Eine Microservice-Architektur für die Semantische Analyse im Kontext der Computerlinguistik

Masterarbeit

Martin Zloch

24. Oktober 2016

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
Dr. Arne Johanson
Philipp Döhring

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

In dieser Arbeit beschäftigen wir uns mit dem Architekturstil Microservices für die semantische Analyse von Texten im Umfeld der Computerlinguistik. Zunächst zeigen wir einige Anwendungsmöglichkeiten für die Textanalyse auf und begründen, dass sich die semantische Analyse zur Umsetzung als Microservice-Architektur eignet.
Um die Grundlagen für diese Arbeit deutlich zu machen, erläutern den Begriff Microservices und die Charakteristiken von Microservices sowie einige für die Arbeit relevanten Technologien. Zudem geben wir einen kurzen Einblick in die Computerlinguistik und insbesondere die Textanalyse als Umfeld der semantischen Analyse.
Auf dieser Basis präsentieren wir eine ressourcenbasierte und eine funktionale Gliederungen der Textanalyse und erläutern, wie diese Gliederungen zur Umsetzung der semantischen Analyse als Microservice-Architektur genutzt werden können. Neben der Gliederung eines Systems gibt es bei Microservice-Architekturen weitere Aspekte zu beachten. Diese sind zum Beispiel Kommunikations- und Koordinationsmethoden, Service-Entdeckung und Skalierbarkeit. Wir erläutern einige Aspekte und nennen mehrere Alternativen zur Realisierung. Die Realisierungsalternativen werden für das Umfeld der semantischen Analyse gegeneinander abgewogen. Aus diesen Abwägungen resultiert die von uns vorgestellte Architektur.
Mit StanbolµS präsentieren wir eine auf Apache Stanbol basierende Implementierung der Architektur. Anhand von StanbolµS evaluieren wir den Einfluss der Microservice-Architektur auf die Performance. Wir zeigen, dass Rechenzeit, Speicherbedarf und Prozessorauslastung im Microservice-Ansatz signifikant höher sind, als bei einer vergleichbaren monolithischen Lösung. Außerdem weisen wir nach, dass unser Microservice-System durch die Skalierungsmöglichkeiten mehr Durchsatz erreichen kann, als ein unskaliertes monolithisches System. Anschließend stellen wir einige verwandte Arbeiten vor, die sich ebenfalls mit Microservice-Architekturen, Apache Stanbol und dem Einfluss von Microservices auf die Performance beschäftigen.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 ppi Media GmbH . . . 3
1.2 Ziele . . . 4
1.2.1 Ziel 1: Architekturentwurf . . . 4
1.2.2 Ziel 2: Implementierung . . . 4
1.2.3 Ziel 3: Evaluierung . . . 4
1.3 Aufbau der Arbeit . . . 5

2 Grundlagen und Technologien **7**
2.1 Computerlinguistik . . . 7
2.1.1 Phasen der Textanalyse . . . 7
2.1.2 Semantische Analyse . . . 8
2.2 Microservices . . . 9
2.2.1 Charakterisierung von Microservices und Microservice-Architekturen 10
2.2.2 Vor- und Nachteile von Microservices . . . 10
2.2.3 Vertikalen . . . 11
2.3 Nanoservices . . . 12
2.4 Serviceorientierte Architektur (SOA) . . . 12
2.5 Open Services Gateway initiative (OSGi) . . . 13
2.6 Apache Stanbol . . . 13
2.7 Hypertext Transfer Protocol (HTTP) . . . 14
2.8 Representational State Transfer (REST) . . . 15

3 Eine Microservice-Architektur für die Semantische Analyse **17**
3.1 Partitionierung von Computerlinguistik-Anwendungen . . . 17
3.1.1 Definition von Partitionierung . . . 18
3.1.2 Bedeutung der Partitionierung in Microservice-Architekturen . . . 18
3.1.3 Integration der Vorbereitungsphasen . . . 19
3.1.4 Partitionierung der Semantischen Analyse . . . 22
3.2 Weitere Aspekte von Microservice-Architekturen . . . 29
3.2.1 Kommunikation . . . 29
3.2.2 Service-Entdeckung . . . 32
3.2.3 Koordination von Arbeitsabläufen . . . 35
3.2.4 Grafische Benutzeroberflächen und Ergebniskomposition . . . 40
3.2.5 Datenhaltung . . . 41
3.2.6 Entwicklung . . . 43

3.2.7 Betrieb . . . 44
3.3 Architekturbeschreibung . . . 47
3.3.1 Gesamtbild der Architektur . . . 48
3.3.2 Interner Aufbau der Microservices . . . 49
3.3.3 Koordination und Kommunikation . . . 50

4 Die Implementierung *StanbolμS* **53**
4.1 Architektur von StanbolμS . . . 53
4.1.1 Umsetzung der Konzepte in StanbolμS . . . 55

5 Evaluierung **61**
5.1 Ziel 1: Analyse der Performance von StanbolμS . . . 61
5.1.1 F1.1: Vergleich der Zeit für die Verarbeitung von Texten . . . 62
5.1.2 F1.2: Messung der Systemauslastung . . . 66
5.2 Ziel 2: Analyse der Skalierung von StanbolμS . . . 71
5.2.1 Experiment . . . 71
5.2.2 Ergebnisse . . . 73
5.2.3 Diskussion der Ergebnisse . . . 76
5.3 Zusammenfassung . . . 77

6 Verwandte Arbeiten **79**
6.1 Microservice-Architekturen . . . 79
6.2 Apache Stanbol . . . 80
6.3 Performance Evaluierungen . . . 80

7 Fazit und Ausblick **83**

A Handbuch der Implementierung **85**
A.1 Inhalt des Datenträgers . . . 85
A.2 Bauen der Anwendungen . . . 86
A.2.1 Apache Stanbol . . . 86
A.2.2 StanbolμS . . . 87
A.2.3 Monolith . . . 87
A.2.4 System-Monitor . . . 87
A.3 Starten der Anwendungen . . . 88
A.3.1 StanbolμS . . . 88
A.3.2 Monolith . . . 89
A.3.3 System-Monitor . . . 89
A.4 Entwicklung . . . 90
A.5 Python-Skripte . . . 90

Bibliografie **91**

Abkürzungsverzeichnis

VM Virtuelle Maschine

NLP Natural Language Processing [Indurkhya und Damerau 2010]

SOA Serviceorientierte Architektur [Perrey und Lycett 2003]

CMS Content-Management-System [Baxter und Vogt 2002]

SCMS Semantisches Content-Management-System [Christ und Nagel 2011]

DDD Domain-Driven Design[Evans 2004]

GUI Grafische Benutzeroberfläche

IDL Interface-Beschreibungssprache

WSDL Web Service Description Language [Christensen u. a. 2001]

DNS Domain Name System [Mockapetris und Dunlap 1988]

API Anwendungs-Programmierschnittstelle

DRY Don't Repeat Yourself! [Haoyu und Haili 2012]

SRP Single Responsibility Principle [Martin 2002]

REST Representational State Transfer [Richardson u. a. 2007]

JSON JavaScript Object Notation [Crockford 2006]

XML Extensible Markup Language [Bray u. a. 1998]

IDE Integrierte Entwicklungsumgebung

EDA Event Driven Architecture [Michelson 2006]

HTTP Hypertext Transfer Protocol [Fielding u. a. 1999]

ESB Enterprise Service Bus [Chappell 2004]

EAI Enterprise Application Integration [Linthicum 2000]

OSGi Open Services Gateway initiative [Hall u. a. 2011]

IP Internet Protokoll [Postel 1981]

URI Uniform Resource Identifier [Berners-Lee u. a. 2005]

GQM Goal-Question-Metric-Verfahren [Basili 1992]

IKS Interactive Knowledge Stack [Pereira 2012]

RDF Resource Description Framework [Manola u. a. 2004]

JVM Java Virtuelle Maschine [Lindholm u. a. 2014]

AWS Amazon Web Services [Amazon 2015]

SOAP Simple Object Access Protocol [Box u. a. 2000]

XML-RPC Extensible Markup Language Remote Procedure Call [Winer 1999]

Kapitel 1

Einleitung

Die Entstehung und Entwicklung von Sprache wird meist nicht von zentraler Stelle festgelegt, sondern entsteht ungeplant. Dies führt dazu, dass Sprache oft nicht eindeutig ist. So hängt die Bedeutung eines Wortes von seinem Kontext ab. Zu diesem Kontext können neben dem Satz und dem Text, in dem das Wort vorkommt, auch die Betonung sowie zeitliche und räumliche Faktoren gehören. Außerdem können sich die Grammatik und das Vokabular einer Sprache sowie die Bedeutung und Verwendung von Wörtern mit der Zeit ändern. Menschen können mit dieser Mehrdeutigkeit umgehen, da sie intuitiv den Kontext erfassen und Texte anhand dessen interpretieren.
Da natürliche Sprache in gesprochener und schriftlicher Form die primären Mittel menschlicher Verständigung sind, werden auch Informationen, die Menschen über Computer verteilen, häufig in natürlicher Sprache verfasst. Beispiele für Informationskanäle in natürlicher Sprache sind Statusupdates auf Sozialen Netzwerken, Sofortnachrichten, Emails und Nachrichtenmeldungen. Die weite Verwendung dieser Kanäle führt dazu, dass sehr viele Texte erstellt werden: Über *Whatsapp*[1] wurden an einem Tag 20 Milliarden Nachrichten versendet,[2] auf *Twitter*[3] im Jahr 2013 im Schnitt etwa 500 Millionen Tweets pro Tag veröffentlicht,[4] und allein die deutsche Ausgabe der Online-Enzyklopädie *Wikipedia*[5] umfasst über 1,9 Millionen Artikel.[6]
Die verbreitete Nutzung von Computern für den Informationsaustausch zwischen Menschen hat also zur Folge, dass sehr viele Texte für Computer zugänglich sind. Diese Schriftstücke sind für Maschinen aber durch ihre Mehrdeutigkeit aber nicht ohne weiteres verständlich. Die enthaltenen Informationen könnten jedoch für viele Zwecke genutzt werden. So wäre es sozialen Netzwerken möglich, Informationen zu ihren Nutzern gewinnen, um zielgerichtet Werbung anzuzeigen. Staatliche Einrichtungen könnten öffentliche Mitteilungen und Forenbeiträge analysieren, um kriminelle Aktivitäten zu erkennen. Durch inhaltliche Beziehungen zwischen wissenschaftlichen Texten oder Nachrichtenmeldungen und Wörterbucheinträgen würde außerdem die Recherche von Journalisten oder Forschern erleichtert werden. Unternehmen wäre es möglich, automatisiert Nachrichten und Statusmeldungen zu finden, die sie betreffen und Statistiken, wie zum Beispiel Meinungsbilder,

