Design-Analyse und Performancemessungen von Microservicearchitekturen für die Transformation von Desktopanwendungen zu Microservices

Bachelorarbeit zur Erlangung des Bachelor-Grades
Bachelor of Science im Studiengang Medientechnologie
an der Fakultät für Informations-, Medien- und Elektrotechnik
der Technischen Hochschule Köln

vorgelegt von:	Nataliya Elchina
Matrikel-Nr.:	11134108

eingereicht bei:	Prof. Dr. Björn Krüger
Zweitgutachter:	Tobias Franz

Köln, 30.08.2023

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, 30.08.2023 — Ort, Datum

N. Elchina — Rechtsverbindliche Unterschrift

Kurzfassung/Abstract

Die Bachelorarbeit widmet sich der Designmuster-Analyse von Microservices im Kontext der Transformation von Desktopanwendungen zur Cloud-Anwendung. Insbesondere wird die Relevanz von Microservices in der Cloud hervorgehoben und die Notwendigkeit betont, Performance-Aspekte bei der Architekturumstellung zu berücksichtigen. Die Untersuchung erfolgt an einer Desktop-basierten MBSE-Software, die frühzeitige Phasen einer Raumfahrtmission abbildet. Durch die Analyse von Designmustern sowie gezielten Performancemessungen wird ein Ansatz erarbeitet, um eine Migration von Desktopanwendungen zu Microservices zu ermöglichen.

Schlüsselwörter: Microservices, Cloud Computing, Designmuster, Analyse Performance, Raumfahrt

The bachelor's thesis focuses on the design pattern analysis of microservices in the context of transforming desktop applications into cloud-based applications. In particular, it highlights the significance of microservices in the cloud and emphasizes the necessity of considering performance aspects during the architectural transition. The investigation is conducted on a desktop-based MBSE software that models early phases of a space mission. Through the analysis of design patterns and targeted performance measurements, an approach is developed to facilitate the migration from desktop applications to microservices.

Keywords: Microservices, Cloud Computing, Design Patterns, Performance Analysis, Spaceflight

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Für die Erstellung von Raumfahrtmissionen, ist eine sorgfältige und genaue Planung des Flugkörpers notwendig. Daten und Informationen müssen vorerst entsprechend geplant, berechnet und verifiziert werden, bevor die Montage und der darauffolgende Raketenstart die Raumfahrtmission in ihre Umlaufbahn in das Weltall bringen kann.

Im Deutschen Zentrum für Luft- und Raumfahrt am Standort Braunschweig wurde die Software Virtual Satellite für die Planung der ersten Phasen von Raumfahrmission entwickelt. Erstellt und gewartet von der Projektgruppe Modellierung und Simulation am Institut für Softwaretechnologie, ist die Anwendung eine zentrale Entwicklungsumgebung für den Entwurf, Konfiguration und Verifizierung von Daten und Komponenten eines Satelliten. Hauptsächlich angewendet wird die Software in der Concurrent Engineering Facility (CEF) in Bremen. Das CEF ist ein Entwurfslabor für frühzeitige Phasen von Raumfahrtmissionen, in denen Ingenieure gemeinsam an der Planung des Flugkörpers arbeiten. Dabei setzen die Mitarbeiter auf das Konzept von concurrent engineering, welches ihnen die Vorgehensweise ermöglicht, gleichzeitig an einem Entwicklungsprozess zu arbeiten.

Aktuell ist Virtual Satellite eine Desktopanwendung und nur für Endnutzer verwendbar, wenn sie diese auf ihrem Rechner installieren. Um eine benutzerfreundlichere und flexiblere Benutzung von der Anwendung zu ermöglichen, ist es das Ziel, die Software mit ihren Systems Engineering Werkzeugen in eine Webanwendung umwandeln. Dabei ergeben sich viele Fragen. Was ist der performanteste und effizienteste Ansatz, die Software in eine Cloud-Architektur umzuwandeln? Wie kann festgestellt werden, wie sich die Anwendung, sowie die vielen, komplexen Berechnungsketten auf verteilten Webservices verhalten? Inwieweit kann die Anwendung skaliert werden, welche Grenzen werden erreicht und welchen Einfluss haben verschiedene architektonische Designmuster?

1.1 Motivation

In den letzten Jahren hat sich das Konzept von Microservices als eine effiziente Softwarearchitektur für die Cloud etabliert [1].

Microservices sind eine Architekturform, die mit ihrer Struktur eine Segmentierung von Softwareanwendungen in autonome, unabhängige Dienste ermöglichen kann [2]. Diese Dienste können beliebig skaliert werden, um erhöhte Lasten zu bewältigen [3].

Durch die Aufteilung der Anwendung in einzelne Service und die dadurch entstehende modulare Struktur, können die einzelnen Komponenten wartungsfreundlicher aktualisiert, verändert und deployed werden [2]. Dies führt zu einer Verbesserung der Agilität im Entwicklungsprozess [4]. Entwickler können gezielt an bestimmten Diensten arbeiten, ohne die Gesamtintegrität der Anwendung zu beeinträchtigen [4].

Die damit einhergehende Dezentralisierung von Microservices begünstigt die Ausfallsicherheit einer Anwendung [4]. Die Modularität und unabhängige Arbeitsweise der Dienste voneinander unterstützt die Minimierung von Fehlern, da mögliche Dienstausfälle oder Defekte in der Architektur nicht die vollständige Anwendung beeinflussen [4]. Andere Service können die ausgefallene Funktionen oder den Datenaustausch übernehmen [4].

Komponenten können wiederverwendet werden und andere externe Service können

durch eine Schnittstelle, wie z.B eine API, integriert werden [5]. Dadurch erlauben Microservices einen einfachen Datenaustausch, sowie auch die Entwicklung von plattformübergreifender Anwendungen [5].

Die Transformation zu einer Microservice-Architektur eröffnet somit viele neue Möglichkeiten monolithisch strukturierte Anwendungen in eine webbasierte Lösung umzuwandeln [4]. Insbesondere Desktopanwendungen nehmen stark von dieser Umwandlung Gebrauch [4]. Endnutzern wird der Zugriff auf die Anwendung über verschiedene Geräte und Plattformen hinweg ermöglicht, ohne dass eine Installation auf jedem Gerät erforderlich ist [4].

1.2 Ziel der Arbeit

Das Ziel der Bachelorarbeit ist es, den aktuellen Forschungsstand von Microservices zu ermitteln, um mit dem Wissen eine Designmuster-Analyse und Performancemessungen für die Software Virtual Satellite durchzuführen. Die erzielten Ergebnisse dienen als Grundlage zur Beurteilung der Migration von einer Desktop-basierten MBSE-Anwendung zu einer skalierbaren Cloud-Architektur.

Anhand der Design-Analyse soll eine mögliche Transformation der Desktop-Architektur zur Cloud-Anwendung vorgelegt werden. Dabei sollen verschiedene Aspekte und Schritte evaluiert werden, wie die Migration zur Cloud-Architektur stattfinden kann und welche Entwurfsmuster für die Anforderungen für Virtual Satellite in der Cloud maßgebend sind. Zur Beurteilung werden dabei insbesondere verschiedene Kommunikationsmechanismen evaluiert, sowie Datenverwaltungsmöglichkeiten und Deploymentstrategien.

Im Fokus der Arbeit liegt dabei auch die Fragestellung, mit welcher Leistung Anfragen in einer Microservice-Architektur verarbeitet werden können und welche Faktoren die Performance einer Microservice-Architektur beeinflussen. Mit einem Prototypen werden dafür Perfomancemessungen durchgeführt, um eine erste Einschätzung darüber zu erhalten, wie sich ein Service im Bezug auf Virtual Satellite verhält und festzustellen, ob Berechnungen, wie sie in der CEF stattfinden, performant durchgeführt werden können.

1.3 Aufbau der Bachelorarbeit

Die Bachelorarbeit ist in mehrere Kapitel aufgegliedert. Das Kapitel 2 beschreibt den aktuellen Forschungsstand und erläutert architektonische Muster für die Implementierung einer Microservice-Architektur. Im Kapitel 3 wird unter Berücksichtigung einer Anforderungsanalyse für Virtual Satellite eine Designmuster-Analyse entworfen. Es werden Komponenten erläutert, die die Umwandlung einer Desktop- zur Cloud-Anwendung ermöglichen. Für eine erste Einschätzung über die Performance eines Services wird ein Prototyp implementiert, welcher in Kapitel 4 beschrieben ist. Anschließend wird in Kapitel 5 die Testphase konzipiert und in Kapitel 6 werde die Ergebnisse dargestellt. Es folgt darauf eine Diskussion in Kapitel 7. Mit einem Ausblick und Fazit werden im letzten Abschnitt die Ergebnisse der Bachelorarbeit evaluiert und zusammengefasst.

2 Literatur

2.1 Modellbasierte Systementwicklung

Modellbasiertes Systems Engineering (MBSE) definiert die Methodik, Systemanforderungen und Design-Analysen, sowie Verifikationsverfahren und Validierungen von Entwicklungsprozessen mithilfe von Modellen durchzuführen [6].

Das Ziel von MBSE ist es, die Entwicklung von komplexen Systemen übersichtlich und effizient zu halten [6]. Ingenieure, die an einem System arbeiten, sollen bei den Entwicklungsprozessen gemeinsam die Zusammenhänge zwischen den Systemanforderungen nachvollziehen und Änderungen über die Design-Prozesse automatisch miteinander teilen können [6]. MBSE bietet somit einen effizienten Verwaltungsaufwand bei Systementwicklungen und fördert eine frühzeitige Fehlererkennung [6]. Dies ermöglicht es den beteiligten Ingenieuren zu Beginn der Entwicklung die Komplexität und Kommunikation eines Projektes zu lenken [6].

Im Kontext von Raumfahrtmissionen beschreibt Modellbasiertes Systems Engineering das Designen und Modellieren von Raumfahrtmission, sowie den Austausch der zugehörigen Informationen und Daten zwischen den Ingenieuren für die Planung der Systementwicklung [7]. So werden innerhalb eines Projektes, verschiedene Daten und Systemeigenschaften untereinander anhand eines Systemmodells ausgetauscht, welches die zentrale Aufgabe übernimmt, Daten über mehrere Prozesse zugänglich zu machen [7]. Das Systemmodell wird dabei mittels der Sprache einer konzeptuellen Datenmodells beschrieben (CDM), auch genannt als Meta-Modell [7]. Dieses konzeptuelle Datenmodell gibt eine bestimmte Syntax und Semantik vor, mit der die Systemmodellsprache erstellt werden kann und womit die Konzepte für das Designen einer Raumfahrtmission unter den Ingenieuren gemeinsam zusammengefasst und ausgetauscht werden können [8].

Bei der Erstellung des eigentlichen Systemmodells werden anhand von abstrakten Komponenten und den zugehörigen Interface-Schnittstellen der CDM, explizite Instanzen eines Raumfahrzeuges erstellt und miteinander verknüpft [7]. So kann zum Beispiel eine Instanz eines On-Board Computers als Komponente mit mehreren Interface-Endpunkten zu mehreren Reaktionsrädern verbunden werden [7].

Die Entwicklung einer Raumfahrtmission bis hin zur Herstellung erfolgt in mehreren Phasen die den Lebenszyklus definieren [7]. Diese sind von den Standards der European Cooperations for Space Standardization (ECSS), der European Space Agency (ESA) und der National Aeronautics and Space Administration (NASA) definiert [7]. Die Entwicklung teilt sich dabei in insgesamt sieben Phasen auf, die die unterschiedlichen Aufgaben, Zustände und Ziele einer Mission beschreiben [7]. Die Phasen werden dabei in Phase 0, A, B, C, D, E und F benannt [7]. In den ersten Phasen werden dabei Machbarkeitsstudien für die Raumfahrtmission durchgeführt [7]. Diese führen dann über detaillierte Entwurfsplanungen zu der Montage bis hin zum Betrieb und der anschließenden Entsorgung [7].

Das Institut für Softwaretechnologie im Deutschen Zentrum für Luft- und Raumfahrt hat die Software *Virtual Satellite* entwickelt, die das Konzept von MBSE umsetzt und eine konzeptuelle Datenbank definiert [7]. Mit Virtual Satellite werden frühzeitige Phasen

umgesetzt, die die Modellierung und Planung einer Raumfahrtmission erlaubt [7]. Der Einsatz der Software fokussiert sich hauptsächlich auf das Entwurfslabor für das concurrent enginnering (CEF) in Bremen, wo Virtual Satellite den dort arbeitenden Ingenieuren die Möglichkeit bietet, anhand von Design-Prozessen das System einer Raumfahrtmission zu planen [7].

2.1.1 Concurrent engineering in der CEF

Der Begriff *concurrent engineering* bezeichnet den Prozess einer gleichzeitigen Bearbeitung von Aufgaben innerhalb eines Projekts [8]. Mögliche Probleme sollen dabei frühzeitig erkannt werden, sodass Projekte von Raumfahrtmissionen bereits zu Beginn erkannt werden können [8].

Beim concurrent engineering arbeiten mehrere Ingenieure in einer Einrichtung oder einem Labor an einem gemeinsamen Projekt zusammen [8]. Solch eine erste Einrichtung für die Entwicklung von Raumfahrtmissionen wurde 1995 für NASA's Jet Propulsion Laboratory errichtet [8]. Europäische Forschungseinrichtungen haben sich an der von der ESA gegründeten Concurrent Design Facility (CDF) im Jahr 1998 ein Beispiel genommen [8]. Es zeigte sich, das das Konzept von CE eine positive Minimierung des Zeitaufwand für das Designen von Raumfahrtmissionen ermöglicht [8]. So hat auch das Deutsche Zentrum für Luft- und Raumfahrt ein Entwurfslabor für das Concurrent Engineering einer Raumfahrtmission entwickelt [8]. Die Concurrent Engineering Facility (CEF) in Bremen, gibt Ingenieuren die Möglichkeit, Designs von Satelliten zu erarbeiten [8]. Es ist für die Machbarkeitsstudien der Phase A ausgelegt und implementiert das Konzept von Concurrent Engineering anhand des *IPSP Ansatzes* - Initiation, Preparation, Study, Processing [8]. In den ersten beiden Phasen wird die Raumfahrtmission initialisiert und vorbereitet [3]. In der letzten Phase, die Processing Phase, werden die Ergebnisse der Studie zusammengefasst und dann für weitere Zwecke analysiert [8]. Die dritte Phase beschäftigt sich mit der tatsächlichen Machbarkeitsstudie im Entwurfslabor [8].

Das Entwurfslabor ist mit mehreren Arbeitsplätzen ausgestattet, an denen die Machbarkeitsstudien durchgeführt werden [8]. Diese erfolgen anhand eines geregelten Ablaufs und können eine Dauer von bis zu drei Wochen annehmen, abhängig von der Komplexität der Studie und der Teilnehmendenzahl [8]. Am Ende einer solchen Studie ist das Ziel ein vorläufiges Design des Systems entwickelt zu haben [8]. Wobei mehrere 3D Modelle und ein parametrisches Systemmodell mit Subsystemen und der benötigten Ausrüstung vorgelegt werden kann [8]. Des weiteren werden auch Budgets der physikalischen Eigenschaften wie Masse und Leistung eines Systems bestimmt [8].

Für die Umsetzung dieser Machbarkeitsstudien wird eine Software benötigt, die die geforderten Konzepte und Informationen repräsentiert und verwaltet [8]. Diese Anforderungen erfüllt ein konzeptuelles Datenmodell, anhand dessen die Ingenieuren die Möglichkeit haben, deren Arbeit in einem gemeinsamen Systemmodell zusammenzufassen und zu beschreiben [8]. Hierfür wird im CEF die Software Virtual Satellite mit dem implementierten konzeptuellen Datenmodell verwendet [8].

Abbildung 1 Entwurfslabor im CEF [9]

2.1.2 Virtual Satellite

Virtual Satellite ist die Software die im CEF das konzeptuelle Datenmodell beschreibt und das Konzept von Model-Based System Engineering umsetzt [8]. Die Anwendung bietet den Ingenieuren eine zentrale Entwicklungsumgebung zum Entwerfen, Konfigurieren und Verifizieren von Daten und Komponenten, sodass ein erstes Modell einer Raumfahrtmission erstellt werden kann [8].

Implementiert in Java mit der Eclipse Rich Client Platform hat die Software für die Benutzung ein User Interface mit verschiedenen Editoren, wodurch Informationen der Konzepte, sowie Entwicklungswerkzeuge für die Modellierung und Veränderung der Daten angezeigt werden können [8]. Die Software ist mit einer Versionskontrolle ausgestattet, die den Nutzern eine flexible Arbeit mit den Änderungen erlaubt [8].

Damit die Daten des Systemmodells entsprechend strukturiert und miteinander verknüpft werden können, lassen sich in Virtual Satellite hierarchische Baumstrukturen arrangieren [8]. Dadurch kann das Systemmodell in mehrere Subsysteme gruppiert werden, die mit weiteren, einzelnen Elementen ausgestattet sind [8]. Dies ermöglicht es, eine relative Beziehung zwischen den Komponenten herzustellen und die Komplexität des Systems auf eine einfachere Ebene zu bringen [8]. Die Erstellung der verschiedenen Level-Stufen sind Virtual Satellite dabei grenzenlos [8].

Um die Komponenten innerhalb der Baumstrukturen zu beschreiben, werden den Elementen anhand von Parametern, physikalische Eigenschaften wie Masse, Größe oder Leistung zugewiesen [8]. Virtual Satellite nutzt dafür den Standard *QUDV* (Quantities, Units, Dimensions and Values) [8]. Das QUDV ist eine Modellbibliothek für die Nutzung von SySML-Modellen und beschreibt Einheitssystemen wie das International System of Quantites (ISQ) und das International System of Units (SI) [10]. Die Nutzung dieser Bibliothek in Virtual Satellite ermöglicht die Verwaltung der Einheiten bei Berechnungen und Konver-

tierungen [8]. Während einige Ingenieure z.B in Milliwatt rechnen, können andere die Daten in Watt modellieren, ohne das eine Barriere beim Verständnis über die Daten vorliegt [8].

