem oftmals Container für die Kapselung genutzt. Diese haben den Vorteil, dass sie wesentlich leichtgewichtiger sind und somit in sehr dynamischen

Systemen eine schnellere Skalierung mit weniger Overhead ermöglichen. Jedoch ist die Software innerhalb eines Containers nicht vollständig isoliert, da beispielsweise mehrere Container auf einem Host denselben Kernel nutzen. Daraus resultieren erhebliche Einbußen hinsichtlich der Sicherheit und Isoliertheit eines serviceorientierten Systems, da sich Software, die in separaten Containern jedoch auf demselben Host läuft, gegenseitig über den Host-Kernel korrumpieren kann.
In beiden Fällen wird jedoch eine Kapselung der verwendeten Technologie erreicht, sodass für die Auslieferung und Inbetriebnahme lediglich die API der Virtualisierungs- beziehungsweise Containerisierungslösung bekannt sein muss. Es wird kein weiteres Wissen hinsichtlich der Technologie des Services benötigt. Die Vorteile im Auslieferungsprozess werden allerdings mit erhöhter Komplexität beim Bauprozess aufgewogen, da hier zusätzlich die Images für die VMs oder Container erstellt werden müssen.
Die Verwendung einer SOA und die gewonnene Flexibilität durch die daraus resultierende lose Kopplung ermöglicht zusätzlich eine wesentlich agilere Auslieferung von Aktualisierungen und flexiblere Möglichkeiten für die Skalierung eines Softwaresystems.
Im Folgenden werden diese Möglichkeiten genauer erläutert.

5.1 Skalierung

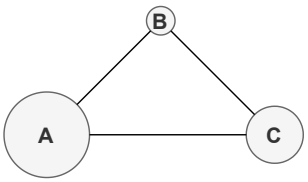
Ein System, bestehend aus autonomen Services kann sowohl horizontal als auch vertikal unproportional skaliert werden. Unproportionale Skalierung meint hierbei, dass einzelne Services des Systems in beliebige Richtung skaliert werden können, während andere Services ihre Kapazität halten. Somit kann sich das Gesamtsystem dynamisch an beliebige Lastszenarien anpassen [Na16].
Weitere oder leistungsfähigere Serviceinstanzen können somit bei Bedarf ausgeliefert und zugeschaltet werden. Man nennt dieses Vorgehen auch *deploy on demand*.
Dabei sollte beachtet werden, dass eine Skalierung lediglich dann erfolgen sollte, wenn mittel- oder langfristige Lastveränderungen im System auftreten oder eine Lastveränderung aufgrund eines vorhersehbaren Ereignisses bevorsteht. Die ständige Anpassung an kurzfristige Szenarien sorgt ansonsten für enormen Overhead aufgrund der Auslieferung und Inbetriebnahme beziehungsweise dem Herunterfahren von Services. Zusätzlich erfolgen ständige Änderungen im Lastenverteilungssystem durch zu- beziehungsweise abschalten neuer Services oder Ressourcen.
Vertikale Skalierung beschreibt in diesem Kontext die Erhöhung beziehungsweise Reduktion der verfügbaren Ressourcen eines Services. Dies kann beispielsweise bedeuten, eine Serviceinstanz durch eine andere zu ersetzen, die auf einem leistungsfähigeren Host läuft oder die zugeteilten Ressourcen der VM, in welcher die Serviceinstanz läuft, zu erhöhen [Na16].
Bei der horizontalen Skalierung werden mehrere Instanzen des gleichen Services eingerichtet und zugeschaltet. Die Gesamtlast des Systems wird dann über ein Lastenverteilungssystem auf die unterschiedlichen Instanzen aufgeteilt.
Abbildung 7 stellt die unterschiedlichen Skalierungsmöglichkeiten bildlich dar.

Belastung der einzelnen Services		
A	**B**	**C**
mittel	mittel	mittel



Unproportionale vertikale Skalierung

Belastung der einzelnen Services		
A	**B**	**C**
hoch	niedrig	mittel



Unproportionale horizontale Skalierung

Belastung der einzelnen Services		
A	**B**	**C**
hoch	mittel	mittel

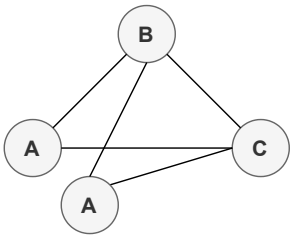


Abb. 7: Darstellung der beispielhaften Skalierungsmöglichkeiten eines Systems bestehend aus drei Services A, B und C, entsprechend bestimmter Lastszenarien