[1] https://www.whatsapp.com/?l=de
[2] https://twitter.com/WhatsApp/status/451198381856014337
[3] https://twitter.com/?lang=de
[4] http://www.internetlivestats.com/twitter-statistics/
[5] https://wikipedia.de/
[6] https://de.wikipedia.org/wiki/Spezial:Statistik

daraus zu generieren. Suchmaschinen könnten durch semantische Informationen ihre Ergebnisse verbessern, indem nicht nur textuell, sondern auch inhaltlich relevante Begriffe gesucht werden.
Die *Computerlinguistik* (eng: Natural Language Processing (NLP)) beschäftigt sich laut Chowdhury [2003] damit, Computer fähig zu machen, natürliche Sprache zu verstehen und erzeugen. Dazu wird bei der Interpretation eines Textes ein Modell des Texts erzeugt. Das Modell enthält die Informationen des Texts in einer für Computer verständlichen Form. Die Modellbildung nennen wir *Analyse des Textes* oder *Textanalyse*. Da das Modell nicht nur die Struktur des Texts, sondern auch Verweise auf die Bedeutung enthalten soll, müssen semantische Informationen ermittelt werden. Dieser Teil der Textanalyse wird *semantische Analyse* genannt.
Es gibt bereits Anwendungen, die diese Aufbereitung von Texten anbieten: Neben dem in Python[7] geschriebenem *Natural Language Toolkit*,[8] der Sammlung von Computerlinguistik-Anwendungen *Stanford CoreNLP*,[9] und *Apache Stanbol*[10] existieren noch weitere. Diese Anwendungen sind meist Bibliotheken, die in andere Anwendungen eingebunden werden können und nicht als eigenständiges System operieren sollen.
Natürliche Sprache kann, wie eingangs beschrieben, mehrdeutig sein. Zudem ist der Aufbau von Sprachen vielfältig und meist nicht streng geregelt. Die Mehrdeutigkeit und Diversität von Sprachen führt dazu, dass manche Aufgaben im Rahmen der Analyse von Texten komplex sind. Außerdem entwickeln sich Sprachen mit der Zeit und machen somit Anpassungen der Textanalyse erforderlich. Diese Bedingungen stellen hohe Anforderungen an die Wartbarkeit von Computerlinguistik-Implementierungen und Software-Systemen, die solche Implementierungen verwenden.
Gute Wartbarkeit ist eines der Ziele bei der Verwendung von Microservice-Architekturen, denn Microservices lassen sich unabhängig voneinander entwickeln und ausliefern. Dies ermöglicht es, verschiedene Aspekte der Textanalyse isoliert voneinander zu betrachten. Die Technologiefreiheit von Microservices erlaubt es außerdem, für jeden Service die passende Technologie einsetzen zu können. So kann für Teilaspekte jeweils die beste Implementierungen genutzt werden, ohne sich auf eine einzige Computerlinguistik-Bibliothek festlegen zu müssen.
Die Anzahl der zu verarbeitenden Texte kann stark variieren. Das macht erforderlich, das dafür verwendete System skalieren zu können. Durch ihre Autonomie und die lose Kopplung können Microservices es vereinfachen, gezielt die Methoden der Computerlinguistik zu skalieren, die den größten Rechenaufwand bedeuten.
Bei manchen Anwendungsfällen ist die Menge der Texte so groß, dass die Verarbeitung kontinuierlich durchgeführt werden muss. Das Computerlinguistik-System kann in diesem Fall nicht ohne weiteres ausgeschaltet und aktualisiert werden. Das Microservice-Konzept sieht bereits auf Architekturebene vor, dass Services ausfallen und ersetzt werden. Dadurch

[7] https://www.python.org/
[8] http://www.nltk.org/
[9] http://stanfordnlp.github.io/CoreNLP/
[10] http://stanbol.apache.org/

wird die Wartung im laufenden Betrieb vereinfacht.
Microservice-Architekturen haben jedoch auch Nachteile. Die Verwendung eines verteilten Systems erschwert das Testen und Beobachten der Anwendung und kann zu Einbußen in der Performance führen.
Wir untersuchen in dieser Arbeit, wie sich die Verfahren der Computerlinguistik, die für die semantische Analyse nötig sind, in Microservices aufteilen lassen und ob die angeführten Vorteile die Nachteile einer solchen Architektur in diesem Anwendungsbereich überwiegen.
Die ppi Media GmbH unterstützt diese Arbeit mit Ressourcen wie der in Abschnitt 5.2 auf Seite 71 verwendeten Amazon Web Services (AWS) Instanzen und mit der Betreuung durch Philipp Döhring.

1.1. ppi Media GmbH

Die ppi Media GmbH[11] ist ein seit 1984 bestehendes deutsches Software-Unternehmen mit 140 Mitarbeitern und Standorten in Hamburg, Kiel und Chicago. Die Firma hat sich auf den Bereich Publishing fokussiert und bietet Software-Lösungen, die den kompletten Produktionsablauf von Tageszeitungen und Magazinen abdecken. Hierbei werden analoge und digitale Kanäle unterstützt. Die angebotenen Produkte umfassen unter anderem Lösungen zum Planen und Editieren von Ausgaben, zum Schalten von Werbung, zum Tracking von Online-Ausgaben und zum Druck von Ausgaben. Die ppi Media GmbH hat einen internationalen Kundenkreis mit über 100 Kunden in Deutschland, Skandinavien, Kanada, Indien und Südafrika.
Mit Konzepten wie SOA, die mit Microservices verwandt sind, hat ppi Media GmbH bereits seit längerem Erfahrung. Die Arbeit mit Microservices begann vor etwa einem halben Jahr und befindet sich zur Zeit in einer experimentellen Phase. Von dieser Arbeit erwartet die ppi Media GmbH , Technologien, die Microservice-Architekturen unterstützen, zu evaluieren und mögliche Verwendungszwecke des Microservice-Architekturstils zu ermitteln.
Die semantische Analyse kann im redaktionellen Umfeld vor allem bei der Recherche angewendet werden, um die Suche nach relevanten Artikeln zu verbessern. Eine Implementierung als Service erleichtert hierbei, die Analyse in verschiedene Produkte zu integrieren. So kann für verschiedene Produkte die gleiche Wissensbasis verwendet werden. Das hat den Vorteil, dass mehr Wissen generiert wird und sich so die Qualität der Ergebnisse verbessern kann. Zudem kann der Service als Dienstleistung angeboten werden und muss nicht als Software vermarktet werden, was weitere geschäftliche Möglichkeiten eröffnet.

[11] https://www.ppimedia.de/ppi/

1.2. Ziele

In diesem Abschnitt definieren wir die Ziele dieser Arbeit. Das erste Ziel ist ein Architekturentwurf für die semantische Analyse mit Microservices. Eine Implementierung dieser Architektur zu schaffen ist das nächste Ziel. Das letzte Ziel ist die Evaluierung der Architektur und ihrer Implementierung sowie des Ansatzes, eine Microservice-Architektur für die semantische Analyse zu verwenden.

1.2.1. Ziel 1: Architekturentwurf

Für Microservice-Systeme ist die Partitionierung der Anwendung wichtig. Daher ist die Partitionierung der für die semantische Analyse benötigten Verfahren ein Teilaspekt des Architekturentwurfs. Die Partitionen sollen den in Abschnitt 2.2 gegebenen Definitionen von Microservices entsprechen. Der Datenaustausch zwischen verschiedenen Microservices sowie Möglichkeiten, um die Anwendung zu skalieren, sind ebenfalls Teil des Architekturentwurfs. Zudem muss der Architekturentwurf die Koordination der Microservices beschreiben, falls es Aktionen gibt, die die Ausführung mehrerer Schritte in einer festen Reihenfolge erfordern.

1.2.2. Ziel 2: Implementierung

Als zweites Ziel soll eine Implementierung des in Kapitel 3 vorgestellten Architekturentwurfs erstellt werden. Diese Realisierung dient der beispielhaften Veranschaulichung und der Evaluierung der Arbeit. Die Implementierung soll Komponenten von Apache Stanbol verwenden und mindestens die Funktionalität einer Konfiguration der monolitischen Stanbol-Variante enthalten. Das entwickelte System soll eine REST-Schnittstelle [Richardson u. a. 2007] bereitstellen und das Spring-Framework [Chapman 2015] verwenden. Des Weiteren sollen einzelne Microservices auch nur die von ihnen benötigten Komponenten enthalten. Den vollständigen Monolithen in jeden Microservice zu integrieren widerspricht dem Microservice-Konzept und möchten wir daher vermeiden.

1.2.3. Ziel 3: Evaluierung

Nach dem Erstellen des Architekturentwurfs und seiner Implementierung ist das letzte Ziel, den Microservice-Ansatz für die semantische Analyse zu evaluieren. Dafür erstellen wir mittels Goal-Question-Metric-Verfahren (GQM) Metriken, anhand derer wir bestimmen, welche Vor- und Nachteile die Implementierung als Microservices im Vergleich zur monolithischen Version hat. Die zu vergleichenden Qualitätsmerkmale sind Effizienz, Wartbarkeit und Übertragbarkeit wie durch *ISO 9216* [2000] definiert, soweit sich die Merkmale im Rahmen dieser Arbeit prüfen lassen.

1.3. Aufbau der Arbeit

Dieses Kapitel liefert eine Einleitung zu dem Thema der Arbeit sowie die Motivation für die Arbeit. Zudem wird mit der ppi Media GmbH das betreuende Unternehmen vorgestellt und die Ziele der Arbeit erklärt. In Kapitel 2 stellen wir die grundlegenden Konzepte und Technologien dieser Arbeit vor: Computerlinguistik, Semantische Analyse, Microservices und Apache Stanbol. In dem darauf folgenden Kapitel 3 erklären wir zwei Ansätze, die Textanalyse zu partitionieren und stellen einige Aspekte von Microservice-Architekturen vor. Anschließend präsentieren wir den Architekturentwurf mit Microservices für semantische Analyse. Die Implementierung der Architektur StanbolµS mit Apache Stanbol wird in Kapitel 4 vorgestellt. In Kapitel 5 evaluieren wir die Performance von StanbolµS im Vergleich mit einem Monolithen. Kapitel 6 und Kapitel 7 schließen die Arbeit mit dem Blick auf verwandte Arbeiten, einem Fazit und einem Ausblick auf mögliche anschließende Forschungsthemen ab. In Anhang A dokumentieren wir Details zur Implementierung.

Kapitel 2

Grundlagen und Technologien

Für das Verständnis dieser Arbeit ist die Kenntnis einiger Konzepte und Technologien erforderlich. In diesem Kapitel erläutern wir daher die Begriffe *Computerlinguistik*, *semantische Analyse* und *Microservice*. Wir grenzen zudem SOA und *Nanoservices* von Microservices ab und stellen mit HTTP, OSGi und REST weitere Technologien vor. Außerdem beschreiben wir Details zu *Apache Stanbol*, welches die Grundlage für unsere Implementierung StanbolµS ist.

2.1. Computerlinguistik

Computerlinguistik bezeichnet nach Chowdhury [2003] ein wissenschaftliches Feld, das sich mit der Verarbeitung natürlicher Sprache durch Computer beschäftigt. Das Ziel der Computerlinguistik ist, Informationen aus natürlichsprachlichen Quellen für Computer nutzbar zu machen und natürlichsprachliche Ausgabemethoden zu entwickeln. Die Quellen und Ausgabemethoden können textuell oder auditiv sein.
Einige Anwendungsgebiete nennen wir in der Einleitung. Weitere sind automatische Übersetzung, Textanalyse und Zusammenfassung, Benutzerschnittstellen, Spracherkennung, künstliche Intelligenz und Expertensysteme.
Im Rahmen dieser Arbeit beschäftigen wir uns mit der semantischen Analyse, welche Teil der Textanalyse ist. Wir erläutern daher zunächst die Phasen der Textanalyse, um das Umfeld der semantischen Analyse zu verdeutlichen.

2.1.1. Phasen der Textanalyse

Dale [2010] zufolge wird in der Computerlinguistik die Textanalyse anhand der Aufteilung der theoretischen Linguistik in Phasen unterteilt: Syntax, Semantik und Pragmatik. Die Phasen heißen entsprechend *syntaktische Analyse*, *semantische Analyse* und *pragmatische Analyse*. Syntax bezeichnet hier den Aufbau von Wörtern und Sätzen, die Semantik ist die intendierte Bedeutung von Wörtern, Sätzen und Texten. Pragmatik bezeichnet den Gebrauch der Sprache, also in welchem Kontext eine Äußerung steht und wie sich dadurch ihre Bedeutung ändert. Andere Teilgebiete der Linguistik wie *Phonetik* und *Phonologie*, die sich mit gesprochener Sprache beschäftigen, finden in anderen Bereichen der Computerlinguistik Anwendung.