Die Werte der Parameter können dann mit unterschiedlichen mathematischen Berechnungen verarbeitet werden [8]. So kann zum Beispiel eine Gesamtmasse eines Subsystems aus mehreren Elementen berechnet werden [8].

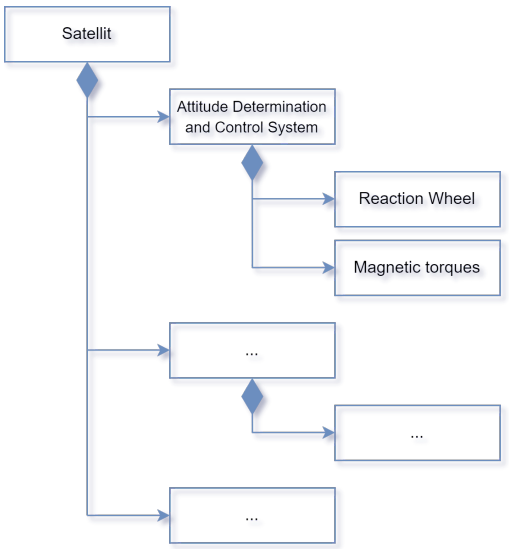

Abbildung 2 Hierarchische Baumstruktur (Angelehnt an [8])

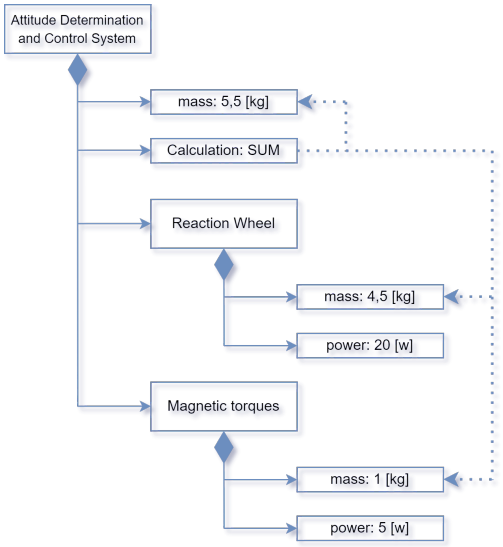

Abbildung 3 Parameterzuweisung (Angelehnt an [8])

Eine weitere Funktion erlaubt die Definition und den Umgang der Zustände von bestimmten Elementen [8]. Eine Kamera zum Beispiel, die an einem Raumfahrzeug angebracht ist, ist während des Start aus und wird erst in Einsatz gebracht, wenn das Raumfahrzeug bereit ist, die Kamera einzusetzen [8]. Der Zustand wenn die Kamera an oder aus ist, hat Auswirkungen auf den Leistungsumsatz und ist bei der Modellierung der Leistung zu beachten [8].

Um einen Überblick über die Masse- und Leistungswerte der Komponenten eines Satelliten zu erhalten, sind sogenannte Massen- und Leistungsbudgets in Virtual Satellite implementiert [8]. Das Massenbudget legt fest, wie viel Masse das System oder seine Komponenten insgesamt haben dürfen, während das Leistungsbudget die maximale Energieaufnahme des Systems definiert [8]. Diese Funktionen sind notwendig, um sicherzustellen, dass das System innerhalb der vorgegebenen Beschränkungen operiert und gleichzeitig alle erforderlichen Funktionen erfüllt [8].

Über die Jahre kann es auch zu Änderungen von Designkonzepten von Raumfahrtmissionen kommen, somit lässt sich das Datenmodell flexibel an neue Konzepte anpassen [7]. Mit Hinblick auf die weiterführende Pläne der Implementierung für Virtaul Satellite, ist das Ziel die Anwendung auch den vollständigen Lebenszyklus einer Raumfahrtmission begleiten zu können [7].

2.2 Microservices

Der Begriff Microservices wurde das erste Mal im Jahr 2011 auf einem Softwarearchitekten Workshop in der Nähe von Venedig eingeführt [2]. Nach Martin Fowler und James Lewis wird die Architektur als ein Ansatz beschrieben, die eine Anwendung in eine Sammlung aus mehreren, kleinen und unabhängigen Dienste unterteilt [2]. Ein Dienst ist dabei eine einzelne, selbständige Komponente die eine eigene Datenbank haben und mit verschiedenen Programmiersprachen sowie unterschiedlichen Datenbanktechnologien implementiert werden können [2]. Die Kommunikation zwischen den Diensten findet dabei mit Kommunikationsprotokollen oder APIs statt [2]. Die Unabhängigkeit der Dienste voneinander ermöglicht es, dass diese einzeln deployed werden können, sodass Änderungen an einem Dienst die anderen Dienste nicht beeinträchtigt oder dazu führt, dass man alle neu deployen muss [2].

Um ein besseres Verständnis über die Microservice-Architektur zu haben, ist es hilfreich, diese mit einem monolithischen Aufbau zu vergleichen [2]. Der Autor von Building Microservices, Sam Newman, beschreibt Monolithen als eine Komponente, die vollständig mit all ihren Funktionen deployed wird [11]. Dabei gibt es drei gängige Konzepte, wie ein solcher Monolith aufgebaut werden kann [11].

Single-Process Monolith Die einfachste und bekannteste Variante eines Monolithen, ist eine Anwendung, bei der die vollständige Software innerhalb eines Prozesses fungiert und auf eine Datenbank zugreift [11]. Solch ein Ansatz ist eine gute Wahl für kleine Organisationen mit kleinem Software Bedarf [11].

Modular Monolith Bei einem modularen Monolithen, besteht die Software aus mehreren Modulen, die aber gemeinsam innerhalb eines Prozesses miteinander agieren [11]. Diese Architektur erlaubt es gleichzeitig an den Modulen Änderungen durchzuführen [11].

Distributed Monolith Ein verteilter Monolith ist eine Anwendung, die einer Microservice-Architektur ähnlich ist, da diese aus mehreren Servicen besteht, jedoch aber eine starke Kopplung aufweist [11]. Solche Architekturen können des öfteren durch den Mangel an Isolation von Daten entstehen [11].

Monolithische Anwendungen haben den Vorteil, dass sie durch deren starke Kopplung aller Funktionen einfacher zum Entwickeln und Deployen sind, sowie das Bestandteile des Codes einfacher wieder zu verwenden sind [11]. Die zusammenhängende Architektur ermöglicht es auch, die Anwendung einfacher zu testen und leichter zu überwachen [11].

Auf der anderen Seite haben monolithische Anwendungen über die Zeit jedoch den Nachteile, dass je größer die Anwendung wird, desto schwieriger und verzweigter wird ihre Komplexität [2]. Änderungen an der Software haben größere Auswirkungen auf die gesamten Funktionen und eine Skalierung der Anwendung ist nur durch eine mehrfache Instanziierung der gesamten Software möglich [2]. Diesem Problem steuert die Microservice-Archtiktur mit ihren einzelnen, voneinander unabhängigen Servicen entgegen und erlaubt somit eine spezifische Skalierung an die geforderten Anforderungen einer Software [2].

Chris Richardson definiert in seinem Buch *Microservices Patterns* die Architektur anhand eines Models für die Skalierung von Software, welches Martin Abbott und Michael Fischer in ihrem Buch *The Art of Scalability* präsentieren [4]. Die beiden Autoren stellen in ihrem

Buch drei verschiedene Möglichkeiten der Skalierung einer Software anhand eines Würfels vor [12].

So beschreibt die X-Achse die horizontale Skalierung, d.h die Anzahl der gleichen Instanzen [4]. Die Y-Achse beschreibt die funktionale Zerlegung und stellt die Anzahl der verschiedenen Dienste in einer Anwendung dar [4]. Entlang der Z-Achse erfolgt die Datenpartitionierung [4].

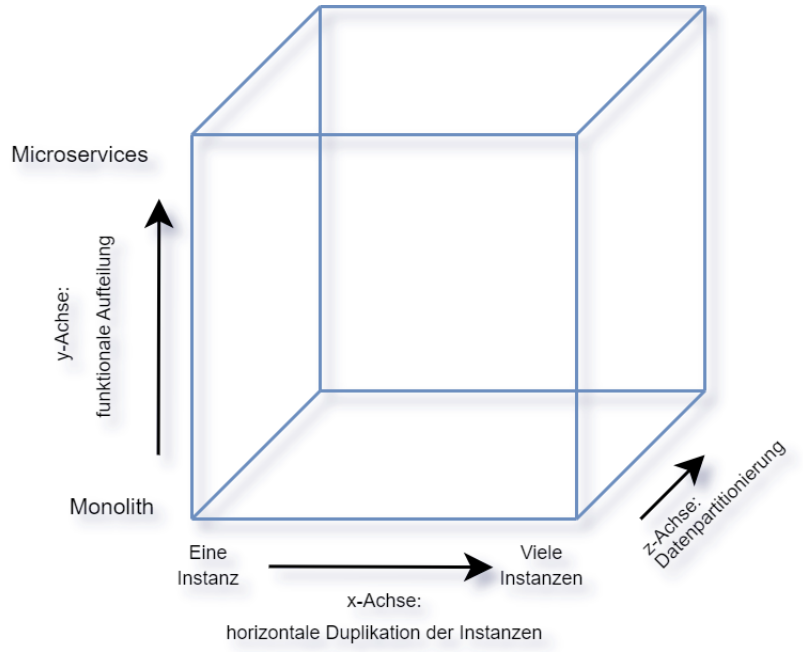

Abbildung 4 Skalierungswürfel (Angelehnt an [12])

Neben dem großen Vorteil der Skalierbarkeit, bringen Microservices den Vorteil, dass sie jeweils für ihre eigenen Aufgaben zuständig sind [4]. Entwickler können sich autonom von Anderen auf ein Service konzentrieren und dieses pflegen [4]. Sie können auch einfacher neue Technologien oder neue Aspekte in der Software ausprobieren, ohne die eigentliche Software zu verändern. Durch die Unabhängigkeit und lose Verbindung der Dienste untereinander, kann eine bessere Fehleranalyse garantiert werden [4]. Sollte ein Service ausfallen, können andere Serviceinstanzen die benötigte Aufgabe übernehmen [4].

Jedoch gibt es aber eine Reihe von Herausforderung, die bei einer Microservice-Architektur berücksichtigt werden sollten [4]. Durch die Zerlegung einer monolithischen Anwendung in verschiedene Komponenten, ist die Struktur und der Aufbau komplexer [4]. Service müssen miteinander kommunizieren, sodass zusätzlich die Kommunikationswege implementiert werden müssen [4]. Des weiteren ist es nicht immer eindeutig, in wie viele und in welche Komponenten die Anwendung am besten zerlegt werden kann [4]. Durch die Verteilung und Zerlegung in Dienste, ist es schwierig die Datenkonsistenz zu erhalten, da einzelne Dienste eine eigene Datenbank haben und somit Anfragen über mehrere Service arrangiert werden müssen [4]. Für die Entwickler einer solchen Microserivce-Architektur besteht zwar der Vorteil der Autonomie, aber dies ist auch mit zusätzlicher Arbeit verbunden die Service

zu koordinieren [4].

Um eine erfolgreiche Microservice-Architektur aufzubauen, ist es wichtig verschiedene Aspekte wie Zerlegung der Anwendung, Kommunikation zwischen Diensten, Datenkonsistenz der Services und Deployment in Betracht zu ziehen [4]. Dafür eigenen sich unterschiedliche Designmuster, die bei einer Umstellung auf eine Microservice-Architektur helfen [4].

2.3 Designmuster

Ein Architekturmuster beschreibt den Zusammenhang zwischen einem Kontext, einem Problem und einer Lösung [13]. Der Kontext stellt dabei eine Situation dar, die auf ein Problem hinleitet [13]. Bei dem Problem stellen sich Qualitätsattribute heraus, die für das Lösen des Problems wichtig sind [13]. Mit dem Designmuster werden dafür die entsprechenden Architekturkomponenten, die eine Sammlung von Interaktionsmechanismen und topologischen Strukturen darstellen, als Lösung vorgeschlagen [13].

Das Azure Architecture Center (AAC)[1], eine Cloud-Computing Platform von Microsoft, stellt verschiedene Cloud-Dienste zur Verfügung und beschreibt mehrere Cloud-Architekturen, unter anderem Microservices. Das AAC erklärt die Implementierung, Nutzen und Vorgehen mit Cloud-Architekturen und beschreibt dafür in einem Musterkatalog 42 Muster für die Implementierung einer Cloud-Anwendung [14]. Diese werden in die drei große Kategorien unterteilt: Datenverwaltung, Entwurf und Implementierung, Nachrichten [14].

Von diesen Mustern wurden vierzehn in einer empirischen Studie von Vale et al. analysiert, um die Signifikanz von Microservice-Muster in der Industrie festzustellen [15]. An der Befragung nahmen neun Mitarbeitende aus der Industrie teil. Die Ergebnisse der Studie ergaben, dass Microservices ein beliebter Architekturstil ist, da es insbesondere die Skalierung der Anwendung unterstützt [15].

Chris Richardson beschreibt in seinem Buch *Microservice Patterns* eine Sammlung aus Microservice-Mustern, die er als eine Muster-Sprache zur Implementierung einer Microservice-Architektur definiert [4]. In seiner Zusammensetzung der Muster lassen sich diese in drei Schichten unterteilen - Infrastrukturmuster, Applikationsinfrastrukturmuster und Applikationsmuster [4]. Diese beschreiben wiederum weitere Kategorien wie Kommunikationsmuster, Muster zur Datenkonsistenz und Bereitstellungsmuster, mit denen die entsprechenden Probleme in der Microservice-Architektur gelöst werden können [4].

Die Autoren G. Marquez und H. Astudillo haben in einer Literaturstudie den Nutzen von Entwurfsmustern in Open-Source Projekten und den Zusammenhang zwischen Qualitätsattributen und der Muster analysiert [16]. Die Studie hat ergeben, dass es viele Muster für die Microservice-Architektur gibt, jedoch aber mit einer hohen Rate an Redundanz, sodass nach einer Klassifikation nur 17 Muster analysiert wurden [16]. Es zeigte sich, dass das Qualitätsattribut Verfügbarkeit am häufigsten unter den Mustern vertreten ist [16]. Das Qualitätsattribut beschreibt inwieweit die Abhängigkeiten zwischen Komponenten in einer Microservice-Architektur zuverlässig sind [15]. Da in vielen Fällen Service voneinander abhängig sind, müssen diese für eine funktionierende Zusammenarbeit miteinander agieren können und somit verfügbar sein [15].

[1] https://learn.microsoft.com/de-de/azure/ (besucht am 03.06.2023)

Zusammen mit M. Villegas haben die Autoren im selben Jahr eine Mustersprache aus verschiedenen Entwurfsmuster explizit für die Skalierbarkeit der Microservices entwickelt [17]. Angelehnt ist diese an den Skalierbarkeitswürfel von Abbott und Fischer, den Chris Richardson für die Definition der Microservice-Architektur verwendet [MoMG?]. Der Würfel ordnet bei der Mustersprache für die Skalierbarkeit jeder Achse eine bestimmte Gruppe an Mustern zu [17]. So beschreibt die X-Achse Load Balancer Patterns, die Y-Achse Zerlegungsmuster und die Z-Achse Gruppenmuster [17]. Die Ergebnisse können als ein Leitfaden für die Implementierung von Skalierbarkeit in Projekten verwendet werden [17].

Akhan Akbulut und Harry G. Perros haben drei wichtige Muster auf Performance analysiert und deren Nutzen und Effizienz in einer Microservice-Architektur belegt [18].

2.3.1 Designmuster für Microservice-Architekturen

API-Gateway

Das API Gateway ist eine zentrale Komponente in einer Microservice-Architektur und dient als Schnittstelle zwischen dem Client und den einzelnen Servicen [19]. Es nimmt Anfragen vom Client an, kann diese authentifizieren und leitet sie an die entsprechenden Service weiter [19]. Ohne ein Gateway müsste der Client direkt mit den Servicen kommunizieren, was zu einer hohen Komplexität auf der Seite des Clients führt, da dieser alle Endpunkte von den Servicen kennen muss [19]. Es würde eine zu starke Kopplung zwischen Client und den Servicen entstehen und die Rate der Netzwerkroundtrips würde sich bei einem Aufruf von mehreren Servicen erhöhen [19].

Mit dem API-Gateway können solche Probleme gelöst werden [19]. Es implementiert drei wichtige Entwurfsmuster - Gatewayrouting, Gatewayaggregation und Gatewayabladung - die zentrale Funktionen für die Microservice-Architektur übernehmen [19]. So wird mit dem Gatewayrouting der Client von den Servicen entkoppelt und dessen Anfragen werden durch das API-Gateway, welches dann als Reverseproxy fungiert, an die entsprechende Dienste geschickt [19]. Die Gatewayaggregation erlaubt eine Zusammenstellung der Informationen, wenn mehrere Dienste für die Bearbeitung einer Client-Anfrage benötigt werden und verringert somit die Kommunikation zwischen den beiden Seiten [19]. Das API-Gateway kann auch übergreifende Aufgaben wie Authentifizierung, Protokollierung und SSL-Terminierung, die für die gesamte Microservice-Architektur benötigt werden, anhand der Gatewayabladung auf das API-Gateway übertragen [19].

