Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit (Informatik)

Microservices als Basis für skalierbare Web-Applikationen

Autoren	Daniel Zürrer Hans-Daniel Graf
Hauptbetreuung	Walter Eich
Nebenbetreuung	Dr. Mark Cieliebak

Datum	09.06.2017

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……...

……

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Dank der Entwicklung und Verbreitung von Cloud-Computing-Technologien werden Microservice-Architekturen immer häufiger als Alternative zu monolithischen Applikationsdesigns eingesetzt. Microservices sollen Vorteile für Skalierbarkeit einzelner Applikationsteile, Robustheit gegen Ausfälle und Innovationsfähigkeit mit sich bringen.

In dieser Arbeit fassen wir verschiedene Lösungsansätze für diese Problemstellung zusammen, zeigen deren Herausforderungen auf und evaluieren, ob Microservice-Architekturen die genannten Vorteile bieten können. Das Ziel ist eine Übersicht von Lösungsmöglichkeiten mit Empfehlungen für die Umsetzung einer Applikation mit Microservices.

Zu diesem Zweck implementieren wir mehrere Prototypen eines Chatbots mit verschiedenen Architekturen. Neben einem monolithischen Ansatz werden zwei Microservice-Architekturen mit unterschiedlich umfangreichen Services untersucht. Anhand dieser Prototypen werden die Vor- und Nachteile der jeweiligen Architektur evaluiert. Mit Beispielanfragen werden Antwortzeiten in Bezug auf die Skalierung einzelner Services gemessen. Durch die Arbeit an den Prototypen und Erweiterungen an der Funktionalität wird beurteilt, welche Vorteile für die Verfügbarkeit und Erweiterbarkeit durch Microservices entstehen.

Zusammen mit dem Studium von aktuellen Empfehlungen bieten diese Erfahrungen die Grundlage für die Beurteilung der Microservice-Architektur. Die gewonnen Vorteile stehen den Herausforderungen durch höhere Komplexität auf Applikations- und Infrastrukturebene gegenüber. Es ist uns gelungen, die wichtigsten Grundlagen betreffend Microservices zusammenzutragen. Ausserdem zeigen unsere Experimente klare Vor- und Nachteile der verschiedenen Prototypen auf.

Wir haben festgestellt, dass der Serviceschnitt einen der wichtigsten Punkte für den erfolgreichen Einsatz von Microservices darstellt. Wird dieser nicht optimal durchgeführt, so werden die Vorteile der Architektur zunichte gemacht.

Eine weitere wichtige Erkenntnis ist die Tatsache, dass wir keine generelle Antwort auf die Frage *Microservice oder Monolith* liefern können. Welcher Architekturstil besser geeignet ist, hängt stark von den Anforderungen der Applikation ab.

Abstract

Thanks to the development and spreading of cloud computing technologies, microservice architectures are used more and more frequently as an alternative to monolithic application designs. Microservices are supposed to yield benefits in scaling single parts of applications, resiliency against system failures and ability to innovate.

We summarize different approaches for solutions to these problems, show their respective challenges and evaluate if microservice architectures are able to yield the stated benefits. The goal is an overview of possible solutions with recommendations of how to implement an application as a set of microservices.

To achieve this goal we implement multiple chatbot prototypes with different architectures. In addition to a monolithic approach we examine two microservice architectures composed of services of different scopes. Using these prototypes we evaluate the benefits and drawbacks of the corresponding architecture. Response times are measured to judge the scaling of the services for different sample questions. By working on the implementations and by extending the functionality we gauge the benefits for resiliency and extensibility.

This experience adds to the knowledge gained from studying current recommendations and provides the foundation to evaluate the microservice architecture. The benefits during development brought about by microservices are opposed to the challenges of more complex applications and infrastructure. We were able to collect the essential principles of microservice architectures. In addition our experiments show the stark differences between the different prototypes.

We determined that one of the most important aspects for a successful use of microservices is the decomposition of an application into services. Any benefits brought about by a microservice architecture can be nullified by a suboptimal service decomposition.

We also realized that there is no general answer to the question *microservice or monolith*. Which style of architecture is best suited depends heavily on the needs of the application.

Vorwort

Dieses Dokument wurde im Rahmen der Bachelorarbeiten 2017 erstellt. Das gewählte Thema *Microservices* zählt zu den neusten Trends in der Softwareentwicklung und verbindet viele für uns interessante Aspekte. Gerade dies hat uns bei der Einarbeitung ins Themengebiet sehr geholfen und den Lernprozess bedeutend erleichtert.

Mit unserer Arbeit möchten wir ein theoretisches Grundverständnis für Microservices in Web-Applikationen vermitteln und zugleich ein erweiterbares Grundgerüst für Chatbot-Applikationen erstellen.

Wir bedanken uns an dieser Stelle bei Walter Eich und Mark Cieliebak für ihre Unterstützung und die interessanten Diskussionen während unseren regelmässigen Sitzungen.

Inhaltsverzeichnis

1 Einleitung

Microservices zeichnen sich durch eine komplexe Struktur und verteilte Komponenten aus. Der Architekturstil versucht die Vorteile der Strategien aus der serviceorientierten Architektur (SOA) zu übernehmen und gleichzeitig mehr Flexibilität zu bieten. Villamizar *et al.* beschreiben den Unterschied zwischen Microservices und SOA wie folgt: „[...] microservices adopt SOA concepts that have been used during the last decade, but it is an architecture style more focused in achieving agility and simplicity [...]“ [1]. Dabei werden bei Microservices die einzelnen Services mit genau einer Verantwortlichkeit entworfen. Der Begriff *micro* bezieht sich dabei nicht auf die Grösse der Services selbst, sondern auf den Umfang der Funktionalität eines Microservices. Abbildung 1.1 von Martin Fowler soll den konzeptionellen Unterschied einer Microservice-Architektur im Vergleich zu einem monolithischem Ansatz verständlich beschreiben.

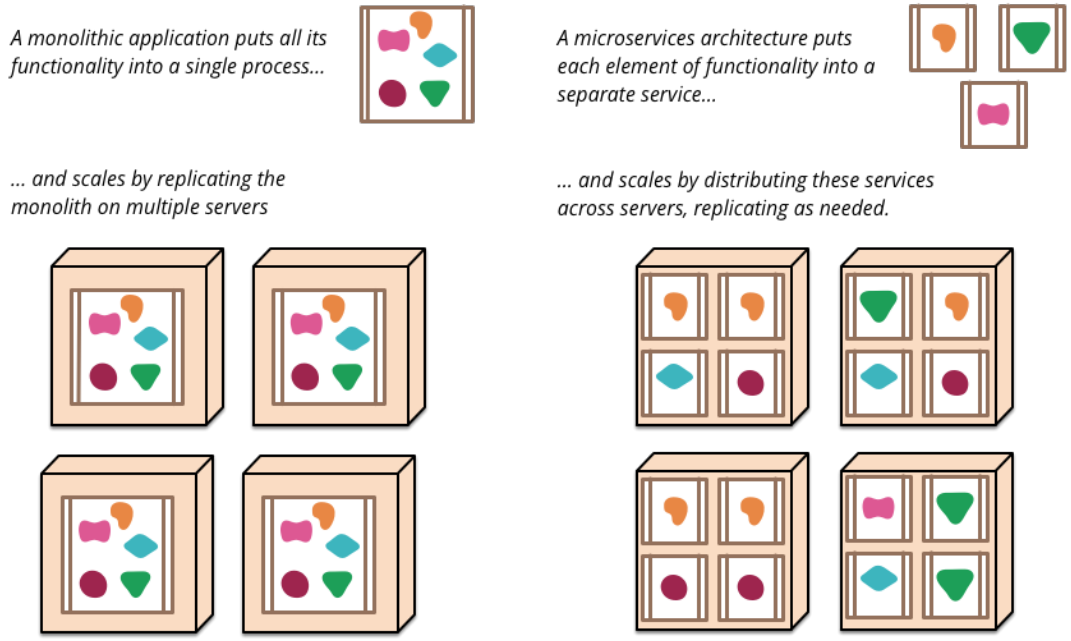

Abbildung 1.1: Monolithische und Microservice-Architektur im Vergleich [2]. Die farbigen Symbole stehen für einzelne Funktionalitäten.

Wie in Abbildung 1.1 dargestellt, zeichnet sich eine Microservice-Architektur durch die Aufteilung einer Applikation in verschiedene, lose gekoppelte Services aus. Im Zu-

sammenhang mit Microservices werden Vorzüge wie flexible Skalierbarkeit einzelner Services, verbesserte Verfügbarkeit der Applikation sowie vergleichsweise einfachere Erweiterbarkeit als positive Merkmale genannt [3]. Im Allgemeinen wird davon ausgegangen, dass Microservices einer klassischen, monolithischen Architektur in diesen Punkten überlegen sind. Ziel der Arbeit ist es, unter anderem diese drei Aspekte zu evaluieren und allfällige Grenzen aufzuzeigen.

Microservices bieten nicht nur Vorzüge, da durch die verteilten Komponenten auch Herausforderungen entstehen, welche bei einem monolithischen Ansatz nicht auftreten. In einer Microservice-Architektur müssen Probleme wie Service-Discovery, Monitoring inklusive Health-Management und Inter-Prozess-Kommunikation gelöst werden. Diese Herausforderungen stehen ebenfalls im Fokus der vorliegenden Arbeit und werden in den folgenden Kapiteln genauer besprochen.

Neben Prinzipien in der Softwarearchitektur setzt die effiziente Entwicklung von Microservices auch organisatorische Unterstützung voraus. Teams, welche an unabhängigen Services arbeiten, müssen in der Lage sein, selbstständig über Aspekte wie eingesetzte Technologien oder Deployment zu entscheiden und diese umzusetzen. DevOps-Strategien kommen damit auch in der Welt der Microservices zur Anwendung. Die Arbeit befasst sich vor allem mit den Merkmalen der Microservice-Architektur. Konsequenzen für die Organisation und Arbeitsweise werden nur am Rande beschrieben.

1.1 Ausgangslage

Der Begriff Microservice hat in den letzten Jahren an Bedeutung gewonnen. Vor allem durch das Aufkommen von Cloud-Computing-Umgebungen haben Microservice-Architekturen grosse Popularität gewonnen. Unter anderem aufgrund der losen Kopplung von einzelnen Services passt dieser Architekturstil sehr gut auf ein verteiltes Cloud-Computing-System. So haben viele Firmen mit der Umstellung in die Cloud auch ihre Software von einer monolithischen Architektur in eine Microservice-Architektur umgewandelt. Bekannte Beispiele dafür sind die Firmen SoundCloud und Netflix.

Netflix hat gemäss eigener Aussage während der Umstellung viel zur Microservice- und Cloud-Computing-Bewegung beigetragen. Dies vor allem, weil Netflix weitestgehend auf etablierte Open-Source-Software (OSS) zurückgegriffen hat, um den Eigenanteil an der Entwicklung möglichst gering zu halten [4]. Selbst entwickelte Software wurde wiederum der Öffentlichkeit zur Verfügung gestellt [5]. Ein beliebtes Beispiel dafür ist Eureka[1], eine Service-Datenbank in welcher jede aktive Instanz eines Services eingetragen wird. Gemäss Netflix habe sich die Umstellung zur Microservice-Architektur sehr gelohnt. Der Architekturstil habe perfekt zu den drei Prioritäten von Netflix gepasst, welche der Abbildung 1.2 entnommen werden können.

[1] `https://github.com/Netflix/eureka` [Stand 01.06.2017]

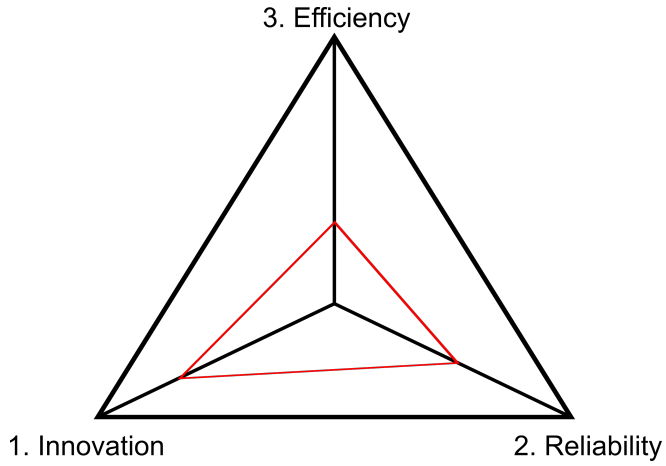

Abbildung 1.2: Qualitatives Netzdiagramm der Netflix Firmenprioritäten (farblich für bessere Lesbarkeit abgeändert) [4].

Gerade die Innovationsfähigkeit habe unter der monolithischen Architektur gelitten, da die verschiedenen Teams stark von einander abhängig waren. Ein neuer Release konnte erst veröffentlicht werden, wenn alle Teams ihre Neuerungen abgeschlossen haben und alle Tests erfolgreich waren. Durch den Wechsel zur Microservice-Strategie, habe sich auch die Firmenstruktur stark verändert [4]. Heutzutage arbeitet Netflix in DevOps-Teams, von welchen jedes für ihre eigenen Microservices von der Entwicklung bis zum Kundenservice zuständig sind. Somit kann auch sichergestellt werden, dass Erweiterungen zum Produkt nicht von der bereits vorhandenen Applikation abhängig sind oder diese beeinträchtigen können. Weiter kann jeder Service die Technologie verwenden, welche die neue Funktionalität am besten erfüllen kann.

Ein weiterer Vorteil, den Netflix durch die Umstellung erreicht hat, ist die Möglichkeit, ihr Produkt genauer skalieren zu können. So müssen nur diejenigen Services vertikal (über zusätzliche Ressourcen) skaliert werden, welche einer grossen Belastung ausgesetzt sind. Dies war zuvor nicht möglich, da nur dem kompletten monolithischen Prozess mehr Ressourcen zugewiesen werden konnten. Andererseits können die Microservices sinnvoller horizontal skaliert werden. Im Gegensatz zur vertikalen Skalierung werden hierbei mehrere Instanzen des gleichen Services erstellt, welche Anfragen parallel bearbeiten. Bei dieser Skalierungsform können neue Instanzen und Maschinen theoretisch unbegrenzt zum laufenden System hinzugefügt werden.

Um die Ausfallsicherheit der neuen Plattform zu garantieren, hat sich Netflix dafür entschieden, mit destruktiven Tests zu arbeiten. Dies bedeutet, dass im laufenden Betrieb in unregelmässigen Zeitabständen zufällige Instanzen der Applikationen zum Absturz gebracht werden. Dies erfordert ein robustes Health-Management, welches den Traffic auf bestehende Instanzen umleiten kann, bis eine neue gestartet ist.

Während Netflix sich dafür entschieden hat, während der Umstellung zwei Implementierungen (Monolith und Microservices) gleichzeitig zu betreiben, schlug Sound-

Cloud eine andere Richtung ein. SoundCloud hat sich für den Wechsel entschieden, weil der eingesetzte Monolith wiederholt an seine Grenzen gestossen war. „In the end we have decided to fundamentally change the way we build products, as we felt we were always patching the system and not resolving the fundamental scalability problem“ [6]. Entsprechend fiel der Entscheid, dass der Monolith flexibler gestaltet werden und laufend in einzelne Microservices ausgelagert werden müsse. Der Fortschritt wurde im Entwicklerblog der Firma dokumentiert [6]-[8] und im Artikel [9] wurde das Fazit gezogen. Diese Berichte beziehen sich mehr auf den Serviceschnitt ausgehend vom Monolith als auf die Vorteile, welche durch die neue Architektur hervorgingen.

Wie bei Netflix hat sich auch bei SoundCloud die Firmenstruktur drastisch geändert. Grosse Entwicklerteams wurden während des Wechsels in mehrere kleine Teams aufgeteilt. Jedes dieser neuen Teams ist für einen spezifischen Bereich zuständig. Die Entwickler konnten die eingesetzten Technologien jedoch nicht völlig frei wählen, da der Unterhalt zu komplex geworden wäre. So gibt SoundCloud ihren Teams den groben Technologiestack vor, um sicherzustellen, dass jeder Mitarbeiter der Firma im Ausnahmefall das nötige Wissen mitbringt, um beispielsweise Wartungsarbeiten am Code durchführen zu können [8].

Mit der neu verwendeten Architektur hat auch SoundCloud vor allem im Bereich Erweiterbarkeit gute Erfahrungen gemacht. So können neue Features viel schneller implementiert werden als zuvor. Weil kein einzelnes, grosses Team für alle Aspekte der Entwicklung zuständig ist, fällt viel Zeit weg, welche früher für Besprechungen und Entscheidungen benötigt wurde.

Nicht nur in der Industrie wird das Thema Microservice-Architektur behandelt. So hielt Martin Fowler and der XConf 2014 einen der bekanntesten Vorträge im Bereich Microservices [2], [10]. Dabei betont er, dass Microservices keine Patentlösung für jede Software sei. Sam Newman veröffentlichte im Jahr 2015 sein Buch *Building Microservices*, welches als generische Anleitung für das Erstellen von Miroservices beschrieben werden kann [11]. Aktuell[2] arbeitet Chris Richardson [12] am Buch *Microservice Patterns*, das voraussichtlich im Herbst 2017 veröffentlicht wird [13]. Im Kapitel 2 dieser Arbeit gehen wir genauer auf die grundlegende Theorie ein, welche wir uns unter anderem mit den genannten Büchern, Vorträgen und Artikeln angeeignet haben.

1.2 Zielsetzung

Das Ziel der vorliegenden Arbeit ist es, die gesammelten Erfahrungen im Bereich der Microservice-Entwicklung aufzulisten und daraus die Vor- und Nachteile des Entwicklungsprozesses und dessen Endprodukts abzuleiten. Dabei wird das Augenmerk vor allem auf die Evaluation von Skalierbarkeit, Verfügbarkeit und Erweiterbarkeit der Applikation gelegt. Die offizielle Aufgabenstellung befindet sich im Abschnitt 8.1 im Anhang der Arbeit.

[2] Stand 09.06.2017

Um die gesetzten Ziele zu erreichen, wird für die Evaluation ein Prototyp eines Chatbots anhand der aktuellen Empfehlungen für Microservice-Architekturen entwickelt. Das Sammeln und Beurteilen der Empfehlungen ist ebenfalls Bestandteil dieser Arbeit. Der Chatbot ist eine Entwicklung der ZHAW und dient als Demonstration für Natural Language Processing (NLP) und Datenverarbeitung. Die Applikation nimmt eine Frage in natürlicher Sprache als Input entgegen, wandelt diese in eine für Computer verständliche Abfrage um und beantwortet die Frage des Nutzers anhand den entsprechenden Informationen aus einer Datenbank. Speziell können mit dieser Software Unterhaltungen über Filme und Fernsehserien geführt werden. Als Datengrundlage stehen dem Chatbot die Internet Movie Database (IMDb) und DBpedia zur Verfügung.

Eine bestehende Implementierung des Chatbots dient als Orientierungshilfe für die Funktionen der Software. Diese Erstimplementierung ist als monolithische Applikation aufgebaut und muss deshalb für diese Arbeit komplett neu als verschiedene Microservices Implementiert werden. Für die Evaluation der Microservice-Architektur wird ausserdem nur eine Teilmenge des bestehenden Funktionsumfangs umgesetzt, um den Umfang der Arbeit dem gegebenen Zeitrahmen anzupassen. Die ausgewählten Funktionen sind genug umfangreich gewählt, dass die Prinzipien einer Microservice-Architektur zur Anwendung gebracht werden können.

1.2.1 Funktionsumfang des Chatbots

Die Implementierung des Chatbots im Rahmen dieser Arbeit ist in der Lage, fünf verschiedene Fragetypen zu den Themen Film und Fernsehserien zu beantworten. Die Fragetypen wurden so ausgewählt, dass anhand deren Umsetzung einige Prinzipien von Microservice-Architekturen getestet werden können. In Tabelle 1.1 sind die implementierten Fragetypen beschrieben.

Fragetyp	Beschreibung
Regie	Regisseurin/Regisseur anhand eines Filmes
Regisseurin/Regisseur	Filme anhand einer Regisseurin/eines Regisseurs
Genre	Filme anhand eines Genres
Erscheinungsjahr	Filme anhand eines Erscheinungsjahrs
Erscheinungsjahr & Genre	Filme anhand der Kombination eines Erscheinungsjahrs und Genres

Tabelle 1.1: Beschreibung des Funktionsumfangs des Chatbots.

Mit dem Chatbot kann über ein Webinterface interagiert werden. Der Fokus dieser Arbeit liegt bei der Architektur der Software. Es wurde Wert darauf gelegt, dass die einzelnen Komponenten der Software repräsentativ für die Chancen und Herausforderungen in einer Microservice-Architektur sind. Die funktional korrekte oder vollständige Implementierung ist dagegen nebensächlich. Die resultierende Applikation dient

zur Evaluation der in der Zielsetzung genannten Aspekte und ist nicht für den Einsatz in Produktionsumgebungen konzipiert.

1.2.2 Vergleich verschiedener Architekturen

Um die Vorzüge und Nachteile einer Microservice-Architektur zu evaluieren, setzen wir drei verschiedene Prototypen um. Alle Prototypen implementieren den im Abschnitt 1.2.1 beschrieben Funktionsumfang und werden auf der gleichen Infrastruktur getestet. Ebenfalls werden wo möglich die gleichen Programmiersprachen und Algorithmen eingesetzt. Die Implementierungen unterscheiden sich einzig bei der Architektur, nämlich bei der Aufteilung einzelner Funktionalitäten in unterschiedliche Services. Damit können wir den Einfluss von externen Variablen bei der Evaluation möglichst gering halten. Die drei umgesetzten Architekturen werden in Kapitel 4 genauer besprochen. In Tabelle 1.2 sind die verschiedenen Service-Aufteilungen kurz beschrieben.

Architektur	Beschreibung
Monolith	Komplette Logik in einem ausführbaren Prozess. Einzelne Datenbank mit komplettem Datenbestand.
Aggregiert	Ein Service für den Datenbank-Zugriff. Einzelne Datenbank mit dem kompletten Datenbestand. Umgebende Applikationslogik ist in anderen Prozessen implementiert.
Atomar	Jeder Themenbereich wird von einem eigenen Service abgedeckt. Mehrere Datenbanken mit spezifischem Datenbestand für je einen einzelnen Fragetypen. Umgebende Applikationslogik ist in anderen Prozessen implementiert.