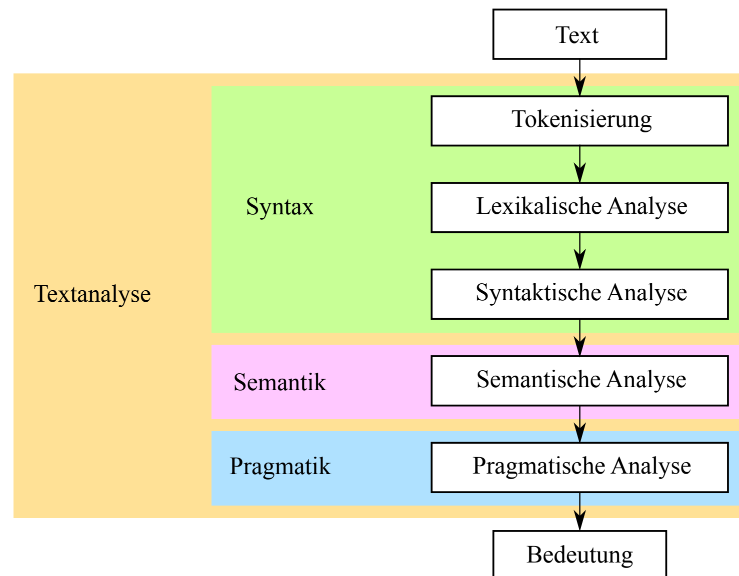


Abbildung 2.1. Phasen der Textanalyse nach Dale [2010]

Dale [2010] beschreibt die in Abbildung 2.1 ersichtliche feinere Aufteilung der Phasen. Die syntaktische Analyse wird in Tokenisierung, lexikalische Analyse und syntaktische Analyse aufgeteilt. Tokenisierung ist ein vorbereitender Schritt für die anderen Phasen und dient der Zerlegung des Textes in zusammenhängende Einheiten. Die lexikalische Analyse entspricht der linguistischen Kategorie *Morphologie*, also der Bildung von Wörtern.

2.1.2. Semantische Analyse

Die Phasen der Syntax-Analyse erheben vor allem strukturelle Daten über den zu analysierenden Text. Die semantische Analyse hingegen hat als Aufgabe, die intendierte Bedeutung zu erfassen und im Text genannte Umstände mit Gegenständen und Gegebenheiten der realen Welt zu verknüpfen.
Goddard und Schalley [2010] unterscheiden zwei Kategorien von Semantik: Lexikalische und supralexikalische Semantik. Lexikalische Semantik wird auch Wortsemantik genannt und bezeichnet die Bedeutung einzelner Wörter. Mit supralexikalischer Semantik ist die Bedeutung von Gruppen zusammenhängender Wörter gemeint. Da diese Gruppen häufig Sätze oder Teilsätze sind, nennt man die supralexikalische Semantik auch Satzsemantik. In Abbildung 2.2 zeigen wir Beispiele für beide Kategorien anhand des Satzes „Microservices dienen dazu, Software in Module aufzuteilen und dadurch die Änderbarkeit der Software zu verbessern.“ [Wolff 2016a].

Wort	Lexikalische Semantik
Microservices	Architekturstil im Softwareengineering
Software	Nicht technisch-physikalischer Funktionsbestandteil einer Datenverarbeitungsanlage
Modul	Austauschbares, komplexes Element innerhalb eines Gesamtsystems, eines Gerätes oder einer Maschine, das eine geschlossene Funktionseinheit bildet
Änderbarkeit	Eigenschaft, geändert werden zu können

(a) Lexikalische Semantik nach duden.de [2016]

- ▻ Microservices teilen Software in Module auf
- ▻ Durch die Aufteilung von Software verbessert sich die Änderbarkeit

(b) Supralexikalische Semantik

Abbildung 2.2. Beispiele für die Kategorien der Semantik

In Abschnitt 3.1.4 auf Seite 23 erläutern wir, dass die Unterscheidung zwischen diesen Arten der Semantik aber in der Computerlinguistik häufig keine harte Grenze bildet. Wir benutzen daher ein zweites Verfahren zur Kategorisierung von Semantik. Wir verwenden die von Gangemi [2013] aufgezählten Methoden der semantischen Analyse zur Kategorisierung der Semantik. Die Methoden repräsentieren jeweils die Kategorie der Semantik, zu der ihre Ergebnisse gehören. Die zur Methode *Individuenerkennung* gehörende Semantische Kategorie ist zum Beispiel *Erkannte Individuen*. Weitere Methoden zählen wir in Abschnitt 3.1.4 auf Seite 24 auf.

2.2. Microservices

Wie Lewis und Fowler [2014] schreiben, gibt es keine zentrale Definition des Microservice-Architekturstils. Wir fassen hier daher einige Charakteristika zusammen und geben eine eigene Definition, auf die wir uns innerhalb dieser Arbeit beziehen.
Lewis und Fowler [2014] beschreiben Microservices als Architekturstil, bei dem eine einzelne Anwendung als Komposition mehrerer kleiner Services implementiert wird. Microservices sind demnach ein Ansatz zur Modularisierung von Software. Die Services sind eigene Prozesse, die nach Wolff [2016a] nur eine einzelne Aufgabe haben. Wolff erwähnt außerdem, dass Microservices *Unterstützungsdienste* mitbringen könnten. Beispiele seien Suchmaschinen oder eine spezielle Datenbank. Um diese Unterstützungsdienste in Microservices integrieren zu können, dürfen Microservices nach ihm nicht nur Prozesse, sondern auch virtuelle Maschinen sein.

Newman [2015] charakterisiert Microservices als automatisch und unabhängig auslieferbar. Um dieses Kriterium zu erreichen, fordern Lewis, Fowler und Wolff Protokolle zur Kommunikation, die lose Kopplung unterstützen. Beispiele dafür sind REST und Messaging. Hierdurch wird auch eine technologieneutrale Schnittstelle erreicht, die Newman empfiehlt, um bei Microservice-Architekturen heterogene Technologien verwenden zu können. Microservices sollen möglichst unabhängig voneinander geändert werden können. Lewis und Fowler raten daher dazu, zentrale Verwaltungskomponenten soweit möglich zu vermeiden. Außerdem sollen Microservices keine gemeinsamen Datenbanken verwenden. Wolff zufolge sind geteilte Datenbanken hingegen erlaubt, solange jeder Microservice seine eigenen Schemata darin hat.

2.2.1. Charakterisierung von Microservices und Microservice-Architekturen

Der Kern der genannten Kriterien ist, dass Microservices unabhängig voneinander ausgeliefert und geändert werden können. Dafür ist es erforderlich, dass verschiedene Microservices keine geteilte Datenhaltung betreiben, da gemeinsame Datenbanken oder Schemata Änderungsabhängigkeiten bedeuten würden. Microservices sollen außerdem heterogene Technologien verwenden können. Unsere Charakterisierung von Microservices lautet also:

> Ein Microservice ist ein funktionaler Teil einer Anwendung, der möglichst unabhängig von anderen Microservices geändert und ausgeliefert werden kann. Falls er eine Datenhaltung benötigt, ist diese Teil des Microservice. Microservices bieten technologieneutrale Schnittstellen. Der Code eines Microservice ist von der Größe so beschränkt, dass ein Team ihn entwickeln und eine einzelne Person ihn verstehen kann.

Eine Microservice-Architektur charakterisieren wir auf der Basis von Microservices:

> Bei einer Anwendung mit Microservice-Architektur ist der Funktionsumfang auf mehrere Microservices verteilt. Auf die Microservices wird ausschließlich über ihre Anwendungs-Programmierschnittstelle (API) zugegriffen.

2.2.2. Vor- und Nachteile von Microservices

Microservices sind verteilte Systeme. Newman [2015] betont, dass daher Auslieferung, Überwachung und Betrieb von Systemen mit Microservice-Architektur schwerer sind, als bei monolithischen Lösungen. Zudem sei das Testen schwerer und die verteilte Datenhaltung komplexer als in zentralen Datenbanken. Wolff [2016a] nennt als weiteren Nachteil, dass die Logik der Anwendung in den Beziehungen der Services untereinander versteckt sei und so der Überblick über die Gesamtarchitektur komplexer werde. Außerdem sei das verschieben von Funktionen zwischen Microservices schwierig und daher die anfängliche Wahl der Aufteilung des Systems wichtig. Er betont auch die Komplexität des Betriebs und

der Entwicklung durch die Kommunikation über Netzwerke. Wir zeigen in Abschnitt 5.1 auf Seite 61 außerdem, dass die Performance gegenüber monolithischen Systemen schlechter sein kann, obwohl der Ressourcenbedarf höher ist.
Monolithische Systeme werden mit wachsendem Funktionsumfang und durch Wartungen häufig größer und komplexer und die Modularisierungsgrenzen werden überschritten. Dadurch wird der Code schwerer zu verstehen und auch zu warten. Microservices bieten nach Wolff [2016a] eine starke Modularisierung und konstant geringe Größe, die die Wartbarkeit aufrecht erhalten. Einige weitere von ihm genannte Vorteile zählt auch Newman [2015] auf. Dazu gehört die Technologieunabhängigkeit und unabhängige Skalierbarkeit der Services. Außerdem lassen sich einzelne Microservices ausliefern, ohne dass das Gesamtsystem angehalten werden muss. So kann ein System im laufenden Betrieb aktualisiert werden. Durch eine eigene Datenhaltung und Zugriff exklusiv über die definierte Schnittstelle können Microservices unabhängig voneinander geändert und ausgeliefert werden. Teams, die sich mit der Entwicklung unterschiedlicher Microservices beschäftigen, sind dadurch weitestgehend unabhängig voneinander. Der Entwicklungsprozess skaliert so besser bei wachsender Projektgröße. Durch die Technologieunabhängigkeit kann jeder Microservice mit den optimalen Technologien implementiert werden. Falls eine effizientere Technologie gefunden wird, können Microservices zudem durch ihre Größe einfacher ersetzt werden. Newman [2015] nennt noch die Beständigkeit gegen Fehler einzelner Komponenten als weiteren Vorteil. Da in verteilten Systemen immer mit dem Fehlschlagen von Kommunikation gerechnet werden muss, sollten Microservices gegen diesen Fall abgesichert sein. So führt der Absturz einzelner Services nicht zu einem Versagen des Gesamtsystems.

2.2.3. Vertikalen

Kraus u. a. [2013] und Steinacker [2014] beschreiben den Einsatz von *Vertikalen* und Microservices im Umfeld der Online-Plattform des Versandhandels Otto.[1] Die Vertikalen charakterisieren sie als separat auslieferbare Anwendungskomponenten. Dafür sollen sie ein eigenes Frontend, eigene Datenhaltung und eigene Code-Basis haben. Die Kommunikation zwischen den Vertikalen geschieht über REST-Schnittstellen und es gibt keinen geteilten Zustand zwischen den Vertikalen. Innerhalb dieser Vertikalen kann es verschiedene Microservices geben. Diese Microservices sind ebenfalls unabhängig auslieferbar und so klein, dass sie von einem einzelnen Programmierer verstanden werden können. Bei Steinacker [2014] haben Microservices keine eigene Datenbank, damit sie leicht ersetzt, skaliert und ausgeliefert werden können. Die Microservices nutzen stattdessen die Datenhaltung ihrer Vertikalen. Es dürfen demnach auch mehrere Microservices innerhalb einer Vertikalen die selbe Datenbank nutzen. Die Vertikalen von Kraus u. a. [2013] entsprechen den Kriterien der anderen Quellen für Microservices. Die von Steinacker [2014] beschriebenen Microservices verstehen wir eher als Nanoservices (siehe Abschnitt 2.3).

[1] `https://www.otto.de/`

2.3. Nanoservices

Wolff [2016a] schreibt, dass manche Menschen unter Microservices extrem kleine Services mit unter 100 Zeilen Code verstünden. Für diese führt er den Begriff *Nanoservice* an. Im Gegensatz zu Microservices dürfen Nanoservices auf einer gemeinsamen virtuellen Maschine oder in einem Prozess laufen. Um das zu realisieren, erlaubt Wolff Einschränkungen der Technologiefreiheit gegenüber Microservices. Nanoservices enthalten aufgrund der geringen Größe keine eigene Datenbank und können sich daher Datenbanken teilen.