5.2 Aktualisierung

Wie bereits im Zusammenhang mit dem Entwicklungsprozess erläutert, bietet die lose Kopplung viele Möglichkeiten für die agile Weiterentwicklung eines serviceorientierten Systems. Einzelne Services können beispielsweise vollständig unabhängig voneinander weiterentwickelt und aktualisiert werden.
Daraus resultieren zumeist wesentlich frequentiertere Rollouts. Änderungen können auf Serviceebene ausgeliefert werden. Somit kann das Gesamtsystem kleinschrittig weiterentwickelt und aktualisiert werden, ohne dass jemals das komplette System neu aufgesetzt werden muss. Dabei ist zu beachten, dass dies nur möglich ist, wenn die Schnittstellen der einzelnen Services lediglich erweitert, nicht aber verändert oder verkleinert werden. Ansonsten kann es zu Inkompatibilitäten innerhalb des Gesamtsystems kommen.

Im Folgenden werden nun drei Verteilungsstrategien für die Auslieferung von Aktualisierungen eines SOA Systems vorgestellt [St21].

Rollende Auslieferung

Bei der rollenden Auslieferung von Services werden in einer geklusterten Umgebung nach und nach einzelne Hosts aus der Produktivumgebung genommen. Die neue Version wird auf dem Host eingerichtet. Anschließend wird der Host wieder in die Produktivumgebung eingepflegt. Dies ermöglicht die schrittweise Aktualisierung des gesamten Systems.
Der Vorteil ist, dass nicht direkt alle Instanzen aktualisiert werden, wodurch das Risiko im Fehlerfall begrenzt wird. Außerdem ist die Umsetzung der rollenden Auslieferung relativ einfach. Nachteil an dieser Verteilungsstrategie ist, dass einzelne Hosts zeitweise vom Netz genommen werden.
Abbildung 8 visualisiert den Ablauf einer rollenden Aktualisierung.

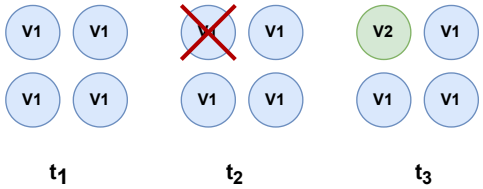


Abb. 8: Ablauf einer rollenden Aktualisierung

Blue-Green Auslieferung

Es wird parallel eine zweite Server-Instanz mit dem aktualisierten Service gestartet. Anschließend werden alle Nachrichten, die an den Service adressiert sind, über ein Lastenverteilungssystem an die aktualisierte Instanz geleitet. Im Falle eines Fehlers wird die Last wieder auf die alte Instanz geleitet. Tritt kein Fehler auf, so kann die alte Instanz heruntergefahren werden.
Vorteil dieser Strategie ist, dass praktisch keine Downtime auftritt. Außerdem kann ein Rollback relativ einfach realisiert werden, indem die Last wieder auf die alte Instanz geleitet und die aktualisierte Instanz wieder heruntergefahren wird.
Der Nachteil dieser Strategie ist, dass im Falle eines Fehlers 100 % der Benutzer betroffen sind, bis der Fehler erkannt und die Last wieder auf die alte Instanz umgeleitet ist.
Abbildung 9 visualisiert den Ablauf einer Blue-Green Aktualisierung.