Tabelle 1.2: Beschreibung der implementierten Architekturen des Chatbots.

1.2.3 Anforderungen an das Resultat der Arbeit

Das Resultat dieser Arbeit soll demonstrieren, welchen Einfluss die in der Einleitung genannten Vor- und Nachteile bei der Umsetzung einer Applikation mit Microservice-Architektur haben. Es soll evaluiert werden, ob die Vorzüge der Skalierbarkeit die höhere Komplexität durch die benötigte Inter-Prozess-Kommunikation überwiegen. Durch eine Erweiterung der Funktionalität wird zudem getestet, wie komplex die Weiterentwicklung einer Microservice-Applikation im Vergleich zu einer monolithischen Software ist.

Weiter werden in der Arbeit Lösungsansätze und Technologien für die Herausforderungen, welche durch eine Microservice-Architektur entstehen, besprochen. Da durch die gewählte Infrastruktur (vgl. Abschnitt 3.9.1) diese Probleme bereits gelöst sind, werden sie in dieser Arbeit nur theoretisch behandelt. Zu diesem Zweck werden verschiedene Erfahrungsberichte und Empfehlungen gesammelt und besprochen.

1.2.4 Übersicht über den Bericht

Im Kapitel *Theoretische Grundlagen* werden Erfahrungen und Empfehlungen gesammelt und besprochen. Das Kapitel gibt einen Überblick über die Komponenten und Muster, welche zur Umsetzung einer Microservice-Architektur benötigt werden. Das nächste Kapitel, *Architekturen und Implementierungen*, enthält die Umsetzung der Chatbot-Prototypen. Anschliessend wird das Vorgehen (Kapitel *Vorgehen und Methoden*) sowie die Resultate (Kapitel *Resultate*) der Evaluation beschrieben. Zum Schluss folgt im Kapitel *Diskussion und Ausblick* eine kurze Reflexion über die Ergebnisse sowie einige Ideen, wie die Resultate dieser Arbeit weiter verwendet werden können.

1.2.5 Voraussetzungen an das Vorwissen des Lesers

In dieser Arbeit werden generelle Begriffe aus der Softwareentwicklung verwendet. Einige Begriffe werden im Glossar definiert, um klarzustellen, wie diese in unserer Arbeit verwendet werden. Dies betrifft vor allem Begriffe, welche mehrdeutig sind oder im allgemeinen Umgang oft ungenau verwendet werden.

Grundlegende Kenntnis über die Prozesse in der Softwareentwicklung sind Voraussetzung, um diese Arbeit vollständig nachzuvollziehen. Vorkenntnisse in den Bereichen Microservices, Cloud-Computing und der Programmiersprache Python sind hilfreich, aber nicht zwingend für das Verständnis der Arbeit.

2 Theoretische Grundlagen

In diesem Kapitel werden wir auf auf die Begriffe Microservices, Microservice-Architektur und Serviceschnitt genauer eingehen. Dabei sollen grundlegende Begriffe und Konzepte erläutert werden, welche wichtig für das Verständnis unserer Arbeit sind.

2.1 Der Microservice

Microservice ist ein Begriff, der lange Zeit ohne konkrete Definition von Entwicklern verwendet wurde. Eine formale Definition des Microservice-Architekturstils fehlt bis heute und wird nach unserer Einschätzung nicht möglich sein. Bei der Beschreibung von Microservices werden grundlegende Prinzipien einer standardisierten Definition vorgezogen. Martin Fowler und James Lewis haben Microservices wie folgt beschrieben:

„In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies“ [3].

Diese Beschreibung deckt die wichtigsten Aspekte eines Microservices ab und erklärt diese sogleich. Der Begriff *micro* bedeutet in diesem Fall nicht, dass die Services nur aus wenigen Zeilen Code bestehen, sondern dass die Funktionalität eines einzelnen Service auf ein Minimum beschränkt ist. Sam Newman definiert Microservices über sieben Prinzipien, welche in Abbildung 2.1 dargestellt sind.

In den meisten Fällen, in welchen eine Microservice-Architektur verwendet wird, sind diese auf einer containerbasierten Lösung deployt. Diese Beobachtung lässt sich vor allem auf die höhere Portabilität und schnelleren Deployment-Vorgang zurückführen [15]. Ein zusätzlicher Faktor ist das Angebot von vielen Cloud-Computing-Plattformen, welche während des Deployment-Vorgangs automatisch einen Container mit allen nötigen Abhängigkeiten um die Applikation erstellen. Beispiele dafür sind Google Cloud Platform (GCP) und Swisscom Application Cloud. Je nach Aufbau der Cloud-Computing-Infrastruktur können virtuelle Maschinen (VMs) aber einen Leistungsvorteil gegenüber Containern bieten. Bei Amazon Web Services (AWS) bieten VMs bessere Performance, da hier Container in VMs betrieben werden und nicht direkt auf einer physikalischen Host-Maschine. Container werden so auf einer höheren

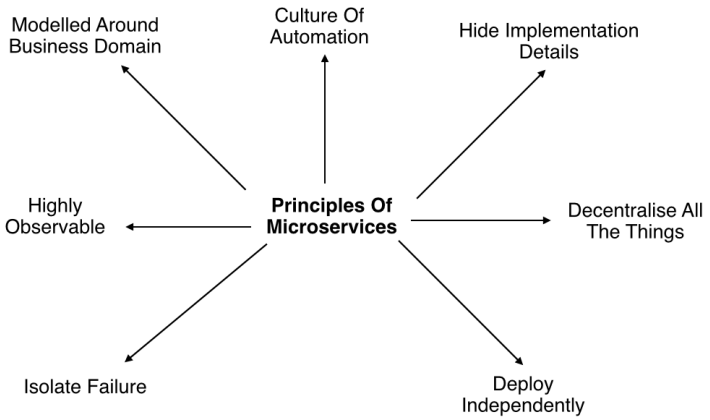

Abbildung 2.1: Sieben Microservice-Prinzipien nach Sam Newman [14].

Ebene als VMs betrieben, weshalb der Container schlussendlich weniger Leistung zur Verfügung hat [15].

Den Entwicklern muss bewusst sein, dass die Kommunikation über das Netzwerk nicht immer erfolgreich sein wird. Möglicherweise ist das Netzwerk zum Zeitpunkt eines Requests überlastet oder der angesprochene Service steht gerade nicht zur Verfügung. Um diesem Umstand entgegenzuwirken, bietet sich defensives Programmieren an. Das bedeutet, jeder Service implementiert entsprechende Gegenmassnahmen, wie dieser sich in einem solchen Fall verhalten soll. Wird dies bei der Entwicklung nicht beachtet, geht ein grosser Vorteil von Microservices, die modulare Unabhängigkeit, verloren.

Anhand der Abbildung 2.2 soll eine mögliche Applikationslogik erklärt werden, welche kaskadierende Ausfälle im System verhindert. Hierbei wird `Service A` vom Client angesprochenen und leitet die Anfrage an `Service B` weiter. `Service B` wird aus einem unbekannten Grund unerreichbar. `Service A` liefert dem Client darauf eine Standardantwort aus. Ab diesem Zeitpunkt wird `Service A` bis zum Ablauf eines Timers keine Anfragen zu `Service B` weiterleiten. Dies soll dazu beitragen, dass `Service B` sich wieder erholen kann. Nach Ablauf des Timers bei `Service A` leitet dieser einen Teil der Anfragen zu `B` weiter. Erhält `A` von `B` darauf korrekte Antworten, wird der Normalzustand wiederhergestellt. Ist dies nicht der Fall, beginnt der Timer bei `A` wieder von vorne. `Service A` folgt hierbei dem Circuit-Breaker-Design-Pattern. Nach diesem Pattern kann `A` sich in einem von drei Zuständen befinden, welche in Tabelle 2.1 beschrieben sind [16], [17].

Der Vollständigkeit halber wird der Unterschied zwischen den Begriffen *Service* und *Instanz* in Zusammenhang mit Microservices erklärt. Ein Service bezeichnet eine einzelne Codebasis, welche von Entwicklern geschrieben, kompiliert und deployt wird. Eine Instanz wiederum bezeichnet den laufenden Prozess eines Services. So können mehrere Prozesse desselben Services zeitgleich laufen und die Anfragen untereinander aufteilen. Diese Begriffe werden im weiteren Verlauf der Arbeit des Öfteren verwendet.

Zustand	Beschreibung
Geschlossen	Die Sicherung ist geschlossen. Die Kommunikation von `A` nach `B` funktioniert normal
Offen	Im übertragenen Sinne ist die Leitung von `A` nach `B` unterbrochen. `A` leitet keine Anfragen zu `B` weiter, bis ein interner Timer abgelaufen ist.
Halboffen	`A` leitet einen Teil der Anfragen zu `B` weiter. Erhält `A` von `B` eine Antwort so wird `A` wieder in den Status Geschlossen, andernfalls in den Status Offen versetzt.

Tabelle 2.1: Beschreibung der möglichen Zustände im Circuit-Breaker-Pattern.

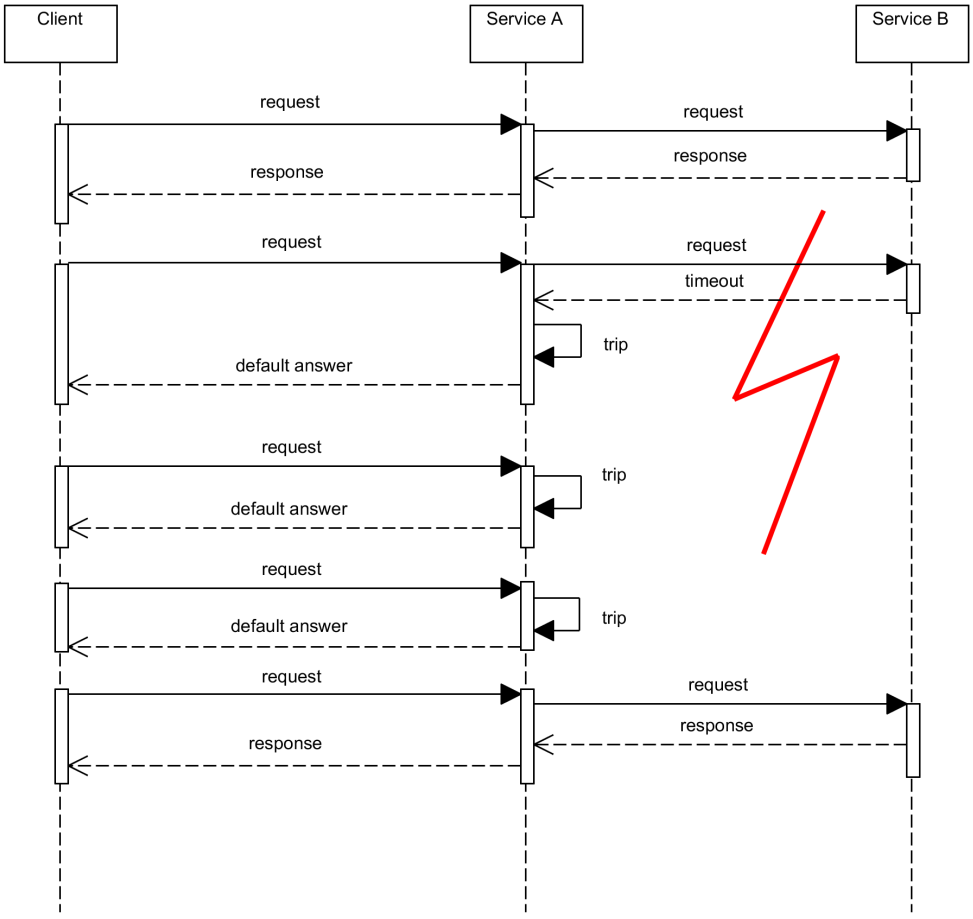

Abbildung 2.2: Beispiel eines Circuit-Breaker-Verhaltens angelehnt an das Beispiel von Martin Fowler [17].

2.2 Die Architektur

Damit eine Microservice-Architektur optimal bestehen kann, sollten gewisse technische und organisatorische Voraussetzungen existieren. Diese können in die Kategorien Team-Management, Daten-Management, Infrastruktur und Automation aufgeteilt werden. In diesem Abschnitt gehen wir auf diese Grundbausteine und -konzepte von Microservices ein. Viele Methoden, die in dieser Architektur eingesetzt werden, wurden aus Bereichen wie SOA und Cloud-Computing abgewandelt und finden auch hier ihren Einsatz.

Grundvoraussetzung für Microservices ist eine Struktur, die eine Applikation als Menge von kleinen und lose gekoppelten Services definiert. Auf dem Würfel der Skalierung (vgl. Abbildung 2.3) wird die Applikation in die Y-Dimension skaliert.

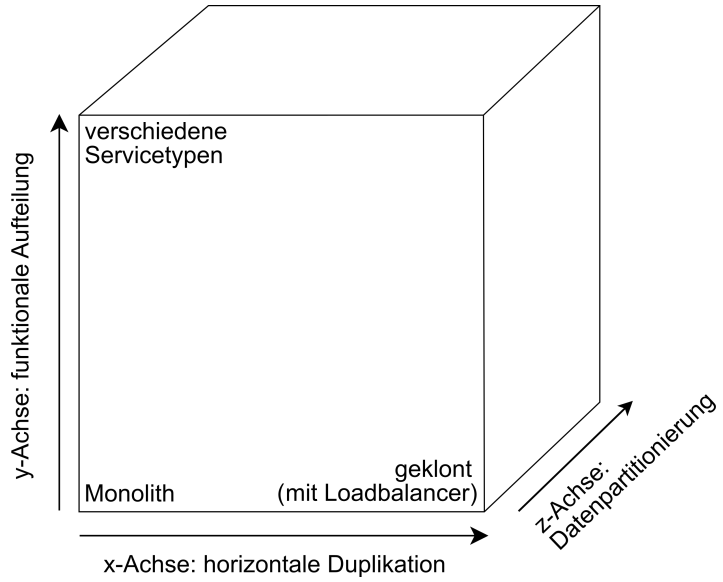

Abbildung 2.3: Würfel der Skalierung angelehnt an das Original vom Buch *The Art Of Scalability* [18].

Es sollte möglich sein, von jedem der Services mehrere Instanzen starten zu können. Das bedeutet, dass die Services selbst keinen Zustand speichern dürfen. So wird jede Instanz sogar während einer Session ersetzbar. Die grosse Ausnahme bilden hierbei jegliche Services, welche Aufgaben im Bereich Persistenz erfüllen müssen. Ein Beispiel dafür ist ein Cache-Service. Um die Konsistenz der Daten gewährleisten zu können, kann eine ereignisgetriebene Architektur verwendet werden. Sobald die Daten eines Services oder einer Instanz verändert werden, wird die Änderung mit einem Event den anderen Services bzw. deren Instanzen bekannt gemacht. Diese können dann ihrerseits die betroffenen Daten anpassen [19].

Was bei einer Microservice-Architektur ebenfalls beachtet werden muss, ist der Zugriff auf die einzelnen Dienste. So muss dem Client die Möglichkeit geboten werden,

die benötigten Funktionen ansprechen zu können. Die Funktionen werden aber von dynamischen Serviceinstanzen bereitgestellt. Auf Seiten des Clients entsteht so das Problem, dass dieser nicht weiss, wie die richtigen Funktionen erreicht werden können.

Mithilfe des API-Gateway-Pattern soll dieses Problem gelöst werden. Hierbei wird dem Client ein einzelner Einstiegspunkt zur Applikation geboten. Die Aufgabe des API-Gateways ist es, die Anfrage des Clients zu verarbeiten und an die zur Erfüllung der Funktionalität benötigten Instanzen weiterzuleiten. Das Pattern sieht auch die Möglichkeit vor, mehrere API-Gateways für verschiedene Clienttypen bereitzustellen. So können API-Gateways für mobile Geräte, eine Web-Applikation oder Applikationen von Drittanbietern bereitgestellt werden. Die API-Gateways können ebenfalls skaliert werden, da sie andernfalls zu einem Bottleneck der Applikation werden könnten [20]. Unsere Chatbot-Prototypen arbeiten ebenfalls mit einem API-Gateway, welcher vom Webclient angesprochen wird.

2.2.1 Team Management

Um optimal von der Flexibilität und der losen Kopplung von Microservices zu profitieren, müssen gewisse Prozesse überarbeitet werden. Conway's Law besagt, dass durch die Struktur einer Organisation implizit auch die Struktur des Produktes vorgegeben wird [21]. Oft werden Entwicklungs- und Betriebsteams nach ihren technologischen Spezialgebiet aufgeteilt. So bilden sich UI-, Automation-, Datenbank- und Backend-Teams, welche für die jeweiligen Bereich des Gesamtprodukts verantwortlich sind. Das führt dazu, dass Änderungen am Produkt Auswirkungen auf jedes Team haben. Somit muss auch jedes Team in den Entscheidungsprozess miteinbezogen werden. Dies kann bereits bei kleineren Aktualisierungen zu langwierigen und teuren Entscheidungsphasen führen. Weiter lässt sich der Wissensaustausch zwischen diesen Fach-Teams nur schwer bewerkstelligen. So ist es schwieriger, durch den Dialog interessante Alternativen zur verwendeten Technologie zu finden.

Im Gegensatz zu diesen Fach-Teams können so genannte Feature-Teams gebildet werden. Diese bestehen aus Spezialisten von verschiedenen Technologien und sind für einzelne Funktionen (Features) der Applikation zuständig. So tangieren Änderungen am Produkt häufig nur noch einzelne Teams und nicht mehr die gesamte Entwicklungs- oder Betriebsabteilung. Abbildung 2.4 zeigt den Vergleich der Verantwortlichkeiten zwischen Feature- und Fach-Teams anhand einer gesamten Applikation dar. Die Applikation wird hierbei als Blackbox betrachtet.

2.2.2 Daten Management

Grundsätzlich gilt bei Microservices die Regel, dass jeder Service wenn möglich und falls nötig seine eigene Datenquelle besitzt. Dies führt zu einem dezentralen Daten-Management, da es keine grosse Datenbanken mehr gibt, welche alle Informationen in verschiedenen Schemata oder Tabellen abspeichert. Die kleineren Datenbanken sollten

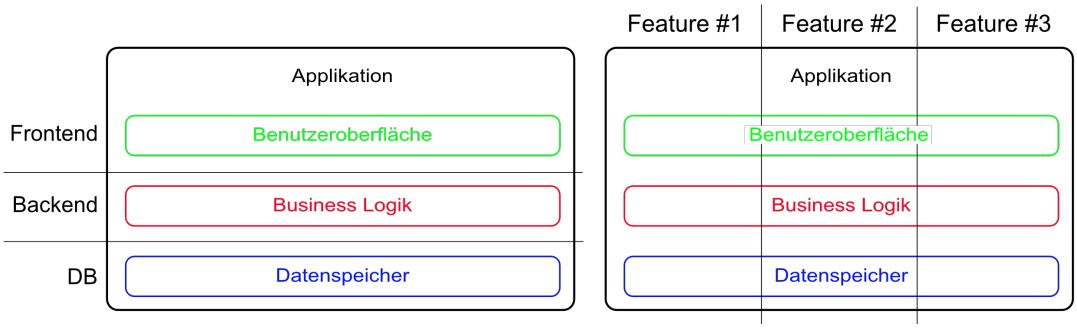

Abbildung 2.4: Aufgabenbereiche im Vergleich: Fach-Teams (links) und Feature-Teams.

weniger komplex und genau auf die Bedürfnisse des Services zugeschnitten sein, welcher sie verwendet. Dieses Design-Pattern wird Database-per-Service genannt [22].

Weiter möchten wir das Shared-Database-Pattern erwähnen, bei welchem mehrere Microservices auf eine zentrale Datenbank zugreifen. Dabei muss darauf geachtet, dass die Services mittels lokalen ACID-Transaktionen auf die Daten zugreifen, damit diese konsistent bleiben [23]. Wir raten aber von diesem Pattern ab, da dies eines der Kernprinzipien von Microservice-Architekturen verletzt: Die Services sollten unabhängig von einander sein. Das bedeutet auch, dass sie nicht von einer gemeinsamen Datenbank abhängen. Beim Database-per-Service-Pattern müssen Daten von verschiedenen Quellen auf der Applikationsebene aggregiert werden, was die Performance beeinträchtigen kann. Diese Beeinträchtigung kann jedoch aufgrund der gewonnenen Modularität in Kauf genommen werden.

Dezentrales Datenmanagement bedeutet jedoch nicht, dass die gleichen Daten in verschiedenen Datenbanken für unterschiedliche Services gespeichert werden. So hat ein Microservice, der beispielsweise mehrere Daten aus verschiedenen Quellen zusammenführt und diese verarbeitet, keine eigene Datenbank im Hintergrund. Dieser erhält die Daten indirekt von denjenigen Microservices, welche für die benötigten Informationen verantwortlich sind. Dies schafft eine neue Herausforderung. Die Kommunikation zwischen den Microservices muss klar definiert sein. Es muss vor der Umsetzung bekannt sein, in welcher Form oder Struktur die Rohdaten gesendet respektive empfangen werden. So können Informationen beispielsweise als JSON, XML oder YAML beschrieben werden. Die Struktur kann aber auch durch die verwendete Kommunikationstechnologie vordefiniert sein. Gerade in diesem Gebiet kann eine Microservice-Architektur stark von der Cloud-Computing-Technologie profitieren.

Zur Kommunikation in der Cloud kann einfaches HTTP verwendet werden. Des Öfteren wird aber so genannte nachrichtenorientierte Middleware (Message-Oriented-Middleware (MOM)) verwendet.