2.4. SOA

Newcomer und Lomow [2005] beschreiben SOA als organisatorisches Prinzip, mit dem Unternehmenslösungen realisiert werden können. Bei SOA werden Anwendungen als Service zur Verfügung gestellt. Dafür gibt es die Web Service Description Language (WSDL) zur Beschreibung von Service-Schnittstellen. Auf diese Weise können Arbeitsabläufe über Anwendungsgrenzen hinaus koordiniert werden. Die Verbindung der Services heißt Enterprise Service Bus (ESB). Dieser ist häufig ein anwendungsübergreifendes Netzwerk. Die Steuerung von Arbeitsabläufen erfolgt von zentraler Stelle. Da diese Steuerung der Anleitung eines Orchesters durch einen Dirigenten gleicht, nennt man den Vorgang auch Orchestrierung. Wir bezeichnen daran angelehnt die steuernde Stelle als *Dirigent*.
Als Unterschiede zwischen Microservice-Architekturen und SOA zählt Wolff [2016a] auf, dass SOA eine Unternehmensweite Struktur beschreibt, während Microservices die Architektur einer Anwendung bilden. Weiterhin sei der Fokus von SOA die Flexibilität bei der Nutzung der Services, welche durch Orchestrierung erreicht wird. Microservices hingegen haben als Ziel, Flexibilität in der Entwicklung erreichen. Dafür soll jeder Microservice einzeln ausgeliefert werden können, während SOA-Services monolithisch ausgeliefert werden dürfen.

2.5. OSGi

Die *OSGi Alliance* ist ein aus Großunternehmen bestehendes Industriekonsortium. Der Name OSGi kommt von *Open Services Gateway initiative*, der früheren Bezeichnung der OSGi Alliance. Mit OSGi bezeichnen wir die von der OSGi Alliance spezifizierte Plattform.
Die OSGi-Plattform ist ein auf Java [Gosling 2000] aufbauendes Framework zur Einbindung von dafür entwickelten Softwarekomponenten und deren Verwaltung. Kern der Spezifikation ist eine Definition von Programmierschnittstellen zur Verwendung und Konfiguration der Komponenten. OSGi-Komponenten werden *Bundles* genannt. Sie sind Java-Archive mit Einträgen im Manifest über den Namen, implementierte Funktionen, sowie Abhängigkeiten.
Implementierungen der OSGi-Plattform wie *Apache Felix* [Gédéon 2010] ermöglichen die separate Auslieferung von Bundles. Instanzen von Bundles können im laufenden Betrieb über die Verwaltungsschnittstelle installiert, aktualisiert oder entfernt werden.
Die OSGi-Plattform dient wie Microservices der Modularisierung von Software. Im Gegensatz zu Microservices werden die Bundles einer OSGi-Installation aber in einem Prozess, der Java Virtuelle Maschine (JVM), und nicht verteilt betrieben. Außerdem müssen OSGi-Bundles in Java implementiert sein.

2.6. Apache Stanbol

Apache Stanbol [Foundation 2010] ist eine Open-Source-Anwendung mit dem Ziel, Content-Management-Systeme (CMSs) um semantische Funktionen zu erweitern. Die Entwicklung wurde 2009 im Rahmen des Projekts Interactive Knowledge Stack (IKS) gestartet. Christ und Nagel [2011] stellen es als Referenzimplementierung für Semantische Content-Management-Systeme (SCMSs) vor. Seit 2012 ist Stanbol ein Apache-Projekt [Pereira 2013].

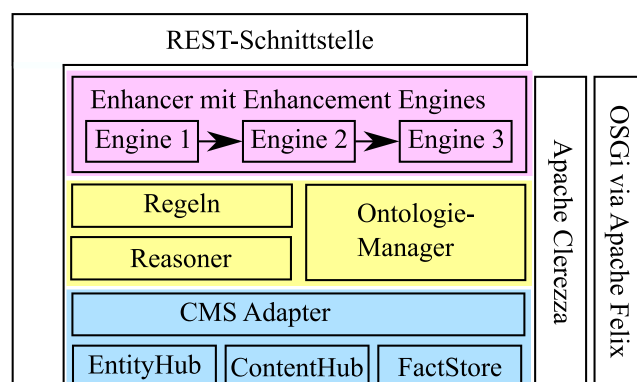


Abbildung 2.3. Architektur von Apache Stanbol nach Foundation [2010]

Abbildung 2.3 zeigt die Architektur von Stanbol. Alle Komponenten bieten eine REST-Schnittstelle und werden über die OSGi Implementierung *Apache Felix* [Gédéon 2010] eingebunden. *Apache Clerazza* [Foundation 2011] bietet Funktionen zur Verwaltung der RDF-Tripel [Manola u. a. 2004], durch welche die semantischen Informationen des *Enhancers* repräsentiert werden.
Der Enhancer beinhaltet eine Menge von *Enhancement-Engines*. Diese Enhancement-Engines werden pipelineartig zu *Enhancement-Chains* zusammengesetzt. Der Enhancer ruft die Elemente der aktiven Enhancement-Chain nacheinander auf, um den zu analysierenden Text mit syntaktischen oder semantischen Informationen anzureichen. Diese Enhancement-Engines implementieren Vorbereitungsphasen (siehe Abschnitt 3.1.3 auf Seite 19) oder Methoden der semantischen Analyse (siehe Abschnitt 3.1.4 auf Seite 24). Enhancement-Engines können angeben, dass sie parallele Ausführungen unterstützen. Stanbol kann mehrere Texte parallel analysieren, falls die Enhancement-Engines dies erlauben.
Der *Ontologie-Manager* verwaltet die Ontologien, die Metadaten zu den Texten speichern. Die *Regel*-Komponente bietet Regeln zum umorganisieren von semantischen Informationen. Diese Regeln können zum Beispiel verwendet werden, um die Informationen in ein anderes Format zu konvertieren. Der *Reasoner* kann anhand bekannter Regeln neues Wissen inferieren.
Der *FactStore* speichert die inferierten Beziehungen. Der *CMS Adapter* bietet CMS-kompatible Schnittstellen für den Zugriff auf die in Stanbol gespeicherten Daten. Der *ContentHub* persistiert Analysierte Texte mit den notwendigen Metadaten, um semantische Informationen zuordnen zu können.

2.7. HTTP

Das Hypertext Transfer Protocol [Fielding u. a. 1999] ist ein Protokoll zur Übertragung von Daten zwischen Computersystemen. Es wird im ISO/OSI-Referenzmodell [Zimmermann 1980] der Anwendungsschicht zugeordnet. HTTP ist zustandslos. Das heißt, zwischen verschiedenen Sitzungen werden keine Daten gespeichert.
HTTP-Nachrichten setzen sich aus einem Header und einem optionalen Body zusammen. Der Header beinhaltet Informationen über den Inhalt des Bodys, wie etwa die Kodierung des Inhalts. Bei den Nachrichten wird zwischen Anfrage und Antwort unterschieden. Jede Anfrage wird vom Empfänger mit einer Antwort erwidert. Bei den Anfragen gibt es verschiedene Methoden, wie zum Beispiel *PUT*, *POST* und *GET*. In jeder Anfrage wird der Name einer Methode spezifiziert. Diese Namen sind semantische Informationen über den Zweck der Anfrage. Eine PUT-Nachricht zum Beispiel dient zum Hochladen von Daten und eine GET-Nachricht fordert Daten vom Empfänger an.

2.8. REST

Fielding [2000] führt die Bezeichnung Representational State Transfer ein. Er beschreibt REST als eine Menge von architekturellen Bedingungen an Software-Systeme. Die Einhaltung der REST-Kriterien soll Skalierbarkeit von Interaktion zwischen Komponenten, Allgemeinheit der Interfaces und unabhängige Auslieferung der Komponenten bewirken. Dadurch können Interaktionslatenzen verringert, die Sicherheit des Systems verbessert und Altsysteme gekapselt werden. REST setzt eine Client-Server-Architektur voraus. Als Protokoll zur Kommunikation wird häufig HTTP eingesetzt.
Fowler [2010] unterscheidet verschiedene Level von REST:

Level 0: **Client-Server mit Webtechnologien**: Client und Server senden XML-, SOAP- oder XML-RPC-Nachrichten über HTTP. Es wird nur eine einzelne HTTP-Methode verwendet und der Server wird über eine einzelne Uniform Resource Identifier (URI) adressiert.

Level 1: **Ressourcen**: Es werden verschiedene URIs verwendet, um verschiedene Ressourcen anzusprechen.

Level 2: **HTTP-Methoden**: HTTP-Methoden werden möglichst entsprechend ihrer semantischen Bedeutung verwendet.

Level 3: **Hypermedia-Navigation** Daten zu Ressourcen enthalten Informationen (zum Beispiel URIs), die Operationen auf den Daten ermöglichen.

Kapitel 3

Eine Microservice-Architektur für die Semantische Analyse

In diesem Kapitel präsentieren wir unseren Entwurf einer Microservice-Architektur für die semantische Analyse im Kontext der Computerlinguistik. Dafür erläutern wir in Abschnitt 3.1 zwei Möglichkeiten, den Vorgang der Textanalyse in Microservices aufzuteilen. Das erste Aufteilungskriterium ist die Kategorie der Semantik und das zweite ist durch die Methoden der semantischen Analyse definiert. Für unseren Architekturentwurf entscheiden wir uns für die zweite Möglichkeit.
In Abschnitt 3.2 gehen wir auf Herausforderungen bei Microservice-Architekturen im Allgemeinen ein und wägen Lösungsalternativen im speziellen Umfeld der semantischen Analyse ab. Dazu gehören neben Kommunikationsarten, wie die von uns gewählte asynchrone *One-To-One*-Kommunikation mit textbasierten Nachrichtenformaten, auch Koordinationsmuster. Dabei entscheiden wir uns für asynchrone direkte Aufrufe. Ein weiteres Thema ist die Registrierung und Entdeckung von Services. Hierbei wählen wir die Muster *Selbstregistrierung* und *Server-seitige Entdeckung*. Des Weiteren erläutern wir, warum eine Grafische Benutzeroberfläche (GUI) in unserem Anwendungsfall optional ist und dass die Nachteile durch *Eventual Consistency* so gering sind, dass skalierbare Datenbanken verwendet werden können. Ein weiterer Punkt ist die Handhabung von geteiltem Code, welchen wir zwischen Microservices nur erlauben, wenn dadurch keine Abhängigkeiten bei der Auslieferung entstehen. Als Beispiel dafür führen wir Maßnahmen zur Ausfallsicherheit an, die wie eine externe Bibliothek behandelt werden sollen.
Abschnitt 3.3 präsentiert die aus diesen Entscheidungen resultierende Architektur. Wir zeigen die Gesamtarchitektur und den internen Aufbau der Microservices und erläutern das Verhalten zur Laufzeit an einem Beispiel.

3.1. Partitionierung von Computerlinguistik-Anwendungen

In diesem Abschnitt definieren wir zunächst den Begriff Partitionierung und verdeutlichen ihre Bedeutung in Microservice-Architekturen. Anschließend wägen wir die Integrationsmöglichkeiten für notwendige Vorbereitungen der semantischen Analyse ab und zeigen zwei verschiedene Partitionierungsmöglichkeiten der semantischen Analyse.

3.1.1. Definition von Partitionierung

Im Rahmen dieser Arbeit definieren wir *Partition* und *Partitionierung* wie folgt:

> Partitionen sind disjunkte Teilmengen der Funktionen eines Systems. Die Menge der Partitionen heißt Partitionierung.