Das API-Gateway ist ein weit verbreitetes Muster, was am häufigsten von allen Mustern in Open-Source Projekten verwendet wird, da es den zentralen Kontaktpunkt zwischen dem Client und der eigentlichen Microservice-Architektur bildet, sowie die Qualitätsattribute Erreichbarkeit und Wartbarkeit unterstützt [16]. Insbesondere das Entwurfsmuster Gatewayrouting wird häufig eingesetzt, da es die Kopplung zwischen Client und Service auflöst [15]. Jedoch stellt es auch einen möglichen, zentralen Problempunkt dar [15]. Da alle Anfragen zuerst das API-Gateway passieren, stellt die Funktionalität des Gateways den Eingangpunkt zu den Services dar und kann bei Komplikationen oder Ausfällen somit zu einem zentralen Fehlerpunkt führen [15].

Einen weiteren großen Vorteil bietet das API-Gateway, da damit die Möglichkeit besteht, das Backends for frontends - Muster zu implementieren [4]. Für jeden Client-Typ wird

dabei ein eigenes API Gateway implementiert [4]. Dies ermöglicht verschiedenen Clients wie zum Beispiel Mobil-Clients, Browser-Clients oder auch Dritt-Anwendungen, ihre eigene Schnittstelle und API zu verwenden [4]. Dies unterstützt, dass eine bessere Zuverlässigkeit, Beobachtbarkeit und Skalierung der eigenen API garantiert werden kann [4]. Viele Entwickler verwenden dieses Muster für die Microservice-Architektur, da es insbesondere die Performance der Anwendung steigert, auch wenn es komplexer zu implementieren ist [15].

Koordination von Microservices

Mehrere Microservices können auch sequentiell, hintereinander ausgeführt werden, wenn das Ergebnis auf eine Anfrage über mehrere Dienste verarbeitet werden muss [18]. Dieses Designmuster beschreibt dann eine Zuständigkeitskette (Chain of responsability), die mehrere Dienste miteinander verknüpft [18]. Ein Beispiel von so einer Kette kann ein Bildverarbeitungsprogramm sein, bei dem ein Bild über mehrere Service unterschiedliche Prozesse der Bildverarbeitung, wie Größenanpassung oder Graustufenkonvertierung durchläuft [18].

Die Autoren A. Akbulut und H. G. Perros haben einen solchen Prototypen auf die CPU- und RAM-Anforderungen analysiert und mit einem weiteren Prototypen verglichen, der die gleichen Funktionen der Service in einem einzigen Service implementiert [18]. Dabei wurden die Auswirkungen auf die Hosting-Kosten betrachtet [18]. Die Ergebnisse zeigen, dass die Art und Weise der Aufteilung der Funktionen in einzelne Dienste direkte Auswirkungen auf die Kosten und die Effizienz der Hardwarenutzung hat [18]. Die Nutzung des Chain-of-Responsibility-Musters für sequenzielle Datenverarbeitung ermöglicht eine effizientere Ressourcennutzung und senkt die Hosting-Kosten im Vergleich zu einem Megaservice-Ansatz, indem sehr viele Funktionen zusammen vorliegen [18]. Des weiteren bietet eine Chain-of-Responsibility eine bessere Skalierbarkeit der Service, sodass nur die entsprechenden Funktionen bei Bedarf skaliert werden müssen, während bei einem Service alle Funktionen skaliert werden und sich somit ein größerer Ressourcenverbrauch ergibt [18].

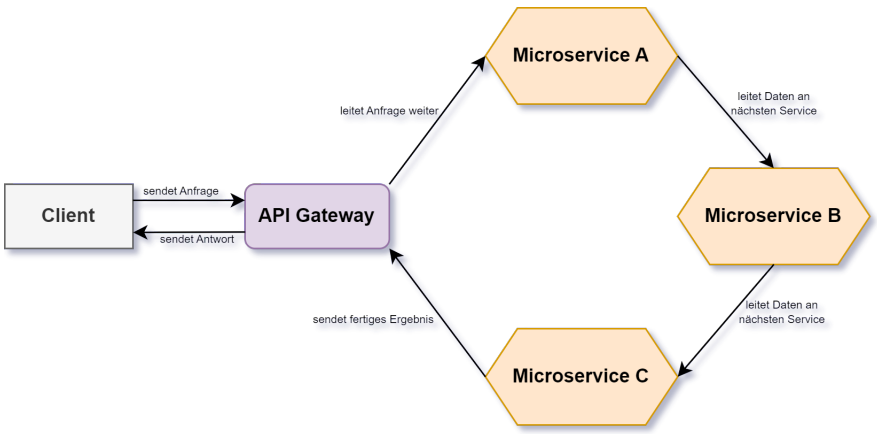

Abbildung 5 Chain of responsability (Angelehnt an [18])

Auslagerung von Funktionen

Bestimmte Funktionen können auch in einem Sidecar oder einem Ambassador Muster, neben der eigentlichen Anwendung ausgelagert werden [20].

Das Sidecar-Muster isoliert unterstützende Funktionen sprachübergreifend von einer Anwendung, indem es sie in einem eigenen Prozess oder Container bereitstellt, wobei dann das Sidecar den Lebenszyklus der Anwendung teilt [20]. Das Muster kann Überwachung, Protokollierung, Konfiguration und andere periphere Aufgaben übernehmen, die zur Anwendung gehören, aber nicht zu ihrer Hauptfunktionalität zählen [20]. Dies eignet sich z.B für Service, die in verschiedenen Programmiersprachen implementiert sind und mit der Funktionalität eines Sidecars keine Abhängigkeiten auf die zentralen Funktionen benötigen werden [20]. Des weiteren gibt es keine zusätzliche Latenz, da Sidecars in demselben Host gehostet werden [20].

Das Ambassador-Muster kann als Sidecar bereitgestellt werden. Dabei handelt es sich um einen Proxy, welcher außerhalb der Anwendung liegt [21]. Es lagert ebenfalls sprachunabhängige Funktionen, wie Überwachung und Routing aus und wird in der Regel bei älteren Anwendungen eingesetzt [21]. Es ermöglicht somit eine Remote-Verbindung mit einer älteren Software [22]. Da sich das Ambassador-Muster als ein Proxy außerhalb der Anwendung agiert, besteht jedoch eine höhere Latenz [21].

Die Einsatz des Sidecars und des Ambassador Musters wird im Bezug auf die Möglichkeit, Überwachung und Wartbarkeit zu implementieren, gerne von Entwicklern für Microservice-Architekturen verwendet [15]. Das Ambassador Muster wird insbesondere für das Qualitätsattribut der Verfügbarkeit verwendet, sowie der Sicherheit, da es aufgrund des Proxies eine sichere Schicht zwischen den Instanzen ermöglicht [15].

Migration von Monolith zu Microservices

Um bei einer Migration von monolithischen Anwendung zur Microserive-Architektur eine relevante Kommunikation zu erhalten gibt es zwei Entwurfsmuster, die eine Verbindung zwischen der Legacy-Anwendung und der sich entwickelnden Cloud-Anwendung ermöglichen [23], [24]. Das Strangler-Muster und die Antibeschädigungsebene sind beides Muster, die eine Umwandlung zur neuen Architektur migrieren könnnen [23], [24].

Da eine Anwendung nicht immer direkt in kurzer Zeit vollständig auf eine Microservice-Architektur migriert werden kann, bietet das Strangler-Muster die Möglichkeit, anhand einer Fassade neue und alte Komponenten der Anwendung zu verwenden [24]. Dabei kümmert sich die Fassade um das Routing der Client-Anfragen [24]. Anfragen die noch vom alten System bearbeitet werden müssen, werden entsprechend an das alte System weiter geleitet und anders herum auch für die neuen Dienste [24]. Dadurch lässt sich eine Anwendung über einen längeren Zeitraum zur neuen Architektur migrieren [24]. Dies ist aber auch mit einer gewissenhaften Plannung verbunden, die Zeiträume entsprechend zu koordienieren [15]. Das Strangler-Muster wird oft von Entwicklern eingesetzt, da es die Qualitätsattribute Skalierbarkeit und Wartbarkeit unterstützt [15].

Ein weitere Lösung für die Migration auf eine neue Architektur ist die Antibeschädigungsebene, die zwischen den Systemen implementiert wird, Anforderungen der beiden Architekturen zu übersetzen [23]. Da Systeme sich verändern können und ältere Technologien nicht den Anforderungen von Neuen entsprechen, ist bei einer Migration eine solche

Übersetzung notwendig [23]. Der Nachteil jedoch bei einer solchen Ebene ist die Latenz durch die zusätzliche Kommunikation [23], sowie auch die Komplexität, die durch weitere Implementierungsarbeit steigt [15].

Fehlererkennung

Aufgrund von möglichen Fehlern, Wartungen eines Services oder aber auch bei einer Überlastung, kann es zu Dienstausfällen kommen [4]. Dies führt zu Kommunikationsproblem in einer verteilen Architektur, da die Erreichbarkeit nicht gewährleistet ist [4]. Um diese Fehler zu umgehen, muss das Circuit Breaker Muster implementiert werden, welches eine zuverlässige Kommunikation und den Umgang mit Fehlern gewährleisten kann [4].

Das Circuit Breaker Muster zählt dabei wie oft fehlerhafte Anfragen an einen Service geschickt werden und bricht direkt nach einer bestimmten Anzahl, weitere Anfragen ab [4]. Es wird dann angenommen, dass der Dienst zurzeit nicht erreichbar ist und wird somit für einen limitierten Zeitraum blockiert [4]. Danach wird der Dienst wieder zur Verfügung gestellt [4]. Die Implementierung eines solchen Mechanismus stärkt die Stabilität in der Anwendung, da Fehler somit erkannt, protokolliert und behoben werden können [25].

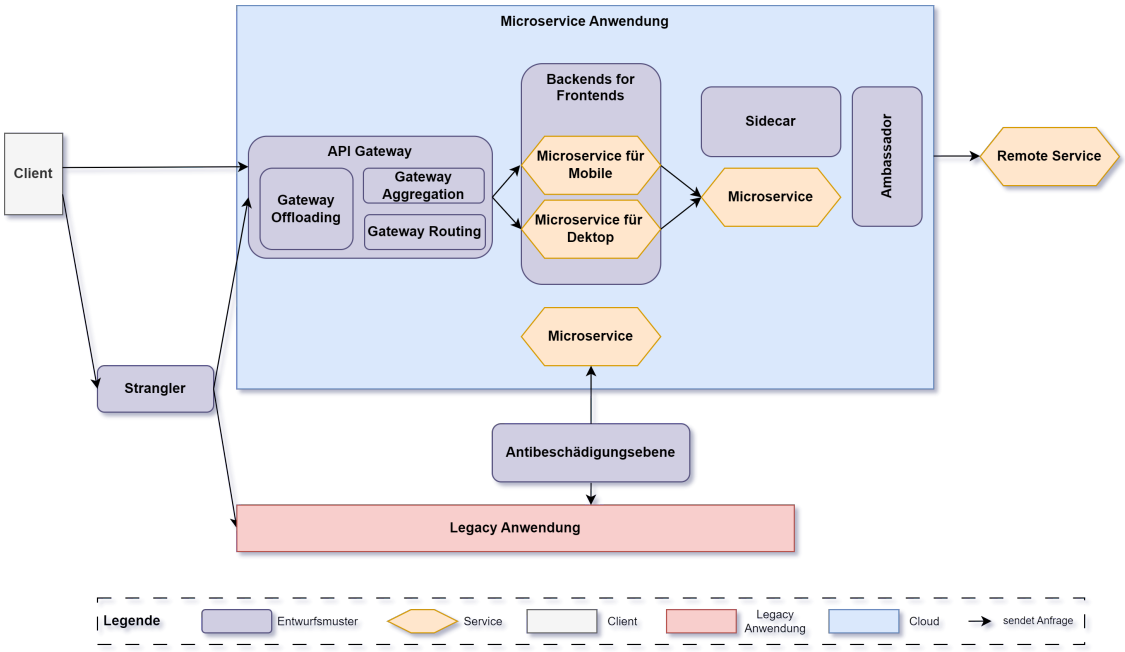

Abbildung 6 Anordnung von Entwurfsmuster für eine Microservice-Architektur (Angelehnt an [26])

2.3.2 Kommunikation

Während in einer monolithischen Anwendung die Kommunikation durch Methodenaufrufe in der Software selber stattfindet, muss bei einer Microservice-Architektur die Kommunikation anhand von Kommunikationsmechanismen erstellt werden [2]. Somit ist die Interprozesskommunikation zwischen den einzelnen Microservices ein wichtiger Bestandteil in einer verteilten Architektur, da es aufgrund der Datenübertragung zu Latenzproblem füh-

ren sowie über die Anwendungsverfügbarkeit bestimmen kann [4]. Damit eine stabile und zuverlässige Kommunikation in einer Microservice-Architektur vorliegen kann, muss bei der Implementierung einige Aspekte berücksichtigt werden [4]. So muss der richtige Kommunikationsstil, sowie die Art und Weise wie Service Nachrichten versenden, gewählt werden [4]. Service müssen sich gegenseitig finden können, die Zuverlässigkeit bei Serviceausfällen muss gewährleistet werden und das Design von externen APIs sollte für die Kommunikation mit Clients bedacht werden [4].

Allgemein kann in einer Microservice-Architektur die Kommunikation mit drei verschiedenen Methoden durchgeführt werden [27]. Ein Client oder ein Service kann eine Abfrage erstellen, einen Befehl ausrufen oder ein Ereignis auslösen [27]. Diese Möglichkeiten können dabei in synchroner oder asynchroner Art und Weise implementiert werden [27]. Bei einer synchronen Kommunikation erwartet der Client eine zeitnahe Antwort, während bei einer asynchronen Kommunikation keine direkte Antwort benötigt oder überhaupt erwartet wird [4]. Durch die Verteilung der Microservices sind auch verschiedene Kommunikationsbeziehungen zwischen mehreren, einzelnen Diensten möglich, sodass auch eine Interaktion zwischen genau zwei Servicen oder auch zwischen einem und mehreren Servicen stattfinden kann [4].

Synchrone Kommunikation

Bei einer synchronen Kommunikation schickt der Client oder ein Service dabei eine Anfrage an einen anderen Service und wartet auf eine zeitnahe Antwort [4]. Das Konzept der synchronen Kommunikation wird auch als Remote procdeure invocation Muster bezeichnet [4]. Hierbei werden synchrone Protokolle wie gRPC und REST verwendet [4].

Das synchrone Protokoll REST arbeitet mit den HTTP-Methoden und kann die Zustände einer Ressource wiedergeben [4]. Aufgrund dessen Popularität und der Einfachheit, wird das Protokoll von vielen Entwicklern verwendet [4]. Jedoch können mit REST-Protokollen nicht direkt mit einer Anfrage mehrere Ressourcen angefordert werden [4]. Die Implementierung mit Abfrageparametern ist bei manchen Szenarien ausreichend, sobald aber die Information komplexer wird, reichen diese nicht aus [4].

Das gRPC ist ein sprachunabhängiges Framework, welches ein binär-basiertes Nachrichtenprotokoll bietet [4]. Dabei wird eine API anhand von Protokol Buffer erstellt, [4]. gRPC bietet eine hohe Effizienz für den Austausch von vielen, großen Daten und ein bidirektionales Streaming [4]. Es unterstützt auch die Interoperabilität zwischen den Diensten [4].

Die Autoren Weerasinghe und Perera haben die Kommunikationsprotokolle HTTP, gRPC und WebSockets auf Performance untersucht und konnten feststellen, dass das gRPC-Protokoll deutlich mit weniger Latenz kommuniziert, als das HTTP-Protokoll [1]. Dies ist darauf zurückzuführen, dass gRPC eine einzelne TCP Verbindung benutzt, während beim HTTP-Protokoll die Verbindung zuerst erstellt und wieder geschlossen werden muss [1].

Jedoch haben beide Protokolle den Nachteil, dass durch die synchrone Kommunikation, diese in einer Microservice-Architektur die Verfügbarkeit und Erreichbarkeit einschränken [4]. Mögliche Serviceausfälle verhindern die Kommunikation, weswegen entsprechende

Maßnahmen implementiert werden müssen [4]. Hierbei wird das Circuit Breaker Muster verwendet, welches im vorherigen Kapitel bereits erwähnt wurde [4].

Asynchrone Kommunikation

Bei einer asynchronen Kommunikation wird häufig ein externer Vermittler für die Datenübertragung verwendet [4]. Solch ein Nachrichtenvermittler oder Nachrichtenbus kann Nachrichten in einer Warteschlange aufhalten, bis der Empfänger die Nachricht verarbeiten kann [4]. Dies ermöglicht eine zuverlässige Vermittlung aller gesendeten Nachrichten [4]. Des weiteren erlaubt solch ein System eine lose Koppelung zwischen Client und Service, da die Instanzen, zwischen denen ein Nachrichtenvermittler eingesetzt wird, nicht den jeweiligen Netzwerkort kennen müssen, um die Datenübertragung durchzuführen [4].

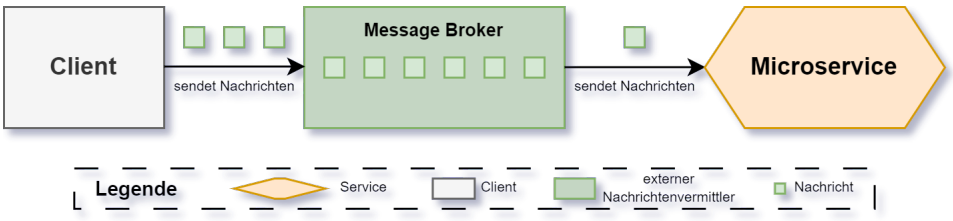

Abbildung 7 Message Broker (Angelehnt an [4])

Bekannte Technologien für solche Nachrichtenvermittler sind RabbitMQ, ActiveMQ und Apache Kafka [4]. Die Auswahl eines geeigneten Nachrichtenvermittler hängt von den Eigenschaften des Vermittler und den geforderten Anforderungen einer Anwendung ab, da verschiedene Vermittler unterschiedliche Funktionen und Möglichkeiten implementieren [4].