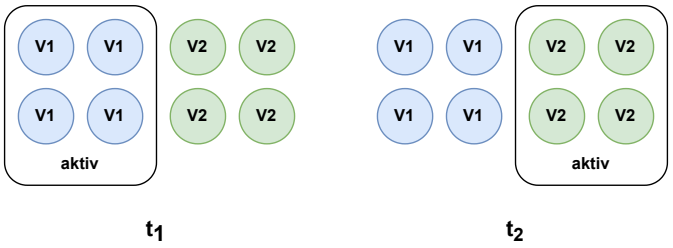


Abb. 9: Ablauf einer Blue-Green Aktualisierung

Canary Auslieferung

Die Canary Strategie ähnelt der Blue-Green Strategie, soll jedoch deren Probleme bei Auftreten eines Fehlers beheben. Sie kann angewendet werden, wenn jeweils mehrere Instanzen der zu aktualisierenden Services im System aktiv sind. Bei der Canary Strategie werden zunächst nur wenige aktualisierte Serviceinstanzen gestartet. Ein Teil der Anfragen werden anschließend über das Lastenverteilungssystem auf die neuen Instanzen geleitet. Treten bei der Benutzung keine Fehler auf, werden anschließend alle zu aktualisierenden Services im Blue-Green Verfahren ersetzt. Die Verwendung der Canary Strategie ist nur möglich, wenn gleichzeitig mehrere Versionen eines Services laufen können.
Vorteil dieser Strategie ist, dass im Fehlerfall nur eine Teilmenge der Benutzer betroffen ist. Außerdem gelten die gleichen Vorteile, die bereits bei der Blue-Green Strategie genannt wurden.
Der Nachteil dieser Strategie ist eindeutig die Komplexität in der Umsetzung. Die Verwendung der Canary Strategie ist nur zu empfehlen, wenn sie über ein Continuous Delivery (CD) System automatisiert ist.
Abbildung 10 visualisiert den Ablauf einer Canary Aktualisierung.

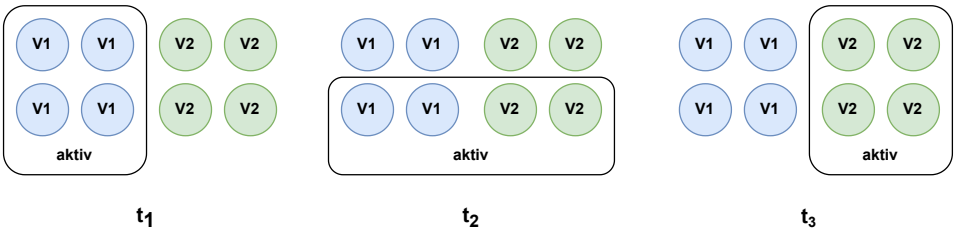


Abb. 10: Ablauf einer Canary Aktualisierung

5.3 Zusammenfassung der Auswirkungen

Es wurden unterschiedliche Aspekte der Auslieferung von serviceorientierten Systemen beleuchtet. Im Folgenden werden nun die konkreten Auswirkungen einer SOA auf den Auslieferungsprozess von Softwaresystemen zusammengefasst.

Zunächst ist die Einrichtung der einzelnen Services, unabhängig der Implementierungs-Technologie über das API der Virtualisierungs- oder Containersoftware ein wichtiger Punkt. Virtualisierung wird zwar auch bei Systemen mit anderen Architekturen verwendet, ist aber zentral für SOA, da sie homogene Auslieferungs- und Installationsprozesse für heterogene Services ermöglicht.

Auch die verteilte Auslieferung der Systeme und die daraus resultierende erhöhte Komplexität des Auslieferungsprozesses ist eine zentrale Auswirkung der Verwendung einer SOA. Alle Services des Systems müssen einzeln ausgeliefert werden und werden zudem üblicherweise auf unterschiedlichen Hosts eingerichtet. Dadurch wird der Auslieferungsprozess wesentlich komplexer als beispielsweise die Auslieferung eines monolithischen Systems. Um diese Komplexität zu verringern, wird zunehmend *serverlos* ausgeliefert. Dabei wird der Code der Services an einen Anbieter für serverlose Auslieferung weitergegeben, welcher dann beliebig viele Instanzen in gewünschter Konfiguration auf eigener Infrastruktur ausliefert. Dabei übernimmt dieser Anbieter die gesamte Verwaltung der Infrastruktur und stellt die Funktionalität der Services über ein Netzwerk bereit. Die Verwendung einer serverlosen Auslieferung hat den Vorteil, dass der Auslieferungsprozess erheblich vereinfacht wird und keine eigenen Hosts mehr verwaltet werden müssen. Im Austausch mit diesen Vorteilen stehen die Vertrauenswürdigkeit der Anbieter und die Tatsache, dass der Herausgeber der Services keinerlei Kontrolle darüber hat, wo diese laufen [St21].

Auch die Auswirkungen auf die Skalierung eines Systems wurden bereits beschrieben. Bei der Auslieferung von monolithischen Systemen wird beispielsweise eine Instanz des Systems auf einem Host eingerichtet. Um diese Systeme skalieren zu können, wird je nach Art des Systems entweder die bestehende Instanz auf einen leistungsfähigeren Host verschoben oder es werden weitere Instanzen des Systems auf anderen Hosts aufgesetzt, die sich die vorhandene Last anschließend teilen.
Bei serviceorientierten Systemen werden die einzelnen Services hingegen einzeln ausgeliefert und skaliert.
Für die flexible Skalierbarkeit der Systeme und dem damit einhergehenden *deployment on demand* wird eine Art Service Provider benötigt, der bei Bedarf weitere oder leistungsfähigere Serviceinstanzen bereitstellen und ausliefern kann [Na16]. Diese Möglichkeiten führen zu deutlich erhöhter Komplexität der Infrastruktur, wenngleich sie enorme Vorteile für die Flexibilität eines Systems bieten.