Microservices sind die technische Realisierungen von Partitionen. Für jede Partition gibt es genau einen Microservice, der verschiedene Instanzen haben kann.
Microservice-Systeme müssen nicht komplett aus Microservices bestehen. Es kann zum Beispiel eine weitere Schicht geben, die die Funktionalität des Systems nach außen repräsentiert. Kraus u. a. [2013] stellen zudem eine Datenintegrations-Schicht vor. Ein weiteres Beispiel kann das Service-Register sein, welches wir in Abschnitt 3.2.2 auf Seite 32 erläutern. Es kann also Komponenten geben, die keiner Partition zugeordnet sind. Die Partitionierung schließt diese Komponenten nicht mit ein.

3.1.2. Bedeutung der Partitionierung in Microservice-Architekturen

Wolff [2016a] schreibt, dass die Qualität eines Microservice-Systems stark davon abhängt, wie die Funktionalität auf die Services aufgeteilt ist.
Als Beispiel dafür führt er an, dass eine zu feine Aufteilung einen größeren Kommunikations-Overhead bedeutet, da Microservices Daten austauschen oder sich gegenseitig aufrufen. Außerdem wachsen Anforderungen an die Infrastruktur, da mehr einzelne Komponenten verwaltet und betrieben werden müssen. Wie Richardson [2016a] beschreibt werden Maschinen häufig nicht ideal ausgelastet, sodass eine größere Anzahl an Maschinen zu mehr nicht genutzter Rechenkapazität und somit auch zu höheren Kosten führt. Automatisches Clustermanagement kann die Ausnutzung der Maschinen verbessern, sodass die Auswirkungen dieses Nachteils geringer werden. Mit sinkender Größe von Microservices und somit weitläufigerer Verteilung von Daten werden Transaktionen auf den Daten auch zunehmend schwerer. Richardson [2015a] beschreibt als Lösungsansätze für verteilte Transaktionen, wie die Datenbank als Nachrichtenwarteschlange verwendet werden, ein Transaktionsprotokoll genutzt oder *Event Sourcing* [Fowler 2005] angewendet werden kann. Diese Ansätze führen zu einem größerem Organisationsaufwand als es in monolithischen Systemen der Fall wäre. Zudem sinkt die Konsistenz der Daten häufig durch die Verteilung, da Daten Änderungen erfahren können, während andere Services zeitgleich mit älteren Kopien der Daten arbeiten.
Andererseits führt eine zu grobe Partitionierung dazu, dass die Vorteile des Microservice-Stils nicht mehr zur Geltung kommen. Wolff [2016b] beschreibt, dass jeder Microservice von einem Team entwickelt werden sollte. Wenn ein Microservice zu viel Funktionalität enthält, wird die Entwicklung für wenige Personen aber zu langwierig. Größere Teams hingegen steigern den organisatorischen Aufwand und widersprechen dem von Wolff [2016a, Seite 38] empfohlenem Prinzip, dass es für jeden Service möglich sein sollte, von einer einzelnen Person verstanden zu werden. Ein weiterer Nachteil von zu grob partitionierten

Microservices ist, dass die Ersetzbarkeit leidet, da mehr Code neu geschrieben werden muss. Newman [2015, Seite 7] zählt Ersetzbarkeit aber aufgrund der geringen Größe und der losen Kopplung zu den Vorteilen von Microservices.
Wie Wolff [2016a, Seite 14] beschreibt, ist es schwer ist, Funktionalitäten zwischen Microservices zu verschieben. In monolithischen Anwendungen können einzelne Komponenten meist einfach und durch eine Integrierte Entwicklungsumgebung (IDE) unterstützt in ein anderes Modul verschoben werden. In Microservice-Architekturen hingegen betrifft die Änderung alle Schichten der Microservices. Die Funktionalität muss aus dem Interface, der Logik und der Datenhaltung des einen Service entfernt und in einen anderen integriert werden. Falls die Microservices verschiedene Technologien verwenden, ist eine Neuimplementierung der Funktion erforderlich. Nach dieser Umstrukturierung sind alle Services und andere Komponenten, die diese Funktionalität nutzen, zu aktualisieren und die Auslieferung für mehrere Microservices zu koordinieren. Insgesamt entsteht ein hoher organisatorischer Aufwand, der durch die Verwendung einer Microservice-Architektur eigentlich vermieden werden soll.
Die Partitionierung ist also von großer Bedeutung für die Effizienz des Systems und der Entwicklung und kann zu späteren Zeitpunkten nur unter größerem Aufwand geändert werden.

3.1.3. Integration der Vorbereitungsphasen

In Abschnitt 2.1 beschreiben wir die Aufteilung der Textanalyse in Tokenisierung, lexikalische Analyse, syntaktische Analyse und semantische Analyse. Wie in Abbildung 2.1 ersichtlich sind diese Phasen in einer festen Reihenfolge durchzuführen. Die Phasen vor der semantischen Analyse bezeichnen wir im Folgenden auch als Vorbereitungsphasen. Wir betrachten die in Abbildung 3.1 gezeigten Möglichkeiten, wie diese in das System integriert werden können.

Integration als Bibliothek

Die in Abbildung 3.1a gezeigte Möglichkeit ist, die Phasen in eine Bibliothek zusammenzufassen. Diese kann dann in die Microservices integriert werden, die sie benötigen. Dadurch entsteht kein zusätzlicher Kommunikationsaufwand über das Netzwerk, die Programmierung von zusätzlichen APIs entfällt und die Fehleranfälligkeit auf Kommunikationsebene sinkt. Die Bibliothek könnte von den Entwicklern der Microservices für die semantische Analyse produziert werden, da aber die Vorbereitungsphasen nach Palmer [2010] komplex sein können, empfiehlt sich eher ein eigenes Team dafür. Außerdem gibt es noch weitere Nachteile von geteiltem Code, die wir in Abschnitt 3.2.6 vorstellen. Hinzu kommt, dass die Bibliothek nur für einen Teil der Services verwendbar ist, wenn Microservices inkompatible Technologien einsetzen. Dann müssten redundante Bibliotheken für die Technologien produziert werden. Dadurch sind diese schwerer zu warten und bedeuten einen höheren

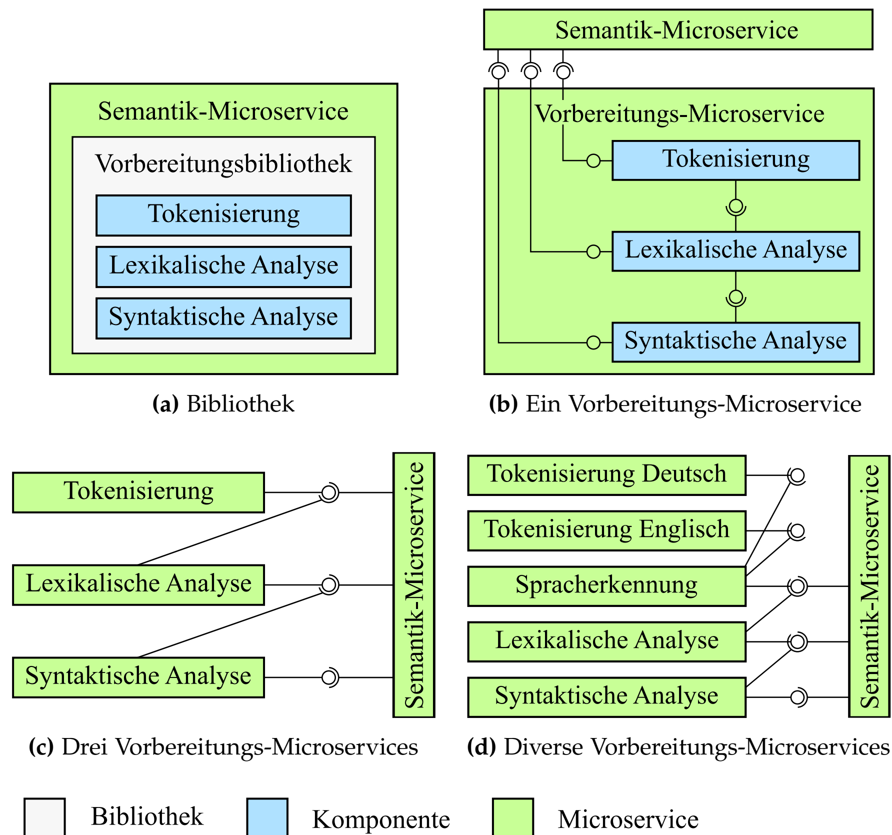


Abbildung 3.1. Möglichkeiten der Integration der Vorbereitungsphasen

Entwicklungsaufwand. Zudem ist die Programmiersprache der Bibliothek durch die sie nutzenden Microservices festgelegt und könnte dadurch nicht optimal sein. Diese Gründe führen dazu, dass wir von der Nutzung der Vorbereitungsphasen als Bibliothek abraten.

Integration als ein Microservice

Abbildung 3.1b zeigt, wie die Phasen in einem Vorbereitungs-Microservice zusammengefasst sind. In diesem Fall müsste ein zu analysierender Text zuerst einer Instanz des Vorbereitungs-Microservice zugeführt werden und anschließend an die gewünschten Microservices der semantischen Analyse weitergeleitet werden. Das erfordert einen höheren Organisationsaufwand und zusätzliche Ressourcen, da mehr Microservices beteiligt sind. Zudem kann die Kommunikation des Vorbereitungs-Microservice mit denen der semantischen Analyse über das Netzwerk erfolgen, was zu Einbußen in der Performance führt. Ein

weiterer Nachteil ist, dass eine Microservice-konforme API-Schicht bereitgestellt werden muss. Im Gegenzug können die Vorbereitungsphasen unabhängig von der semantischen Analyse entwickelt und ausgeliefert werden. Änderungen an den Vorbereitungsphasen können so in Betrieb genommen werden, ohne dass die sie nutzenden Microservices ebenfalls aktualisiert werden müssen. Diese Lösung bietet sich an, wenn die Vorbereitungsphasen in ihrer Komplexität so geartet sind, dass der Vorbereitungs-Microservice noch den Anforderungen an die Größen einer Partition entspricht. Wenn die einzelnen Vorbereitungsphasen zu komplex werden, sodass die Nachteile einer zu groben Partitionierung wie in Abschnitt 3.1.2 beschrieben auftreten, kann eine feinere Aufteilung gewählt werden.

Integration als drei Microservices

Bei der Integration als drei Microservices bildet jede der Vorbereitungsphasen einen eigenen Microservice, wie in Abbildung 3.1c ersichtlich. Das verstärkt die Nachteile der Microservice-Lösung, da es mehr Microservices gibt, die koordiniert und betrieben werden müssen. Andererseits können die Vorbereitungsphasen nach Palmer [2010] so aufwändig sein, dass die Anforderung, eine Person soll den gesamten Service verstehen können, sonst nicht eingehalten werden könnte. Zudem besteht eine klare funktionale Trennung der Phasen, sodass eine Partitionierung nach den Funktionen nahe liegt.

Integration als diverse Microservies

In der letzten Lösung, die in Abbildung 3.1d dargestellt ist, gehen wir darauf ein, dass sich der Aufbau von Sprachen stark unterscheidet. Durch die Unterschiede ist es möglich, dass die Umsetzungen der Vorbereitungsphasen in Abhängigkeit von der Sprache verschieden sein können. Diese Lösung unterteilt daher die Microservices der Vorbereitungsphasen weiter in Versionen für die verschiedenen Sprachen. Durch eigene Microservices pro Sprache erreichen wir Technologiefreiheit für die einzelnen Implementierungen. Wenn also die Umsetzung der Tokenisierung zum Beispiel für deutsche Texte in einer Skriptsprache und für Chinesisch durch Programmiersprachen mit Patternmatching effizienter ist, kann so für jede Sprache eine geeignete Technologie gewählt werden. Diese Aufteilung führt dazu, dass die Sprache eines Textes bekannt sein muss. Dafür kann ein weiterer Microservice verwendet werden, den wir *Spracherkennung* nennen. Bei dieser Lösung entstehen mehrere Services für einzelne Vorbereitungsphasen und ein neuer Service, der sich den bisher verwendeten Phasen nicht zuordnen lässt, wodurch die Aufteilung zwischen Tokenisierung, semantischer Analyse und lexikalischer Analyse weniger streng wird. Dafür sinkt aber die Komplexität pro Microservice weiter und das System kann anhand der Sprachen in Z-Richtung skaliert werden (siehe Abschnitt 3.2.7 auf Seite 44).