A. Akbulut und Harry G. Perros haben mit einem Prototypen das Asynchrone Messaging Muster mit RabbitMQ als Nachrichtenvermittler auf dessen Nützlichkeit untersucht [18]. Dabei wurden Transaktionen zwischen Diensten mit unterschiedlichen Arbeitszeiten durchgeführt [18]. Deren Ergebnisse ergaben, dass das Konzept eines Nachrichtensystems in einer Anwendung eine optimale CPU Nutzung mit 50 Prozent weniger Verbrauch unterstützt [18]. Außerdem betonen die Autoren, dass durch die Stauchung der Nachrichten in einer Warteschlange, alle Anfragen verarbeitet werden können [18].

Xian Jun Hong et al. führten Simulationstests durch, um das Verhalten der Performance von RESTful APIs und dem AMQP-Protokoll in einer Microservice-Architektur zu ermitteln [28]. Dabei wurde die Geschwindigkeit der Anfrage- und Antwortzeit gemessen, wobei die Anzahl der Nutzer in den Testszenarien variiert wurde [28]. Als Message Broker wurde RabbitMQ verwendet [28]. Das Resultat ergab, dass bei einer Erhöhung der Nutzeranzahl die RESTful API bis zu einem bestimmten Punkt performanter ist, während die Verwendung von RabbitMQ gleich blieb [28]. Übersteigt die Nutzeranzahl jedoch einen zu hohen Ausmaß, so bricht die Performance der RESTful API ein [28].

2.3.3 Datenmanagement

Die einzelnen Dienste, die in einer Microservice-Architektur miteinander kommunizieren, besitzen meistens eine eigene Datenbank [2]. Microservices können auch eine Datenbank

miteinander teilen, dies wird jedoch aufgrund von zu starker Kopplung vermieden [11]. Die Unabhängigkeit von Microservices untereinander ermöglicht es, dass Änderungen an einem Dienst keine Auswirkungen auf andere Dienste haben und Informationen privat gehalten werden [11]. Eine geteilte Datenbank führt jedoch dazu, dass Änderungen an der Datenbank direkt mehrere Service betrifft und die gleichzeitige Nutzung auch zu Performanceproblemen führen kann [11].

Datenbanktechnologien

Für die Verwendung von Datenbanken gibt es zwei verschiedene Datenbanktechnologien: SQL und NoSQL [29].

Die SQL (Structred Query Language) Datenbank ist seit den 70ern die meist verbreitete Datenbank und verwendet ein relationales Konzept, in dem die Daten in Tabellen und in Reihen gespeichert werden [29]. SQL-Datenbanken unterstützen das Konzept der Replikation aller gleichen Instanzen, wenn Änderungen an einer durchgeführt wurden [29].

Seit den 00ern hat sich eine neue Datenbanktechnologie etabliert, die nicht-relationale Datenbank NoSQL [29]. Hierbei handelt es sich um eine Datenbanktechnologie, die verschiedene Strukturen und Formen von Daten persistent halten kann und somit einen flexibleren und dynamischeren Umgang mit Daten erlaubt [29]. NoSQL Datenbanken werden des öfteren bei sich ändernden Datenmodellen, mit unstrukturierten Daten eingesetzt [29]. Es ist für eine horizontale Skalierung besser geeignet als eine SQL Datenbank und unterstützt somit die Lastenverteilung auf mehrere Server [29].

Die Wahl einer Datenbank für einen Microservice spielt aufgrund der verschiedenen Eigenschaften der Datenbanken eine wichtige Rolle [30].

Datenkonsistenz

Damit in einer Anwendung die Datenkonsistenz erhalten bleibt, spielt das Konzept von Transaktionen eine wichtige Rolle [4]. Eine Transaktion ist dabei entweder eine einzelne Aufgabe oder eine Abfolge von Aufgaben, welche eine Zustandsänderung und eine Aktion durchführen [31]. Transaktionen haben die ACID-Eigenschaften (*atomic, consistent, isolated, durable*) und im Rahmen eines Dienstes sind diese Eigenschaften gegeben [31]. Finden aber Änderungen dienst-übergreifend statt, so reicht eine ACID-Transaktion nicht aus [31].

Generell sind bei einer Microservice-Architektur, in der mehrere Dienste miteinander kommunizieren, mehrere Datenbanken und Nachrichtenvermittler vorhanden sind, die Ausführung von Transaktionen komplex [4]. Informationen die zur Verarbeitung von Anfragen erforderlich sind, sind über mehrere Dienste verteilt [4]. Verteilte Transaktionen, die den üblichen Weg für Datenkonsistenz mit dem Zweiphasencommit-Protokoll (2PC) darstellen, sind darauf ausgelegt, dass bei jeder Transaktion ein Commit oder ein Rollback durchgeführt werden werden muss [4]. So kann dieses Protokoll von vielen SQL-Datenbanken und einigen Nachrichtenvermittlern unterstützt werden, jedoch aber nicht für moderne Technologien wie NoSQL-Datenbanken wie MonogoDB oder Cassandra, sowie auch nicht für moderne Nachrichtenvermittler wie die im vorherigen genannten Kapitel RabbitMQ und Apache Kafka [4]. Des weiteren basieren verteile Transaktionen auf synchroner Kommunikation, welche nicht in allen Szenarien gefordert wird und auch die Verfügbarkeit der Service unbedingt einhalten muss, da die Service für die verteilte Transaktionen perma-

nent erreichbar sein müssen [4].

Um diese Probleme mit verteilten Transaktionen in einer Microservice-Architektur zu umgehen, werden Sagas verwendet [4]. Saga ist ein Designmuster für die Erhaltung der Datenkonsistenz und beschreibt eine Sequenz aus lokalen Transaktionen [4]. Lokale Transaktionen beziehen sich dabei auf die einzelne Dienste, in denen Daten aktualisiert werden [4]. Dabei wird die erste Saga mit der Anfrage vom Client ausgelöst und alle darauffolgenden werden nach Vollendigung der Datenverarbeitung der vorherigen Saga durchgeführt [4]. So können auch Services, die ihren Zustand ändern, anhand mit Sagas beschrieben werden [32]. Sagas basieren auf der asynchronen Kommunikation [4]. Somit kann gewährleistet werden, dass die Sequenz vollständig durchgeführt wird, da die Transaktionen erst weitergeführt werden, wenn die vorherige Saga abgeschlossen ist [4]. Falls eine Transaktion fehlerhaft ist, so wird eine kompensierende Transaktion durchgeführt, die die Änderung wieder rückgängig macht [4].

Die Koordination einer Saga-Sequenz kann auf zwei verschiedene Weisen implementiert werden - Choreographie und Orchestrierung [4]. Choreographie beschreibt dabei den Ansatz, dass Sagas Ereignisse über ihre eigenen Änderungen in der Datenbank an die anderen teilnehmenden Service veröffentlichen und sich gegenseitig auch zuhören, sodass sie die Änderungen mitbekommen [4]. Es ist eine einfache Möglichkeit die Sagas zu koordinieren, jedoch wird es komplexer und unübersichtlicher, je mehr Service an dem Konzept teilnehmen [4]. Im Unterschied dazu, wird mit dem Ansatz der Orchestrierung ein Service gewählt, welcher als Orchestrator agiert und der die Aufgaben der Sagas verwaltet [4]. Hierbei ist die Koordinierung übersichtlicher und es gibt keine zyklischen Abhängigkeiten zwischen den Services [4]. Die Verwendung des Orchestrierung-Konzepts wird bei komplexen Sagas, aufgrund der besseren Übersicht, bevorzugt [4]. Für die Implementierung von kleineren und unkomplizierten Sagas kann eine Choreography verwendet werden [4].

Eine weitere Herausforderung durch die Verteilung der Dienste und Datenbanken, ist die Implementierung von Abfragen, die mehrere Informationen von verschiedenen Diensten benötigen [4]. Dieses Problem kann mit den beiden Mustern API Composition und Command query responsibility segregation (CQRS) gelöst werden [4].

Mit dem API Composition Muster können Daten oder Ergebnisse von mehreren Diensten für eine Abfrage zusammengefasst werden [4]. Dabei kann ein Client, ein API Gateway oder auch ein eigener Service als API Composer agieren und die benötigten Daten der Dienste abfragen und zusammenfassen [4]. Jedoch bringt dieses Verfahren auch Nachteile wie z.B eine reduzierte Erreichbarkeit und mögliche Probleme mit der Datenkonsistenz [4]. Je mehr Service in eine Verarbeitung einer Abfrage involviert sind, desto stärker sinkt die Erreichbarkeit der Service [4].

Probleme bei der Datenkonsistenz entstehen hierbei, wenn der API composer mehrere Anfragen über die Dienste erstellt, die noch nicht die entsprechende Zustandsänderungen durchgemacht hat [4]. So ist das Muster nicht für alle Szenarien eine richtige Wahl und kann zu ineffizienten Abfragen führen [4].

Aufgrund dessen können solche Schwierigkeiten mit dem Command query responsibility Muster implementiert werden [4]. Das Konzept des Musters ist die Aufteilung des Datenmodells [4]. Die Datenbanken werden in zwei Instanzen aufgeteilt, wobei eine Instanz

die Bearbeitungsoperationen Create, Update und Delete implementiert und die andere nur die Abfrageoperation Get ermöglicht [4]. Dabei abonniert die Abfrageinstanz, die Bearbeitungsinstanz, um alle Änderungen zu erhalten [4]. Das Muster erlaubt somit eine effiziente Implementierung von Abfragen [4].

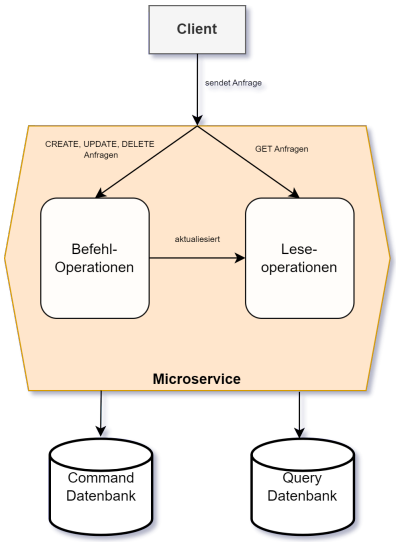

Abbildung 8 CQRS (Angelehnt an [4])

2.3.4 Deployment

Damit eine Anwendung für Nutzer möglich ist, müssen die Funktionen und Ressourcen dafür entsprechend zugänglich gemacht werden [4].

Das Konzept des Deployments kann als eine Kombination aus Prozess und Architektur beschrieben werden [4]. Dabei beschreibt der Prozess, die notwendigen Vorgänge und Aufgaben von Entwicklern, eine Software bereitzustellen und die Architektur beschreibt die Entwicklungsumgebung, die für das Deployment benötigt wird [4]. Über die Jahre hat sich die Art und Weise des Deployments von physischen über virtuelle Maschinen bis hin zu der modernen Technologie von Containern und der Möglichkeit, Anwendungen heutzutage serverlos zugänglich zu machen, verändert [4]. Da Microservice-Anwendungen heutzutage aus vielen einzelnen Diensten bestehen, ist eine automatisierte Möglichkeit des Deployments stärker gefragt [4].

Deployment mit Containern

Das Deployment mit Containern ist eine moderne und virtualisierte Möglichkeit, Anwendungen zugänglich zu machen [4].

Ein Container ist dabei eine isolierte Einheit auf einer Maschine, in der mehrere Prozesse parallel laufen können [4]. Container haben ihre eigenen Root-Dateisysteme und können sich die Betriebssystemmechanismen teilen [4]. Um einen solchen Container zu erstellen, wird anhand eines Tools, welches den Quellcode eines Services einliest, ein Containerbild generiert [4]. Das Bild wird dann in einem Bildregister gespeichert und kann für die Erstellung eine Containers verwendet werden [4].

Für die Bereitstellung der Anwendung mit Containern ist Docker[2] die führende Platt-

[2]https://www.docker.com/ (besucht am 21.06.2023)

form [33]. Mit Docker können Service in isolierte Container verpackt werden und mit ihren Abhängigkeiten bereitgestellt werden [33]. Somit eignet sich Docker für eine Microservie-Architektur, da sich die einzelnen Service schnell, unabhängig und automatisiert mit Docker verwaltet werden können [33].

Die Autoren Singh und Peddoju haben mit einer Microservice und einer monolithischen Architektur Performance Messungen durchgeführt, um das Verhalten der Prototypen bezüglich Anfrage- und Bereitstellungzeit zu analysieren [3]. Die Ergebnisse zeigen, dass bei einer steigenden Anzahl von Threads, die Anfragezeit bei einer Microservice-Architektur deutlich geringer ist, während diese beim Monolithen steigt [3]. Dies ist auf die Lastverteilung der Anfragen in der Microservice-Architektur zurückzuführen [3]. Auch die Bereitstellungs- und Aktualisierungszeit ist deutlich geringer als bei einer monolithischen Anwendung [3]. Zusätzlich haben die Autoren Container im Vergleich zu Virtuellen Maschinen evaluiert und können zeigen, dass in Bildgröße, Bereitstellung und Aktualisierungen Container schneller und performanter agieren [3].

2.3.5 Load Balancing

Load Balancing beschreibt das Verfahren, Anfragen zwischen mehreren Instanzen zu verteilen [34]. Dadurch kann eine bessere Verteilung der Last erzielt werden, welche auch gegen Fehler bei einer Überlastung eines Servers wirkt [34]. Es verbessert somit die Performance und ermöglicht eine höhere und stabile Erreichbarkeit der Anwendung [34].

Neue Server können flexibel zum Load Balancer hinzugefügt oder entfernt werden, sodass die Instanzen beliebig skaliert werden können [34]. So können zum Beispiel auch problemlos Serverwartungen durchgeführt werden, während der Anfragenverkehr unter den anderen Servern weiterläuft [34].

Für die Lastverteilung gibt es unterschiedliche Algorithmen, die die Verteilung der Last auf die Server berechnen [34]. Dabei wird zwischen gewichteten und ungewichteten Algorithmen unterschieden [34]. Bei gewichteten Algorithmen erhalten Load Balancer ein Gewicht, welches je nach Wert, mehr oder weniger Anfragen verarbeitet als die anderen Instanzen [34]. Algorithmen können auch statisch oder dynamisch sein [35]. Dynamische Load Balancer passen sich dabei an die Echtzeit Eigenschaften des Servers an [35].

2.4 Performance

2.4.1 Performance in Microservices

Performance ist ein fundamentales Qualitätsattribut in der Softwareentwicklung und beschreibt die Fähigkeit einer Anwendung, zeitliche Anforderungen zu erzielen [36]. Da Ausführungen eines Systems eine bestimmte Zeit beanspruchen, ist diese Eigenschaft eine zentrale Komponente, die Performance zu bestimmen [36].

Performance Modellierung

Aufgrund der Interaktion zwischen den einzelnen Diensten, das Qualitätsattribut der Performance auch eine hohe Relevanz in der Microservice-Architektur [37]. Die Eigenschaften der Netzwerkkommunikation haben einen starken Einfluss auf das Verhalten der Architektur, da diese sich in der Latenz oder der Geschwindigkeit von Anfragen wiederspiegeln [37]. Das Design der Interprozesskommunikation in einer Microservice-Architektur hat somit einen starken Einfluss auf die allgemeine Performance einer Anwendung [1].

Die Zeit die verbraucht wird, um eine Anfrage vom Client anhand eines Kommunikationsprotokolls, gegebenenfalls durch einen Nachrichtenvermittler, zum Service zu schicken, welche dann entsprechend verarbeitet wird und eine Transaktion in der Datenbank auslösen kann, wird als Bearbeitungszeit einer Anfrage angesehen [38]. Diese kann anhand der Dauer für die jeweiligen Komponenten gemessen werden [38].

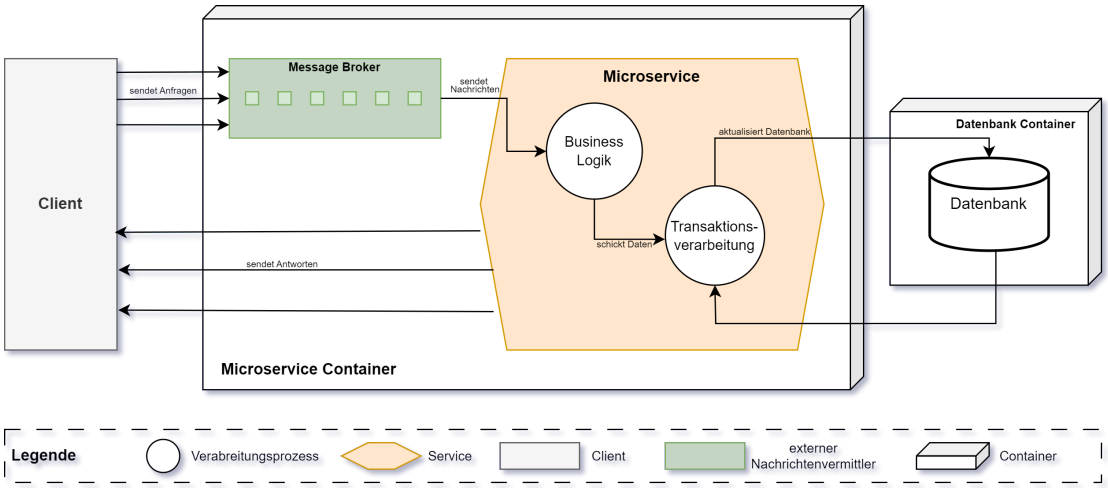

Abbildung 9 Performance Modellierung in einem Service (Angelehnt an [38])

Die Performance einer Microservice-Architektur bzw. die von einem Service kann somit anhand von vier Rubriken beschrieben werden [38]. Die erste Kategorie ist Bedeutung der Anfragen [38]. Die Performance des Services hängt von der Anzahl ab, sowie die Dauer, die sie in einer Warteschlange verbringen, bevor sie ausgeführt werden [38]. Die zweite Kategorie beschreibt die Ausführung der Businesslogic innerhalb eines Microservices [38]. Hierbei ist unter anderem die Komplexität des Services relevant, da sich dadurch die Verarbeitungszeit der Anfrage stark auf die Performance auswirkt [38]. Darüber hinaus verübt die darauffolgende Transaktionszeit in einer Datenbank, abhängig von der Anzahl der Einträge, eine weitere Verzögerung [38]. Der vierte Aspekt, mit der man die Performance manipulieren kann, ist die Leistung der Hardware und die Eigenschaften der Komponenten [38].