Hierbei übernimmt eine spezialisierte Software oder ein Protokoll das Management der Kommunikation. Diese Middleware kann in zwei Typen klassifiziert werden [24]:

- Brokered
- Brokerless

Im *brokered* Ansatz übernimmt ein sogenannter Messagebroker die Orchestrierung der Nachrichten. Jede Applikation ist mit einem zentralen Broker verbunden, welcher die Nachrichten an die entsprechenden Empfänger weiterleitet. Wie diese Aufgabe gelöst wird, ist dem Broker überlassen. Die offensichtlichste Herangehensweise ist das Publish-Subscribe-Verfahren, bei dem eine Applikation sich als Sender (Publisher) oder Empfänger (Subscriber) an einer Message-Queue anmelden kann. Das Paradebeispiel für eine brokered MOM ist RabbitMQ[1]. Die Arbeit mit einem zentralen Broker bedeutet aber auch, dass dieser zu einem Bottleneck werden kann.

Mit dem *brokerless* Ansatz wird dieses potentielle Bottleneck eliminiert. In diesem Lösungsansatz wird direkt zwischen den entsprechenden Applikationen kommuniziert. Die Middleware übernimmt hierbei nur noch das Caching und den Nachrichtentransfer. Die Vorzeigebeispiele für diesen Ansatz sind ZeroMQ[2] und OpenDDS[3].

In unseren Prototypen verwenden wir ausschliesslich HTTP als Kommunikationsmittel, da dieses sehr einfach zu implementieren ist. Vor allem aber benötigt die Kommunikation über HTTP keine weitere Komponente, welche zuvor eingerichtet werden muss. Dies vermeidet unnötige Komplexität und vereinfacht die Wartung der Prototypen.

2.2.3 Infrastruktur

Während Microservices die direkte Komplexität eines Services durch das Aufteilen in kleinere Services vereinfachen kann, steigt bei diesem Ansatz zeitgleich die Komplexität der benötigten Infrastruktur. Alleine der Umstand, dass die Systeme verteilt werden, bedeutet, dass Komponenten wie Load-Balancing, Service-Discovery, zentralisierte Logs und Health-Management vorhanden sein und durch die Infrastruktur verwaltet werden müssen.

Load Balancing

Das Load-Balancing kann in zwei Varianten umgesetzt werden. Eine Möglichkeit ist, dass die Clientseite das Verteilen der Anfragen übernimmt. Dabei wird dem Client ein Teil der Komplexität in Form von LOC abgegeben. In dieser Variante werden die für den Client relevanten Adressen in dessen Quellcode bekanntgegeben. Dieser kann dann

[1] `http://www.rabbitmq.com` [Stand 10.05.2017]
[2] `http://zeromq.org` [Stand 10.05.2017]
[3] `http://opendds.org` [Stand 10.05.2017]

über ein Verfahren wie Round-Robin seine Requests aufteilen. Zusammen mit einer Microservice-Architektur muss bei der Client-Variante beachtet werden, dass auch die Services untereinander kommunizieren. Das heisst, dass auch jeder Microservice sein eigenes Load-Balancing durchführen muss. Umgesetzt wird das oft so, dass die gleiche Logik, die auf dem Client eingebunden wird, auch in den einzelnen Services verwendet wird.

Für eine reine Webanwendung wird des Öfteren auf eine serverseitige Lösung gesetzt. Hierbei übernimmt eine dedizierte Anwendung das Balancing aller eingehenden Anfragen. Dadurch bleibt die gesamte Logik auf der Serverseite. Der Client kann also die im Web zur Verfügung gestellten Inhalte konsumieren, ohne eigene Load-Balancing-Logik zu implementieren. Allerdings kann bei serverseitigem Load-Balancing das Problem eines Bottlenecks auftreten, falls auf der Serverseite nicht genügend Instanzen zum Bearbeiten der Anfragen bereitstehen.

Beide Lösungsansätze weisen also bedeutende Schwächen auf. In Produkten wie Kubernetes wird die Möglichkeit geboten, einen Kompromiss zwischen beiden Ansätzen zu schliessen. Hierbei kann vor jeden Service ein dedizierter Load-Balancer geschaltet werden. Dieser übernimmt Verteilung der Anfragen auf die zur Verfügung stehenden Instanzen. Die Client-Anfragen werden direkt an einen zentralen Load-Balancer gesendet. Dieser ist seinerseits vor einen API-Gateway geschaltet, welcher dem Client als Einstiegspunkt in die Web-Applikation dient.

Service Discovery

Der Begriff Service-Discovery oder auch Service-Registry beschreibt eine spezielle Anwendung, welche für jeden Service die erreichbaren Instanzen und deren Adresse speichert und verwaltet. Da Service-Instanzen dynamisch hoch- und heruntergefahren werden, ändern sich auch laufend die IP-Adressen, unter welcher ein spezifischer Service erreichbar ist. Die Service-Registry hat die Aufgabe, laufend die entsprechenden Adressen der Instanzen anzupassen, sodass beispielsweise ein Load-Balancer eine Anfrage an `Service B` an eine Instanz von `Service B` weiterleiten kann. Der genannte Beispielfall wird in Abbildung 2.5 noch einmal dargestellt. Die Service-Registry könnte auch vom Client direkt abgefragt werden. Dies wäre vor allem dann sinnvoll, wenn ein clientseitiger Load-Balancer eingesetzt wird.

Die Service-Registry kann durch zwei Pattern aktuell gehalten werden. Entweder implementiert der Service selbst eine entsprechende Logik (Self-Registration-Pattern) oder ein dedizierter Registrator übernimmt diese Aufgabe (Third-Party-Registration-Pattern). Die Prinzipien in der Tabelle 2.2, welche implementiert werden müssen, sind bei beiden Ansätzen identisch. Eine beliebte Service-Registry ist Eureka. Sie wurde von Netflix entwickelt und wird über das OSS-Repository zusammen mit einer Dokumentation angeboten [5]. Kubernetes umgeht das Problem der Service-Discovery mit einer eigenen Lösung [25].

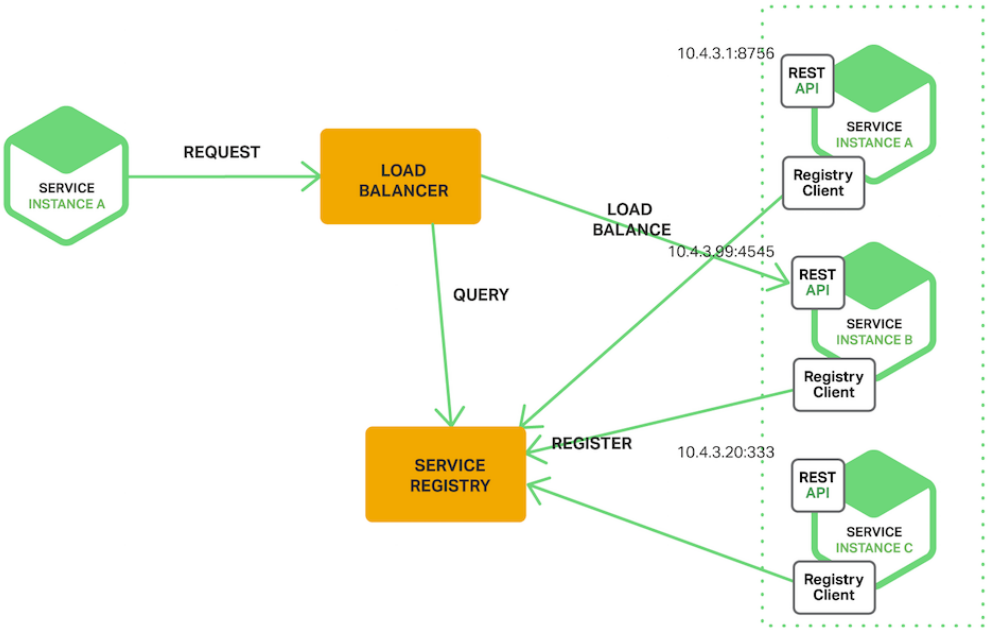

Abbildung 2.5: Serverseitiges-Discovery-Pattern [25].

Prinzip	Beschreibung
Register	Die erste Meldung an die Service-Registry. Sie beinhaltet den Servicenamen und die IP Adresse der Instanz.
Heartbeat	Der Service-Registry wird in regelmässigen Abständen mitgeteilt, dass die Instanz noch erreichbar ist. Fällt dieser Heartbeat aus, so markiert die Registry die Instanz nach einer definierbaren Zeit als inaktiv und löscht den Eintrag zu einem späteren Zeitpunkt.
Unregister	Wird die Instanz ordnungsgemäss heruntergefahren, so wird dies der Registry während des Herunterfahrens gemeldet.

Tabelle 2.2: Beschreibung der Registrierungsprinzipien in einer Service-Registry [25].

Health Management

Obwohl das Health-Management eng mit dem Thema Service-Discovery verwoben ist, muss es dennoch als eigene Komponente in der Infrastruktur betrachtet werden. Ob die Komponente im Endeffekt in einer Service-Registry oder als separater Monitoring-Service implementiert ist, spielt hier keine Rolle. Zu den Aufgaben des Health-Managements gehört es, den Ist- gegenüber dem Sollzustand abzugleichen und auf eine allfällige Diskrepanz zu reagieren. Der Sollzustand wird vom Entwickler festgelegt und in einem Speicher abgelegt, der für das Health-Management-System zugänglich ist. Der Istzustand wird meist direkt vom Health-Managment-System ausgelesen.

Damit die nötigen Informationen von einem externen System verwendet werden können, sollten die Services einen „[...] health check API endpoint (e.g. HTTP /health) that returns the health of the service“ implementieren [26]. Dabei sollte der Service gemäss [26] folgende Informationen auf dem Endpoint anbieten:

- Verbindungsstatus zu den Teilen der Infrastruktur, die der Service benötigt
- Stand der Ressourcen auf dem Host/Container der jeweiligen Instanz
- Applikationsspezifische Logik

Die Informationen werden jeweils periodisch abgefragt und geprüft. Als mögliche Reaktion auf eine Diskrepanz kann das System einen Alarm auslösen, der in Form eines E-Mails oder SMS an die zuständigen Personen eskaliert wird. Grundsätzlich sollten die Gesamtsysteme so automatisiert wie möglich verwaltet werden. In diesem Fall könnte das Health-Management-System automatisch eine Instanz neu starten, löschen oder initiieren. Zeitgleich kann das System das Routing zur beschädigten Instanz verhindern.

Mögliche Produkte, welche die Aufgaben eines Health-Management-Systems übernehmen können, sind HM9000[4] von Cloud Foundry oder Consul[5]. Consul ist ein Hybrid, der unter anderem Service-Discovery und Health-Management übernehmen kann. HM9000 dagegen ist ein reiner Health-Manager, welcher in der Cloud Foundry-Plattform implementiert ist.

Log Management

Die Kernaufgabe eines Log-Management-Systems in einer Microservice-Architektur ist es, die verschiedenen Logs der Serviceinstanzen an einem zentralen Ort zu speichern und allenfalls darzustellen. So wird die Problemstellung von verschiedenen Logfiles an verteilten Orten gelöst. Die Informationen können durch einen Log-Service oder durch eine dafür zuständige Softwarebibliothek zusammengetragen werden.

Ein Log-Service kopiert das Logfile von jeder Instanz periodisch an einen zentralen Ort, wo es mit den Logfiles von anderen Instanzen aggregiert wird. Während des Kopiervorgangs werden die Logfiles mit weiteren Metadaten wie Servicenamen und Instanzinformationen erweitert. Diese Lösung hat den Nachteil, dass die zentralisierten Informationen nicht zu jedem Zeitpunkt auf dem aktuellsten Stand sind. Vorteilhaft ist, dass sich die einzelnen Services nicht um ihren Logoutput kümmern müssen. Diese Aufgabe fällt ausschliesslich dem Log-Service zu.

Sollen die Logfiles durch eine Softwarebibliothek zusammentragen werden, muss bei der Entwicklung jedes einzelnen Services darauf geachtet werden, dass die Bibliothek eingebunden und korrekt verwendet wird. Die Bibliothek ist dafür zuständig, alle Logs

[4] `https://github.com/cloudfoundry/hm9000` [Stand 23.05.2017]
[5] `https://www.consul.io/intro/index.html` [Stand 23.05.2017]

der einzelnen Instanzen direkt an einen zentralen Ort umzuleiten und diese möglicherweise mit Metadaten wie dem Servicenamen und Instanzinformationen anzureichern. So kann jedes Logfile ein standardisiertes Format implementieren. Der grosse Nachteil dieser Lösung ist die Abhängigkeit aller Services von der gleichen Bibliothek.

Um die Logs unserer Prototypen zu standardisieren, haben wir eine eigene Softwarebibliothek geschrieben, die von den einzelnen Services implementiert wird. Zu beachten ist, dass das Zusammentragen der Logs bereits durch die zugrundeliegende Infrastruktur erledigt wird. Dadurch verringert sich die Komplexität der eigenen Bibliothek massiv. Die Bibliothek wird im Abschnitt 3.6.4 genauer erklärt.

Viele Plattform as a Service (PaaS) Lösungen implementieren die beschrieben Komponenten der Infrastruktur bereits von Haus aus. So können sich die Entwickler ausschliesslich mit den Problemstellungen ihrer Services befassen und das fertige Produkt deployen. Die Google App Engine (vgl. Abschnitt 3.9.1) beispielsweise bindet den Service während des Deploy-Vorgangs automatisch an die nötige Infrastruktur. Dieser Vorgang nennt sich auch *staging*.

2.2.4 Automation

Im Bereich der Microservice-Architektur sind automatisierte Deployments und automatisierte Tests sehr zu empfehlen. Sie unterstützen Entwickler dabei, für das Produkt ständig neue Features zu implementieren. Diese Features können in Form von neuen oder aktualisierten Services auftreten. Updates für eine Applikation sollten zu jeder Zeit deploybar sein und zeitgleich ein kleines Delta für jedes Update aufweisen. So wird das Risiko für grosse Probleme beim Deployment vermindert. Generell ist eine solche Deployment-Pipeline auch zu empfehlen, wenn keine Microservice-Architektur verwendet wird [27].

Das Testing dient dabei unter anderem zur Sicherstellung der Qualität des Produktes. Damit sind nicht nur die verschiedenen Testtypen wie Unit-Tests oder Integration-Tests gemeint, sondern auch destruktive Tests, welche während des normalen Betriebs der Applikation durchgeführt werden können. Wie in Abschnitt 2.1 bereits angesprochen, muss bei der Microservice-Architektur mit Fehlern gerechnet werden. Die destruktiven Tests sollen in einem kontrollierbarem Umfeld das Verhalten des Systems aufzeigen und bestätigen, dass sich die Applikation von einem Fehler erholen kann. Oft werden diese Art von Tests auf einer Testumgebung durchgeführt, welche das Verhalten in der Produktionsumgebung simuliert. Das Beispiel von Netflix zeigt, dass die Tests auch direkt auf der Produktionsumgebung durchgeführt werden können [4].

Obwohl die Microservices möglichst unabhängig voneinander sein sollten, können gewisse Abhängigkeiten nicht vermieden werden. Gerade beim Einführen von neuen Features oder Produktupdates kann dies zum Problem werden. „In the case of collaborating services, one of the stickiest points is evolution“ [28]. Für diese Problemstellung bietet das Tolerant-Reader-Pattern von Martin Fowler eine mögliche Lösung. Dieses Pattern besagt, dass die Services in einer Übermittlung jeweils nur die Daten beach-

ten, die sie zum Ausführen ihrer Logik benötigen. In unseren Prototypen beispielsweise erhalten wir vom externen Service api.ai-Daten in Form von JSON. Dieses JSON wird von unseren Microservices weiter verarbeitet. Unsere Services lesen ausschliesslich die Felder des Dokuments aus, welche für ihre Funktionalität wichtig ist. Würde api.ai im Zuge eines Updates die gesendeten Daten weiter anreichern, so wären unsere Services nicht davon betroffen und würden trotzdem weiter funktionieren.

2.3 Serviceschnitt

Das etablierte Pattern für das Designen von Microservice-Architekturen ist Bounded-Context. Das Pattern ist ein zentraler Teil aus der Disziplin des Domain-Driven-Designs. Das Pattern wurde bereits vor dem Aufkommen von Microservice-Architekturen zur Aufteilung von sehr grossen Domänen verwendet. Ursprünglich wurde die Aufteilung vor allem zur einfacheren Kommunikation dank homogenen Themengebieten durchgeführt. Dies erlaubt es, mit Stakeholdern über ihre jeweiligen Fachgebiete zu reden. Werden wie bei Bounded-Context die Domänen als einzelne Kontexte betrachtet, wird klar, dass die gleiche Technik auch für das Definieren von einzelnen Services Sinn macht. Bei Bounded-Context wird auf einem Domänenmodell eine Gruppe von eng zusammenhängenden konzeptionellen Klassen in einen Kontext zusammen gefasst. Als eng zusammenhängend kann beispielsweise eine Reihe von Klassen bezeichnet werden, welche auf einer gemeinsamen Datenbasis arbeiten. Ein Beispiel einer solchen Aufteilung ist in Abbildung 2.6 dargestellt. In einem idealen Serviceschnitt lässt sich jeder Bounded-Context in einen Microservice umwandeln, welcher über eine klar definierte API angesprochen werden kann [29].

Bei einer Umstellung vom Monolith zu Microservices ist es sehr zu empfehlen, eines der Gang of Four (GoF) Pattern wie Fassade oder Adapter zu verwenden. So kann während der Umstellung zwischen einem Microservice und einer noch nicht migrierten Funktion gewechselt werden. In diesem Zusammenhang wird die Verwendung solcher Patterns auch *Antikorruptionsschicht* genannt [29].

Der Serviceschnitt vom aggregierten Chatbot wurde ebenfalls mit der Methode Bounded-Context durchgeführt. Während wir beim atomaren Chatbot die kleinst möglichen Funktionen geschnitten haben. So konnten wir auf Basis von zwei Prototypen unsere Experimente durchführen.

2.4 Aktueller Forschungsstand

Gerade weil Microservices einen relativ neuen Interessenbereich[6] in der Informatik darstellen, sind dazu in den letzten zwei Jahren viele Forschungsarbeiten veröffentlicht worden. Dies bestätigt auch eine kürzlich publizierte Übersichtsarbeit, in der

[6] Definition von Fowler und Lewis stammt aus dem Jahr 2014 [2], die von Newman aus 2015 [11].

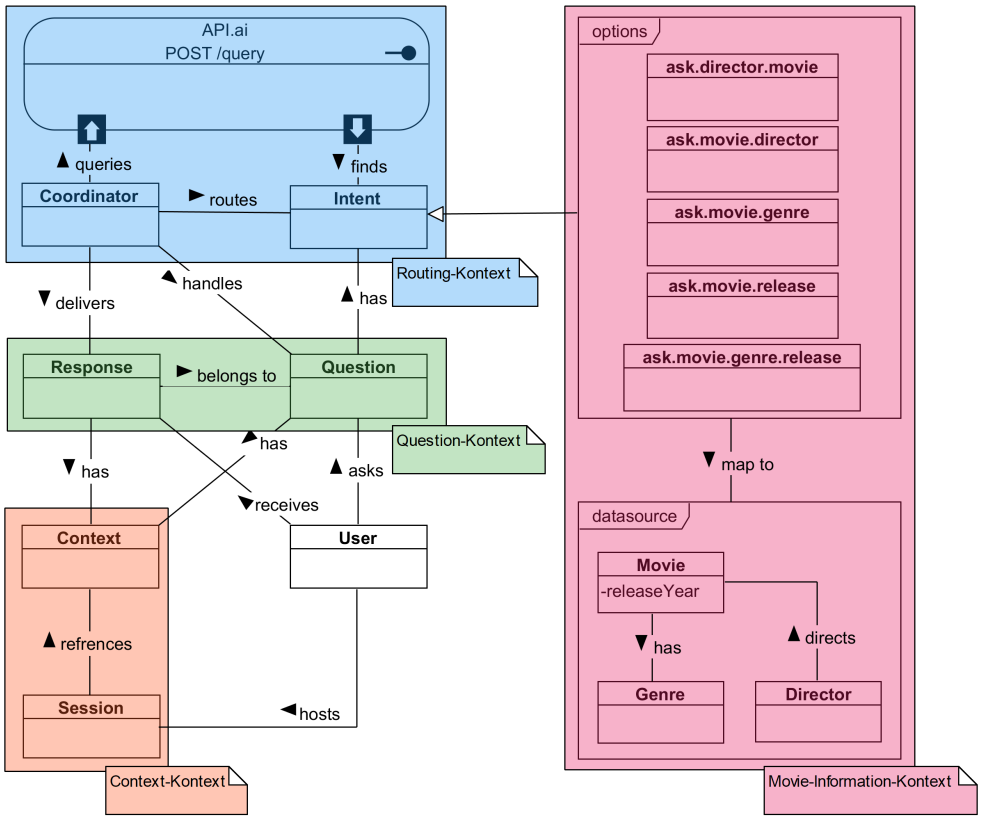

Abbildung 2.6: Beispiele von Bounded-Contexts anhand des Chatbot-Prototypen.

71 veröffentlichte Papers von vier verschiedenen Quellen[7] ausgewertet wurden. Dabei stellten sie fest, dass vor allem in den Jahren 2015 und 2016 viele Arbeiten zum Thema Microservices veröffentlicht wurden. Weiter wird aufgezeigt, dass Performanz, Unterhalt und Cloud-Technologien zu den populärsten Themen im Bereich Microservices gehören, während Themen wie Sicherheit und Testbarkeit seltener behandelt werden. Generell wurde mehr Forschung für spezifische Probleme betrieben, statt im Bereich der allgemeinen Evaluation [30].

Im Jahre 2015 veröffentlichten Villamizar *et al.* an der Computing Colombian Conference ihre Arbeit mit dem Titel *Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud.* In ihrer Arbeit testeten sie anhand eines Prototypen die Nutzbarkeit von Microservices im direkten Vergleich zum Monolithen. Zudem berechneten sie die Kostendifferenz beider Lösungen. Als Cloud-Service-Provider wählten sie AWS. Während ihrer Arbeit machten sie folgende Feststellung: „Based on the case study, microservice architecture should be employed in businesses that need to scale their applications to hundreds of thousends or millions of users because its implementation requires additional abilities that are not present in many companies. [...] For applications with a small number of users (hundreds or thousands of users), the monolithic approach may be a more practical and faster way to start“ [1]. Sie erläuterten auch, dass die meisten Beispiele aus der Industrie bestehende monolithische Applikationen in Microservices umgewandelt haben, als sie auf Skalierungsprobleme stiessen, statt von Grund auf Microservices zu verwenden. 2017 veröffentlichten Villamizar *et al.* ein weiteres Paper, in dem sie sich mit den Kostenberechnungen von Microservices in einer Cloud-Computing-Umgebung befassten. Genauer gesagt wird bei dieser Arbeit ein Kostenvergleich anhand von drei Prototypen einer Web-Applikation durchgeführt [31].