Weitere Vorbereitungs-Microservices

In der Praxis bieten existierende Anwendungen zur Analyse natürlichsprachlicher Texte häufig zusätzliche Module wie Werkzeuge zur Textextraktion aus verschiedenen Dokumenttypen. Diese lassen sich wie die Spracherkennung in der letzten Lösung nicht in die Vorbereitungsphasen nach Dale [2010] einsortieren. Sie sollten auf die gleiche Weise integriert werden wie die Module der anderen Phasen, um eindeutige Konventionen zu haben.
Einige Anwendungen bieten verschiedene Implementierungen der Phasen an. Apache Stanbol beinhaltet drei verschiedene Tokenisierungsoptionen: *OpenNLP Tokenizer Detection Engine*, *Smartcn Tokenizer Engine* und *Paoding Tokenizer Engine*.[1] Diese könnten in einem Microservice zusammgenfasst werden oder in eigene Services integriert werden. Bei der Integration in mehreren Microservices lässt sich eine abgewandelte Version der letzten Integrationsmöglichkeit der Vorbereitungsphasen verwenden. Die Phasen werden dann nicht nach Sprache aufgeteilt, sondern nach Implementierung und jeder aufrufende Microservice kann eine kompatible Version wählen. Eine Kombination der Aufteilung nach Implementierung und Sprache ist auch möglich, wenn die Komplexität der Implementierungen es erfordert.

Abwägung der Integrationsmöglichkeiten der Vorbereitungsphasen

Welche der Möglichkeiten der Integration der Vorbereitungsphasen am besten ist, hängt davon ab, wie komplex die einzelnen Phasen sind, ob die Technologien für alle Phasen gleich sind und ob sie selbst entwickelt werden. Eine allgemeingültige Empfehlung lässt sich daher nicht geben. Für unseren Architekturentwurf entscheiden wir uns aber für die Lösung aus Abbildung 3.1d mit den meisten Microservices, da sie am flexibelsten ist und so die Skalierung des Entwicklungsprozesses und das Ergänzen von Funktionen am besten unterstützt.

3.1.4. Partitionierung der Semantischen Analyse

Richardson [2014] stellt als Verfahren zur Aufteilung von Systemen Partitionierung nach Verb oder Anwendungsfall, Nomen oder Ressource und das Single Responsibility Principle (SRP) vor. Bei der Partitionierung nach Verb oder Anwendungsfall wird die Funktionalität nach Anwendungsfällen aufgeteilt. Ein Online-Shop könnte zum Beispiel die Anwendungsfälle *Anmelden*, *Bestellen*, und *Artikel suchen* als Grundlage für Partitionen verwenden. Im gleichen Beispiel entsprächen *Benutzer*, *Artikel* und *Bestellungen* Nomen oder Ressourcen. Nach dem SRP wäre die Unterteilung feiner. Es würden Partitionen für *Benutzer anmelden*, *Benutzerdaten ändern* und *neuen Benutzer anlegen* existieren.

[1] http://stanbol.apache.org/docs/trunk/components/enhancer/engines/list

Wir nutzen die Partitionierung nach Nomen oder Ressource zunächst, indem wir die semantische Analyse nach Kategorie der Semantik partitionieren. Anschließend zeigen wir die Aufteilung nach Methoden der semantischen Analyse als Umsetzung der Partitionierung nach Verb oder Anwendungsfall.

Partitionierung nach Kategorien der Semantik

Goddard und Schalley [2010] schreiben, dass die semantische Analyse bisher nicht so weit erforscht sei und nicht so intensiv betrieben werde wie die syntaktische Analyse. Daher sei eine allgemein gültige feinere Unterteilung in diesem Bereich noch nicht möglich. Es ist aber möglich, Semantik zu kategorisieren und anhand dessen eine Partitionierung zu erstellen. Die Kategorien sind, wie in Abschnitt 2.1.2 auf Seite 8 erklärt, lexikalische Semantik (Wortsemantik) und supralexikalische Semantik (Satzsemantik). Zusätzlich führen wir die Kategorie *Inferenz* (Schlussfolgerungen) ein. Sie beinhaltet Regeln zur Schlussfolgerung neuer Fakten. Diese Unterteilung setzt die in Abschnitt 3.1.3 beschriebene thematische Verfeinerung der Syntax auf die drei Vorbereitungsphasen fort. Die resultierende Partitionierung besteht aus den folgenden Partitionen:

▻ Supralexikalische Semantik

▻ Lexikalische Semantik

▻ Inferenz

Jede Partition ist für alle Daten ihrer Kategorie der Semantik zuständig. Da sich diese Aufteilung auf die produzierten und verwendeten Ressourcen bezieht, ist es eine Partitionierung nach Nomen oder Ressource. Für jede Kategorie von Semantik kann eine geeignete Repräsentationsform gewählt werden. Durch diese Aufteilung wird die Datenhaltung klar abgegrenzt. Die Anwendungslogik, also die Methoden der semantischen Analyse, werden ebenfalls in eine der Kategorien einsortiert und in den entsprechenden Microservice integriert.
Im Bezug auf die Datenhaltung sind die Grenzen der Partitionen so klar festgelegt. Goddard und Schalley [2010] beschreiben jedoch, dass sich lexikalische und supralexikalische Semantik auf viele Weisen beeinflussen und überschneiden, sodass viele Linguisten die strikte Trennung ablehnen. Durch diese Überschneidungen lassen sich auch Methoden der semantischen Analyse häufig nicht klar in eine der Kategorien eingliedern: Manche verwenden Daten aus mehreren Kategorien und erzeugen auch Daten mehrerer Kategorien. Diese Methoden einer der Kategorien zuordnen zu müssen, verwischt die Grenzen zwischen den Partitionen auf funktionaler Ebene. Der Bedarf dieser Methoden nach Daten anderer Partitionen führt zudem zu starken Abhängigkeiten zwischen den Microservices. Diese Abhängigkeiten sollen aber eigentlich vermieden werden, um die einzelnen Microservices möglichst autonom zu halten.

Partitionierung nach Methode der Semantischen Analyse

Gangemi [2013] haben zur Evaluierung von Anwendungen im Bereich der semantischen Analyse Funktionen von existierenden Anwendungen ermittelt. Diese Funktionen nennen wir *Methoden der semantischen Analyse*. Wir erklären sie am folgenden Beispielsatz:

> Eberhard Wolff schrieb 2016 das Buch „Microservices - Grundlagen flexibler Softwarearchitekturen".

- ▷ **Thema-Extraktion** [Berry und Castellanos 2007] dient der Erkennung des Themas des Texts. Dies kann aus einer Textüberschrift oder dem Text selbst extrahiert werden. Als Thema des Beispieltextes kann *Autor des Buches „Microservices - Grundlagen flexibler Softwarearchitekturen"* ermittelt werden.
- ▷ **Individuenerkennung** [Nadeau und Sekine 2007] ist die Identifizierung von Nennungen von Individuen. *Eberhard Wolff* und das Buch *„Microservices - Grundlagen flexibler Softwarearchitekturen"* sind Individuen im Beispieltext. *Buch* hingegen ist kein Individuum sondern ein Oberbegriff.
- ▷ **Individuenauflösung** [Bhattacharya und Getoor 2007] ist die Zuordnung von Individuen zu einer Repräsentation. Im Beispiel wird *Eberhard Wolff* einem eindeutigen Exemplar des Oberbegriffs Mensch zugeordnet, das zum Beispiel durch eine URI [Masinter u. a. 2005] beschrieben wird.
- ▷ **Terminologie-Extraktion** [Hartmann u. a. 2012] dient der Identifikation domänentypischer Begriffe wie *Microservices* und *Softwarearchitektur*. Dies kann verwendet werden, um die Abbildung dieser Begriffe in Ontologien [Hesse u. a. 2004] zu verbessern.
- ▷ **Bedeutungsmarkierung** [Ciaramita und Altun 2006] löst Mehrdeutigkeiten wie Synonymie, Hyponymie und Meronymie auf. So wird im Beispiel erkannt, dass *Buch* in diesem Text *Druckwerk* bedeutet und nicht eine *Wettliste beim Pferderennen* und dass es Synonyme wie *Band* und *Werk* gibt.
- ▷ **Taxonomie-Induktion** [Ponzetto und Strube 2011] schlussfolgert Unterkategorie-Beziehungen zwischen Objekten. Aus dem Beispieltext geht zum Beispiel hervor, dass *Microservices-Architektur* eine Unterkategorie von *Softwarearchitektur* ist.
- ▷ **Beziehungsextraktion** [Ciaramita u. a. 2005] ist die Beziehungserkennung zwischen Individuen. Diese Methode würde im Beispieltext erkennen, dass die Beziehung *schreiben* zwischen den Individuen *Eberhard Wolff* und *Buch "Microservices - Grundlagen flexibler Softwarearchitekturen"* besteht.
- ▷ **Markierung semantischer Rollen** [Moschitti u. a. 2008] ordnet Wörtern Rollen im Kontext des Satzes zu. So ist *Eberhard Wolff* der Akteur für die Verbform *schrieb* und *das Buch "Microservices - Grundlagen flexibler Softwarearchitekturen"* ist ein zugehöriges Argument.

- **Ereigniserkennung** [Hogenboom u. a. 2011] ermittelt im Text genannte Ereignisse. Dazu können Ort, Zeitpunkt und eine Beschreibung gehören. Der Beispielsatz beschreibt ein Ereignis mit unbekanntem Ort, dem Zeitpunkt 2016 und der Beschreibung *Eberhard Wolff schreibt Buch über Microservices*.
- **Rahmenerkennung** [Coppola u. a. 2009] erkennt Bedeutungskontexte. Diese bieten Domänenspezifische Rollenmodelle. Ein Rahmen im Beispieltext ist *Schriftstellerei* mit den Rollen *Autor* und *Werk*.

Die Partitionierung nach diesen Methoden ist in Abbildung 3.2 abgebildet. Für jede Methode der semantischen Analyse gibt es eine Partition. Im Vergleich zu der Partitionierung nach Kategorien der Semantik werden deutlich mehr Microservices gebildet. Dies führt zu einem höherem Verwaltungsaufwand, da mehr Komponenten konfiguriert, getestet, ausgeliefert, betrieben und überwacht werden müssen. Diese Aufteilung entspricht aber eher dem Microservice-Stil nach Wolff [2016a]: Es gibt kleine Einheiten, die eine klar definierte Aufgabe im Gesamtsystem haben.

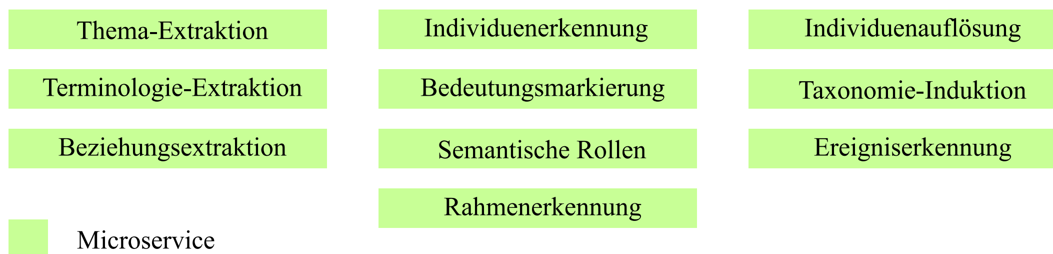


Abbildung 3.2. Partitionierung nach Methode der semantischen Analyse

Diese Partitionierung ist deutlich feiner und dadurch sind auch die resultierenden Microservices kleiner. Das macht sie weniger komplex und erlaubt kleinere Teams, die sich auf genau eine Aufgabe spezialisieren können. Bei kleineren Teams gibt es auch weniger Personen, mit denen Änderungen abgesprochen werden müssen, wodurch der Koordinationsaufwand sinkt. Zudem sind die Microservices eher für eine einzelne Person komplett zu erfassen, sodass Änderungsvorschläge schneller verständlich sind, da jeder beteiligte Entwickler die entsprechenden Stellen gut kennt.
Wenn es verschiedene Funktionalitäten gibt, die auf der gleichen Datenhaltung arbeiten, ist es schwerer, Änderungen an den Daten zurückzuverfolgen. Da bei dieser Partitionierung jeder Microservice genau eine Aufgabe hat, ist also die Beobachtbarkeit der einzelnen Services besser als bei der Aufteilung nach Art der Semantik. So können Fehlerursachen schneller identifiziert und behoben werden. Das Beobachten des Gesamtsystems wird zugleich komplexer, da es mehr Einheiten gibt, die für eine Überwachung konfiguriert und betrieben werden muss.