Die Anzahl der CPU Kerne, der Arbeitsspeicher und deren Auslastung zum Zeitpunkt der Verarbeitung der Anfrage, sind Faktoren die dabei eine Rolle spielen [38].

Performance Optimierung

Für einen effizienteren Umgang mit Anfragen gibt es die Möglichkeit, mehrere Anfragen gebündelt zu verschicken [39]. Dies kann mit dem Request Bundle Muster implementiert werden [39]. Mehrere Anfragen werden dabei in einen Datencontainer zusammengestellt, mit Metadaten für die Identifizierung versehen und als eine einzelne Nachricht an den Empfänger geleitet [39]. Die Implementierung dieses Musters ist jedoch aber mit einer höheren Komplexität des API Endpunktes verbunden, sowie auch einem höheren Payload der übertragenen Nachrichten [39].

A. Malki und U. Zdun haben für dieses Muster eine empirische Studie durchgeführt, in der sie das Request Bundle Muster in open-source Microservice-Projekten auf die Performance analysiert haben [40]. Dabei haben ihre Ergebnisse ergeben, dass die Zeit, die benötigt wird, um eine Anfrage zu verschicken, deutlich geringer ist, bei der Verwendung von Bündelungen und somit einen effezienten Einfluss auf die Performance hat [40].

2.4.2 Performance Tests

Die Art und Weise, wie eine Anwendung auf verschiedene Situationen reagiert, kann viel über die Performance des Systems sagen [41]. So ist die Performance, aber auch die Zuverlässigkeit einer Anwendung unter anderem von der Nutzer Anzahl und der Komplexität der Anfragen abhängig [41]. Aber auch die Dauer, wie lange Nutzer die Anwendung verwenden können, kann die Eigenschaften des System beschreiben [41].

Um die Performance einer Anwendung zu überprüfen, gibt es verschiedene Testmöglichkeiten für die Evaluierung und Analyse von Eigenschaften wie z.B. die Anfragendauer [41]. Solche Tests werden Load Tests genannt und erlauben es, verschiedene reale Szenarien nachzustellen und zu überprüfen [41]. In Tabelle 1 sind unterschiedliche Varianten aufgelistet.

Tabelle 1 Arten von Performance Tests

	Testdauer	Virtuelle Nutzer	Verwendung
Load Test	5-60 Minuten [42]	10-100 [42]	Überprüfung der Stabilität der Anwendung bei einer durchschnittlichen Arbeitsbelastung [42]
Stress Test	5-60 Minuten [43]	10-200 [43]	Überprüfung der Stabilität der Anwendung bei einer überdurchschnittlichen Arbeitsbelastung [43]
Soak Test	3-72 Stunden [44]	10-100 [44]	Überprüfung der Stabilität der Anwendung, wenn diese über eine sehr lange Zeit funktionieren muss [44]
Spike Test	1-4 Minuten [45]	~2000 [45]	Überprüfung ob die Anwendung unter kurzfristigem, hoher Anfragenlast sperformant bleibt [45]

3 Methodik

Die Software Virtual Satellite soll zu einer Microservice-Architektur migriert werden und concurrent engineering Prozesse für Raumfahrtmissionen in der Cloud ermöglichen. Das Ziel einer Cloud-Architektur ist es, von den Vorteilen und Möglichkeiten der Microservice-Architektur zu profitieren. Somit soll die Transformation der Software den Ingenieuren neue Mittel in ihrer Verwendung bieten, aber auch mit der neuen architektonischen Microservice-Struktur eine effiziente Skalierung und Ressourcenverteilung, sowie eine leichtere Wartbarkeit für die Entwickler ermöglichen.

Da während einer CE-Studie mehrere Ingenieure miteinander an verschiedenen Modellkomponenten arbeiten und mehrere Berechnungen durchführen, ist eine performante Implementierung der Architektur zu berücksichtigen. Bei den Berechnungen kann es sich dabei um einzelne oder aber auch um komplexe Berechnungsketten handeln. Die Bearbeitungszeit eines Microservices, der die Anfragen erhält, muss somit eine entsprechende Geschwindigkeit leisten und die CE-Studien sollen mit Virtual Satellite performant in der Cloud ausgeführt werden können.

Die Wahl von angemessenen Designmuster, die den Anforderungen der Architektur entsprechen, ist somit ein wichtiger Aspekt für die Umwandlung. Insbesondere die Wahl der Kommunikationsart ist eine für die Performance maßgebende Entscheidung, da aufgrund der Netzwerkkommunikation Latenz und Fehler entstehen können.

Für den Beginn der Transformation wird eine Designmusteranalyse konzipiert, in der verschiedene Entwurfsmuster für die Cloud-Architektur zusammengestellt werden, die architektonische Herausforderungen für die Software lösen. Dafür wird zuerst eine Anforderungsanalyse erstellt, die die zentralen Aspekte für die Transformation von Virtual Satellite zu Microservices zusammenstellt.

Um eine erste Einschätzung zu erhalten wie man einen Service auf seine Performance untersuchen kann, wird ein Prototyp mit einem bereits verwendeten Service erarbeitet, anhand dessen Funktionen und Grundstrukturen von Virtual Satellite abgebildet und analysiert werden können.

3.1 Anforderungsanalyse für Virtual Satellite

Virtual Satellite ist mit ihrer aktuellen Desktop-Architektur eine modulare und flexible Anwendung. Mit Plug-Ins lassen sich neue Funktionen und Methoden implementieren. Datenmodelle können erweitert oder neu erstellt werden. Dennoch ist aber aufgrund des monolitischen Aufbaus die Desktopanwendung nur lokal auf den Arbeitsplätzen im CEF erreichbar. Mit der Umwandlung zur Cloud soll den Mitarbeitern im CEF die Option vorliegen, sich auch Remote über eine URL zu den CE-Studien zu zuschalten und an den Systemmodellen mit zu arbeiten. Es soll in Zukunft auch die Möglichkeit geben, Daten von anderen Geräten, wie z.B Mobile-Telefonen, abzurufen.

Das Datenmodell von Virtual Satellite soll auch für andere Applikationen zur Verfügung stehen. In der zu entwickelnden Cloud-Variante von Virtual Satellite sollen die Daten in Form einer API für Andere zugänglich gemacht werden. So soll z.B eine CAD-Anwendung,

auf die Strukturdaten des Satelliten zugreifen können, um die existierende Grundstruktur des Satelliten mit mechanischen Details zu erweitern.

Folgende Anforderungen müssen somit für die Architektur in der Cloud erfüllt sein:

R1 Modular
Konzepte zum Designen von Raumfahrtmissionen und die dazu benötigten Modellierungswerkzeuge können sich über die Jahre oder aufgrund von neuen Anforderungen ändern. Somit soll auch Virtual Satellite in der Cloud flexibel an die neuen Anforderungen anpassbar sein. Es soll die Möglichkeit geben neue Funktionen zu implementieren, neue Editoren für die Benutzung der Daten zu erstellen, Datenmodelle anzupassen und zu erweitern.

R2 Interoperabilität
Da Virtual Satellite ein konzeptuelles Datenmodell implementiert, soll es die Möglichkeit geben, dass andere Anwendungen oder Dritte auf die Daten von Virtual Satellite zugreifen können. Es soll eine Interoperabilität möglich sein, sodass Projektpartner, die ähnliche Anwendungen oder Datenmodelle haben, die Daten von Virtual Satellite verwenden können. So ist an vielen DLR-Projekten z.B die ESA beteiligt, die ähnliche Systems Engineering Prozesse und Werkzeuge verwendet. In der aktuellen Umsetzung mit Desktopanwendungen mangelt es jedoch häufig an Import- und Exportfunktionen, um einen effizienten Datenaustausch herzustellen.

R3 Dezentral
Ein Cloud-basierter Nachfolger von Virtual Satellite sollte dezentral implementiert sein. So sollen z.B Virtualisierungskomponenten auf Hochleistungsrechner bereitgestellt und bereits existierende Service in die Anwendung integriert werden können. Des Weiteren kann es notwendig sein, das für Datensätze auf Grund von rechtlichen Rahmenbedingungen unterschiedlichen Verfügbarkeiten vorliegen.

3.2 Desginmusteranalyse

Für die Musteranalyse werden verschiedene Designkonzepte für Virtual Satellite zusammengestellt, mit denen eine Migration stattfinden kann und wie die Services miteinander agieren können. Es wird erläutert wie die Muster funktionieren und wie sie in Virtual Satellite eingesetzt werden könnten.

Migration von Desktopanwendung zur Cloud-Architektur
Für die Umwandlung von Virtual Satellite als Desktopanwendung zur Microservice-Architektur ist die Organisation, wie die Software in unabhängige Dienste aufzuteilen, ein entscheidender Punkt für eine erfolgreiche Migration. Um dieses Vorgehen zu ermöglichen, ist eine Kombination aus dem Strangler-Muster mit der Verwendung der Antibeschädigungsebene eine mögliche Wahl.

Das Strangler-Muster erlaubt eine sukzessive Migration der Software, wobei eine Fassade über die Desktopanwendung und die neu erstellten Service implementiert wird. Somit können die Clients auch die entsprechenden Funktionen wiederfinden bzw. sie werden an die richtigen Funktionalitäten weitergeleitet. Das Team kann dabei unabhängig voneinander an verschiedenen Servicen arbeiten und schrittweise die Desktop-Version von Virtual

Satellite auflösen.

Damit aber dennoch eine kompatible Kommunikation zwischen den Cloud-Servicen und der Desktopanwendung vorliegt, kann zusätzlich eine Antibeschädigungsebene zwischen den beiden Versionen implementiert werden. Diese kann Anfragen oder Befehle zwischen den Systemen übersetzen und unterstützt somit die Anforderung R2 Interoperabilität zwischen den beiden Systemen. Bei der Implementierung ist jedoch zu beachten, dass eine solche Ebene mit einer erhöhten Latenz verbunden ist und somit die Performance der Kommunikation beachtet werden muss.

API-Gateway

Das API-Gateway ist eine wichtige und zentrale Komponente für eine Microservice-Architektur. Die Implementierung dieses Musters ist obligatorisch für eine sichere, funktionale und strukturierte Kommunikation zwischen Client und Backend-Services. Es fungiert als Schnittstelle für Anfragen und externe Zugriffe. Dadurch kann der Datenverkehr verwaltet und protokolliert werden.

Somit sollte die Cloud-Architektur für Virtual Satellite ein API-Gateway implementieren. Anhand zusätzlicher Entwurfsmuster wie Gatewayrouting und Gatewayabladung können weitere Funktionen, wie z.B Authentifizierung oder Protokollierung, eingesetzt werden.

Insbesondere ist es mit dem API-Gateway möglich, eine Implementierung des Backends-for-Frontends Musters zu erreichen. Für jeden Client kann für dessen Anforderungen ein zugeschnittenes API-Gateway implementiert werden. Dadurch wird eine performantere und effizientere Datenübertragung erreicht, da für jedes Frontend spezifisch auf dessen Anforderungen eingegangenen werden kann.

Für Virtual Satellite ist dies ein wichtiger Punkt, da auch andere Clients, wie z.B. eine CAD-Anwendung, über ein API-Gateway auf die Strukturdaten eines Satelliten zu greifen und die Datensätze einlesen kann. Solche Clients benötigen ein komplexeres API-Gateway, während Mobil-Geräte weniger komplexere Daten zur Verfügung gestellt bekommen. Damit wird es verschiedenen Clients mit eigenen Frontends und Komplexität ein spezifischerer Zugang zu den benötigten Daten ermöglicht.

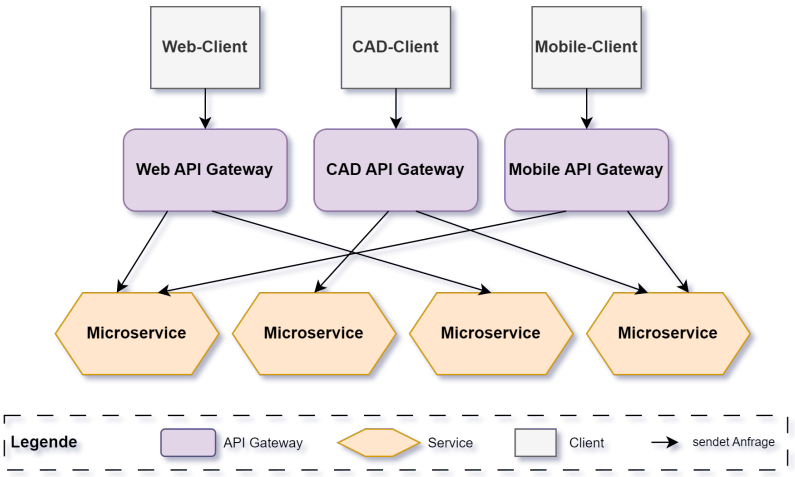

Abbildung 10 Backends for Frontends (Angelehnt an [4])

Das API-Gateway unterstützt in vielen Hinsichten somit auch die Interoperabilität und die Modularität der Architektur. Externe Services können über eine Schnittstelle integriert werden und bleiben in ihrer inneren Struktur unberührt. Die Schnittstelle kann sich um die Übersetzung und Protokollierung zwischen Clients und Backendservice kümmern, sodass die Service problemlos auf verschiedene Clients agieren kann. Da das API-Gateway auch als Vermittler agiert, welcher die Standorten der Dienste kennt, können so die Informationen und Details für die Clients verborgen werden. Da in der Raumfahrt Unternehmen auch Informationen geheim halten müssen, ist es wichtig das Services auch ihre Informationen nur über bestimmte Endpunkte zugänglich machen können.

Koordination von Servicen

Einzelne Services können in einer Microservice-Architektur auf verschiedene Art und Weisen miteinander verknüpft werden. So können Dienste als eine einzige Komponente für eine Aufgabe zu ständig sein oder aber auch mehrere Verarbeitungsschritte nacheinander implementieren.

In Virtual Satellite gibt es verschiedene Funktionen wie z.B die Berechnung von Massen- und Leistungsbudgets, Zustandsdeklarierung von Komponenten oder aber auch die Erstellung von hierarchischen Baumstrukturen. Der Aufbau, welche Funktionen in welche Service zusammengestellt werden können sowie die Interaktion und Datenverwaltung ist eine wichtige Entscheidung um einen performanten Betrieb der Anwendung zu gewährleisten.

Für die Erstellung eines Systemmodells hat Virtual Satellite die Möglichkeit, das System in mehrere Baumstrukturen abzubilden. Baumkomponenten können dabei in anderen Bäumen instanziiert werden und Daten vererben. Die Bäume werden zur Konfigurationskontrolle genutzt, um zum Beispiel verschiedene Ausprägungen eines Systems abzubilden, ohne jedes Mal alle Details vollständig modellieren zu müssen.

Um die hierarchische Baumstruktur in Virtual Satellite zu implementieren, kann das Konzept einer Saga verwendet werden:

Dabei wird zunächst ein NodeService erstellt, der sich um die Verwaltung der gesamten Baumstruktur kümmert und ein ParentService, welcher sich um die relative Beziehung zwischen den Knoten kümmert. Der NodeService kann neue Knoten an die entsprechende Stelle in der Baumstruktur hinzufügen oder entfernen. Die Verwaltung findet dabei mit Sagas statt, welche durch eine Anfrage vom Client ausgelöst werden. Erhält der NodeService eine Anfrage, z.B das ein neuer Knoten dem Produktbaum hinzugefügt werden soll, so startet der NodeService die lokale Transaktion und fügt einen neuen Knoten in die Datenbank. Der ParentService erhält eine Benachrichtigung von NodeService, dass eine neue Komponente vorliegt, und aktualisiert dessen Beziehung zu dessen Elternknoten. Die Saga ist dann beendet, wenn der letzte Schritt, in dem Fall die Aktualisierung der Datenbank des Parents Services, vollendet ist.