Eine weitere interessante Arbeit von Do *et al.* befasst sich mit skalierbaren Routing für Microservices, welche zwingend einen Zustand haben müssen. Dies stellt eine Abweichung von der Norm dar, da Microservices idealerweise zustandslos sind. Ein Beispiel dafür wäre ein Warenkorb in einem Onlineshop. Dieser Service müsste eine konsistente Ansicht bieten, auch wenn sich der Benutzer von verschiedenen Quellen aus anmelden kann. Die Evaluation ihrer Arbeit führten sie anhand eines Reservationssystems für eine Cloud-Computing-Infrastruktur durch. In ihrem Prototypen konnten sich Benutzer eigene VMs reservieren. Während des Reservierungsprozess mussten Do *et al.* sicherstellen, dass während einer Session die Anfragen des Benutzers an die Serviceinstanzen weitergeleitet werden, welche zu Beginn der Session angefragt wurden. Sie konnten dabei einen effizienten Routingmechanismus entwickeln, der ihre Anforderung erfüllt [32].

Neben den oben beschrieben Arbeiten wird auch weiterhin an neuen Publikationen im Themenbereich Microservices gearbeitet. Wie bereits im Abschnitt 1.1 erwähnt, arbeitet Chris Richardson am Buch *Microservice Patterns*, welches eine Sammlung von wichtigen und nützlichen Design-Patterns für Microservice-Architekturen beinhalten wird. Dieses kann mit dem Buch der GoF für objektorientierte Programmiersprachen

[7] IEEE Xplore, Scopus, Web of Science, ACM Digital Library

verglichen werden. Geplant ist ebenfalls, dass die in Abschnitt 2.1 aufgeführte Arbeit von Salah *et al.* weitergeführt wird. Sie möchten ihre Experimente auch auf weiteren Cloudplattformen wie GCP und Microsoft Azure durchführen [15].

Weitere interessante Arbeiten, die sich unter anderem mit dem Themengebiet Microservice in IoT [30] befassten, werden in dieser Arbeit aufgrund mangelnder Relevanz nicht vertiefter beschrieben.

3 Architekturen und Implementierungen

In diesem Kapitel werden die im Abschnitt 1.2.2 kurz beschriebenen Architekturen detailliert besprochen. Die jeweils gewählte Aufteilung in einzelne Services, wichtige Implementierungsdetails sowie das Deployment sind Gegenstand der folgenden Abschnitte.

3.1 Webserver

Für alle drei Architekturen haben wir uns entschieden, einen separaten Webserver für die statischen HTML-, CSS- und JavaScript-Dateien zu implementieren. Alle drei Implementierungen verwenden den selben, in Python geschriebenen Webserver. Die statischen Files sind äquivalent.

Der Fokus der Arbeit liegt auf der Architektur, weshalb das UI einfach gehalten wurde. Abbildung 3.1 zeigt das UI für das Resultat der Anfrage *„Movies by Stanley Kubrick.“*

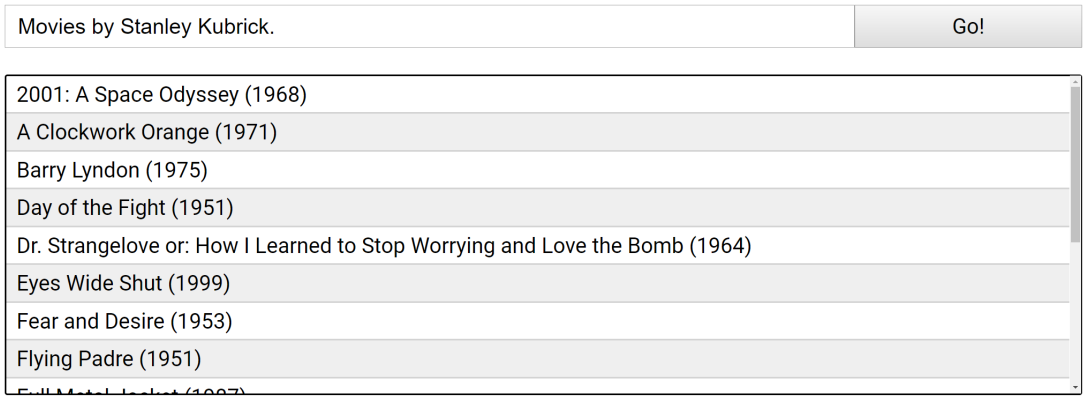

Abbildung 3.1: UI für das Resultat der Anfrage *„Movies by Stanley Kubrick.“*

Die Clientlogik ist ebenfalls sehr einfach gestaltet. Zum Absenden einer Anfrage wird die Backend-URL mit den benötigten Parametern (Frage, Session-ID, und Geheimnis zur einfachen Authentifizierung (vgl. Abschnitt 3.8.2)) ergänzt. Danach wird per AJAX ein GET-Request an das Backend geschickt und die Antwort im Browser dargestellt.

3.2 api.ai

Um den Intent eines User-Inputs zu erkennen, wird von allen Implementierungen api.ai[1] verwendet. Api.ai wurde für die unterstützten Fragetypen mit dazugehören Intents konfiguriert. Über eine bereitgestellte, REST-ähnliche HTTP-API können Fragen an den Webservice geschickt werden. Als Antwort wird der zur Frage passende Intent sowie alle in der Frage vorkommenden Parameter zurückgeschickt. Im Codeausschnitt 3.1 ist ein Beispiel-JSON-File ersichtlich, welches api.ai als Antwort liefert.

```
{
  "timestamp": "2017-05-15T09:56:24.791Z",
  "lang": "en",
  "result": {
    "resolvedQuery": "Movies by Stanley Kubrick.",
    "parameters": {
      "director": "Stanley Kubrick",
      "resultType": "movie"
    },
    "metadata": {
      "intentName": "ask.movie.director"
    }
  }
}
```

Codeausschnitt 3.1: Ausschnitt aus der Antwort von api.ai auf die Anfrage *„Movies by Stanley Kubrick.“*

Weil api.ai nur eine limitierte Anzahl an Anfragen pro Zeiteinheit erlaubt, werden bereits gestellte Fragen und die dazugehörigen Antworten von api.ai zwischengespeichert. Dies bietet gleichzeitig den Vorteil, dass wiederholte Fragen durch das Entfallen eines HTTP-Requests schneller bearbeitet werden können.

3.3 Atomare Architektur

Für die atomare Architektur werden die einzelnen Funktionen in kleinstmögliche Services aufgeteilt. Jeder Intent wird von einem dafür zuständigen Service abgedeckt. Gemäss den Empfehlungen für Microservice-Architekturen verwenden die einzelnen Intent-Services wie auch der Kontext-Service grundsätzlich eigene Datenbanken. Da für Regisseur-Informationen genau eine Tabelle notwendig ist, teilen sich die beiden Services `ask.director.movie` und `ask.movie.director` dieselbe Datenbank. Die sehr ähnlichen Funktionen der beiden Services sind ein Anzeichen dafür, dass diese als ein Microservice implementiert werden sollten. Für die atomare Aufteilung haben wir uns

[1] `https://api.ai` [Stand 15.05.2017]

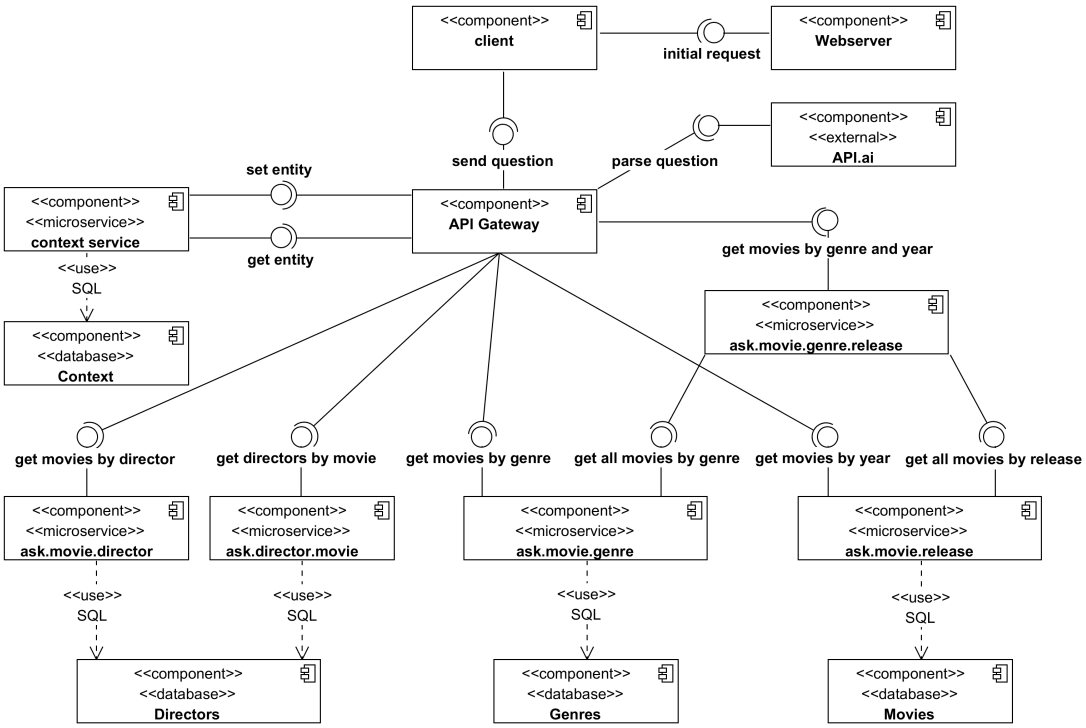

Abbildung 3.2: Komponentendiagramm der atomaren Chatbot-Architektur.

dennoch dazu entschieden, die Intents zu trennen, um einen Extremfall aufzeigen zu können.

Der Service `ask.movie.genre.release` dagegen hat keinen direkten Zugriff auf eine Datenbank. Um kombinierte Anfragen zu Genre und Erscheinungsdatum (z. B. *„Show me 22 action films from 2011.“*) beantworten zu können, müssen die unterliegenden Services beansprucht werden. Der `ask.movie.genre.release`-Service führt einen Application-Level-Join durch. Dazu werden alle passenden Daten zu Genre und Erscheinungsdatum von den dafür zuständigen Services bezogen. Auf diesen Daten wird anschliessend ein Join ausgeführt, um die gemeinsamen Filmtitel zu finden. Zuletzt wird eine zufällige Auswahl gemäss der angefragten Anzahl aus den verbliebenen Daten getroffen und an den API-Gateway geschickt.

Die von den Services angebotenen Schnittstellen (vgl. Komponentendiagramm in Abbildung 3.2) sind alle über HTTP erreichbar. Für den ursprünglichen Funktionsumfang ist dieser simple Request/Response-Applikationsfluss ausreichend, da keine Verzweigungen auftreten. Für den Kontext-Service wurde ebenfalls eine HTTP-Schnittstelle implementiert, um die Komplexität für das Evaluieren der Erweiterbarkeit nicht weiter zu erhöhen. In den meisten Fällen werden synchrone HTTP-Requests verwendet. Einzig die `set entity` Schnittstelle des Kontext-Services wird asynchron aufgerufen.

Die einzelnen Services, welche auf die Filmdaten zugreifen, sind jeweils auch für das Rendering der Daten zuständig. Dazu bieten diese Services verschiedene Formate für die Resultate an. Der Anfragesteller kann zwischen HTML, JSON oder Rohdaten wählen. Der API-Gateway fordert immer HTML-formatierten Text an, der direkt an den Client weitergesendet werden kann. Für andere Systeme könnte ein zweiter API-Gateway erstellt werden, der die Daten in einem Zwischenformat erhält und diese noch weiter verarbeitet. In der aktuellen Implementierung sind alle HTML-Resultate identisch gerendert. Dennoch wird für das Rendering kein separater Service oder eine Library verwendet. Dies führt zu Codeduplizierung, welche wir in diesem Fall für die gewonnene Flexibilität für allfällige Änderungen in Kauf nehmen.

Da auf den Filmdaten nur Lesezugriffe erfolgen, müssen keine Mechanismen zur Synchronisation der Daten vorhanden sein. Ebenfalls sind keine Transaktionen mit mehr als einer Datenbank-Query notwendig. Dies vereinfacht die Aufteilung der Daten in mehrere Datenbanken. Strategien für die Synchronisation von Datenbeständen zwischen Datenbanken werden im Abschnitt 2.2.2 besprochen.

3.4 Aggregierte Architektur

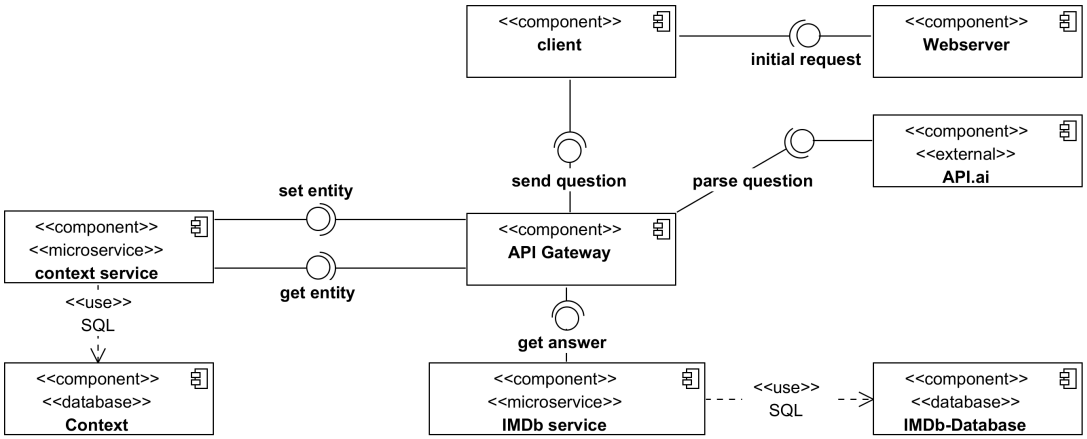

Abbildung 3.3: Komponentendiagramm der aggregierten Chatbot-Architektur.

In der aggregierten Architektur wird der Ansatz verfolgt, dass zusammenhängende Daten von genau einem Microservice verarbeitet werden. Alle Filminformationen von IMDb sind sehr eng verwandt, weshalb diese als einzelne, komplette Datenbank implementiert wird. Die Kontextinformationen werden für einen separaten Zweck benötigt und werden deshalb in einer zweiten Datenbank verwaltet. Beide Datenbanken können von je einen Service angesprochen werden.

Der im Komponentendiagramm 3.3 ersichtliche IMDb-Service bietet die gleichen Funktionen wie die fünf intentbasierten Services in der atomaren Architektur. Dieser

Service bietet eine einzelne Schnittstelle an, um alle Intents abzudecken. Damit muss der API-Gateway kein Routing durchführen und kann die Antworten von api.ai direkt an den IMDb-Service weiterleiten. Weil datenspezifische Aufgaben nicht vom API-Gateway durchgeführt werden, wird dessen Kohäsion erhöht. Sollen neue Intents vom Chatbot abgedeckt werden, muss der API-Gateway nicht angepasst werden. Gleichzeitig geht die Möglichkeit verloren, viel benötigte Services für beliebte Anfragen individuell zu skalieren.

Ein wichtiger Vorteil der aggregierten gegenüber der atomaren Architektur ist die Möglichkeit, SQL-Funktionen (in diesem Fall hauptsächlich Joins) zu verwenden. Da diese Operationen nicht auf Application-Level durchgeführt werden müssen, können viel kleinere Datenmengen zwischen den Services übertragen werden. Die Auswirkung dieser Ersparnis wird im Abschnitt 5.1.1 detailliert beschrieben.

3.5 Monolithische Architektur

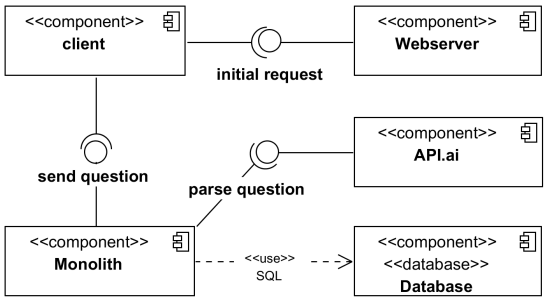

Abbildung 3.4: Komponentendiagramm der monolithischen Chatbot-Architektur.

In der monolithischen Implementierung ist die gesamte Applikationslogik in einem Prozess abgebildet. Alle Funktionen für die Beantwortung der Anfragen inklusive Gesprächskontext sind in einem Service vereint. Als Datenbasis wird eine einzelne Datenbank für den kompletten Chatbot verwendet. Die Datenbank ist grundsätzlich identisch mit der `IMDb-Database` der aggregierten Architektur. In einer zusätzlichen Tabelle werden die Kontext-Informationen gespeichert. Intern werden die einzelnen Funktionen ähnlich zu den bisher besprochenen Architekturen in verschiedene Module aufgeteilt.

Wie in Abbildung 3.4 ersichtlich ist, wird der Webserver für alle statischen Dateien als separate Komponente implementiert, identisch zu den oben beschriebenen Architekturen. Diese Strategie erlaubt es, die Serverlast von der aufwendigen Applikationslogik zu trennen. So kann die Latenz beim Laden der Web-Applikation für den User klein gehalten werden, auch wenn der Chatbot viel Last erfährt.

3.6 Wichtige Packages

Alle Services sind in Python geschrieben. In diesem Abschnitt werden die wichtigsten Packages besprochen, welche verwendet wurden.

3.6.1 Flask[2]

`Flask` wurde für die Umsetzung der HTTP-Endpoints verwendet. Die Kommunikation zwischen Client und Backend sowie unter den einzelnen Services läuft komplett über HTTP. Im Codeausschnitt 3.2 ist die Definition der Endpoints des Kontext-Services ersichtlich. In der definierten URL können direkt Funktionsparameter angegeben werden. Im Beispiel des Kontext-Services werden die Parameter verwendet, um die Session-ID und den Entitätstypen (vgl. Abschnitt 4.1.3) anzugeben.

```
1 @app.route('/setEntity/<sessionId>/<entityType>', methods=['POST'])
2 def setEntity(sessionId, entityType):
3     # save entity in database
4
5
6 @app.route('/getEntity/<sessionId>/<entityType>')
7 def getEntity(sessionId, entityType):
8     # get entity from database
```

Codeausschnitt 3.2: Definition der HTTP-Endpoints des Kontext-Services mittels `Flask`.

Standardgemäss erlaubt `Flask` CORS für alle Endpoints. Um den Chatbot gegen unbefugten Zugriff zu schützen, wurde CORS auf unseren Webserver beschränkt. Über eine Regex können erlaubte Domänen festgelegt werden. Die Konfiguration für die atomare Architektur, welche über die Domäne `http://ba17-us.appspot.com/`[3] erreichbar ist, ist im Codeausschnitt 3.3 ersichtlich.

```
1 app = Flask(__name__)
2 CORS(app, resources={r"/*": {"origins":
  ↪  r"https?://(www\.)?ba17-us.appspot.com.*"}})
```

Codeausschnitt 3.3: Definition der erlaubten Domänen für CORS mittels `Flask`.

[2] `http://flask.pocoo.org/` [Stand 17.05.2017]
[3] Zugriff auf Domäne ist nur für Mitglieder des Projektteams möglich (vgl. Abschnitt 3.8).

3.6.2 urlfetch[4]

Für Anwendungen in der Google App Engine wird von Google empfohlen, `urlfetch` für HTTP-Requests zu verwenden. Weil alle Microservices in unserer Applikation über HTTP Kommunizieren, ist das korrekte Funktionieren der HTTP-Requests kritisch. Daraus resultierte die Entscheidung, der Empfehlung von Google zu folgen. Mit Ausnahme des Speichern von Entitäten für den Gesprächskontext werden alle Requests synchron durchgeführt. Weil der Applikationsfluss hinter dem API-Gateway sequenziell abläuft, sind synchrone Requests ausreichend und vereinfachen die Applikationslogik.

3.6.3 MySQLdb[5]

Um den Zugriff auf die GCP-SQL-Datenbanken zu ermöglichen, soll gemäss Google die `MySQLdb`-Bibliothek verwendet werden.[6] Die Informationen der Datenbank werden im `app.yaml` (vgl. Abschnitt 3.7) als Umgebungsvariablen definiert. Wie für HTTP-Requests wird die von Google vorgeschlagene Lösung für die Datenbankverbindung verwendet.

3.6.4 Eigene Packages

Für die Evaluation der Microservices haben wir zwei eigene Packages implementiert, welche von unseren Services verwendet werden. Die beiden Packages sind für einheitliches Logging und Timing zuständig. Die Packages sind über ein privates GitHub-Repository (sowie auf der CD im Anhang dieser Arbeit) verfügbar und können über `pip`[7] installiert werden.

Die Logs aller Microservices in der Google App Engine werden in der GCP automatisch gesammelt und zentralisiert angezeigt. Um die Logs übersichtlich zu halten, verwenden alle Services ein spezielles Logformat, bei dem der Servicename mitgeloggt wird. So können die einzelnen Logs schnell unterschieden werden, was Debugging und die Evaluation vereinfacht. Ausschnitt 3.4 zeigt einen Teil der Logs für eine Anfrage.

Das zweite Package wurde zur Messung der Antwortzeiten (vgl. Abschnitt 5.1) eingesetzt. Da die Zeitmessung sowie das Logging über die gesamte Lebensdauer der Applikation einheitlich bleibt, wurden diese zwei Funktionalitäten als separate Packages realisiert.