Eine weitere Auswirkung der SOA auf den Auslieferungsprozess sind wesentlich kleinschrittigere Aktualisierungen und daraus resultierend wesentlich frequentiertere Auslieferungen ohne Downtime des Systems (beispielsweise durch Blue-Green oder Canary Auslieferung).

Insgesamt ist also erkennbar, dass die Verwendung einer SOA zu einem komplexeren Auslieferungsprozess führt. Um diese Komplexität handhabbar zu machen, werden zumeist umfangreiche DevOps Systeme für die Unterstützung im Entwicklungs- und Auslieferungsprozess verwendet.
Diese führen bei zielgerichteter Benutzung zu einer besseren Organisation, die sich bereits im Entwicklungsprozess positiv auswirkt. So ist beispielsweise eine gut organisierte Quellcodeverwaltung eine essenzielle Basis, um die isolierte Entwicklung homogener Services gut strukturiert durchführen zu können und aus diesen zu einem späteren Zeitpunkt komplexe Systeme zusammensetzen zu können. Darüber hinaus unterstützen diese Systeme das Konfigurations- und Releasemanagement, um die Zusammensetzung, Parametrisierung und Versionierung der serviceorientierten Softwarelösungen zu automatisieren. Dies reduziert Fehler und Inkonsistenzen, sowohl im Stadium der Entwicklung, als auch im Auslieferungsprozess. Eine gute Organisation und konsistente Daten sind beispielsweise auch nötig, um Systeme für Wartungsarbeiten, Recovery- und Testszenarien reproduzierbar zu machen.
In vielen DevOps Systemen lassen sich außerdem sogenannte *Pipelines* einrichten, über welche umfangreiche Ausführungsfolgen automatisiert werden können. Eine solche Pipeline kann zum Beispiel nach Fertigstellung einer Aktualisierung automatisiert gestartet werden. Im Durchlauf der Pipeline werden dann beispielsweise die Paketierung und Einrichtung der Software in einer VM angestoßen werden und der aktualisierten Service wird als VM-Image für den Kunden bereitgestellt oder direkt auf dessen Zielhosts ausgerollt.

Die Möglichkeit, Systeme in Form von einzelnen Services auszuliefern und dadurch unabhängig voneinander erweitern, skalieren und aktualisieren zu können, birgt je nach Anwendungsfall also enorme Vorteile, führt im gleichen Zug jedoch auch zu erheblicher Steigerung der Komplexität bei der Auslieferung. Diese Komplexität kann durch geeignete Hilfssysteme zwar wieder reduziert werden, dies ist jedoch mit sehr hohem initialen Aufwand verbunden.
Die Verwendung einer SOA lohnt sich hinsichtlich der Auswirkungen auf den Auslieferungsprozess also nur, wenn die dadurch gewonnenen Vorteile auch ausgeschöpft werden.

Literatur

[Ad10] Adnan Gohar: Analyzing Service Oriented Architecture (SOA) in Open Source Products, Master Thesis, School of Innovation, Design and Engineering, 7.10.20210, URL: https://www.diva-portal.org/smash/get/diva2:360992/FULLTEXT01.pdf.

[ADM06] Ang, H.-W.; Dandashi, F.; McFarren, M.: Tailoring DoDAF For Service-Oriented Architectures. In: MILCOM 2006 - 2006 IEEE Military Communications conference. S. 1–8, 2006, ISBN: 2155-7586.

[Ax22] Axiomatics, Hrsg.: eXtensible Access Control Markup Language (XACML), 3.08.2022, URL: https://axiomatics.com/resources/reference-library/extensible-access-control-markup-language-xacml.

[Bo93] Booch, G.: Object-Oriented Analysis and Design with Applications (2nd Ed.) Benjamin-Cummings Publishing Co., Inc, USA, 1993, ISBN: 0805353402.

[Du22] Duden: monolithisch auf Duden online, 2022, URL: https://www.duden.de/rechtschreibung/monolithisch.

[Er09a] Erl, T.: Service-oriented architecture: Concepts, technology, and design. Prentice Hall PTR, Upper Saddle River, NJ und Munich, 2009, ISBN: 0131858580.