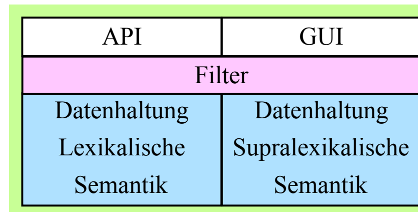


Abbildung 3.3. Interner Aufbau der Microservices bei Partitionierung nach Methode

Der interne Aufbau der Microservices ist in Abbildung 3.3 skizziert. Die oberste Schicht beinhaltet eine API für andere Services und eine optionale GUI, die das Testen dieses Services erlaubt. Vor- und Nachteile der Integration einer GUI in die Microservices erläutern wir in Abschnitt 3.2.4 auf Seite 40. Die darunterliegende Schicht zeigt, wie vorhandene Pipeline-Strukturen bei der Migration von Altsystemen genutzt werden. Die für eine Funktion benötigten Filter werden extrahiert und mit der Datenhaltung in einen Microservice integriert. Die Pipeline muss dann aus verschiedenen Microservices zusammengesetzt werden. Möglichkeiten, diese Zusammensetzung zu koordinieren, erläutern wir in Abschnitt 3.2.3 auf Seite 35.
Bei dieser Partitionierung ist die Haltung von Daten mehrerer semantischer Kategorien in einem Service erlaubt. Dadurch wird die Kritik von Goddard und Schalley [2010] an der strikten Trennung von lexikalischer und supralexikalischer Semantik berücksichtigt und jeder Microservice kann genau die Daten halten, die er benötigt. Dies führt dazu, dass Daten einer semantischen Kategorie auf verschiedene Microservices verteilt werden. Die Unterteilung, welcher Microservice für welche Daten zuständig ist, erfolgt nun anhand der Methode der semantischen Analyse. Jeder Microservice ist der Besitzer aller Daten, für die er zuständig ist. Das bedeutet, er legt das Datenschema fest und andere Services dürfen diese Daten nur replizieren. Außerdem sind die Daten eines Microservices für einen anderen nur durch eine Programmierschnittstelle erreichbar. Direkte Zugriffe auf die Datenbanken anderer Services sind nicht erlaubt. Dadurch wird gewährleistet, dass Änderungen an der Repräsentation der Daten einer Methode nicht dazu führen, dass andere Services ebenfalls geändert werden müssen. Zudem kann für jeden Microservice eine optimale Repräsentationsform gewählt werden.
Ein Nachteil dieser Aufteilung ist, dass mitunter mehrere Datenbanken pro Microservice gehalten werden. Das macht die Skalierung einzelner Services schwieriger, da mehrere Datenbanken mit skaliert werden müssen. Falls mehrere Microservices die gleiche Technologie zur Repräsentation der Daten nutzen, müssen außerdem so mehr Instanzen der gleichen Technologie betrieben werden. Das führt zu höherem Aufwand für den Betrieb. Falls jedes Team für den Betrieb seines Microservice zuständig ist, müssen mehr Personen mit dem Betrieb und der Programmierung der Technologie vertraut sind, als es bei der Partitionierung nach Art der Semantik der Fall wäre.
Die Partitionierung nach Methode der semantischen Analyse und die dafür nötige Anpassung der Datenhaltung führen dazu, dass die Abhängigkeiten zwischen den Partitionen

geringer sind als bei der Aufteilung nach Art der Semantik. Die Microservices können weitestgehend eigenständig entwickelt werden. Wenn eine neue Funktion hinzu kommt, muss sie nicht anhand der semantischen Kategorie in einen vorhandenen Microservice integriert werden, sondern kann von einem neuen Team unabhängig entwickelt werden. Dadurch skaliert der Prozess der Entwicklung besser: Der Anstieg des Kommunikationsbedarfs bei wachsender Funktionalität ist wesentlich geringer, als es bei einem Monolithen oder der anderen Partitionierung wäre.

Strategic Design und Bounded Contexts

Wolff [2016a] schlägt für die Partitionierung *Strategic Design* als Teil des Domain-Driven Design (DDD) vor. *Bounded Contexts* (Begrenzte Kontexte) sagt als zentrales Konzept des Strategic Design aus, dass jedes Modell eines Datums nur in gewissen Grenzen eines Systems sinnvoll ist. Für die Rechnungsabwicklung eines Online-Shops sind zum Beispiel andere Daten eines Kunden erforderlich als für Produktempfehlungen. Beim Strategic Design werden dann zwei verschiedene Modelle einem allgemeingültigen vorgezogen.
Methoden der semantischen Analyse können unterschiedliche Modelle eines Texts erfordern. Ein Modell kann zum Beispiel der durch die Rahmenerkennung erkannte Rahmen sein. Diese Modelle bilden dann nach den Bounded Contexts die Kontexte, anhand derer eine Partitionierung erstellt wird. Die Kontexte in Abbildung 3.4 sind Vorbereitung, Rahmen und Beziehungen. Alle Methoden der semantischen Analyse, die das gleiche Modell erfordern, sind in einem gemeinsamen Bounded Context.
Wenn jede Methode der semantischen Analyse ein eigenes Datenmodell benötigt, gibt es auch für jede Methode einen eigenen Kontext. In diesem Fall gleicht die Partitionierung nach Bounded Contexts der nach Methode der semantischen Analyse. Die Partitionierung nach Art der Semantik schließt alle Modelle, die für eine Kategorie der Semantik benötigt werden, in eine Partition ein. Dies widerspricht laut Wolff [2016a] den Bounded Contexts.
Abbildung 3.4 zeigt beispielhaft, wie die Beziehungen der Modelle untereinander im Strategic Design benannt sind. Die Vorbereitungsphasen werden hier zur Vereinfachung wie ein Bounded Context mit einem einheitlichen Modell behandelt. Die Methode Individuenerkennung benötigt das Modell der Methode Rahmenerkennung, daher teilen sich diese den Bounded Context *Rahmen*, sind aber weiterhin eigenständige Microservices. Dass die Vorbereitungsphasen ihr Modell dem Kontext *Rahmen* zur Verfügung stellen, heißt *CUSTOMER/SUPPLIER*-Beziehung. Die Beziehungsextraktion hat den Kontext *Beziehungen* und nutzt ebenfalls das Modell der Vorbereitungsphasen nach dem CUSTOMER/SUPPLIER-Schema. Das Modell der Vorbereitungsphasen bildet einen gemeinsamen Kern der Kontexte *Rahmen* und *Beziehungen*. Das nennt man dann einen *SHARED KERNEL*.

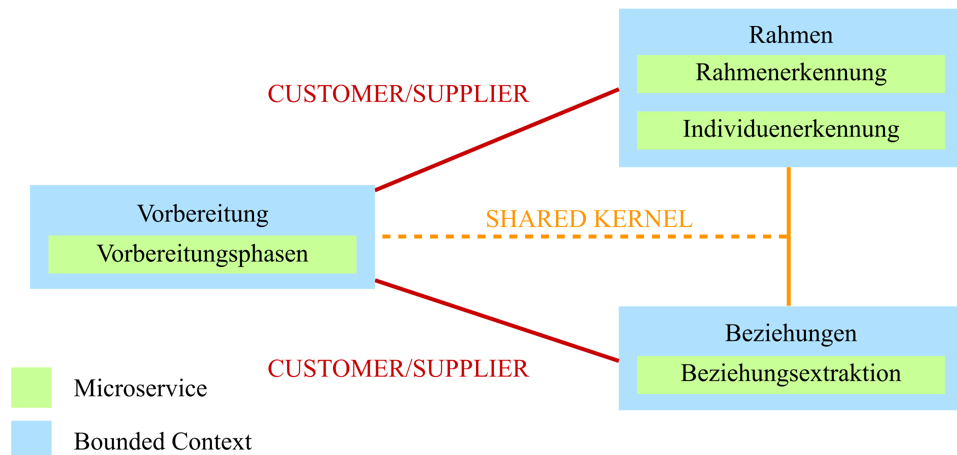


Abbildung 3.4. Beziehungen von Bounded Contexts untereinander in der semantischen Analyse

Abwägung der Partitionierungsmethoden

Welche Partitionierungsmethode besser ist, hängt von verschiedenen Faktoren ab. Ein Faktor ist, ob ein neues System geschaffen oder ein altes migriert werden soll. Bei der Migration ist ein weiterer Faktor, mit welcher Aufteilung vorhandene Strukturen am besten genutzt werden. Es kann schwer sein, die Datenabhängigkeiten nach der Art der Semantik aufzuteilen. Bei einem neuen System lässt sich diese Trennung vor Beginn der Implementierung einführen und somit leichter einhalten.
Ein weiterer Faktor ist die Komplexität der Methoden der semantischen Analyse. Wenn man sich für die Partitionierung nach Art der Semantik entscheidet, wählt man größere Microservices. Falls die Services zu groß werden sind die Vorteile durch die Microservice-Architektur abgeschwächt, sodass die Partitionierung nach Methode der semantischen Analyse ratsamer wäre.
Insgesamt bietet die Partitionierung nach Methoden der semantischen Analyse in den meisten Fällen mehr Vorteile als die Aufteilung nach Art der Semantik: Die Microservices haben eine angemessenere Größe, der agile Prozess skaliert besser und die Aufteilung der Funktionen ist nicht künstlich erzwungen. Wir entscheiden uns daher bei unserem Architekturentwurf in Abschnitt 3.3 ab Seite 47 für die Partitionierung nach Methoden der semantischen Analyse.

3.2. Weitere Aspekte von Microservice-Architekturen

In diesem Abschnitt des Kapitels befassen wir uns mit Architekturentscheidungen abseits der Partitionierungen. Wir nennen verschiedene Aspekte, die es zu berücksichtigen gilt und wägen alternative Lösungsansätze für die einzelnen Probleme ab. Hierbei orientieren wir uns an den von Richardson [2015c], Kraus u. a. [2013], Wolff [2016a] und Newman [2015] genannten Thematiken. Wir nennen Vor- und Nachteile im Bereich der semantischen Analyse und treffen darauf basierend Entscheidungen für unseren Architekturentwurf.

3.2.1. Kommunikation

Wolff [2016a] fordert, dass Microservices möglichst autonom sind. Das bringt verschiedene Vorteile: Sie sind einfacher zu testen, das System wird unanfälliger für Fehler und die Services sind leichter auszuliefern. Die semantische Analyse basiert aber auf anderen Phasen, die wie in Abschnitt 3.1.3 beschrieben ebenfalls als Microservices realisiert werden können. Dadurch sind Microservices der semantischen Analyse nicht komplett autonom. Zudem basieren manche Methoden der semantischen Analyse auf anderen: Der Individuenauflösungs-Service könnte zum Beispiel den Individuenerkennungs-Service benötigen. Um diese Abhängigkeiten zu erfüllen, muss einerseits die Kommunikation und andererseits die Koordination der Services organisiert werden.
In diesem Unterabschnitt stellen wir zunächst einige Kommunikationsformen vor, beschäftigen uns dann mit dem Format, in dem Nachrichten übertragen werden und beschreiben zum Schluss Möglichkeiten, die Schnittstellen von Microservices zu definieren.