[4] `https://cloud.google.com/appengine/docs/standard/python/refdocs/google.appengine.api.urlfetch` [Stand 17.05.2017]
[5] `http://mysql-python.sourceforge.net/MySQLdb.html` [Stand 18.05.2017]
[6] `https://cloud.google.com/appengine/docs/standard/python/cloud-sql/` [Stand 18.05.2017]
[7] `https://pip.pypa.io` [Stand 18.05.2017]

```
15:12:36.549 [API_GATEWAY:main.py:198] Question: Movies by Stanley Kubrick.
15:12:36.549 [API_GATEWAY:main.py:104] Routing to service url
↪  https://atomic-director-service-dot-ba17-us.appspot.com/v1/
↪  html?director=Stanley%20Kubrick&resultType=movie&
15:12:36.567 [DIRECTOR_SERVICE:main.py:85] Searching for Stanley Kubrick
15:12:36.567 [DIRECTOR_SERVICE:main.py:46] connecting to gcp cloud SQL
↪  service: directors
15:12:37.545 [CONTEXT_SERVICE:main.py:77] logging to DB:
↪  HD04LQpG56vYUM3ubJL5VGCChsDstAgIuVgeQga7sbsYmWdnL0pILQxRTrhFB6rP - movie
↪  - 2001: A Space Odyssey (1968)
15:12:37.545 [CONTEXT_SERVICE:main.py:26] connecting to gcp cloud SQL
↪  service: context
```

Codeausschnitt 3.4: Ausschnitt aus den Logs zur Anfrage *„Movies by Stanley Kubrick.“*

Für verwendete Packages ist es wichtig, dass die angebotenen Interfaces stabil bleiben. In der atomaren Implementierung wird das Logging-Package von sieben Services verwendet, eine Anpassung in jedem Service wäre mit viel Aufwand verbunden.

3.7 app.yaml

Services in der Google App Engine werden über eine YAML-Datei konfiguriert.[8] Darin sind unter anderem Service-Name, Einstiegspunkte für Endpoints und Umgebungsvariablen definiert. Im Ausschnitt 3.5 ist eine solche Konfigurationsdatei abgebildet.

3.8 Zugriffsbeschränkung

Durch die Limitierung von IMDb für die Nutzung ihrer Daten darf der Chatbot nicht öffentlich zugänglich sein. Um die Applikation vor unbefugtem Zugriff zu schützen, werden drei Mechanismen auf verschiedenen Ebenen eingesetzt.

3.8.1 Webserver

Der Zugriff auf die statischen Files wird über die Konfiguration des Webservers beschränkt. Die Google App Engine bietet die Möglichkeit, den Zugriff auf Services auf ausgewählte Google-Konten zu beschränken. Die Konfiguration wird über die `app.yaml`-Datei gesteuert und die Überprüfung von Google durchgeführt. Über die Option `login` (vgl. Codeausschnitt 3.6) können für verschiedene Endpoints die Zugriffsrechte angegeben werden.

[8] `https://cloud.google.com/appengine/docs/standard/python/config/appref` [Stand 18.05.2017]

```
service: atomic-directing-service
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /.*
  script: main.app

libraries:
- name: MySQLdb
  version: latest

env_variables:
    DB_NAME: directors
    CLOUDSQL_CONNECTION_NAME: ba17-us:us-east1:db-imdb-directors
    CLOUDSQL_USER: root
    CLOUDSQL_PASSWORD: BA17-Main
    LOGGER_NAME: DIRECTING_SERVICE
```

Codeausschnitt 3.5: `app.yaml` für den atomaren `ask.directing` Service.

```
handlers:
- url: /.*
  script: main.app
  login: admin
  secure: always
```

Codeausschnitt 3.6: Sicherheitsrelevanter Konfigurationsausschnitt des Webservers.

3.8.2 API-Gateway

Der API-Gateway kann nicht über ein Google-Konto gesichert werden, weil die Authentifizierung mit dem Google-Konto für AJAX-Requests nicht möglich ist. Deshalb wird ein statischer Schlüssel als URL-Parameter mit jedem Request mitgeschickt, welcher dann vom API-Gateway überprüft wird. Fehlt der Schlüssel oder ist dieser nicht korrekt, wird die Anfrage abgebrochen. Die gesamte Kommunikation verläuft über HTTPS und ist damit verschlüsselt. Dies gilt nicht nur für die Client-Server-Kommunikation, sondern auch für Requests zwischen Microservices.

3.8.3 Inter Service Kommunikation

Die einzelnen Microservices hinter dem API-Gateway sind standardmässig in der Google App Engine über eine URL aus dem Internet erreichbar. Es gibt deshalb die Möglichkeit, für eingehende Requests zu überprüfen, von welchem GCP-Projekt eine An-

frage stammt. Die Kontrolle ist über einen Header im HTTP-Request möglich, welcher nur bei Requests von Google App Engine Services vorhanden ist.[9] Über diesen Mechanismus wird in unseren Implementierungen verhindert, dass über einzelne Microservice auf die IMDb-Daten zugegriffen wird.

3.9 Deployment

Für das Deployment der Chatbot-Implementierungen haben wir uns für die GCP entschieden. Unter dieser Platform bietet Google verschiedene Produkte und Services für Cloud-Computing an. Die einzelnen Services des Chatbots werden über die Google App Engine[10] bereitgestellt. Für die Datenbankinstanzen wird Google Cloud SQL[11] verwendet. In den folgenden Abschnitten wird beschrieben, wie wir diese Produkte für unsere Applikation nutzen.

3.9.1 Google App Engine

Über Google App Engine werden alle Services, deren Versionen und Instanzen verwaltet. Zusätzlich übernimmt die Google App Engine viele für Microservices zentrale Aufgaben. Darunter fallen Skalierung (inklusive Load-Balancing), Service-Discovery, zentralisierte Logs und Health-Management. Weiter entfällt das Management der Infrastruktur, Virtuellen Maschinen oder Container für das Deployment. Neben der Konfigurationsdatei (vgl. Abschnitt 3.7) werden nur der Quellcode und allfällig benötigte Bibliotheken hochgeladen.

Skalierung

Die Services des Chatbots werden von der Google App Engine automatisch skaliert. Dies entspricht der Standardkonfiguration. Die automatische Skalierung basiert auf diversen Metriken, darunter Anfragefrequenz und Antwortlatenz [33]. Die Skalierung wird ebenfalls durch die gewählte Instanzklasse[12] beeinflusst. Die Instanzklasse legt das Limit für Memory und CPU fest. Die meisten Services unserer Prototypen verwenden die Standardklasse `F1`, welche die Ressourcenlimiten bei 128 MB Memory und 600 MHz CPU festlegt. Für die meisten Services sind diese Limiten ausreichend. Einzig der Service `ask.movie.genre.release` benötigt mehr Memory, um den Application-Level Join durchzuführen. Im Konfigurationsfile wird deshalb die Instanzklasse auf `F2` (Limiten 256 MB/1.2 GHz) gesetzt. Eingehende Requests werden von der Google App Engine automatisch auf laufende Instanzen verteilt. Der Entwickler muss sich also nicht um das Load-Balancing kümmern.

[9] `https://cloud.google.com/appengine/docs/standard/python/appidentity/` [Stand 19.05.2017]
[10] `https://cloud.google.com/appengine/` [Stand 23.05.2017]
[11] `https://cloud.google.com/sql/` [Stand 23.05.2017]
[12] `https://cloud.google.com/appengine/docs/standard/#instance_classes` [Stand 23.05.2017]

Service Discovery

Die von der Google App Engine verwalteten Services sind über eine eigene URL erreichbar. Jede URL wird anhand des Service- und Projektnamens zusammengesetzt. Zusätzlich können noch einzelne Versionen der Services adressiert werden, diese Möglichkeit wird von unseren Prototypen aber nicht genutzt. Die Service-URLs werden in den einzelnen Services unserer Prototypen nach Bedarf erstellt. Der Service muss dafür Kenntnis über den Servicenamen haben, was wir über Umgebungsvariablen lösen. Codeausschnitt 3.7 zeigt das Erstellen der Service-URLs für drei Beispielservices.

```
1 def makeServiceURL(serviceName):
2     return 'https://' + serviceName + '-dot-' + (os.environ.get(
      ↪  'GCLOUD_PROJECT')) + '.appspot.com'
3
4 SERVICE_DICTIONARY = {
5     'ask.movie.director': makeServiceURL(os.environ.get('DIRECTOR_SRV')),
6     'ask.recent': makeServiceURL(os.environ.get('RELEASE_SRV')),
7     'context': makeServiceURL(os.environ.get('CONTEXT_SRV'))
8 }
```

Codeausschnitt 3.7: Beispiel für Erstellung von Service-URLs für die Inter-Service-Kommunikation im API-Gateway.

Log Management

Bei verteilten System wie in einer Microservice-Architektur ist es wichtig, den Applikationsfluss anhand der generierten Logs nachvollziehen zu können. Die Fehlersuche in verteilten Logfiles ist zeitaufwendig und komplex. Die durch Services in der Google App Engine generierten Logs werden deshalb automatisch gesammelt und über Stackdriver Logging[13] bereitgestellt. In diesem Tool werden auch alle Applikationsfehler gesammelt und gruppiert. So können häufig auftretende Probleme einfach identifiziert werden. In Abbildung 3.5 sind alle aufgetretenen Fehler des atomaren Prototypen über einen Zeitraum von 30 Tagen aufgelistet. In diesem Fall ist zu erkennen, dass häufig ein `DeadlineExceededError` aufgetreten ist. Diese Fehler stammen hauptsächlich aus der Evaluationsphase für die Resultate aus dem Abschnitt 5.1.1.

Health Management

Die Google App Engine führt automatisch Health-Checks für alle aktiven Instanzen durch. Dazu muss von Entwicklerseite kein zusätzlicher Code geschrieben werden. Standardgemäss wird ein HTTP 404-Statuscode als Antwort auf den Health-Check als erfolgreich angesehen. Wenn benötigt, können eigene Health-Checks implementiert

[13] `https://cloud.google.com/logging/` [Stand 23.05.2017]

Errors in the last 30 days

Occurrences	Error	Seen in	First seen	Last seen	Status
1,576	**DeadlineExceededError: The overall deadline for responding to the HTTP request** routeQuestion (/base/data/home/apps/p~ba17-us/atomic-api-gateway:2017042	atomic-api-gateway:20170424t142444 +10 more	Apr 6, 2017	4 days ago	500
1	**NameError: name 'logFormatm' is not defined** <module> (/base/data/home/apps/p~ba17-us/20170407t150935.40039043795	atomic-api-gateway:20170518t135924	Apr 7, 2017	5 days ago	500
3	NEW **TypeError: cannot concatenate 'str' and 'NoneType' objects** makeServiceURL (/base/data/home/apps/p~ba17-us/atomic-api-gateway:20170	atomic-api-gateway:20170505t142313	May 5, 2017	May 5, 2017	500
2	**ImportError: No module named flask_cors** <module> (/base/data/home/apps/p~ba17-us/atomic-api-gateway:20170420t09	atomic-api-gateway:20170426t110229 atomic-context-service:20170504t124259	Mar 31, 2017	May 4, 2017	500
11	**File "/base/data/home/apps/p~ba17-us/atomic-api-gateway: 20170425t16005** LoadObject (/base/data/home/runtimes/python27/python27_lib/versions/1/goo	atomic-api-gateway:20170425t160050 atomic-context-service:20170503t154956	Mar 31, 2017	May 3, 2017	500
1	NEW **File "<string>", line 1, in reraise: DeadlineExceededError: The overall deadli** handle_user_exception (/base/data/home/apps/p~ba17-us/atomic-genre-servic	atomic-genre-service:20170419t155646	Apr 27, 2017	Apr 27, 2017	500

Abbildung 3.5: Fehlerberichte über 30 Tage für den atomaren Chatbot-Prototypen im Stackdriver Error Reporting Tool.

werden, welche von der Google App Engine überprüft werden sollen. Wird ein Problem bei einer Instanz erkannt, werden keine weiteren Requests an diese Instanz weitergeleitet. Nach einer bestimmten, konfigurierbaren Anzahl erfolgloser Health-Checks wird eine Instanz automatisch neu gestartet [34].

3.9.2 Google Cloud SQL

Für unsere Chatbot-Implementierungen verwenden wir MySQL-Datenbanken. Die Daten der IMDb waren bereits als MySQL-Dump vorhanden und konnten so direkt weiter verwendet werden. Weil der Umgang mit dem Cloud SQL Produkt zum Zeitpunkt der Erweiterung (vgl. Abschnitt 4.1.3) bereits bekannt war, verwendet der Kontext-Service ebenfalls eine MySQL-Datenbank als persistenten Speicher. Die Verbindung auf die Instanzen aus der Google App Engine funktioniert über das im Abschnitt 3.6.3 erläuterte Package und dem Verbindungsnamen (wird von Cloud SQL erstellt) der Cloud SQL Instanz.

Bei Cloud SQL Instanzen können, ähnlich wie bei den Google App Engine Services, verschiedene Klassen von virtuellen Maschinen ausgewählt werden. Weil wir für die Prototypen relativ wenige User erwarten, haben wir uns für die kleinsten zwei Maschinentypen (`db-f1-micro` und `db-g1-small`) entschieden. Diese Typen bieten mit 0.6 GB respektive 1.7 GB Memory genügend Leistung für unseren Verwendungszweck. Unabhängig vom Maschinentypen können auch Speichermedium und -kapazität eingestellt werden. Alle unsere Daten werden auf SSDs gespeichert, als günstigere Alternative stehen auch Festplatten zur Auswahl.

4 Vorgehen und Methoden

4.1 Evaluation Messwerte

Zum Ende unserer Implementierungsphase verfügen wir über drei Prototypen, die wir miteinander vergleichen. Dabei handelt es sich unter anderem um den Prototypen mit kleinstmöglichen Diensten und um die aggregierte Version. Als dritten Prototypen haben wir den ursprünglichen Monolithen so nachgebaut, dass die Applikation über die gleichen Funktionen wie unsere Microservice-Prototypen verfügt. Im Endeffekt wurde beim Monolithen der gleiche Code wie bei der aggregierten Version verwendet. Die einzelnen Python-Komponenten sind aber direkt ins Programm eingebunden, statt dass diese über das Netzwerk miteinander kommunizieren. Somit sind für alle Chatbot-Versionen die möglichst identische Voraussetzungen geschaffen.

4.1.1 Skalierung

Gesamtantwortzeit

Um die Gesamtantwortzeit zu testen, simulieren wir die Nutzung der Applikation unter verschiedenen Bedingungen. Zum einen werden geringe und ausgiebige Nutzung der Applikation getestet. Dieser Parameter variiert in der Anzahl Anfragen pro Zeiteinheit. Zum anderen wird die Antwortzeit für verschieden komplexe Anfragen getestet. Pro Implementierung ergibt dies vier Messwerte, anhand derer wir die Gesamtantwortzeit messen und beurteilen können.

Um Einflüsse durch lokale Netzwerkeigenschaften zu vermeiden, werden alle Zeitwerte serverseitig gemessen. Zusätzlich wird jeder Testfall fünfmal durchgeführt, um einen aussagekräftigen Durchschnittswert zu erhalten. Die Tests werden mit Kombinationen der in Tabelle 4.1 beschriebenen Parameter durchgeführt.

Parameter	Beschreibung
Geringe Nutzung	200 Anfragen mit 2 Anfragen pro Sekunde
Ausgiebige Nutzung	1000 Anfragen mit 100 Anfragen pro Sekunde
Einfache Frage	Frage *„Movies by Stanley Kubrick.“*
Komplexe Frage	Frage *„Show me 20 action movies from 2016.“*

Tabelle 4.1: Beschreibung der Parameter zum Testen der Gesamtantwortzeit.

Mit diesen Versuchen simulieren wir einen Teil der User Experience (UX). Die Antwortzeit auf eine Frage ist neben dem korrekten Inhalt der Antwort ein wichtiges Merkmal für den Benutzer. Gemäss Experimenten von Microsoft haben bereits kleine Verzögerungen in der Antwortzeit einer Web-Applikation einen messbaren Einfluss auf die Benutzer. Bei einer Verzögerung der Antwort um eine Sekunde fiel die Nutzer-Zufriedenheit um 1.6 %, bei zwei Sekunden sogar um 3.8 % [35].

Für die Versuchsdurchführung wird ein speziell von uns für diesen Zweck erstelltes Programm verwendet. Die Software ist in Python geschrieben und sendet in konfigurierbaren Intervallen Anfragen an den Einstiegspunkt der jeweiligen Implementierungen. Für das Senden der Anfragen wurde das `grequests`[1] Modul verwendet, um die Anfragen asynchron zu senden. Für die einzelnen Anfragen wird jeweils ein eigener Thread gestartet, um die für unsere Tests notwendigen Anfrageraten zu erreichen. Der entsprechende Codeausschnitt ist in Abbildung 4.1 ersichtlich.

```
1  def RequestThread():
2      rs = (grequests.get(u) for u in requestUrl)
3      grequests.map(rs)
4
5
6  for i in range(0, numberOfRequests):
7
8      requestUrl = [
9          'https://atomic-api-gateway-dot-ba17-us.appspot.com/?question=' +
10         urllib2.quote(question) +
11         '&requestid=' + str(i) +
12         '&secret=<secret>'
13     ]
14
15     t = threading.Thread(target=RequestThread)
16     t.start()
17     sleep(secondsBetweenRequests)
```

Codeausschnitt 4.1: Erstellen und Senden von Anfragen zum Testen der Gesamtantwortzeit.

Im Gegensatz zum Aufwand des Erstellen und Abschicken der einzelnen Anfragen bedeutet die Erstellung eines neuen Threads viel weniger Zeitaufwand. Die zusätzlich benötigte Zeit hat zur Folge, dass die Anfragen etwas langsamer als angegeben verschickt werden können. Bei zwei Anfragen pro Sekunde macht diese Verzögerung weniger als 1 % aus. Bei den Tests zur ausgiebigen Nutzung liegt die durchschnittliche Verzögerung bei etwa 14 %, was 14 Tausendstelsekunden entspricht. In der Praxis zeigt sich, dass bei unseren Implementierungen auch mit dieser Verzögerung noch ein klarer Unterschied zu den Tests mit geringer Nutzung besteht.

[1] `https://github.com/kennethreitz/grequests` [Stand 27.04.2017]

Die benötigte Antwortzeit wird vom API-Gateway für jede Anfrage geloggt. Anschliessend werden die Log-Daten von einer weiteren, selbst erstellten Software von der GCP heruntergeladen und zur Analyse vorbereitet. Google bietet keine Möglichkeit an, Logfiles direkt von der GCP herunterzuladen, weshalb wir die Logs über eine Beta-Funktionalität des Google Cloud SDK beziehen[2]. Die relevanten Timing-Informationen werden automatisch aus den Logs ausgelesen und für die Analyse in MATLAB aufbereitet. Die Resultate für die Gesamtantwortzeit werden im Abschnitt 5.1.1 besprochen.

Skalierung von Services

Eng zusammenhängend mit der Performance der Prototypen ist die Skalierung der Applikation. In einer Microservice-Architektur können einzelne Services unabhängig vom Rest der Applikation skaliert werden. Die Implementierung der Skalierung wird im Abschnitt 3.9.1 besprochen. Messwerte für das Skalierungsverhalten unserer Prototypen liegen nicht vor. Die Überwachung der erstellten Service-Instanzen sollte über Stackdriver[3] möglich sein. Bei der simulierten Nutzung, ähnlich dem Vorgehen zum Testen der Gesamtantwortzeit, werden über Stackdriver falsche Werte ausgegeben. So werden zum Beispiel über eine komplette Messreihe mit 1000 Anfragen an die aggregierte Version keine aktiven Instanzen gemeldet. Aus diesem Grund beschränkt sich die Evaluation auf theoretische Überlegungen sowie arbeiten Dritter. Die Erkenntnisse werden in Abschnitt 5.1.2 besprochen.

4.1.2 Verfügbarkeit

Um die Verfügbarkeit mit aussagekräftigen Messwerten evaluieren zu können, müssten die Chatbot-Prototypen in einer Produktionsumgebung ausgiebig getestet werden. Dies ist im Rahmen dieser Arbeit nicht möglich. Deshalb beschränken wir uns bei der Evaluation der Verfügbarkeit auf Erfahrungen, welche wir im Verlauf der Arbeit mit den Prototypen gemacht haben.

4.1.3 Erweiterbarkeit

Um die Unterschiede in der Erweiterbarkeit testen zu können, muss die neue Funktionalität komplex genug sein, dass die Erweiterung in jeder Implementierung Herausforderungen bietet. Als Funktionalität wurde deshalb ein einfacher Gesprächskontext gewählt, in dem zuletzt abgefragte Entitäten (Filme und Regisseure) für jede Session gespeichert werden. Diese Entitäten können in nachfolgenden Anfragen wieder verwendet werden, wie in Abbildung 4.1 dargestellt wird.

[2] `https://cloud.google.com/logging/docs/reference/tools/gcloud-logging` [Stand 27.04.2017]
[3] `https://app.google.stackdriver.com/` [Stand 25.05.2017]

```
USER:    Show me 1 comedy movie from 2017.

CHATBOT: Everything Is Wonderful (2017)

USER:    Who made that?

CHATBOT: Pia Mechler
         Stephanie Angel
```

Abbildung 4.1: Beispielkonversation mit kontextbasierter Abfrage.

Umsetzung

Die Erweiterung des Chatbots um einen Gesprächskontext verlangt einige Anpassungen an den registrierten Intents auf api.ai. Damit der Chatbot erkennen kann, dass eine Anfrage über den Kontext geleitet werden muss, sind spezielle Intents nötig. Dies wird über die neu erstellten Intents `ask.director.movie.context` und `ask.movie.director.context` erreicht.

Weiter müssen die bestehenden Intents um einen Parameter erweitert werden, in dem der gesuchte Entitätstyp (Film oder Regisseur) angegeben ist. So wird zum Beispiel bei der Anfrage *„Movies by Stanley Kubrick.“* angegeben, dass Resultate von Typ Film gesucht sind. Mit dieser Information kann die Antwort auf die Anfrage unter dem korrekten Entitätstypen gespeichert werden.