[Er09b] Erl, T.: SOA: Principles of service design. Prentice Hall, Upper Saddle River, NJ, 2009, ISBN: 9780132344821.

[Fr] Frank Buschmann; Regine Meunier; Hans Rohnert; Peter Sornmerlad; Michael Stal: Wiley - Pattern-Oriented Software Architecture: A System of Patterns, Volume 1./.

[G 07] G. A. Lewis; E. Morris; S. Simanta; L. Wrage: Common Misconceptions about Service-Oriented Architecture. In: 2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07). S. 123–130, 2007.

[Gi22] Gillis. Alexander S.: Simple Object Access Protocol (SOAP), 2022, URL: https://www.computerweekly.com/de/definition/Simple-Object-Access-Protocol-SOAP.

[He07] Heutschi, R.: Serviceorientierte Architektur: Architekturprinzipien und Umsetzung in die Praxis ; mit ... 52 Tabellen: St. Gallen, Univ., Diss., 2007. Springer, Berlin und Heidelberg, 2007, ISBN: 978-3-540-72357-8.

[HR10] Haines, M. N.; Rothenberger, M. A.: How a service-oriented architecture may change the software development process. Communications of the ACM 53/8, S. 135–140, 2010, ISSN: 0001-0782.

[KOS06] Kruchten, P.; Obbink, H.; Stafford, J.: The Past, Present, and Future for Software Architecture. IEEE Software 23/2, S. 22–30, 2006, ISSN: 0740-7459.

[LF04] Lascelles, F.; Flin, A.: WS security performance. Secure conversation versus the X509 profile. 2004.

[Na16] Nadareishvili, I.; Mitra, R.; McLarty, M.; Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture./, 2016, URL: https://docs.broadcom.com/doc/microservice-architecture-aligning-principles-practices-and-culture.

[PMA19] Ponce, F.; Márquez, G.; Astudillo, H.: Migrating from monolithic architecture to microservices: A Rapid Review: Concepción, Chile, November 4-9, 2019. In: 2019 38th International Conference of the Chilean Computer Science Society (SCCC). S. 1–7, 2019, URL: http://ieeexplore.ieee.org/servlet/opac?punumber=8956485.

[Ra05] Rausch, T.: Service Orientierte Architektur: Übersicht und Einordnung, 26.12.2005, URL: https://web.archive.org/web/20081010033719/http://www.till-rausch.de/assets/baxml/soa_akt.pdf.

[Ro12] Rotem-Gal-Oz, A.: SOA Patterns. Manning, 2012, ISBN: 978-1933988269.

[SDS04] Stojanovic, Z.; Dahanayake, A.; Sol, H.: Modeling and design of service-oriented architecture. In (Wieringa, P., Hrsg.): 2004 IEEE international conference on systems, man & cybernetics theme. IEEE, Piscataway (N.J.), S. 4147–4152, op. 2004, ISBN: 0-7803-8567-5.

[SHM08] Sanders, D. T.; Hamilton Jr, P. J.; MacDonald, P. R. A.: Supporting A Service-Oriented Architecture./, 2008, URL: https://dl.acm.org/doi/pdf/10.5555/1400549.1400595.

[SM09] Savolainen Juha; Myllarniemi Varvana: Layered architecture revisited — Comparison of research and practice. In: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. S. 317–320, 2009.

[So22] Solutions, Martin Schmidt - Asgard: Serviceorientierte Architektur (SOA) - Knowledge Base - Asgard Solutions, 26.11.2022, URL: https://www.asgard-solutions.de/KnowledgeBase/Softwaretechnik/Serviceorientierte-Architektur-SOA.aspx.

[St21] Storz, C.: The Journey to Microservices & Deployment Strategies: From monolith to microservices! Learn all about microservices & deployment strategies, hrsg. von harness.io, harness.io, 2021, URL: https://harness.io/blog/microservices-deployment-strategies.

[Wi06] Wisniewski, T.; Whitehead, G.; Hinton, H.; Cahill, C.; Cantor, I.; Nate; Klingenstein; Rl, B.; Morgan; John; Bradley; Individual, J.; Hodges; Individual, J.; Brennan, L.; Alliance, E.; Tiffany, L.; Alliance, T.; Hardjono, M.; Trustgenix: Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2. 0–Errata Composite./, 2006.

[ZM09] Ziegler, S.; Müller, A.: Service-orientierte Architekturen: Leitfaden und Nachschlagewerk./, 2009, URL: https://www.bitkom.org/sites/default/files/file/import/bitkom-soa-leitfaden.pdf.