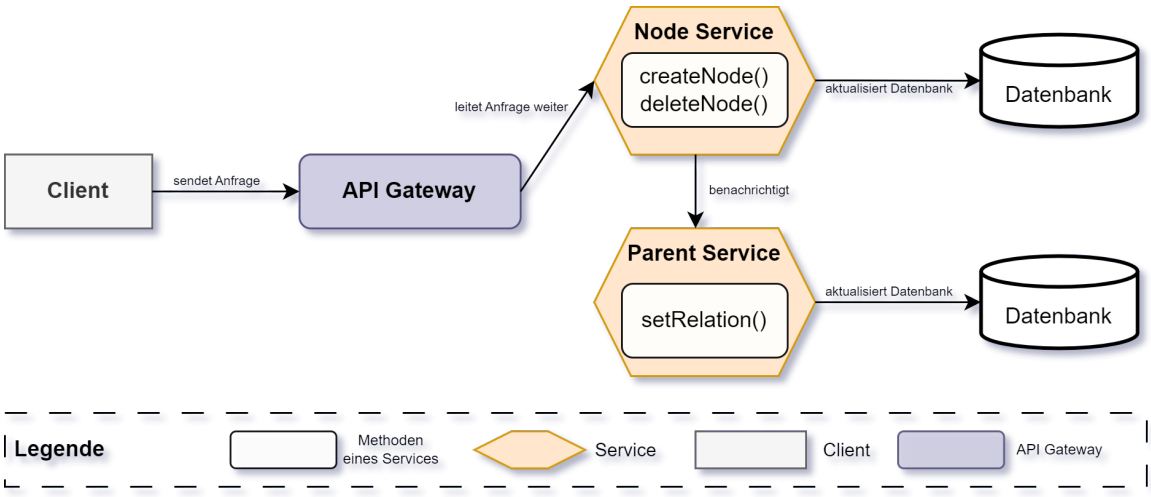

Abbildung 11 Baumstrukturimplementierung mittels Saga

Service können als Chain of responsability aufgebaut werden, wenn sie für das Ergebnis eine abhängige Reihenfolge von aufeinanderfolgenden Verarbeitungsschritten benötigen.

In Virtual Satellite könnte dies für die Implementierung von Simulationsschritte verwendet werden. Zum Beispiel um Thermalanalysen des Satelliten durchzuführen, müssen die Strukturdaten erst verfeinert werden, um danach Materialübergänge zu erkennen und zu analysieren. Diese prozessierten Daten können in existierende Thermalanalysen eingefügt werden, um die Auswirkung der Sonneneinstrahlung auf das Material des Satelliten zu untersuchen.

Kommunikation

Die Kommunikationsart in einer Microservice-Architektur stellt eine zentrale Entwurfsentscheidung dar, da die Wahl der Kommunikationsmechanismen die Performance, sowie die Erreichbarkeit in einem hohen Maß beeinflusst. Die Kommunikation in einer Microservice-Architektur kann unterschiedlich, je nach Anforderung implementiert werden.

So können für Service RESTful APIs erstellt werden, wobei mit einer synchronen Kommunikation die Daten ausgetauscht werden können. Jedoch ist das REST Protokoll am wenigsten performant und kann auch schnell zu Fehlern in der Datenübertragung führen, wenn die Last der Anfragen zu hoch ist. In vielen Hinsichten ist somit die Wahl der asynchronen Kommunikation mit der Verwendung von Nachrichtenvermittler die effizientere Wahl.

In Virtual Satelllite ist die Implementierung von RESTful APIs eine gute Möglichkeit, Service über Endpunkte nach außen zu Verfügung zu stellen. Somit können auch andere Anwendungen auf die Daten vo Virtual Satellite zu greifen.

Datenmanagement

Bei der Wahl der Datenbanktechnologien, ist die Art des Datenaufbaus relevant. Da es in Virtual Satellite keine unstrukturierten Daten gibt, wird für die Services eine SQL-Datenbank verwendet.

Aufgrund der vielen Netzwerkkommunikationen und Datenübertragungen, wird eine Gestaltung der Datenbanken benötigt, die den Anforderungen der Anwendung entsprechen.

Das CQRS (Command Query Responsibility Segregation) bietet eine passende Lösung für eine effiziente Datenverwaltung. Durch die Trennung von Lese- und Schreibvorgängen können die Modelle jeweils optimal auf ihre Aufgaben zugeschnitten werden. Dies führt zu einer verbesserten Skalierbarkeit, da Leseanfragen in der Regel häufiger auftreten als Schreibanfragen und somit unabhängig voneinander skaliert werden können.

In Virtual Satellite kann dies z.B für die Implementierung des QUDV Services verwendet werden. Das QUDV stellt eine Bibliothek von Einheiten zur Verfügung, die für die Berechnung der physikalischen Eigenschaften verwendet wird. Da die Ingenieure während der CE-Studien viel Gebrauch von dieser Bibliothek machen und viele Leseanfragen verschicken, um Einheiten zu erhalten, wird eine höhere Skalierung des Lesevorgangs mehr benötigt als die des Schreibvorgangs.

Deployment

Container, insbesondere die weit verbreitete Plattform Docker, ermöglichen die Einbettung von Anwendungen samt ihren Abhängigkeiten in isolierten, ressourceneffizienten Umgebungen. Diese Isolation vereinfacht die Bereitstellung und Skalierung von Anwendungen und gewährleistet konsistente Umgebungen.

Die Nutzung von Docker und Containern für Virtual Satellite erlaubt eine modulare Strukturierung von Desktopanwendungen in der Cloud, verbessert die Verwaltbarkeit und Flexibilität in einer verteilten Microservices-Umgebung.

4 Prototypenimplementierung

Um eine erste Einschätzung zu erhalten, wie sich Virtual Satellite als eine verteilte Cloud-Anwendung mit Servicen verhält, wird ein Prototyp erstellt, welcher auf die Performance getestet wird. Dabei werden unterschiedliche Load Tests implementiert, die das Verhalten des Services anhand der Eigenschaften von HTTP-Anfragen analysieren, die von virtuellen Nutzern unter bestimmten Zeitspannen, verschickt werden.

Für den Prototypen wird ein bereits existierendes Backend verwendet, welches das QUDV als ein Service implementiert. Das QUDV ist ein Standard, welcher in der Software für die Berechnungen und Konvertierungsaufgaben verwendet wird. Dieser bietet sich als Prototypenimplementierung an, da der Service eine ähnliche Struktur wie Virtual Satellite aufweist. Es besitzt ein Datenmodell, Verarbeitungsfunktionen der Daten und eine UI. Auf diesem Grund sind erste Performancetests dafür geeignet, da anhand des Aufbaus damit Virtual Satellite nachgeahmt werden kann.

Während der CE-Studien werden laufend Budget-Rechnungen ausgeführt, die z.B die Gesamtmasse, Kosten und Leistung des Systems bestimmen und somit zu vielen kumulierten Berechnungen führen. Dabei kann es sich um einzelne oder auch um komplexe Berechnungsketten handeln, die teilweise über mehrere Stunden vorgenommen werden. Dabei stellen sich die Fragen, inwieweit solche Berechnungen aus den CE-Prozessen sich performant in der Cloud verhalten und ob sich die Transformation einer MBSE-Anwendung zu Microservices unter der Berücksichtigung der Performance lohnt.

Somit sollen erste Ergebnisse evaluiert werden, wie sich das QUDV mit vielen Berechnungsanfragen in der Cloud verhalten könnte und wie ein solcher Service für Virtual Satellite auf die Anfragezeit getestet und analysiert werden kann. Des weiteren soll überprüft werden, inwieweit sich eine Skalierung des Services auf die Performance und auf die Lastverteilung auswirkt.

Daraus ergeben sich zentrale Forschungsfragen, die für die Analyse des Prototypens auf die Performance beantwortet werden sollen:

1. Hat die Dauer eine Auswirkung auf die Anfragezeit der Anwendung unter konstanter Nutzeranzahl?
2. Welche Auswirkungen hat eine steigende Nutzeranzahl auf die Anwendung?
3. Wie viele Instanzen werden für eine optimale Performance der Anfragenlast benötigt?

Aufbau des Prototypen
Der Prototyp des QUDV's wird mit dem Django-Webframework implementiert. Als Load Balancer dient ein einfacher HTTP nginx Load Balancer, welcher mit dem Round Robin Verfahren die Anfragen abwechselnd nacheinander auf die Instanzen des QUDV's routet. Die Instanzen werden mit Docker und docker-compose skaliert und deployed. Als Datenbank wird die Default-Datenbank SQLite in Django verwendet.

Eine Synchronisation der Datenbankinstanzen ist in diesem ersten Prototypen nicht implementiert. Da bei den Load Tests für die Berechnungsanfragen keine Synchronisation der

Datenbanken erforderlich ist und auf Operationen wie Löschen und Hinzufügen neuer Einheiten in die Datenbank verzichtet wird. Der Prototyp erlaubt eine Berechnung von zwei Werten mit grundlegenden mathematischen Operationen, Addition, Subtraktion, Multiplikation und Division, sowie eine Konvertierung von einem Wert in eine neue Zieleinheit. In den Testdurchläufen werden jedoch nur einzelne Berechnungsanfragen verschickt. Konvertierungsanfragen sich nicht vorhanden.

Durchgeführt werden die Tests auf einem Ubuntu 20.04.6 LTS Server mit einem Prozessor Intel(R) Core(TM)) i7-4790 CPU @3.60GHz. Die Skalierung auf mehrere Instanzen erfolgt auf einem Rechner, sodass bei den Testdurchläufen mehrere Docker Container sich die Ressourcen eines Servers teilen.

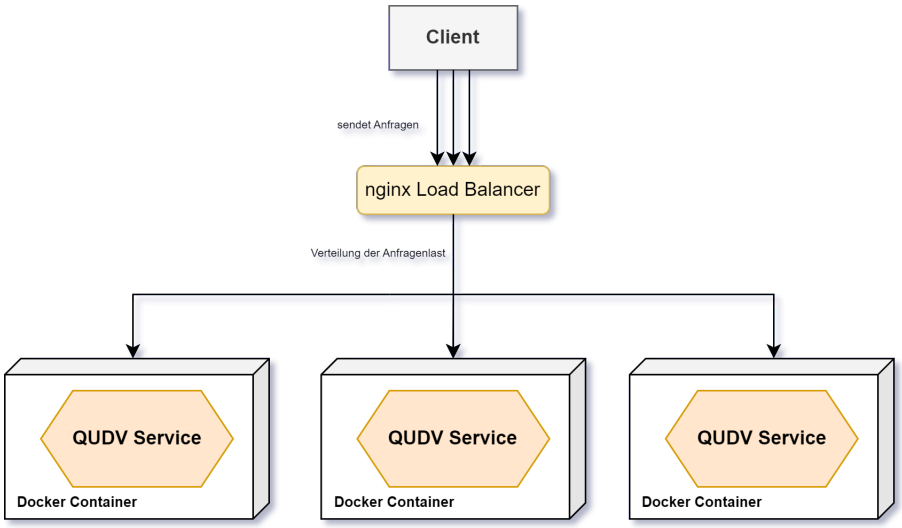

Abbildung 12 Aufbau des Prototypen

5 Testphase

Die Performance Tests werden mit dem Open-Source Tool k6[3] von Grafana Labs durchgeführt. Das Tool bietet die Möglichkeit unterschiedliche Load Tests mit benutzerdefinierten Einstellungen durchzuführen und somit die Zuverlässigkeit, sowie die Performance einer Anwendung zu testen.

Um die Forschungsfragen aus dem vorherigen Kapitel auf die Performance zu evaluieren, werden verschiedene Testszenarien implementiert. Dabei wird die gesamte HTTP-Anfragen-Dauer gemessen, die von dem Testtool k6 mit folgender Formel definiert werden kann [46]. Die gemessene Zeit entspricht somit der Dauer, die eine Anfrage benötigt um zum Server geschickt zu werden, die Wartezeit sowie die Antwort vom Server wieder an den Client zu erhalten.

$$http\ duration = http\ sending + http\ waiting + http\ receiving$$

In den Testszenarien wird die Anzahl der Anwender, sowie die Dauer, wie lange die virtuellen Nutzer Anfragen verschicken können, variiert. Die Variablen basieren dabei auf realistischen Angaben, die in einer CE-Studie durchgeführt werden können. Eine zusammenhängende Sitzung in einer CE-Studie hat in der Regel eine Dauer zwischen 2-4 Stunden, wobei um die 12 Ingenieure und Mitarbeiter daran arbeiten [8]. Somit sind die Testsituationen so aufgebaut, dass realistische Performance-Tests mit dem Prototypen imitiert werden können. Dafür werden unterschiedliche Testszenarien über eine längere Dauer sowie mit steigenden virtuellen Nutzern durchgeführt. In Abb. 13 sind die Testszenarien 1-3 abgebildet und zeigen 9 verschiedene Kombinationen. Eine Skalierung der Instanzen und der Verwendung eines Load Balancers wird bei dem Test mit 2 Stunden und 24 Nutzern umgesetzt, da dies eine typische CE-Studie wieder spiegelt.

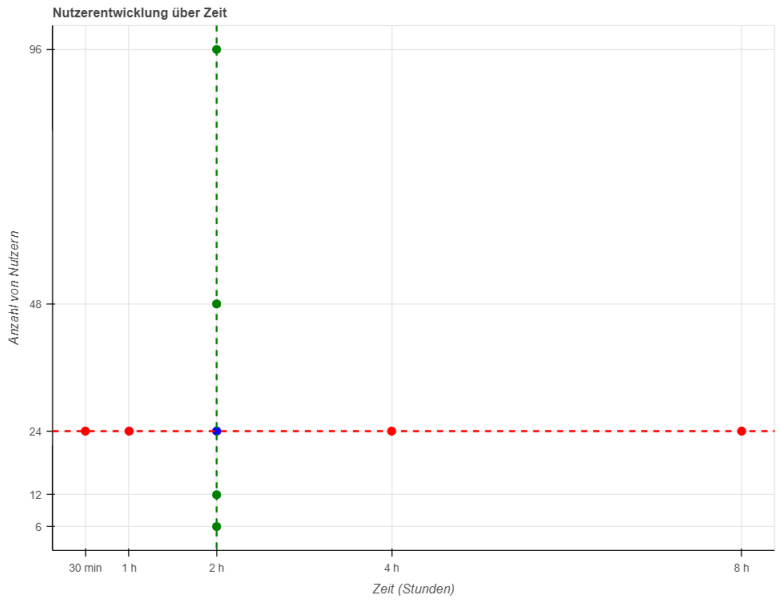

Abbildung 13 Testszenarien 1-3

[3]https://k6.io/ (besucht am 04.07.2023)

1. Hat die Dauer Auswirkungen auf die Anfragendauer? (Soak Tests)

Tabelle 2 Testszenario 1

	1	2	3	4	5
Anzahl Nutzer	24	24	24	24	24
Dauer	30 Minuten	1 Stunden	2 Stunden	4 Stunden	8 Stunden

2. Welche Auswirkungen hat eine steigende Nutzeranzahl? (Stress Tests)

Tabelle 3 Testszenario 2

	1	2	3	4	5
Anzahl Nutzer	6	12	24	48	96
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden	2 Stunden

3. Wie viele Instanzen werden für eine optimierte Performance der Anfragenlast benötigt?

   (a) dynamische Anfragenrate

   (b) konstante Anfragenrate

Tabelle 4 Testszenario 3

	1	2	3	4	5
Anzahl Nutzer	24	24	24	24	24
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden	2 Stunden

4. Wie verhält sich die Anwendung unter sehr vielen Nutzern in kurzer Zeit? (Spike Tests)

   (a) ohne Skalierung

   (b) mit Skalierung

Tabelle 5 Testszenario 4

	1	2	3	4	5
Anzahl Nutzer	96	150	200	250	300
Dauer	15 Minuten	15 Minuten	15 Minuten	15 Minuten	15 Minuten

6 Ergebnisse

Für jedes Testszenario wird eine Tabelle mit der durchschnittlichen, gemessenen *http duration*, sowie der Gesamtanzahl der Anfragen, erstellt. Die Ergebnisse werden in Diagrammen visuell dargestellt. im weiteren Verlauf wird die http duration auch als HTTP-Dauer oder Anfragendauer kenntlich

1. Hat die Dauer Auswirkungen auf die Anfragendauer? (Soak Tests)

Tabelle 6 Ergebnisse Testszenario 1

	1	2	3	4	5
Anzahl Nutzer	24	24	24	24	24
Dauer	30 Minuten	1 Stunden	2 Stunden	4 Stunden	8 Stunden
HTTP-Dauer	424.68 ms	428.1 ms	398.45 ms	389.95 ms	410.76 ms
Anzahl Anfragen	101.616	201.592	433.296	885.474	1.680.788

2. Welche Auswirkungen hat eine steigende Nutzeranzahl? (Stress Tests)

Tabelle 7 Ergebnisse Testszenario 2

	1	2	3	4	5
Anzahl Nutzer	6	12	24	48	96
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden	2 Stunden
HTTP-Dauer	98.8 ms	222.01 ms	398.45 ms	814.97 ms	1.67 s
Anzahl Anfragen	435.636	388.492	433.296	423.878	413.026

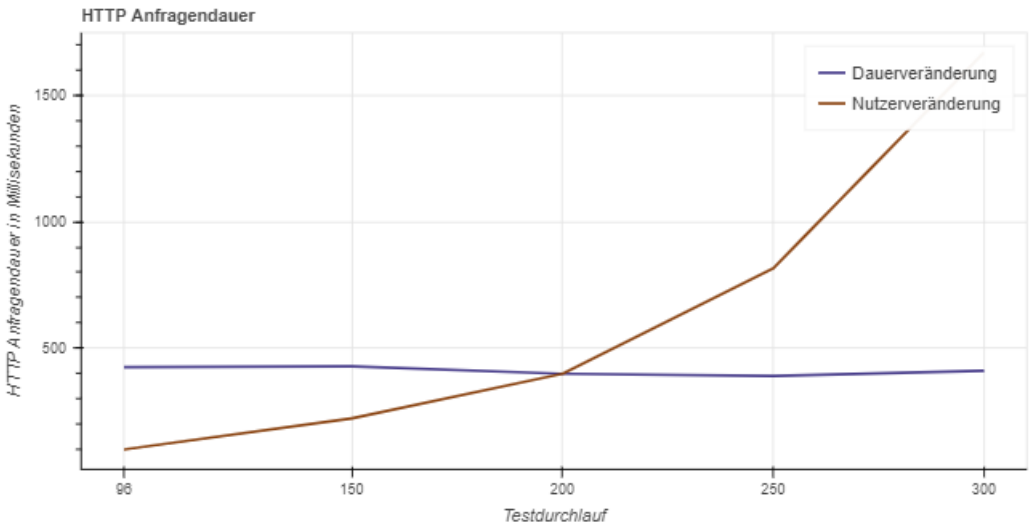

Abbildung 14 Testszenario 1 und Testszenario 2

3. Wie viele Instanzen werden für eine optimierte Performance der Anfragenlast benötigt?

Tabelle 8 Skalierung der Instanzen mit dynamischer Anfragenrate

	1	2	3	4	5
Anzahl Nutzer	24	24	24	24	24
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden	2 Stunden
HTTP-Dauer	398.45 ms	234.98 ms	161.12 ms	118.71ms	100.7ms
Anzahl Anfragen	433.296	734.242	1.069.812	1.438.160	1.684.450