Der Einfachheit halber wird bei einer Antwort mit mehreren Entitäten, wie zum Beispiel einer Liste von Filmen, nur das erste Resultat gespeichert. Dies ist für die Evaluierung der Erweiterbarkeit ausreichend, weil von einer komplexeren Umsetzung einzig die neue Funktionalität, nicht aber der bestehende Teil der Applikation betroffen wäre.

Neben den implementierungsübergreifenden Intents wird auch die clientseitige JavaScript-Datei angepasst. Damit der Gesprächskontext dem aktuellen Dialog zugeordnet werden kann, müssen Sessions eindeutig identifiziert werden. Zu diesem Zweck wird beim Laden des Chatbots ein zufälliger String als Session-ID generiert, welcher bei jeder Anfrage mitgeschickt wird. Diese Session-ID wird mit dem Entitätstypen und dem dazugehörigen Antwortwert gespeichert.

Die Kontextinformationen werden in einer separaten Tabelle einer SQL-Datenbank gespeichert. Als Primärschlüssel wird die Kombination aus Session-ID und Entitätstyp verwendet. Die Tabelle wird bei jeder Anfrage aktualisiert, um die zuletzt aufgetretenen Entitäten zu speichern. Ein Ausschnitt der Kontext-Datenbank mit den Daten aus dem Beispieldialog der Abbildung 4.1 ist in der Tabelle 4.2 ersichtlich.

sessionID	entityType	entityValue
yhcwPSxyy33Uhn3uIjctZ4Y0s98...	director	Pia Mechler
yhcwPSxyy33Uhn3uIjctZ4Y0s98...	movie	Everything Is Wonderful (2017)

Tabelle 4.2: Ausschnitt aus der Kontext-Datenbank.

Zuletzt muss die Applikationslogik implementiert werden, um Entitäten in der Datenbank zu speichern und bei Bedarf wieder zu beziehen. Abbildung 4.2 zeigt, wie der Ablauf einer kontextbasierten Anfrage durch die Applikation aussieht.

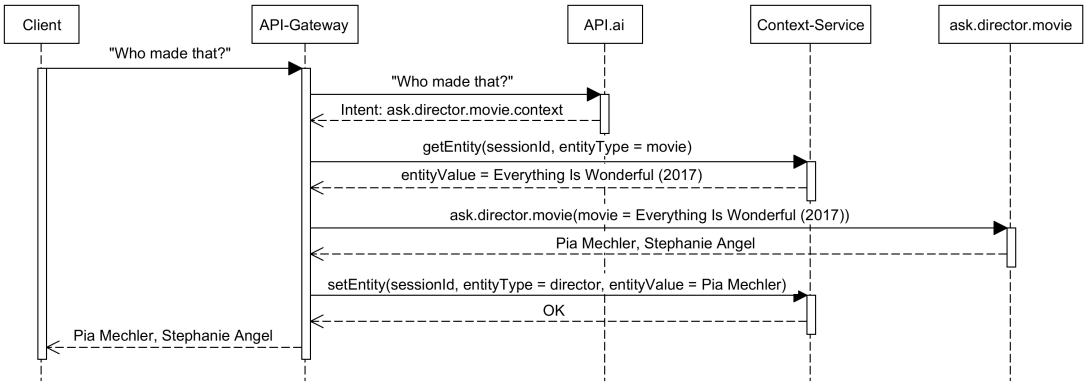

Abbildung 4.2: Sequenzdiagramm für eine kontextbasierte Anfrage.

Die Umsetzung des Gesprächskontexts verfügt über eine eigene Datenquelle und kontextbasierte Anfragen müssen zuerst durch diese neue Funktionalität bearbeitet werden, bevor die bestehenden Services angesprochen werden können. Damit bietet der Kontext als Erweiterung genügend Komplexität, um die Erweiterbarkeit der verschiedenen Implementierung zu bewerten. Die Gemeinsamkeiten und Unterschiede der Erweiterungen der verschiedenen Implementierung werden im Abschnitt 5.3 besprochen.

5 Resultate

5.1 Skalierung

5.1.1 Gesamtantwortzeit

Für den Vergleich der Antwortzeiten unserer drei Implementierungen erstellten wir für jeden der vier Testfälle (vgl. Abschnitt 4.1.1) einen Boxplot pro Implementierung. Damit lassen sich die Antwortzeiten für die verschiedenen Szenarien visuell Vergleichen.

Die MATLAB-Skripte sowie die Rohdaten, mit denen die Boxplots erstellt wurden, sind auf der CD im Anhang gespeichert. Von allen Messreihen wurden die zehn kürzesten und zehn längsten Antwortzeiten für den Boxplot ignoriert, um extreme Ausreisser aus der Evaluation zu entfernen. In den Rohdaten sind diese Extrema weiterhin ersichtlich. In allen Boxplots beschreibt die rote, vertikale Linie den Median. Die blaue Box umfasst alle Werte zwischen dem ersten und dritten Quartil. Die Antennen haben eine maximale Länge von 1.5 mal dem Interquartilabstand. Rote Plus-Symbole (+) markieren Ausreisser.

Abbildung 5.1 zeigt die Antwortzeiten für geringe Last mit einer einfachen Anfrage. Die Durchschnittszeiten liegen für alle Implementierungen um 0.05 Sekunden. Die Unterschiede zeigen sich vor allem in den Ausreissern. Während die atomare Implementierung einige Anfragen sehr schnell beantworten kann, weisen beim Monolithen wenige Anfragen im Vergleich zur Norm längere Antwortzeiten auf. Dazwischen liegt die Architektur mit aggregierten Services, welche weder besonders schnelle noch langsame Antwortzeiten aufweist. Die Unterschiede sind absolut gemessen sehr gering. Alle Werte liegen zwischen 0.01 und 0.20 Sekunden.

Bei einer höheren Anzahl Anfragen pro Sekunde werden die Unterschiede zwischen den Implementierungen klar sichtbar. Wie in Abbildung 5.2 zu erkennen ist, sind die Antwortzeiten der Implementierung mit aggregierten Services extrem gestreut. Die Durchschnittszeiten liegen mit 0.04 (monolithisch), 0.20 (aggregiert) und 0.03 (atomar) Sekunden noch immer relativ nahe beisammen. Die aggregierte Microservice-Architektur weist aber sehr viele Ausreisser mit Antwortzeiten bis 2.76 Sekunden auf. Die anderen Implementierungen können alle Anfragen unter einer halben Sekunde beantworten. Wie bereits bei einer geringeren Auslastung konnten die Anfragen insgesamt am schnellsten von der atomaren Implementierung beantwortet werden. Kumuliert benötigte der Monolith 42 % mehr Zeit für alle Anfragen als die atomare Implementierung.

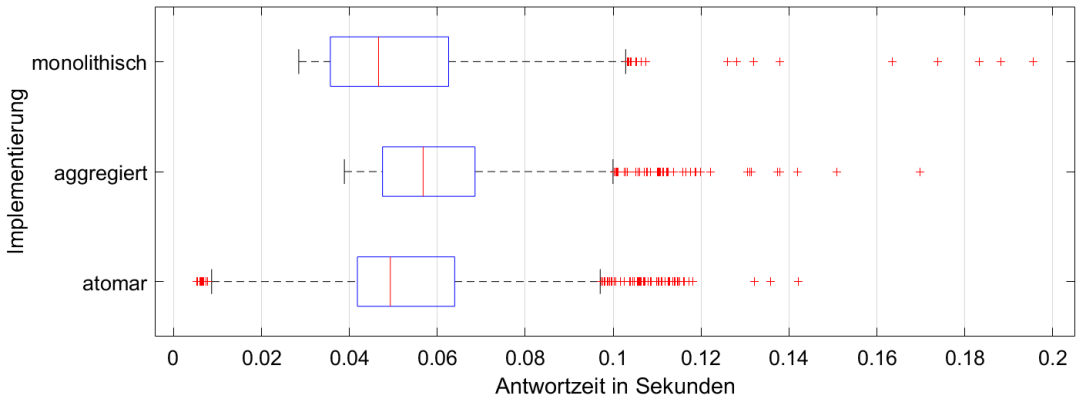

Abbildung 5.1: Antwortzeiten für *„Movies by Stanley Kubrick.“* mit 2 Anfragen pro Sekunde.

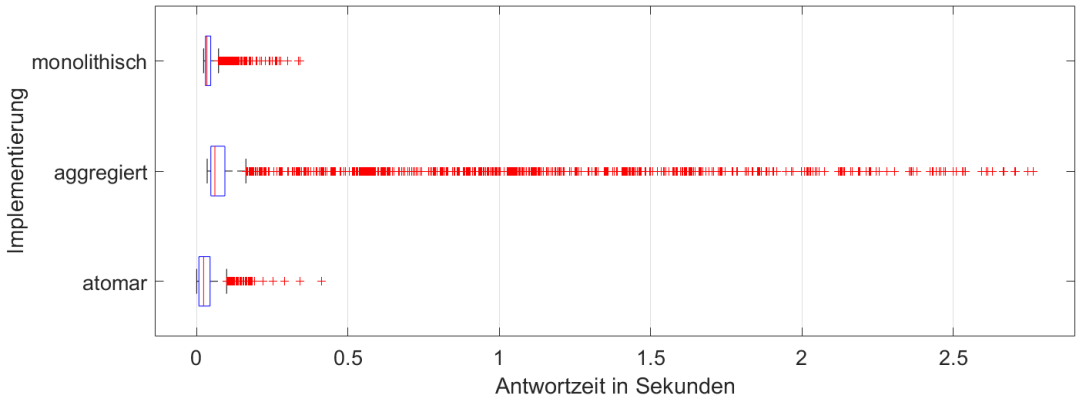

Abbildung 5.2: Antwortzeiten für *„Movies by Stanley Kubrick.“* mit 100 Anfragen pro Sekunde.

Für einfache Anfragen, bei denen nur wenige Services betroffen sind, gibt es in den Antwortzeiten nur vernachlässigbare Unterschiede. Die Situation ändert sich bei komplexeren Anfragen, bei denen grosse Datenmengen zwischen den Services gesendet werden müssen. In Abbildung 5.3 ist klar zu erkennen, dass der Monolith die mit Abstand kürzesten Antwortzeiten aufweist. Die durchschnittliche Antwortzeit beim Monolithen liegt bei 0.79 Sekunden, verglichen mit 12.19 (aggregiert) und 18.46 (atomar) Sekunden.

Beim vierten Szenario mit einer komplexen Frage und hohen Anfragefrequenz konnten keine aussagekräftigen Messwerte erstellt werden. Alle Implementierungen lieferten nur bei einem Bruchteil der Anfragen eine Antwort. Die unterliegenden Datenbank-Instanzen konnten diese hohe Anzahl an Anfragen nicht abarbeiten, wodurch in den

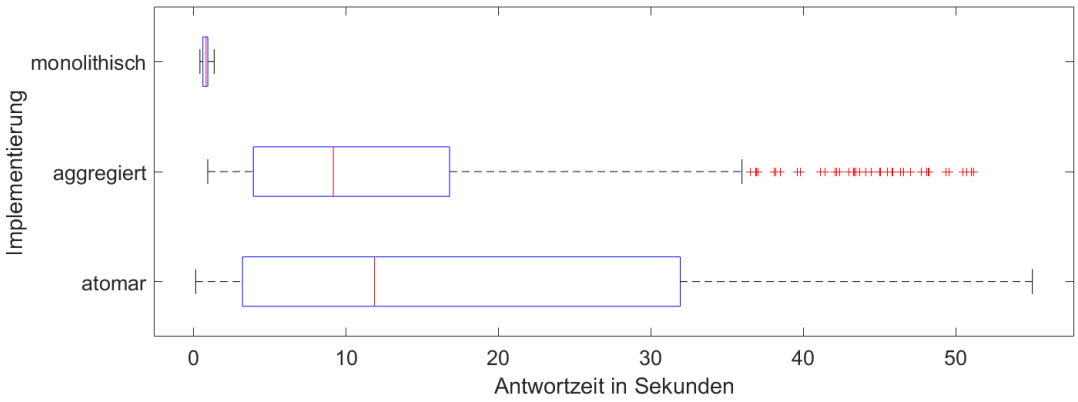

Abbildung 5.3: Antwortzeiten für *„Show me 20 action movies from 2016.“* mit 2 Anfragen pro Sekunde.

meisten Fällen die Anfrage abgebrochen wurde. Bei mehreren Versuchen waren die Resultate so stark gestreut, dass sich keine Aussagen über die Resultate machen lassen.

5.1.2 Skalierung von Services

Villamizar *et al.* beschreiben das Problem der Skalierung wie folgt: „Scaling monolithic applications is a challenge because they commonly offer a lot of services, some of them more popular than others. If popular services need to be scaled because they are highly demanded, the whole set of services also will be scaled at the same time, which generate that non popular services consume large amount of server resources even when they are not going to be used“ [1]. Dieses Problem kann mit kleineren Services, welche granularer skaliert werden können, umgangen werden. Kleinere Services senken dank zielgenauen Ressourcennutzung die Kosten im Betrieb. Anhand der aktuellen[1] Preisliste der Google App Engine Instanzen lässt sich dies nachvollziehen. In Tabelle 5.1 sind die Leistung sowie Preise für Google App Engine Instanzen ersichtlich.

Instanzklasse	Kosten pro Instanz pro Stunde	Memory-Limit	CPU-Limit
`B1`	USD 0.05	128 MB	600 MHz
`B2`	USD 0.10	256 MB	1.2 GHz
`B4`	USD 0.20	512 MB	2.4 GHz

Tabelle 5.1: Liste von Instanzklassen mit Preisen und Limiten verfügbar in der Google App Engine [36], [37].

[1] Stand 25.05.2017

Anhand der Preise aus Tabelle 5.1 lassen sich drei Serviceaufteilungen ableiten, welche den Preisvorteil kleinerer Services (und somit günstigeren Instanzen) illustrieren. Tabelle 5.2 zeigt die verschiedenen Aufteilungen und das Preisverhalten in unterschiedlichen Skalierungsszenarien.

	Monolith	mittlere Services	kleine Services
Benötigte Instanzen	`B4`	`B2, B2`	`B1, B1, B1, B1`
Grundpreis	USD 0.20	USD 0.20	USD 0.20
Skalierung eines Features	USD 0.40	USD 0.30	USD 0.25
Skalierung von zwei Features	USD 0.40	USD 0.30/0.40[a]	USD 0.30
Skalierung von drei Features	USD 0.40	USD 0.40	USD 0.35
Komplette Skalierung	USD 0.40	USD 0.40	USD 0.40

[a]USD 0.30, wenn beide Features im gleichen Service implementiert sind. USD 0.40, wenn beide Services skaliert werden müssen.

Tabelle 5.2: Kostenvergleich von Skalierungsszenarien für verschiedene Serviceaufteilungen.

Resultate von Villamizar *et al.* unterstützen diese Überlegung [31]. Gemäss deren Messungen sind die Kosten pro Request in einer Microservice-Architektur niedriger als bei einem Monolithen. Abbildung 5.4 zeigt die Kosten verschiedener Architekturen im Vergleich.

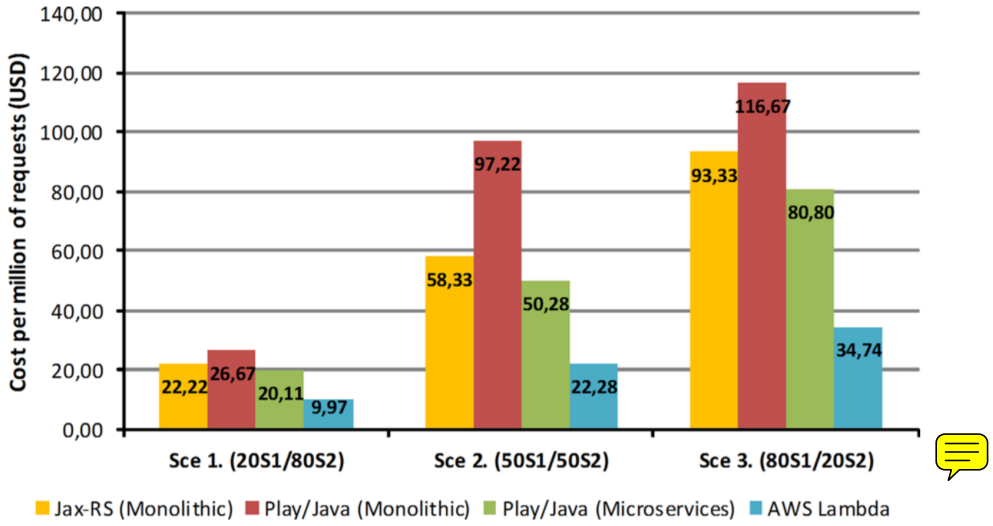

Abbildung 5.4: Kostenvergleich für verschiedene Architekturen pro Million Requests [31].

Weiter muss beachtet werden, wie lange es dauert, bis eine Instanz gestartet ist. Nachdem erkannt wird, dass eine Skalierung notwendig ist, sollte die neue Instanz so schnell wie möglich Anfragen entgegen nehmen. Das Ziel, Services möglichst schnell zu starten, wird auch durch den Einsatz von Containern (vgl. 2.1) unterstützt.

5.1.3 Fazit Skalierbarkeit

Unabhängig vom Umfang der einzelnen Services, von atomaren Services bis zum Monolithen, können ähnliche Antwortzeiten für den User erreicht werden. Die Resultate aus Abschnitt 5.1.1 zeigen, dass der Umfang der Services wenig Einfluss auf die Antwortzeiten hat. Erreicht wird dies mit Skalierung einzelner Services, um genügend Ressourcen für die Bearbeitung der Anfragen bereitzustellen.

Auffällig sind die, im Vergleich zu den anderen Prototypen, langsameren Antwortzeiten der aggregierten Implementierung bei vielen oder komplexen Anfragen. Die Zeiten können durch die relativ komplexe Aufgabe des IMDb-Services erklärt werden. Während dieser Service eine Anfrage bearbeitet, blockiert der API-Gateway. Da Services nicht sofort von der Google App Engine skaliert werden, steigen die Antwortzeiten für Anfragen. Dieses Problem kann entschärft werden, indem für eingehende Requests neue Threads erstellt werden. Unsere Prototypen verwenden für Requests keine eigenen Threads, um die Komplexität der Services tiefer zu halten.

Kleinere Services ermöglichen es, viel benötigte Funktionen separat vom Rest der Applikation zu skalieren. Dies erlaubt es, dynamisch auf die Ressourcenbedürfnisse einzelner Services zu reagieren. Durch den gezielten Einsatz von Ressourcen können Kosten im Betrieb von Applikationen gespart werden.

Die Optimierung der Kosten hängt neben dem Serviceschnitt auch von den Kosten der gewählten Deployment-Plattform ab. Die Auswahl geeigneter Instanztypen aufgrund der benötigten Ressourcen ist nicht trivial. Für jeden Service sollte aufgrund des Umfangs der ideale Instanztyp gefunden werden. Bei einer grossen Anzahl an kleinen Services ist es nötig, im laufenden Betrieb flexibel Änderungen vorzunehmen, um auf aktuelle Gegebenheiten einzugehen. Ändernde Preise bei Cloud-Anbietern oder verändertes Nutzerverhalten können einen erheblichen Verwaltungsaufwand mit sich ziehen. Dieser Aufwand ist bei einer monolithischen Architektur kleiner. Im Falle, dass Nutzeranfragen nicht zufriedenstellend bearbeitet werden, kann eine einzelne neue Instanz bereitgestellt werden. Die einfachere Verwaltung des Monolithen wird durch einen Verlust an Kontrolle und potentiell höheren Kosten erkauft.

5.2 Verfügbarkeit

Während der Implementierungsphase der Prototypen traten regelmässig Probleme mit einzelnen Services auf. Applikationsfehler können aus verschiedenen Quellen wie unerwarteten Werten in den unterliegenden Datensätzen oder von Implementierungsfehlern stammen. Komplett vermeiden lassen sich Fehler in einer Applikation selten. Defensives Programmieren schafft bereits Voraussetzungen, um Ausfälle zu vermeiden. Microservice-Architekturen gehen noch einen Schritt weiter. Selbst wenn ein Fehler in einem Service auftritt, soll der Rest der Applikation weiter verfügbar sein.

Am besten sichtbar sind diese Vorteile bei der atomaren Implementierung. Die Aufteilung der Services nach Intents erlaubt es, bereits funktionierende Fragetypen während der Weiterentwicklung der restlichen Applikation weiter zu verwenden. Auf Abbildung 5.5 ist das Resultat einer Anfrage ersichtlich, bei dem der unterliegende Microservice einen Fehler hat. Das UI und alle anderen Services sind weiterhin funktionsfähig, es fällt nur ein Bruchteil der Funktionalität weg.

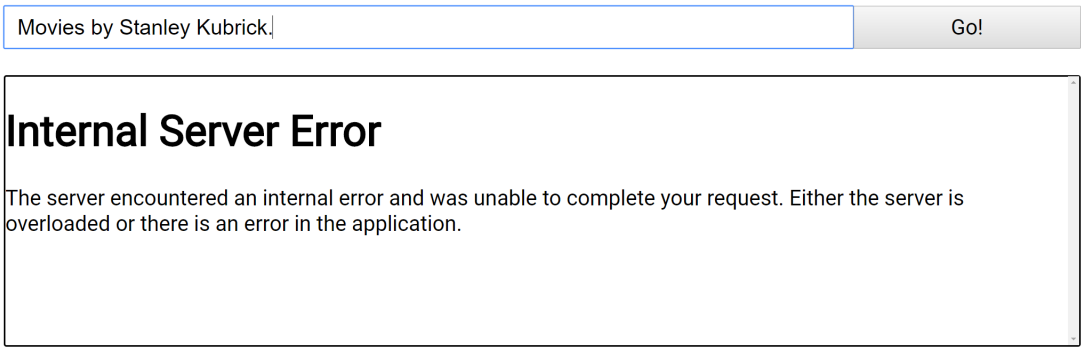

Abbildung 5.5: Beispiel für einen Applikationsfehler in einem unterliegenden Microservice.

Um die Verfügbarkeit in einer Microservice-Architektur abschätzen zu können, müssen auch Abhängigkeiten zwischen den Services beachtet werden. So hängt bei den Chatbot-Prototypen die komplette Funktionalität von den jeweiligen API-Gateways und api.ai ab. Bei einem Problem mit einem dieser Services kann der Chatbot nicht mehr verwendet werden. Weiter ist der `ask.movie.genre.release` Service aus der atomaren Implementierung von den unterliegenden `ask.movie.genre` und `ask.movie.release` Services abhängig. Solche SPOFs sollten nach Möglichkeit beim Entwurf der Architektur vermieden werden. Wenn dies nicht möglich ist, können die Auswirkungen eines Serviceausfalls minimiert werden, indem stets mehrere Instanzen bereitstehen. In der Google App Engine zum Beispiel kann diese Einstellung direkt über das Konfigurationsfile des Services (vgl. Abschnitt 3.7) vorgenommen werden.

Bei der aggregierten und monolithischen Implementierung sind Ausfälle entsprechend des jeweiligen Serviceumfangs weitreichender. Während ein Problem mit dem Kontext-Service in der aggregierten Implementierung nur wenige Fragetypen betrifft, bedeutet ein Ausfall des IMDb-Services, dass keine Fragen mehr beantwortet werden können. Beim Monolithen hat jeder Programmabsturz zur Folge, dass die ganze Funktionalität ausfällt. Ein Update des Monolithen bringt somit stets die Gefahr mit sich, einen Komplettausfall zu verursachen. Dies wiederum wirkt sich negativ auf die Innovationsgeschwindigkeit aus.

5.2.1 Fazit Verfügbarkeit

Der gewählte Serviceschnitt ist für die Verfügbarkeit entscheidend. Grob unterteilte Services bedeuten weitreichende Funktionsausfälle im Problemfall. Durch Services, welche einzelne Funktionalitäten abdecken, können die Auswirkungen eines Serviceausfalls minimiert werden. Wie bereits bei der Skalierbarkeit hat diese feine Kontrolle ein komplexeres Gesamtsystem zur Folge. Neben dem Ausfall von Funktionalität eines Services muss auch beachtet werden, welche Funktionalitäten von diesem Service abhängig sind. Bestehen zu viele Abhängigkeiten, kann es sinnvoll sein, Services zusammenzulegen. Da die Vorteile in den Bereichen Skalierbarkeit und Verfügbarkeit bei starken Abhängigkeiten abgeschwächt werden, kann so das Gesamtsystem vereinfacht werden. Die in den Abschnitten 5.1 und 5.3 beschriebenen Probleme von zu kleinen Services können entschärft werden, während für die Verfügbarkeit keine Einschränkungen auftreten.

Bei umfangreichen Services ist das Risiko gross, dass bei Updates Probleme entstehen. Die GCP bietet die Möglichkeit, Anfragen auf verschiedene Versionen aufzuteilen, womit neue Funktionalitäten mit einer Teilmenge der User getestet werden können.[2] Dieses Vorgehen ersetzt keineswegs das vorgängige, ausgiebige Testen des kritischen Applikationsteils. Dennoch können negative Effekte von fehlerhaften Versionen vermindert werden.

Aufgrund der Kommunikation der Services über ein Netzwerk sind Kommunikationsausfälle viel wahrscheinlicher als bei Methodenaufrufen innerhalb eines Prozesses. Defensives Programmieren ist deshalb zwingend Notwendig, um die Verfügbarkeit des Systems hoch zu halten. Meshenberg beschreibt die Verfügbarkeit wie folgt: „Reliability. It matters, a lot. Especially at scale. Failure happens at distributed systems and the rate of failure is proportional to the amount of change [...] and the scale at which you're running“ [4]. Während der Entwicklung muss mit Ausfällen gerechnet werden. Ein wertvolles Hilfsmittel um eine hohe Verfügbarkeit zu erreichen, ist destruktives Testen (vgl. Abschnitt 2.2.4). Im Produktionsbetrieb dürfen keine Überraschungen durch unvorhergesehene Auswirkungen von Ausfällen auftreten.

5.3 Erweiterbarkeit

5.3.1 Gemeinsamkeiten

Einige der in Abschnitt 4.1.3 beschriebenen Arbeiten sind für alle Implementierungen identisch. Die Anpassungen an den Intents sind nur einmalig nötig, da alle Implementierungen auf den selben Endpoint von api.ai zugreifen. Ähnlich verhält sich die Umsetzung der Identifizierung der Sessions.

[2] `https://cloud.google.com/appengine/docs/standard/python/splitting-traffic` [Stand 05.05.2017]

Bei allen Implementierungen wird dieselbe Logik für den Kontext eingesetzt, womit diese Anpassung bei allen Implementationen identisch ist. Beide Arbeiten sind somit für den Vergleich des benötigten Aufwands irrelevant.

5.3.2 Atomare Implementierung

Für die Speicherung der Kontext-Informationen wird bei der atomaren Implementierung eine dedizierte Datenbankinstanz verwendet. Auf dieser Instanz befindet sich die neu erstellte Datenbank für den Gesprächskontext mit einer Tabelle. Da diese Instanz komplett unabhängig von den Datenbeständen der anderen Services ist, musste bei der Erstellung keine Rücksicht auf Wechselwirkungen genommen werden. Gleichzeitig geht damit die Möglichkeit verloren, eine SQL-basierte Fremdschlüsselbeziehung für die Entitäten festzulegen. Somit könnten in Ausnahmefällen falsche Werte wie zum Beispiel Fehlermeldungen anstelle von eigentlichen Antworten in die Kontext-Datenbank geschrieben werden. Gegen solche Fälle müssen in einer Produktionsumgebung Gegenmassnahmen ergriffen werden. Für die Umsetzung unseres Prototyps haben wir entsprechende Schutzmechanismen weggelassen, um die Komplexität tiefer zu halten.

Der eigentliche Kontext-Service bietet zwei Endpoints. Zum einen um Entitäten zu speichern, zum anderen um diese wieder abzurufen. Die Umsetzung des neuen Services weist viele Parallelen zu den bestehenden Microservices auf, womit viele Konzepte direkt übernommen werden konnten. Aufgaben wie das Bereitstellen von Endpoints oder der Zugriff auf eine Datenbank wurden bereits in anderen Services gelöst. Wie beim Erstellen der neuen Datenbank, fand die Implementierung des Kontext-Sevices komplett unabhängig von den restlichen Services statt. Einzig die vorhandenen Daten und der interne Ablauf einer Anfrage waren ausschlaggebend für die Umsetzung, weil davon direkt die Applikationslogik des Kontext-Services abhängt. Bei der atomaren Architektur wird das Konzept des Information-Hiding praktisch aufgedrängt, was das Erweitern einer Applikation stark vereinfacht.

Während der Umsetzung wurden zwei Microservices (`ask.director.movie` und `ask.movie. director`) leicht angepasst. Beim Ersteren wurde eine Datenbank-Query angepasst, dass auch komplette Filmtitel mit Erscheinungsjahr (vgl. Beispieleintrag in Tabelle 4.2) gesucht werden können. Für letzteren Service wurde der Parametername für den Regisseur von `name` auf `director` konkretisiert, um für allfällige Erweiterungen mit anderen Personentypen diese Typen im Gesprächskontext unterscheiden zu können. Diese beiden Anpassungen waren nicht zwingend für die Erweiterung notwendig, sie erleichterten jedoch die Umsetzung des neuen Services.

Die umfangreichste Anpassung der bestehenden Applikation war im API-Gateway nötig. Durch das Speichern von Entitäten und das eventuelle Abrufen ebendieser ändert sich der Informationsfluss durch die Anwendung.

Der API-Gateway, welcher den Ablauf innerhalb der Applikation steuert, musste deshalb für die neue Funktionalität aktualisiert werden. Dabei handelt es sich um die einzigen Änderungen, welche potentiell die Verfügbarkeit der kompletten Applikation beeinflussen. Alle Anfragen werden durch den API-Gateway geleitet, weshalb die korrekte Funktionsweise entscheidend ist.

5.3.3 Aggregierte Implementierung

Da der Kontext-Service eine unabhängige Datenquelle verwendet, wurde auch für die aggregierte Implementierung eine eigene Datenbank verwendet. Die Erkenntnisse aus der oben beschriebenen atomaren Implementierung gelten somit auch in diesem Fall. Ebenfalls wurde der Quellcode der ersten Kontext-Service-Umsetzung direkt übernommen. Die beiden Kontext-Services unterscheiden sich damit nur in der Konfiguration und können als gleichwertig im benötigten Aufwand betrachtet werden.

Weiter wurde der unterliegende IMDb-Service aktualisiert, um die neue Datenbank-Query und den neuen Parameternamen (vgl. Erklärung im Abschnitt 5.3.2) abzubilden. Die eigentlichen Anpassungen sind zwar identisch, allerdings war die Umsetzung weniger überschaubar. Der aggregierte Service übernimmt die Aufgaben von fünf atomaren Services und ist somit entsprechend komplexer. Durch die zusätzliche Komplexität steigt die Wahrscheinlichkeit für Fehler an. Es gibt mehr Stellen im Quellcode, an denen unvorhergesehene Probleme entstehen können. Dies setzt in einer Produktionsumgebung eine umfangreiche Testabdeckung voraus. Gleiches gilt auch für Services mit kleinerem Funktionsumfang, jedoch ist bei kleinen Services umfangreiches Testen einfacher, da in der Regel die zyklomatische Komplexität niedriger ist.

Die Erweiterung des API-Gateways verlief sehr ähnlich wie bei der atomaren Implementierung. Die Aufteilung der Aufgaben zwischen API-Gateway und den darunterliegenden Services unterscheidet sich bei den zwei Implementierungen. Während der atomare API-Gateway je nach Intent einen anderen Service anfragen muss, sendet der API-Gateway der aggregierten Implementierung alle Anfragen an einen Service. Bei letzterem sind die Anfragen deshalb entsprechend komplexer, da alle benötigten Informationen (wie zum Beispiel der Intent) noch mitgeliefert werden. Dies hat zur Folge, dass die Umsetzung des Kontext-Services etwas komplizierter war. Jedoch hängt dies in diesem Fall hauptsächlich von der gewählten Logik des Kontext-Services (sowie der Aufteilung zwischen API-Gateway und IMDb-Service) ab.

Grundsätzlich ist festzuhalten, dass die Erweiterung sich bei kleineren Services einfacher gestaltet. Die einzelnen Funktionen sind besser gekapselt und die ausgetauschten Daten sind aufgrund des kleineren Funktionsumfang überschaubarer. Sind von Erweiterungen betroffene Microservices auf verschiedene Teams aufgeteilt (vgl. Abschnitt 2.2.1), ist dagegen mehr Koordination nötig. Dies kann sich entsprechend negativ auf die Innovationsgeschwindigkeit auswirken. Es muss somit zwischen Einfachheit der eigentlichen Umsetzung und der zusätzlich nötigen Team-Koordination abgewogen werden, welche Variante im spezifischen Fall besser ist.

5.3.4 Monolithische Implementierung

Im Gegensatz zu den oberen beiden Implementierungen verwendet die Kontext-Erweiterung keine eigene Datenbank. Um die Informationen des Gesprächskontexts zu speichern, wurde eine neue Tabelle in der bestehenden IMDb-Datenbank erstellt. Dies eröffnet die Möglichkeit, mittels einer Fremdschlüsselbeziehung korrekte Werte für die gespeicherten Entitäten zu forcieren. Für komplexere Erweiterungen als das Einfügen einer einzelnen Tabelle in der Datenbank besteht jedoch das Risiko, dass bei der Änderung der Datenbank Probleme auftreten. Dies kann zum Teil- oder Totalausfall der Datenbank führen. Dieses Risiko ist bei komplett unabhängigen Datenbanken nicht oder viel seltener vorhanden.

Die Umsetzung der Kontext-Logik war erwartungsgemäss nochmals schwieriger als bei der aggregierten Implementierung. Durch den umfangreicheren Quellcode gestaltete sich die Änderung entsprechend komplexer. Weil Fehler in der monolithischen Architektur immer das Potenzial haben, die komplette Applikation abstürzen zu lassen, ist besondere Sorgfalt notwendig. Bei einer atomaren Implementierung der Applikation ist das Risiko für einen Totalausfall viel kleiner, was die Entwicklung von experimentellen Features anregen kann.

5.3.5 Fazit Erweiterbarkeit

Für Erweiterungen einer Applikation um neue Features bieten die verschiedenen Implementierungen Vor- und Nachteile. Durch die Unabhängigkeit der atomaren Services können Erweiterungen gekapselt vom Rest der Applikation umgesetzt werden. Der Einfluss auf die restlichen Services ist sehr klein oder im besten Fall gar nicht vorhanden. Dies wirkt sich positiv auf die Innovationsgeschwindigkeit aus. Die Umsetzung des Kontext-Services zeigt aber, dass die Situation nicht immer so einfach ist. Es waren Änderungen am zentralen API-Gateway sowie an einigen anderen Services notwendig. In grösseren Projekten kann dies zu Verzögerungen durch teamübergreifende Kommunikation und Koordination führen. Allerdings waren die Anpassungen pro Service zum Teil mit der Änderung einer einzelnen Zeile sehr klein.

Weiter zu beachten ist der Aufwand bei weitreichenden Erweiterungen einer Applikation. Sollten beispielsweise viele weitere Intents implementiert werden, welche von der IMDb-Datenquelle abgedeckt sind, entsteht für die atomare Implementierung ein erheblicher Mehraufwand. Pro Intent muss ein Service erstellt werden, was zu viel Codeduplizierung, potenziellem Koordinationsaufwand zwischen verschiedenen Teams und einer unübersichtlichen Gesamtarchitektur führt. Bei den anderen Implementierungen muss dagegen nur eine Codebasis verändert werden. Der Gesamtumfang einer Applikation spielt beim Serviceschnitt eine wichtige Rolle, damit die Anzahl der Services nicht unübersichtlich hoch wird.

Durch die umfangreichere Codebasis, die bei der aggregierten und monolithischen Implementierung verändert werden muss, steigt bei diesen Architekturen die Komple-

xität der Erweiterung. Der Quelltext ist weniger überschaubar und das Überprüfen auf allfällige Nebenwirkungen dauert länger als bei atomaren Services. Mit sauber gestaltetem Quellcode und strikter Befolgung der Prinzipien der hohen Kohäsion und tiefen Koppelung lässt sich diesem Problem entgegenwirken. Die Richtlinien der Servicetrennung in einer Microservice-Architektur können auch für die Gestaltung verschiedener Module eines Monoliths verwendet werden. Der Vorteil von der Aufteilung in einzelne Services liegt bei der praktisch erzwungenen Trennung verschiedener Aufgaben. Dies trägt dazu bei, den Quellcode wartbar zu halten und somit zukünftige Erweiterungen zu erleichtern. Die Gefahr für Softwareerosion ist bei getrennten Services geringer als in einem Monolithen.

Je nach Funktionsumfang der Services, Teamaufteilung und bestehender Codequalität kann der Aufwand für eine Erweiterung stark variieren. Aus den oben beschriebenen Erfahrungen lässt sich ableiten, dass weder sehr kleine noch sehr grosse Services vorteilhaft für die Erweiterbarkeit sind. Der ideale Umfang für einen Microservice wird von Richardson wie folgt beschrieben: „The application [should] be decomposed in a way so that most new and changed requirements only affect a single service. That is because changes that affect multiple services requires coordination across multiple teams, which slows down development“ [38]. Diese Aufteilung ist nicht trivial und muss vor und während der Entwicklung einer Applikation fortlaufend diskutiert werden.

Im konkreten Fall des Chatbot-Prototypen erfüllt die aggregierte Implementation diese Richtlinien am besten. Der Zugriff auf die Filmdaten erfolgt über einen einzelnen Service. Änderungen im Bereich der Filmdaten-Abfrage können damit in einem Service bzw. von einem Team umgesetzt werden. Der Kontext-Service, welcher eine komplett andere Funktion übernimmt, ist in einem zweiten Service gekapselt.

6 Diskussion und Ausblick

In der vorliegenden Arbeit haben wir mithilfe von Erfahrungsberichten, Empfehlungen aus der Industrie und dem Studium aktueller Forschungen eine Übersicht über den derzeitigen Wissenstand im Themengebiet Microservices zusammengetragen. Gleichzeitig haben wir Anhand von Prototypen die Vor- und Nachteile von Microservice-Architekturen im Vergleich zu einem monolithischen Ansatz evaluiert.

Mit dem Experiment im Bereich der Skalierbarkeit versuchten wir aufzeigen, wie sehr sich das Skalierungsverhalten der einzelnen Prototypen unterscheidet und was dies für Auswirkungen auf den Betrieb der Applikation sowie die UX hat. Überraschenderweise stellten wir fest, dass sich die Gesamtantwortzeiten zwischen den verschiedenen Applikationen in der Regel um weniger als 0.2 Sekunden unterscheiden. Dies liegt daran, dass die von uns gewählte Plattform Services standardmässig sehr gut skalieren kann. Während die Antwortzeiten für die verschiedenen Implementierungen sehr nahe bei einander liegen, unterscheiden sich unsere Prototypen enorm bei der Granularität der Skalierung. Der monolithische Prototyp lässt sich durch die gegebene Struktur nur als ganzes skalieren. Dies hat zur Folge, dass dem Monolithen mehr Ressourcen als effektiv nötig zugesprochen werden. Währenddessen skalieren beim der aggregierten und atomaren Implementierung nur diejenigen Module, die tatsächlich unter Last stehen. Ein kritischer Einflussfaktor auf die Performanz, welcher durch Skalierung nicht beeinflusst werden kann, ist die Verteilung der Daten in der Applikation. Durch die Aufteilung von Daten in verschiedene Datenbanken in der atomaren Implementierung müssen bestimmte Operationen auf den Daten in der Applikation durchgeführt werden, welche besser von der Datenbank gehandhabt werden. Dies erhöht die Komplexität der Applikationslogik und sorgt für unzumutbare Antwortzeiten.

Um die Verfügbarkeit zu evaluieren, griffen wir auf unsere Erfahrungen zurück, welche wir während der Entwicklung der Chatbots machten. Hier zeigte sich, wenig überraschend, dass die Verfügbarkeit der gesamten Applikation besser ist, je modularer die Struktur aufgebaut ist. Wenn bei der atomaren Implementierung einzelne Fragetypen aufgrund eines Fehler ausgefallen sind, standen die restlichen Features weiterhin zur Verfügung. Beim monolithischen Ansatz kam es im Falle eines Fehlers zum kompletten Absturz der Applikation. Die aggregierte Implementierung ist etwas toleranter für auftretende Fehler. Stürzt jedoch der IMDb-Service ab, so fällt die Hauptfunktionalität des Chatbots komplett aus. Wichtig ist es, systemkritische Elemente bereits beim Entwurf der Architektur zu erkennen und deren Risiko zu minimieren.

Zuletzt evaluierten wir die Erweiterbarkeit der verschiedenen Lösungsansätze. Hier zeigte sich der grösste Nachteil des atomaren Ansatzes. Die einzelnen Services sind sehr

klein geschnitten. Das Risiko ist sehr gross, dass bei einer Anpassung oder Erweiterung einer Applikation eine Vielzahl von Services betroffen sind. Bei der Erweiterung der atomaren Implementierung um einen Gesprächskontext mussten insgesamt drei bestehende Services angepasst werden. Bei grösseren Projekten kann dies zu erheblichen Problemen durch die zusätzlich notwendige Koordination führen. Das Beispiel der atomaren Implementierung soll aufzeigen, dass ein zu kleiner Serviceschnitt sich negativ auf die Ziele einer Microservice-Architektur auswirkt. Die aggregierte Implementierung verfügt zwar über eine grössere Codebasis pro Service und somit auch mehr Komplexität, der Serviceschnitt ist jedoch sinnvoller durchgeführt. Jede Funktionalität im gleichen Kontext ist auch im gleichen Service implementiert. Des Weiteren kann der Codekomplexität mittels Einhaltung von Clean-Code-Regeln, hoher Kohäsion und tiefer Kopplung entgegengewirkt werden.

Zusammengefasst weist die aggregierte Implementierung die Vor- und Nachteile auf, welche von einer Microservice-Architektur erwartet werden. Unsere Erfahrungen decken sich also mit den Berichten und Empfehlungen aus der Industrie. Es ist jedoch wichtig klarzustellen, dass mit dem praktischen Teil unserer Arbeit einzig unser spezifischer Fall abgedeckt wird. Nur das erarbeitete theoretische Wissen ist generell anwendbar. Jede Anwendung weist unterschiedliche Kriterien auf, in jeder Anwendung sind individuelle Bedürfnisse zu beachten. Das heisst auch, dass für jede Anwendung erneut evaluiert werden muss, ob eine Umsetzung mit Microservices sinnvoll ist. Denn Microservices weisen auch klare Nachteile auf, welche im Kapitel 2 adressiert werden. Die Empfehlung von Chris Richardson zum Einsatz einer Micoservice-Architektur entspricht auch unseren Resultaten: „The microservice architecture is not a silver bullet. It has several drawbacks. Moreover, when using this architecture there are numerous issues that you must address“ [12]. Die Aufgabenstellung der Evaluation von Microservices in einer Web-Applikation erachten wir somit als erfüllt.

Der von uns erstellte Chatbot bildet ein Grundgerüst, auf welchem in Zukunft aufgebaut werden kann. Softwarelösungen aus Bereichen wie NLP könnten von unserem aggregierten Prototypen Gebrauch machen und diesen erweitern. Erarbeitete Lösungen von anderen Bachelorarbeiten könnten als neue Services dem aggregierten Chatbot angehängt werden. Bestehende Applikationsteile, wie beispielsweise der Kontext-Service, könnten durch Lösungen ersetzt werden, welche sich explizit mit dieser Problemstellung auseinander setzen. Das jetzige Produkt könnte in Zukunft auch für weitere Experimente verwendet werden. Beispielsweise um die verschiedenen Cloud-Computing-Plattformen zu evaluieren und Unterschiede in ihrer Funktionsweise aufzuzeigen.