Tabelle 9 Skalierung der Instanzen mit konstanter Anfragenrate

	1	2	3	4	5
Anzahl Nutzer	24	24	24	24	24
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden	2 Stunden
HTTP-Dauer	265.22 ms	50.07 ms	23.99 ms	19.55 ms	17.45 ms
Anzahl Anfragen	172.752	172.752	172.752	172.752	172.752

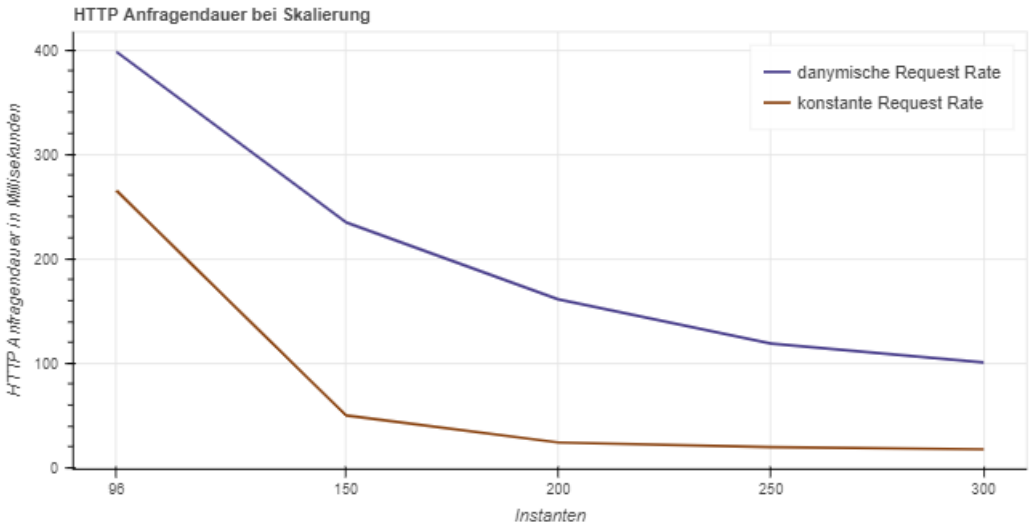

Abbildung 15 Testszenario 3

Tabelle 10 Prozentuale Reduktion der Anfragendauer bei Skalierung der Instanzen

Instanzen	2	3	4	5
Anzahl Nutzer	24	24	24	24
Dauer	2 Stunden	2 Stunden	2 Stunden	2 Stunden
dynamische Reste Rate	41.03 %	59,56 %	70.21 %	74.73 %
konstante Request Rate	81.12 %	90,95 %	92,63 %	93,42 %

4. Wie verhält sich die Anwendung unter sehr vielen Nutzern in kurzer Zeit? (Spike Tests)

Tabelle 11 Spike Test ohne Skalierung

	1	2	3	4	5
Anzahl Nutzer	96	150	200	250	300
Dauer	15 Minuten	15 Minuten	15 Minuten	15 Minuten	15 Minuten
HTTP-Dauer	1.69 s	2.68 s	3.73 s	4.78 s	5.99 s
Anzahl Anfragen	51.030	50.420	48.373	49.162	45.210
Fehlerquote	0.00 %	0.0 %	0.14 %	0.16 %	0.49 %

Tabelle 12 Spike Test mit Skalierung

	1	2	3	4	5
Anzahl Nutzer	96	150	200	250	300
Dauer	15 Minuten	15 Minuten	15 Minuten	15 Minuten	15 Minuten
HTTP-Dauer	624.12 ms	1.03 s	1.43 s	1.85 s	2.31 s
Anzahl Anfragen	138.472	136.294	131.024	121.780	116.722

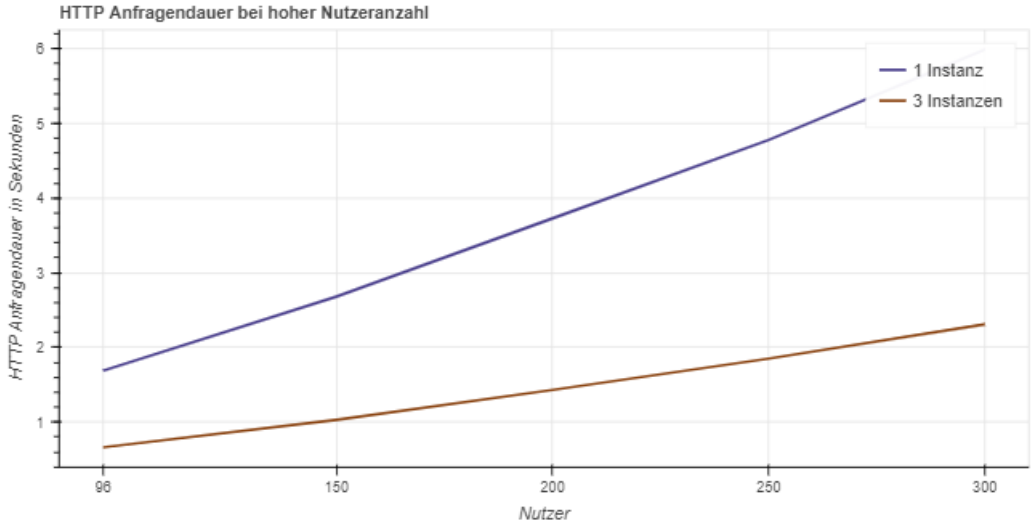

Abbildung 16 Testszenario 4

Tabelle 13 Prozentuale Reduktion der Anfragendauer bei Skalierung der Instanzen

	1	2	3	4	5
Anzahl Nutzer	96	150	200	250	300
Dauer	15 Minuten	15 Minuten	15 Minuten	15 Minuten	15 Minuten
% Reduktion	63,069 %	61,57 %	61,66 %	61,30 %	61,44 %

7 Diskussion

Für die Diskussion der Designmuster-Analyse werden für die Cloud-Architektur die architektonischen Entscheidungen der Komponenten analysiert und auf die zuvor gestellten Anforderungen in Kapitel 3 für Virtual Satellite überprüft. Die Ergebnisse der Performancemessungen werden miteinander verglichen und untersucht, um die Forschungsfragen aus Kapitel 4 zu beantworten. Es werden Schlussfolgerungen aus den Ergebnissen für den QUDV Service in Bezug zu den CE-Studien sowie Virtual Satellite geschlossen und ob sich die Transformation von Virtual Satellite zur Microservice-Architektur eignet.

7.1 Designmusteranalyse

Die erfolgreiche Gestaltung von Microservices-Architekturen erfordert eine entsprechende Auswahl und Implementierung von Designmustern. Im Kontext der Transformation von Desktopanwendungen zu Microservices haben Designmuster eine hohe Priorität, da sie den Übergang von monolithischen Anwendungen zu aufgeteilten, eigenständigen Diensten erleichtern und eine Orientierung in der Umstrukturierung geben, sowie verschiedene Möglichkeiten für die Gestaltung der Interaktion der Dienste und Datenverwaltung ermöglichen.

In Kapitel 3 wurden Aspekte von Designmuster im Kontext von Microservices betrachtet. Dabei wurden grundlegende Entwürfe für die Migration der Desktopanwendung Virtual Satellite zur Cloud Anwendung erstellt. Es wurden mögliche Vorschläge für die Koordination der Funktionalitäten in Virtual Satellite vorgestellt, sowie die Implementierung von Kommunikations- und Datenverwaltungsmechanismen erläutert. Für Virtual Satellite wurden in Kapitel 3 in der Anforderungsanalyse die drei Anforderungen Modularität, Interoperabilität und Dezentralität an die Anwendung gestellt.

R1 Modular
Modularität ist eine Schlüsselanforderung bei der Transformation von Desktopanwendungen zu einer Microservice-Architektur. Die Aufteilung der Anwendung in eigenständige und unabhängige Dienste, die jeweils für spezifische Funktionalitäten verantwortlich sind und die damit einhergehende modulare Struktur erleichtert nicht nur die Entwicklung, Wartung und Erweiterung einzelner Komponenten, sondern fördert auch die Skalierbarkeit und Flexibilität des Gesamtsystems. Durch die klare Abgrenzung der Module können diese unabhängig voneinander aktualisiert werden, ohne dass dies Auswirkungen auf andere Teile der Anwendung hat.

Für Virtual Satellite kann die Anforderung der Modularität anhand der Aufteilung der Dienste erfüllt und erfolgreich umgesetzt werden. Das Konzept der Microservice-Architektur bietet der Desktopanwendung somit vielseitige Lösungen, die Funktionalitäten aufzuteilen und modular zu strukturieren. Neue Funktionen können einfach in die Cloud-Architektur integriert werden, sodass auch z.B Erweiterungen, die in der aktuellen Desktop-Architektur mit Plug-Ins vorgenommen werden, in der Cloud-Anwendung durch eine Integration von neuen Services ermöglicht werden kann.

R2 Interoperabilität

Die Interoperabilität bezieht sich auf die Fähigkeit der einzelnen Microservices, effektiv miteinander zu kommunizieren. Dies bedeutet, dass die Dienste in der Lage sein sollen, Informationen und Daten auszutauschen, Anfragen zu stellen und auf Ereignisse zu reagieren, ohne dass es zu Konflikten oder Inkompatibilitäten führt. Es wird auch die Integration verschiedener Module und Dienste, sowohl innerhalb der Anwendung als auch mit externen Diensten oder Systemen ermöglicht. Eine hohe Interoperabilität erleichtert die Skalierung, die Integration von Drittanbieterdiensten und die Anpassung an zukünftige Änderungen. Eine effektive Interoperabilität erfordert einheitliche Schnittstellen und Protokolle, die die Kommunikation zwischen den Diensten ermöglichen.

Die Anforderung, das Virtual Satellite interoperabel funktionieren soll, ist ein wichtiger Aspekt für die Transformation zur Cloud-Architektur. Da die Daten von Virtual Satellite geteilt werden und dies auch in der Cloud-Architektur ermöglicht werden soll, sind einheitliche Schnittstellen für die Kommunikation und Integration von externen Diensten obligatorisch. Durch die Verwendung von REST-APIs in einer Microservice-Architektur, sowie die Implementierung von angepassten API-Gateways für verschiedene Clients, kann Virtual Satellite eine hohe interoperable Funktionsweise erzielen. So kann damit ein effizienterer und leichterer Zugang für andere beteiligten Teilnehmer von DLR-Projekten, wie z.B die ESA, erlaubt werden.

Interoperabilität von Virtual Satellite kann mit der Implementierung von API-Endpunkten ermöglicht werden. Insbesondere die Implementierung des Backends-for-frontends Musters erlaubt eine effizientere und optimierte Interoperabilität zwischen der Virtual Satellite Cloud-Architektur und externen Zugriffen von anderen Anwendungen.

R3 Dezentral

Die Dezentralität ist ein wesentlicher Aspekt der Microservice-Architektur, der die Unabhängigkeit und Autonomie der einzelnen Dienste betont. Damit wird eine bessere Ausfallsicherheit und Skalierbarkeit ermöglicht, da Ausfälle oder Änderungen in einem Dienst keine Auswirkungen auf andere Dienste haben. Jeder Dienst kann unabhängig skaliert und verwaltet werden, um den Anforderungen gerecht zu werden. Diese Dezentralität erfordert jedoch auch eine effiziente Kommunikation und Koordination zwischen den Diensten, um die Konsistenz und Funktionalität des Gesamtsystems sicherzustellen.

Im Kontext für Virtual Satellite ist die Anforderung dezentral zu funktionieren ein wichtiger Aspekt. Da die Anwendung ihre Daten auch mit anderen Raumfahrtunternehmen, wie der NASA oder der ESA teilen soll, die ihre Server und Hochleistungsrechner an anderen Orten haben, ist eine dezentrale Funktionsweise wichtig. In dem Sinne kann auch die weiterführende Funktion gewährleistet werden, falls Server ausfallen und nicht zu Verfügung stehen. Andere Server können dann weiter für einen Informationsaustausch dienen.

7.2 Performancemessungen

Die Resultate der Performancemessungen geben einen Einblick, wie sich der Prototyp unter verschiedenen Nutzerszenarien verhält. Es lassen sich daraus erste Ergebnisse über die Performance für die Umwandlung zur Cloud schließen und welche Faktoren diese beeinflussen.

Testszenario 1 und Testszenario 2
Die Ergebnisse der ersten beiden Testszenarien stellen die Anfragendauer jeweils unter der Änderung der Zeitperiode sowie der Nutzeranzahl dar. In Abb. 14 wird die Anfragendauer für jeden Testdurchlauf abgebildet.

Für das Testszenario 1, bei welchem die Dauer des Tests unter konstanter Nutzeranzahl erhöht wurde, zeigt sich, dass die Anfragendauer auch nach 8 Stunden die gleiche ist wie nach 30 Minuten. Die zeitliche Periode, wie lange Anfragen von dem Prototypen verarbeitet werden, hat somit keine signifikanten Auswirkungen auf die Performance. Überträgt man das Ergebnis auf eine Sitzung in der CE-Studie bedeutet dies, dass die Arbeitsdauer der Ingenieure keine Rolle für die Performance der Anwendung hat.

Bei dem Testszenario 2 sind im Gegenteil dazu signifikante Unterschiede vorhanden. In Tabelle 7 sind die Werte für jeden Testdurchlauf bei erneuter Verdopplung der Nutzeranzahl angegeben. Es ist zu erkennen, dass eine Verdopplung der Nutzeranzahl auch zu einer Verdopplung der HTTP-Anfragendauer führt. Dadurch sinkt auch entsprechend die Anzahl der Anfragen. Hierbei ist somit auch während einer CE-Studie zu rechnen, dass je mehr Nutzer die Anwendung verwenden, desto langsamer wird die Zeit bis die Mitarbeiter Antworten auf ihre Anfragen erhalten. Somit sinkt die Performance bei einer erhöhten Nutzeranzahl.

Testszenario 3
Im Testszenario 3 wird die Anfragendauer einer typischen Sitzung einer CE-Studie abgebildet, welche mit 24 Mitarbeitern und einer Dauer von 2 Stunden, stattfinden kann. Dabei wurden jeweils 5 Testdurchläufe für die Skalierung mit einer dynamischen Anfragenrate und einer Skalierung mit einer konstanten Anfragenrate durchgeführt.

In Abbildung 15 ist die Anfragendauer für beide Anfragenraten abgebildet. Aus dem Diagramm ist zu erkennen, dass eine Skalierung der Instanzen eine signifikante Auswirkung auf die Anfragendauer und somit auf die Performance hat. Mehr Instanzen des QUDV-Services führen dazu, dass Anfragen schneller verarbeitet werden können. Dies ist darauf zurückzuführen, dass durch die Aufteilung der Last, mehr Anfragen gleichzeitig verarbeitet werden können.

In Tabelle 10 ist die prozentuale Reduktion für beide Anfragenraten abhängig von einer Instanz angegeben. So sinkt die Anfragendauer bei einer dynamischen Anfragenrate bei der Verwendung von 2 Instanzen um 81.12 % und bei einer konstanten Anfragendauer liegt die Reduktion der Anfragendauer bei 41.03 %.

Anhand der Ergebnisse ist zu erkennen, dass bei einer konstanten Anfragenrate, der prozentuale Unterschied der Reduktion von der Anfragendauer nur minimal steigt. So

ist eine Skalierung ab 3 Instanzen für den QUDV Service nicht mehr notwendig, da die Performance dadurch nur sehr gering steigen würde. Die Verwendung von 2 Instanzen verbessert die Performance des Service um 81.12 % und ermöglicht eine viel schnellere Bearbeitungszeit der Berechnungsketten. Eine Skalierung des Services auf das doppelte ist somit eine optimale Lösung, um die Performance der Anwendung zu verbessern.

Testszenario 4
Das Testszenario 4 bildet eine spezifische Situation ab. In kurzer Zeit greifen sehr viele Nutzer auf die Anwendung zu und führen Berechnungen aus. Solch ein Szenario könnte möglich sein, wenn eine besondere Veränderung in Virtual Satellite vorliegt und sehr viele Mitarbeiter, gemeinsam und kurzzeitig Berechnungen ausführen müssen. Für das Szenario wurde eine Zeitdauer von 15 Minuten gewählt mit einer Variation der Nutzeranzahl von 96-300 Nutzern. Dabei wurde das Szenario jeweils ohne Skalierung der Instanzen durchgeführt, sowie mit einer Skalierung von 3 Instanzen. Das Ziel dieses Testszenarios ist es, eine Einschätzung zu erhalten, inwieweit der Service einer hohen Anfragenlast standhalten kann, sowie zu analysieren inwieweit eine Skalierung und die Verwendung eines Load Balancers die Performance bei einer hohen Nutzeranzahl optimiert.