7 Verzeichnisse

7.1 Literaturverzeichnis

[1] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas und S. Gil, „Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud“, in *2015 10th Computing Colombian Conference (10CCC)*, Sept. 2015, S. 583–590.

[2] M. Fowler. (2014). *Microservices* [Online]. URL: `https://martinfowler.com/articles/microservices.html` [Stand: 19. 04. 2017].

[3] M. Fowler. (n.d.). *Microservices Resource Guide* [Online]. URL: `https://www.martinfowler.com/microservices/` [Stand: 12. 04. 2017].

[4] R. Meshenberg. (2016). *Microservices at Netflix Scale - First Principles, Tradeoffs & Lessons Learned* [Online]. URL: `https://youtu.be/57UK46qfBLY` [Stand: 13. 04. 2017].

[5] Netflix. (n.d.). *Netflix open source software center* [Online]. URL: `http://netflix.github.io` [Stand: 20. 04. 2017].

[6] P. Calçado. (2014). *Building Products at SoundCloud Part 1: Dealing with the Monolith* [Online]. URL: `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith` [Stand: 13. 04. 2017].

[7] P. Calçado. (2014). *Building Products at SoundCloud Part 2: Breaking the Monolith* [Online]. URL: `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith` [Stand: 13. 04. 2017].

[8] P. Calçado. (2014). *Building Products at SoundCloud Part 3: Microservices in Scala and Finagle* [Online]. URL: `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle` [Stand: 13. 04. 2017].

[9] T. Stuart. (2016). *Microservices and the monolith* [Online]. URL: `https://developers.soundcloud.com/blog/microservices-and-the-monolith` [Stand: 13. 04. 2017].

[10] M. Fowler. (2014). *Microservices* [Online].
URL: https://www.thoughtworks.com/talks/software-development-21st-century-xconf-europe-2014 [Stand: 28. 04. 2017].

[11] S. Newman, *Building Microservices*. 1st ed. Sebastopol: O'Reilly, 2015, S. 1–22.

[12] C. Richardson. (2017). *Microservice.io* [Online].
URL: http://microservices.io [Stand: 28. 04. 2017].

[13] C. Richardson. (2017). *Microservice patterns* [Online].
URL: https://www.manning.com/books/microservice-patterns [Stand: 28. 04. 2017].

[14] S. Newman. (n.d.). *Principles of microservices* [Online].
URL: http://samnewman.io/talks/principles-of-microservices/ [Stand: 19. 05. 2017].

[15] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri und Y. Al-Hammadi, „Performance comparison between container-based and vm-based services“, in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, Mrz. 2017, S. 185–190.

[16] C. Richardson. (n.d.). *Pattern: Circuit breaker* [Online]. URL:
http://microservices.io/patterns/reliability/circuit-breaker.html [Stand: 12. 05. 2017].

[17] M. Fowler. (2014). *Circuitbreaker* [Online].
URL: https://martinfowler.com/bliki/CircuitBreaker.html [Stand: 12. 05. 2017].

[18] M. Abbot, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. 2nd ed.
Boston: Addison-Wesley Professional, 2015, S. 357–358.

[19] C. Richardson. (n.d.). *Event-driven architecture* [Online].
URL: http://microservices.io/patterns/data/event-driven-architecture.html [Stand: 04. 05. 2017].

[20] C. Richardson. (n.d.). *Pattern: Api gateway / backend for front-end* [Online].
URL: http://microservices.io/patterns/apigateway.html [Stand: 12. 05. 2017].

[21] M. E. Conway. (1967). *Conway's law* [Online]. URL:
http://www.melconway.com/Home/Conways_Law.html [Stand: 05. 05. 2017].

[22] Chris Richardson. (n.d.). *Pattern: Database per service* [Online]. URL:
http://microservices.io/patterns/data/database-per-service.html [Stand: 19. 05. 2017].

[23] Chris Richardson. (n.d.). *Pattern: Shared database* [Online].
URL: http://microservices.io/patterns/data/shared-database.html [Stand: 19. 05. 2017].

[24] ZeroMQ. (2015). *Broker vs brokerless* [Online].
URL: http://zeromq.org/whitepapers:brokerless [Stand: 11. 05. 2017].

[25] C. Richardson. (2015). *Service discovery in a microservices architecture* [Online]. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ [Stand: 15. 05. 2017].

[26] C. Richardson. (n.d.). *Pattern: Health check api* [Online]. URL: http://microservices.io/patterns/observability/health-check-api.html [Stand: 15. 05. 2017].

[27] M. Fowler. (2013). *Continuousdelivery* [Online]. URL: https://martinfowler.com/bliki/ContinuousDelivery.html [Stand: 23. 05. 2017].

[28] M. Fowler. (2011). *Tolerant reader* [Online]. URL: https://martinfowler.com/bliki/TolerantReader.html [Stand: 23. 05. 2017].

[29] V. Reynolds und A. Gupta, *Getting started with microservices*. Ser. Refcardz. DZone, Inc., n.d.

[30] P. D. Francesco, I. Malavolta und P. Lago, „Research on architecting microservices: Trends, focus, and potential for industrial adoption“, in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, S. 21–30.

[31] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano und M. Lang, „Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures“, *Service Oriented Computing and Applications*, Bd. 11, Nr. 2, S. 233–247, Apr. 2017.

[32] N. H. Do, T. V. Do, X. T. Tran, L. Farkas und C. Rotter, „A scalable routing mechanism for stateful microservices“, in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, Mrz. 2017, S. 72–78.

[33] Google Inc. (2017). *An overview of app engine* [Online]. URL: https://cloud.google.com/appengine/docs/standard/python/an-overview-of-app-engine [Stand: 23. 05. 2017].

[34] Google Inc. (2017). *How instances are managed* [Online]. URL: https://cloud.google.com/appengine/docs/flexible/python/how-instances-are-managed.

[35] E. Schurman und J. Brutlag. (2009). *Performance Related Changes and their User Impact* [Online]. URL: https://youtu.be/bQSE51-gr2s [Stand: 27. 04. 2017].

[36] Google Inc. (2017). *App engine pricing* [Online]. URL: https://cloud.google.com/appengine/pricing [Stand: 25. 05. 2017].

[37] Google Inc. (2017). *The app engine standard environment* [Online]. URL: https://cloud.google.com/appengine/docs/standard [Stand: 25. 05. 2017].

[38] C. Richardson. (n.d.). *Pattern: Decompose by business capability context* [Online].
URL: `http://microservices.io/patterns/decomposition/decompose-by-business-capability.html` [Stand: 11. 05. 2017].

7.2 Glossar

ACID Atomic, Consistency, Isolation und Durability. Beschreibt die Anforderungen an Datenbanktransaktionen.

AJAX Asynchronous JavaScript And XML. Technologie um Daten zwischen Client und Server auszutauschen, ohne eine Website komplett neu zu laden.

api.ai Webservice für NLP.

Bottleneck Engpass in einem Systems, an welchem Verzögerungen/Wartezeiten entstehen.

Cache Pufferspeicher, welcher in der Regel vor ein langsames Speichermedium geschaltet wird. Der Cache soll effizientes Abfragen ermöglichen.

Chatbot Applikation, welche eine Unterhaltung mit einem Benutzer simuliert.

Cloud Computing Bereitstellung von Rechenleistung, Netzwerkinfrastruktur und Speicher über das Internet. Kann als (kostenpflichtiger) Service von Kunden verwendet werden.

Cloud Foundry Open Source Cloud-Computing-Plattform, welche das PaaS Servicemodell umsetzt. Kann genutzt werden, um eine private Cloud-Computing-Infrastruktur zu erstellen.

CORS Cross-origin resource sharing. Erlaubt den Zugriff auf Ressourcen von Domänen ausserhalb der Domäne, welche die Ressource bereitstellt.

DBpedia Gemeinschaftsprojekt mit dem Ziel, strukturierte Informationen aus Wikipedia zu extrahieren und Web-Anwendungen zugänglich zu machen.

Defensives Programmieren Vorgehensweise beim Programmieren, um Software gegen Abstürze wegen unvorhergesehener Umstände bei der Ausführung zu schützen.

DevOps Setzt sich aus den englischen Wörter *Development* und *IT Operations* zusammen und hat die Zusammenführung der beiden Teilbereiche zum Ziel. Dadurch soll die Effektivität und Effizient von Entwicklerteams erhöht werden, was wiederum für höhere Qualität im Endprodukt sorgen soll.

Endpoint URL, unter der ein Client auf einen Service zugreifen kann.

GCP Google Cloud Platform. Googles cloudbasiertes Angebot für Rechenleistung, Speicher, Datenverarbeitung etc.

GoF Gang of Four. Bezeichnung für die vier Authoren des Buches *Design Patterns: Elements of Reusable Object-Oriented Software*: Erich Gamma, Richard Helm,

Ralph Johnson und John Vlissides. Ihr Buch über Design-Patterns gilt in der Softwaretechnik als Standardlektüre für jeden Entwickler.

HTTP Hypertext Transfer Protocol. Ist ein zustandsloses Übertragungsprotokoll und wird hauptsächlich zur Datenübertragung von Internetseiten verwendet.

IMDb Internet Movie Database (englisch für Internet-Filmdatenbank). Datenbank zu Filmen, Fernsehserien, Videoproduktionen und Computerspielen sowie über Personen, die daran mitgewirkt haben.

Information Hiding Prinzip in der Softwareentwicklung welches beschreibt, dass Details der Implementierung gegen Aussen versteckt werden.

Intent Englisch für Absicht. Intent beschreibt die Absicht, die hinter einem Text steht.

Kubernetes Open Source System welches zum Management von Softwarecontainer verwendet wird. Ursprünglich entwickelt durch Google.

LOC Lines of Code. Wird oft zur Beschreibung des Programmieraufwands einer Applikation verwendet. Zählt die Anzahl Zeilen im Source Code.

Middleware Programm oder ein Dienst welche Funktionen vom Betriebssystem abstrahieren oder ergänzt und diese vereinfacht einer Applikation zur Verfügung stellt.

Netflix Serien- und Filmstreamingportal. Ursprünglich eine Online-Videothek.

NLP Natural Language Processing (englisch für das Verarbeiten natürlicher Sprache). Disziplin in der Informatik, die sich mit der Interaktion zwischen natürlicher Sprache und Computern befasst.

OSS Software, bei welcher der Quelltext öffentlich zugänglich ist. Wird möglicherweise durch eine Community entwickelt und kann meist gratis bezogen werden.

PaaS Cloud-Servicemodell welches zum Ziel hat, verschiedene Runtime-Environments mit anpassbaren Ressourcen wie Arbeitsspeicher dem Nutzer anzubieten. Ein Beispiel dafür ist die GCP.

pip Kommandozeilenprogramm um Python Packages zu installieren.

Regex Kurzwort für Regular Expression. Zeichenfolge, welche ein Suchmuster für Text definiert.

Round-Robin Einfaches Scheduling-Verfahren, bei welchem die zur Verfügung stehenden Ressourcen der Reihe nach verwendet werden.

SOA Serviceorientierte Architektur. Architekturmuster von 1996 welches zum Ziel hat, Softwarekomponenten und Geschäftsprozesse in wiederverwendbare Dienste zu kapseln.

Softwareerosion Schleichende Verschlechterung der Softwarequalität.

SoundCloud Musik- und Audiostreamingplattform auf der eigene Musik veröffentlichen werden kann. Wird vor allem von DJs stark verwendet.

SPOF Single Point of Failure. Beschreibt einen Punkt im System, dessen Ausfall einen Ausfall für das komplette System zur Folge hat.

Stakeholder Person oder Gruppe, welche Interesse am Verlauf oder Ergebnis eines Projektes hat.

Technologiestack Beschreibt die verwendeten Technologien, welche eine Firma oder ein Projekt verwendet.

ThoughtWorks Internationales IT-Unternehemn welches auf Softwareentwicklung und Auslieferung spezialisiert ist.

UI User Interface. Der Teil einer Anwendung mit welcher ein Benutzer interagieren kann.

URL Uniform Resource Locator. Eine Referenz auf eine Ressource in einem Netzwerk.

UX User Experience (englisch für Anwendererlebnis). Beschreibt die Erfahrungen eines Nutzers bei der Interaktion mit einem Produkt.

XConf Technologiekofnerenz, die von ThoughtWorks organisiert wird.

YAML YAML Ain't Markup Language. Menschenlesbares Dateiformat zur Datenserialisierung. Wird oft für Konfigurationsdateien verwendet.

zyklomatische Komplexität Anzahl linear unabhängiger Pfade auf dem Kontrollflussgraphen eines Software-Moduls.

7.3 Abbildungsverzeichnis

7.4 Tabellenverzeichnis

7.5 Codeausschnittverzeichnis

8 Anhang

8.1 Aufgabenstellung

Microservices sind ein aufkommender Trend in der Software-Architektur, der den Fokus auf kleine, leichtgewichtige Applikationen legt. Damit sollen unwartbare, monolitische Software-Monster vermieden werden.

Microservices sind kleine, unabhängige Prozesse, die miteinander über sprachunabhängige APIs kommunizieren. Dadurch kann man sie in beliebigen Programmiersprachen implementieren, separat weiterentwickeln und einfach skalieren. Eine Applikation wird nach diesem Architekturstil aus vielen solchen Microservices zusammengebaut.

Für das Deployment von Microservices gibt es verschiedene Varianten, z.B. die Container-Virtualisierung Docker, in der jeder Microservice als separater Prozess läuft und einfach gestartet und skaliert werden kann.

In dieser Bachelorarbeit soll an einer praktischen Anwendung elaboriert und demonstriert werden, wie man Microservices entwickeln und einsetzen kann. Als Thema für die praktische Anwendung kann dabei eine konventionell als Deployment-Monolith erstellte Applikation (z.B. MovieMan) genommen werden, um den Architekturstil direkt vergleichen und evaluieren zu können.

Zur Entwicklung steht ebenfalls entsprechende Infrastruktur zur Verfügung (Jenkins Build Server, Host für Docker Images etc.).

Die Aufgabe lässt sich grob in folgende Schritte unterteilen:

1. Einlesen in das Thema Microservices und Aufnehmen der Anforderungen an die Web-Applikation
2. Architektur und Design der Web-Applikation basierend auf einer Microservice-Architektur
3. Aufbau des Technologie-Stacks (Frameworks, Libraries etc.) zur Entwicklung der Microservices
4. Implementation der Web-Applikation (inkl. Continuous Delivery für jeden Microservice)
5. Evaluation der Skalierbarkeit, Verfügbarkeit und Erweiterbarkeit

6. Zusammenfassung der Vor- und Nachteile dieses Architekturstils und der praktischen Erfahrungen

8.2 Dokumentation Software

Um die Abtrennung der Codebasis vom Rest der Applikation für jeden Service zu gewährleisten, wird jeder Service in einem separaten Repository verwaltet. Einige Repositorys werden für mehrere Implementierungen verwendet. In solchen Fällen sind im Repository mehrere Konfigurationsdateien für die verschiedenen Implementierungen vorhanden. Alle Repositorys, welche während dieser Arbeit erstellt wurden, sind auf der beigelegten CD zu finden und in der jeweiligen Readme-Datei beschrieben.

Bevor die Services in die Google App Engine deployt werden können, muss sichergestellt werden, dass alle benötigten Packages vorhanden sind. Python Packages, welche mittels `pip` installiert werden, müssen zusammen mit dem Service deployt werden. Dazu müssen die Packages in einen lokalen Ordner heruntergeladen werden. Die entsprechenden Packages sind in den jeweiligen Repositorys in der `requirements.txt`-Datei aufgelistet. Mittels `pip` können die Packages aus dieser Datei automatisch heruntergeladen werden. Dies geschieht mit dem Befehl aus Codeausschnitt 8.1. Weiter muss konfiguriert werden, in welchem Ordner die Packages zu finden sind. Diese Konfiguration wird in der Datei `appengine_config.py` vorgenommen. Ein Beispiel ist im Codeausschnitt 8.2 abgebildet.

```
1 $ mkdir lib
2 $ pip install -t lib -r requirements.txt
```

Codeausschnitt 8.1: Herunterladen von Python Packages in den `lib`-Ordner.

```
1 from google.appengine.ext import vendor
2
3 vendor.add('lib')
```

Codeausschnitt 8.2: Konfiguration des Ordners für benötigter Python Packages für die Google App Engine.

Die einzelnen Services sind so implementiert, dass sie auf der Google App Engine deployt werden können. Alle Services sind in Python implementiert und sind für die Python Standardumgebung[1] konzipiert. Das Deployment in die Google App Engine wird mittels Google Cloud SDK[2] durchgeführt. Codeausschnitt 8.3 zeigt, wie das Deployment eines Services funktioniert.

[1] `https://cloud.google.com/appengine/docs/standard/python/` [Stand: 29.05.2017]
[2] `https://cloud.google.com/sdk/docs/` [Stand: 29.05.2017]

```
$ gcloud app deploy <app.yaml> --project <project-name>
```

Codeausschnitt 8.3: Deploy-Command per Google Cloud SDK.

Die in der Arbeit verwendeten Filmdaten liegen in der Form eines MySQL-Datendumps vor. Diese Daten stammen von InIT Institut für angewandte Informationstechnologie und können dort bezogen werden. Standardmässig sind die IMDb-Daten in einer Datenbank in verschiedene Tabellen aufgeteilt. Von unseren Prototypen werden die Tabellen `directors`, `genres` und `movies` verwendet. Für den atomaren Prototypen wurden die Tabellen in eigene, unabhängige Datenbanken extrahiert.

Die Konfiguration von api.ai wird über das api.ai-Webinterface vorgenommen. Alle Intents verwenden Parameter, um die benötigten Information (z. B. Name des Regisseurs) aus den Fragen zu extrahieren. Wie api.ai konfiguriert werden kann, ist in der offiziellen Dokumentation über Intents[3] und Parameter[4] ersichtlich. Abbildung 8.1 zeigt als Beispiel die Konfiguration des Intents für Filme anhand des Genres und Erscheinungsjahr. Verwendet wird api.ai über den bereitgestellten `/query`-Endpoint.[5]

[3] `https://docs.api.ai/docs/concept-intents` [Stand: 29.05.2017]
[4] `https://docs.api.ai/docs/concept-actions` [Stand: 29.05.2017]
[5] `https://docs.api.ai/docs/query` [Stand: 29.05.2017]

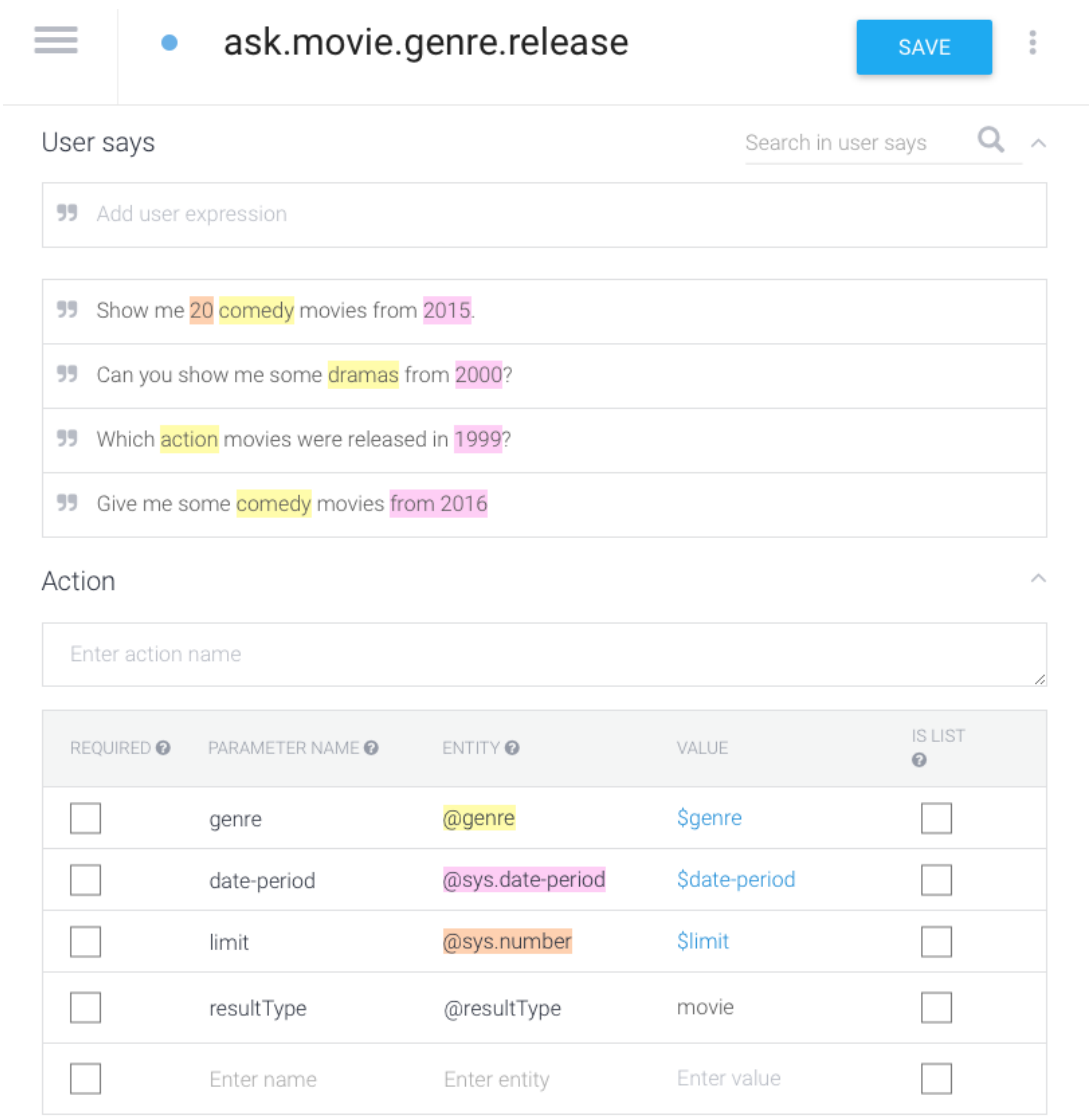

Abbildung 8.1: Api.ai-Konfiguration des Intents `ask.movie.release.genre`.