Tabelle 11 stellt die Ergebnisse der Testdurchläufe mit einer Instanz des QUDV-Services dar. Es ist zu erkennen, dass bereits bei einer Anzahl von 96 Nutzern eine durchschnittliche Anfragendauer von 1.69 s vorliegt. Vergleicht man dieses Ergebnis aus dem Testszenario 2, Tabelle 7, so ist der durchschnittliche Wert der Anfragendauer für 96 Nutzern 1.67 s und dies bei einer Dauer von 2 Stunden. Das Resultat bestätigt, dass die Dauer, wie lange Anfragen an eine Anwendung geschickt werden, keine Auswirkungen auf die Performance hat.

Des weiteren treten Fehler ab einer Anzahl von 200 Nutzern auf. Mit einer Fehlerquote von 0.14 % sind bei 200 Nutzern um die 70 Anfragen fehlgeschlagen. Steigt die Nutzeranzahl, so zieht diese auch eine höhere Fehlerquote mit sich. Bei einer Anzahl von 300 Nutzern steigt die Fehlerquote auf 0.49 % . Ein solches Verhalten ist bei einer Skalierung nicht zu beobachten. Tabelle 12 stellt die Ergebnisse bei einer Skalierung mit 3 Instanzen dar. Bei einer Skalierung sind keine fehlgeschlagenen Anfragen vorhanden. Dies lässt sich darauf zurückführen, dass die Anzahl der Anfragen auf die Instanzen verteilt werden können, sodass alle Anfragen verarbeitet werden können. Es liegt auch eine erhebliche Reduktion der Anfragendauer bei einer Skalierung vor. Tabelle 13 stellt den prozentualen Unterschied der Anfragendauer für jeden Testdurchlauf dar. Mit einem durchschnittlichen Prozentsatz von 61,8 % kann die Performance der Anwendung bei der Verwendung von vielen Nutzern innerhalb einer kurzen Dauer verbessert werden.

Eine ähnliche Studie wurde von X. Hong et. al durchgeführt [28]. Die Autoren haben das REST Protokoll mit dem AMQP Protokoll verglichen und die Antwortzeit bei einer steigenden Nutzeranzahl gemessen [28]. Für das AMQP Protkoll wurde der Nachrichtenvermittler RabbitMQ eingesetzt [28]. Die Ergebnisse ergaben, dass bei Steigerung der Nutzeranzahl ab 250, die Antwortzeit der Anfragen mit der RESTful API deutlich langsamer ist als die, die mit dem Nachrichtenvermittler verarbeitet werden [28]. Die Verwendung des REST Protokolls für eine Nutzeranzahl zwischen 50-200 zeigte jedoch, dass dies schneller für die

Verarbeitung der Anfragen ist als mit AMQP [28]. Es zeigt sich, dass ab einem gewissen Punkt, das REST-Protokoll einen signifikanten Umbruch in der Performance innehat [28]. Ebenfalls wurde in der Studie festgestellt, dass bei steigender Nutzeranzahl mit einer Fehlerquote zu rechnen ist [28]. Die Ergebnisse die mit der Verwendung eines Nachrichtenvermittlers erstellt wurden, zeigen, dass die Performance über die Anzahl der Nutzer relativ konstant ist [28]. Wobei aber auch eine etwas größere Reduktion bei einer Anzahl von 250 Nutzern vorliegt [28]. Zusammenfassend hat die Studie ergeben, dass bei einer kleineren Nutzeranzahl, die Verwendung der REST-Protokolls bis zu einem gewissen Punkt performanter ist als die Verwendung eines zusätzlichen Nachrichtenvermittlers [28]. Wird dieser Punkt jedoch überschritten, ist die das REST-Protokoll gegenüber des Nachrichtenvermittlers sehr ineffizient und mit Fehlern verbunden [28].

Der Punkt, ab dem Fehler im QUDV-Service entstehen, liegt bei einer Nutzeranzahl von 200. Eine möglicher Einsatz eines Nachrichtenvermittlers wäre für eine Anzahl ab 200 Nutzern möglich.

7.3 Design-Analyse und Performancemessungen

Die Wahl geeigneter Designmuster beeinflusst unmittelbar die Struktur, Organisation und Interaktion der Microservices. Hierbei sind Aspekte wie die Kommunikation zwischen den Diensten, die Datenverwaltung und die Skalierbarkeit von hoher Relevanz. Die richtige Anwendung von Designmustern kann eine klare Trennung der Funktionalitäten und eine effiziente Kommunikation fördern, was wiederum die Performance der Gesamtarchitektur beeinflusst.

Nach der Evaluierung der Designmuster zeigt sich, dass bestimmte Muster sich positiv auf die Performance ausüben. Beispielsweise kann die Verwendung von Muster wie das Load Balancing dazu beitragen, die Last auf verschiedene Microservices gleichmäßig zu verteilen und somit Engpässe zu vermeiden. Andererseits kann eine unbedachte Anwendung von Designmustern zu unnötiger Komplexität führen und möglicherweise die Performance beeinträchtigen. Dies kann zum Beispiel mit dem Testszenario 3 belegt werden. Eine Skalierung anhand eines Load Balancers fördert die Performance, jedoch nur zu einem bestimmten Zeitpunkt. Skaliert man mehr, bleibt die Performance unberührt, aber der Arbeitsaufwand mehr Instanzen zu warten wird höher.

Des weitern ist zwar die Wahl einer asynchronen Kommunikation performanter und effizienter für viele Anfragen, jedoch aber auch nicht immer die entsprechende Wahl für eine Anwendung. Virtual Satellite soll ihre Daten interoperabel mit anderen Unternehmen über eine REST-API teilen, weswegen es in manchen Fällen wichtiger ist, eher einen einfacheren Datenaustausch zu ermöglichen und mit einer ineffizienteren Performance zu Arbeiten.

Die Wahl von Designmustern sollte daher stets im Hinblick auf die spezifischen Anforderungen der Anwendung erfolgen. Eine umfassende Analyse der Anwendungsanforderungen und eine sorgfältige Abwägung der Vor- und Nachteile verschiedener Muster sind entscheidend, um eine optimale Lösung für das Zusammenspiel zwischen Performance und Design zu erreichen.

8 Fazit und Ausblick

Im Rahmen dieser Bachelorarbeit wurde eine Untersuchung darüber durchgeführt, wie die Transformation von Desktopanwendungen zu Microservices gestaltet werden kann, wobei sowohl die Analyse von Designmuster als auch die Durchführung von Performancemessungen eine zentrale Rolle eingenommen haben.

Dabei wurde in diesem Kontext die Desktop-basierte MBSE-Anwendung Virtual Satellite für die Planung von frühzeitigen Phasen einer Raumfahrtmission auf die Umwandlung zur Microservice-Architektur analysiert.

Die Ergebnisse der Designmusteranalyse ergaben, dass die feingranulare, modulare Struktur von einzelnen, unabhängigen Services und deren Interaktion über verschiedene Kommunikationsmechanismen, eine effiziente Lösung für die Umwandlung der Software in die Cloud ermöglicht. Die Microservice-Architektur erfüllt die Anforderungen Modularität, Interoperabilität und Dezentralität für Virtual Satellite in der Cloud.

Die Performancemessungen zeigten, dass die Performance des Services von der Nutzeranzahl abhängig ist. Dadurch ist die Verwendung eines Load Balancers für die Skalierung notwendig, um die Anfragenlast auf die Instanzen der Service zu verteilen. Diese Ergbnisse sind in Betracht zu ziehen, wenn Virtual Satellite zukünftig von vielen Nutzern verwendet wird, um die Performance und Zuverlässigkeit der Software zu gewährleisten.

Die Resultate geben eine Grundlage für die Transformation der Desktopanwendung zur Cloud-Architektur. Darauf aufbauend können weitere Untersuchungen und Evaluierungen für die Migration durchgeführt werden.

Für die Optimierung der Performance bezüglich der Bearbeitungszeit der Anfragen, ist es für weitere Forschungszwecke maßgebend, weitere Performancemessungen durchzuführen. Da die Nutzeranzahl einen starken Einfluss auf die Anfragendauer hat können weitere Tests mit hoher Nutzeranzahl wertvolle Ergebnisse liefern.

Im Rahmen dieser Bachelorarbeit wurden Spike Tests durchgeführt. Für eine weitere Evaluierungen kann die Durchführung von Stress Tests sinnvoll sein, um Ergebnisse über eine erhöhte Nutzeranzahl bei einer längeren Zeitperiode zu erhalten. Desweiteren kann eine Implementierung mit einem Nachrichtenvermittler durchgeführt werden, um die Unterschiede zwischen dessen Verwendung und REST-APIs für den Fall des QUDV-Services zu analysieren.

Eine weitere Möglichkeit ist die Evaluierung des Request Bundeling Musters, um einzelne Anfragen in Pakete zu verschicken [39].

In der Studie von P. Kumar et al. haben die Autoren die Performance der beiden Kommunikationsprotokolle gRPC und REST mit einem weiteren Protokoll Thrift evaluiert [47]. Die Resultate ergaben, dass die Performance von Thrift eine weitaus bessere Performance aufgrund von schnellerer Serialisierung für die Kommunikation liefert, als REST und gRPC [47].

Literatur

[1] L. Weerasinghe und I. Perera, „Evaluating the Inter-Service Communication on Microservice Architecture,“ in *2022 7th International Conference on Information Technology Research (ICITR)*, 2022, S. 1–6. DOI: 10.1109/ICITR57877.2022.9992918.

[2] J. Lewis. „Microservices.“ (2014), Adresse: https://martinfowler.com/articles/microservices.html (besucht am 12. 06. 2023).

[3] V. Singh und S. K. Peddoju, „Container-based microservice architecture for cloud applications,“ in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 2017, S. 847–852. DOI: 10.1109/CCAA.2017.8229914.

[4] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018, ISBN: 9781617294549.

[5] E. Yuan, „Architecture Interoperability and Repeatability with Microservices: An Industry Perspective,“ in *2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 2019, S. 26–33. DOI: 10.1109/ECASE.2019.00013.

[6] „Was ist modellbasiertes Systems Engineering? - Fraunhofer IPK.“ (), Adresse: https://www.ipk.fraunhofer.de/de/kompetenzen-und-loesungen/digital-engineering/modellbasiertes-systems-engineering/was-ist-model-based-systems-engineering.html.

[7] P. Fischer, D. Lüdtke, C. Lange, F.-C. Roshani, F. Dannemann und A. Gerndt, „Implementing Model Based System Engineering for the Whole Lifecycle of a Spacecraft,“ Sep. 2016.

[8] P. M. Fischer, M. Deshmukh, V. Maiwald, D. Quantius, A. M. Gomez und A. Gerndt, „Conceptual data model: A foundation for successful concurrent engineering,“ *Concurrent Engineering*, Jg. 26, Nr. 1, S. 55–76, 2018. DOI: 10.1177/1063293X17734592. eprint: https://doi.org/10.1177/1063293X17734592. Adresse: https://doi.org/10.1177/1063293X17734592.

[9] „DLR - Concurrent Engineering Facility (CEF) in Bremen.“ (), Adresse: https://www.dlr.de/de/forschung-und-transfer/forschungsinfrastruktur/grossforschungsanlagen/concurrent-engineering-facility (besucht am 04. 07. 2023).

[10] O. M. G. (OMG), „SysML 1.2 RTF-ANNEX C.5—Quantities, Units, Dimensions, Values (QUDV),“ *Needham, MA: OMG*, 2009.

[11] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015, ISBN: 9781491950333.

[12] M. L. Abbott und M. T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, 1st. Addison-Wesley Professional, 2009, ISBN: 0137030428.

[13] L. Bass, P. Clements und R. Kazman, *Software Architecture in Practice*, 3rd. Addison-Wesley Professional, 2012, ISBN: 0321815734.

[14] „Cloudentwurfsmuster - Azure Architecture Center.“ (17. Apr. 2023), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/ (besucht am 15. 06. 2023).

[15] G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch und J. Bogner, „Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs,“ in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 2022, S. 69–79. DOI: 10.1109/ICSA53651.2022.00015.

[16] G. Márquez und H. Astudillo, „Actual Use of Architectural Patterns in Microservices-Based Open Source Projects,“ in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018, S. 31–40. DOI: 10.1109/APSEC.2018.00017.

[17] G. Márquez, M. M. Villegas und H. Astudillo, „A Pattern Language for Scalable Microservices-Based Systems,“ in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, Ser. ECSA '18, Madrid, Spain: Association for Computing Machinery, 2018, ISBN: 9781450364836. DOI: 10.1145/3241403.3241429. Adresse: https://doi.org/10.1145/3241403.3241429.

[18] A. Akbulut und H. G. Perros, „Performance Analysis of Microservice Design Patterns,“ *IEEE Internet Computing*, Jg. 23, Nr. 6, S. 19–27, 2019. DOI: 10.1109/MIC.2019.2951094.

[19] „API-Gateways - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/microservices/design/gateway (besucht am 25. 07. 2023).

[20] „Sidecar-Muster - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/sidecar (besucht am 29. 07. 2023).

[21] „Botschaftermuster - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/ambassador (besucht am 29. 07. 2023).

[22] N. Mohammadi und A. Rasoolzadegan, „A Pattern-aware Design and Implementation Guideline for Microservice-based Systems,“ in *2022 27th International Computer Conference, Computer Society of Iran (CSICC)*, 2022, S. 1–6. DOI: 10.1109/CSICC55295.2022.9780516.

[23] „Muster „Antibeschädigungsebene“ - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/anti-corruption-layer (besucht am 01. 08. 2023).

[24] „Strangler-Muster - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/strangler-fig (besucht am 01. 08. 2023).

[25] „Trennschalter-Muster - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/patterns/circuit-breaker (besucht am 10. 07. 2023).

[26] „Entwurfsmuster für Microservices - Azure Architecture Center.“ (), Adresse: https://learn.microsoft.com/de-de/azure/architecture/microservices/design/patterns (besucht am 20. 07. 2023).

[27] „Kommunikation zwischen Diensten.“ (), Adresse: https://learn.microsoft.com/de-de/dotnet/architecture/cloud-native/service-to-service-communication (besucht am 02. 07. 2023).

[28] X. J. Hong, H. Sik Yang und Y. H. Kim, „Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application,“ in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, S. 257–259. DOI: 10.1109/ICTC.2018.8539409.

[29] B. Anderson. „SQL vs. NoSQL databases: What's the difference? - IBM blog.“ (12. Juni 2022), Adresse: https://www.ibm.com/blog/sql-vs-nosql/ (besucht am 03. 08. 2023).

[30] „Datensouveränität pro Microservice.“ (10. Mai 2023), Adresse: https://learn.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/data-sovereignty-per-microservice (besucht am 21. 07. 2023).

[31] „Saga-Muster - Azure Design Patterns.“ (2023), Adresse: https://learn.microsoft.com/de-de/azure/architecture/reference-architectures/saga/saga (besucht am 09. 07. 2023).

[32] S. Aydin und C. B. Çebi, „Comparison of Choreography vs Orchestration Based Saga Patterns in Microservices,“ in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 2022, S. 1–6. DOI: 10.1109/ICECET55527.2022.9872665.

[33] D. Jaramillo, D. V. Nguyen und R. Smart, „Leveraging microservices architecture by using Docker technology,“ in *SoutheastCon 2016*, 2016, S. 1–5. DOI: 10.1109/SECON.2016.7506647.

[34] „What is load balancing? | IBM.“ (), Adresse: https://www.ibm.com/topics/load-balancing (besucht am 02. 06. 2023).

[35] H. Wang, Y. Wang, G. Liang, Y. Gao, W. Gao und W. Zhang, „Research on load balancing technology for microservice architecture,“ *MATEC Web of Conferences*, Jg. 336, S. 08 002, Feb. 2021. DOI: 10.1051/matecconf/202133608002.

[36] L. Bass, P. Clements und R. Kazman, *Software Architecture in Practice*, 4th. Addison-Wesley Professional, 2021.

[37] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou und Z. Li, „Microservices: architecture, container, and challenges,“ in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, S. 629–635. DOI: 10.1109/QRS-C51114.2020.00107.

[38] L. Bao, C. Wu, X. Bu, N. Ren und M. Shen, „Performance Modeling and Workflow Scheduling of Microservice-Based Applications in Clouds," *IEEE Transactions on Parallel and Distributed Systems*, Jg. 30, Nr. 9, S. 2114–2129, 2019. DOI: 10.1109/TPDS.2019.2901467.

[39] „Request Bundle." (), Adresse: https://microservice-api-patterns.org/patterns/quality/dataTransferParsimony/RequestBundle (besucht am 02. 08. 2023).

[40] A. El Malki und U. Zdun, „Evaluation of API Request Bundling and its Impact on Performance of Microservice Architectures," in *2021 IEEE International Conference on Services Computing (SCC)*, 2021, S. 419–424. DOI: 10.1109/SCC53864.2021.00060.

[41] „Understanding the different types of load tests: goals and recommendations." (2023), Adresse: https://k6.io/docs/test-types/load-test-types/ (besucht am 14. 08. 2023).

[42] „What is load testing? How to create a load test in K6." (), Adresse: https://k6.io/docs/test-types/load-testing/.

[43] „What is stress testing? How to create a stress test in K6." (), Adresse: https://k6.io/docs/test-types/stress-testing/ (besucht am 10. 08. 2023).

[44] „What is soak testing? How to create a soak test in K6." (), Adresse: https://k6.io/docs/test-types/soak-testing/ (besucht am 10. 08. 2023).

[45] „Spike testing." (), Adresse: https://k6.io/docs/test-types/spike-testing/ (besucht am 10. 08. 2023).

[46] „Built-in metrics." (), Adresse: https://k6.io/docs/using-k6/metrics/reference/ (besucht am 20. 07. 2023).

[47] P. K. Kumar, R. Agarwal, R. Shivaprasad, D. Sitaram und S. Kalambur, „Performance Characterization of Communication Protocols in Microservice Applications," in *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, 2021, S. 1–5. DOI: 10.1109/SmartNets50376.2021.9555425.