Kommunikationsarten

Die Kommunikationsarten lassen sich auf zwei Arten einordnen. Das erste Kriterium ist, ob die Kommunikation synchron oder asynchron stattfindet. Bei synchroner Kommunikation wartet der sendende Prozess immer auf eine Antwort des Empfängers und blockiert so lange. Bei asynchroner Kommunikation wird nicht auf eine Antwort gewartet. Bei asynchroner Kommunikation müssen Antworten besonders gekennzeichnet werden, um sie der ursprünglichen Anfrage zuzuordnen.
Die zweite Art der Unterteilung ist, ob es einen oder mehrere Empfänger gibt. Diese Kommunikationsformen werden *One-to-One* und *One-to-Many* genannt. Tabelle 3.1 ordnet verschiedene Kommunikationsarten in diese Kategorien ein.

Tabelle 3.1. Kommunikationsformen nach Richardson [2015b]

	One-to-One	One-to-Many
Synchron	Request/Response	-
Asynchron	Notification Request/Async Response	Publish/Subscribe Publish/Async Responses

Messaging ist eine asynchrone Form der Kommunikation, die One-to-One und One-to-Many Beziehungen ermöglicht. Dafür stellen Hohpe und Woolf [2002] Integrationsmuster vor. *RappitMQ*,[2] *Apache Kafka*,[3] *Apache ActiveMQ*,[4] und *NSQ*[5] sind verschiedene Systeme, die Messaging anbieten. Zu den Vorteilen von Messaging gehört die Entkopplung von Sender und Empfängern. Kommunikationspartner erhalten Kommunikationskanäle von einer zentralen Stelle und müssen sich nicht gegenseitig kennen. Zudem unterstützen Messaging-Systeme meist das Puffern von Nachrichten. So müssen nicht die einzelnen Services sicherstellen, dass eine Nachricht ankommt. Ein Nachteil ist, dass die zentrale Komponente hoch verfügbar sein muss und einen potentiellen Engpass darstellt. Außerdem sind Antworten schwerer ihren Anfragen zuzuordnen: Es muss beim Senden einer Anfrage explizit ein Kanal bestimmt werden, auf dem die Antwort empfangen wird und ein Identifikator mitgesendet werden.
Richardson [2015b] stellt als Alternative zum Messaging mit *Request/Response* eine synchrone Kommunikationsform vor. Hierbei wird zu jeder Anfrage eine Antwort erwartet und blockiert, bis die Antwort eingetroffen ist. Da diese Antworten auch leer sein können, lassen sich auch Benachrichtigungen im Messaging-Stil nachbilden. Implementierungen sind REST mit HTTP und *Apache Thrift*.[6] Ein Vorteil von Request/Response ist, dass die Zuordnung von Antworten implizit geschieht. Außerdem basiert REST auf Technologien wie der JavaScript Object Notation (JSON) und HTTP, die weit verbreitet und daher vielen Entwicklern bekannt sind. Zudem gibt es im Gegensatz zum Messaging keine zentrale Stelle, die betrieben werden muss. Dafür muss bei Request/Response jeder Microservice das Ziel seiner Anfrage kennen. Hierzu stellen wir Lösungen in Abschnitt 3.2.3 vor. Ein weiterer Nachteil ist, dass Services für die Dauer der Anfrage blockiert und Entwickler daran denken müssen, dass eine Antwort nicht zwingend ankommt. Dies kann der Fall sein, wenn der Empfänger ausfällt oder die Übermittlung scheitert.
Da lose Kopplung für die unabhängige Entwicklung von Microservices wichtig ist, entscheiden wir uns bei unserer Architektur für asynchrone Notifications zur Kommunikation zwischen Microservices. Wir verwenden kein zentrales Messaging-System, um die Abhängigkeit zu einem solchen System zu vermeiden und die Services autonomer zu halten.

[2] https://www.rabbitmq.com/
[3] http://kafka.apache.org/
[4] http://activemq.apache.org/
[5] http://nsq.io/
[6] https://thrift.apache.org/

Nachrichtenformat

Das Nachrichtenformat bestimmt, wie Informationen aufgebaut werden, die zwischen Microservices ausgetauscht werden. Es ist notwendig, dass beide Services dieses Nachrichtenformat kennen, damit die Informationen verarbeitet werden können. Da Microservices mit verschiedenen Technologien implementiert werden können, ist es wichtig, ein Format zu wählen, das möglichst Technologie-unabhängig ist. Die möglichen Formate werden nach Binärformaten und Textformaten unterteilt. Zu den Textformaten gehören JSON und Extensible Markup Language (XML), die Binärformate sind abhängig von der Anwendung. Textformate sind Dateien oder Datenströme, die ihre Daten ausschließlich in einer Repräsentation druckbarer Zeichen kodieren. Sie haben den Vorteil, dass sie für Menschen leichter lesbar sind. Dies erleichtert das Testen und Debuggen und fördert die Verständlichkeit der API. Außerdem sind Textformate unabhängig von der Technologie der Microservices. Als weiteren Vorteil gibt es Sprachen zur Definition von Schemata, mit denen textbasierte Nachrichten syntaktisch validiert werden können. Im Gegenzug müssen Informationen vor dem Senden häufig erst in ein Textformat konvertiert und beim Empfänger wieder eingelesen werden. Das bedeutet zusätzlichen Rechenaufwand, der den Zeitbedarf durch die Kommunikation steigert.
Binärformate hingegen setzen voraus, dass beide Kommunikationspartner eine Möglichkeit haben, die Binärdaten zu verarbeiten. Diese Möglichkeit kann durch das Kommunikationssystem oder ein Framework bereitgestellt werden. Falls es so etwas für eine der verwendeten Technologien nicht gibt, muss eine entsprechende Möglichkeit für diese Technologien selbst erstellt werden. Das erfordert zusätzliche Entwicklungszeit und führt zu möglichen neuen Abhängigkeiten zwischen den Services.
Die Wahl des Nachrichtenformats kann durch die Kommunikationsart eingeschränkt werden. Thrift zum Beispiel erlaubt nur Nachrichten im JSON-Format oder eigenen Binärformaten.
In Abschnitt 3.2.4 beschreiben wir, dass eine grafische Oberfläche im Bereich der Textanalyse häufig überflüssig ist. Ein Textformat erleichtert in diesem Fall das Testen der Services durch bessere Lesbarkeit und existierende Validierungswerkzeuge. Zudem sind Nachrichtenformate Teil der Schnittstellen. Durch Validierungsschemata wird die Schnittstelle klarer und leichter verständlich formuliert, als dies bei Binärformaten möglich ist. Bei Microservices fördert eine eindeutige Schnittstellendefinition die Nutzbarkeit des Services. Wir empfehlen daher die Verwendung eines textbasierten Nachrichtenformats.

Schnittstellendefinition

Die Kombination aus Kommunikationsart und Nachrichtenformaten bilden die Schnittstelle eines Microservice. Diese Schnittstelle sollte klar definiert sein, um die Nutzung des Microservices möglichst einfach zu machen. Eine solche Beschreibung kann explizit über eine Interface-Beschreibungssprache (IDL) erfolgen. Ein Beispiel für IDLs ist WSDL [Christensen u. a. 2001], die im Rahmen von SOA verwendet wird. WSDL kann zur Beschreibung der

Schnittstellen von Microservices genutzt werden, falls unterstützte Technologien benutzt werden. Zur Beschreibung von XML-Dokumenten kann *XMLSchema* [Thompson u. a. 2004] verwendet werden und für JSON-Dokumente gibt es entsprechend *JSON-Schema* [Galiegue und Zyp 2013]. Diese Schemas haben zusätzlich den Vorteil, dass sie leicht zum Validieren der Nachricht verwendet werden können. Das erleichtert Tests und Sicherheitsmaßnahmen im Betrieb. Apache Thrift und viele Messaging-Systeme definieren eigene IDLs, mit denen der Nutzer arbeiten muss. Die Wahl einer IDL kann also implizit durch die Wahl der Kommunikationsart und des Datenformats erfolgen.
Im vorherigen Unterabschnitt begründen wir die Entscheidung für textbasierte Nachrichtenformate in unserer Architektur. Wir empfehlen daher die Nutzung von Schemata, um eingehende Nachrichten validieren zu können und die Schnittstellen zu dokumentieren.

3.2.2. Service-Entdeckung

Microservice-Systeme bestehen typischerweise aus vielen Services, die individuell skaliert und ausgeliefert werden können. Außerdem können Instanzen ausfallen und dafür andere gestartet werden. Das bedeutet einen häufigen Wechsel der Instanzen, der ein manuelles Ausliefern der Services erschweren würde. Daher werden Microservices meist dynamisch oder automatisiert ausgeliefert. Ihre Netzwerkadressen sind dadurch dynamisch und können nicht fest in andere Services integriert werden. Um Microservices trotzdem zu finden, führt Richardson [2015d] zwei Entdeckungsmuster und zwei Registrierungsmuster an.

Entdeckungsmuster

Beide Entdeckungsmuster beinhalten ein *Service-Register*, also einen Ort, an dem die Instanzen der Microservices ihren Netzwerkadressen zugeordnet werden. Dieses Register liegt an einem zentralen Ort, der zum Beispiel über das Domain Name System (DNS) [Mockapetris und Dunlap 1988] auffindbar ist.
In Abbildung 3.6 wird die clientseitige Entdeckung gezeigt. Service A sucht in dem Register nach Instanzen von Service B und führt dann Berechnungen zur Lastverteilung durch, aufgrund derer er sich für Instanz 3 entscheidet. Beim clientseitigen Entdeckungsmuster fragen also die Microservices direkt Adressen der Instanzen anderer Microservices von dem Service-Register ab. Jeder Microservice muss selbst Lastverteilung für alle Services, die er verwendet, betreiben. Das ermöglicht anwendungsspezifische Lastverteilung und Cashing, führt aber auch dazu, dass jeder Service Verfahren dafür implementieren muss. Die erforderliche Logik lässt sich in Programmbibliotheken auslagern, wie in Abschnitt 3.2.6 beschrieben. Netflix’ Eureka [Liu 2014] verfolgt diesen Ansatz. Der Nachteil ist, dass entsprechende Bibliotheken für jede Technologie und daher im Extremfall für jeden Microservice neu implementiert werden müssen.

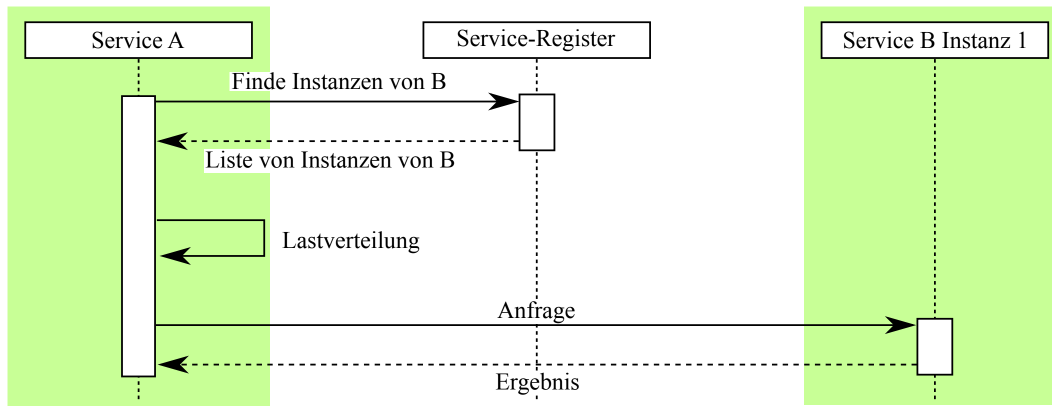


Abbildung 3.5. Clientseitige Entdeckung

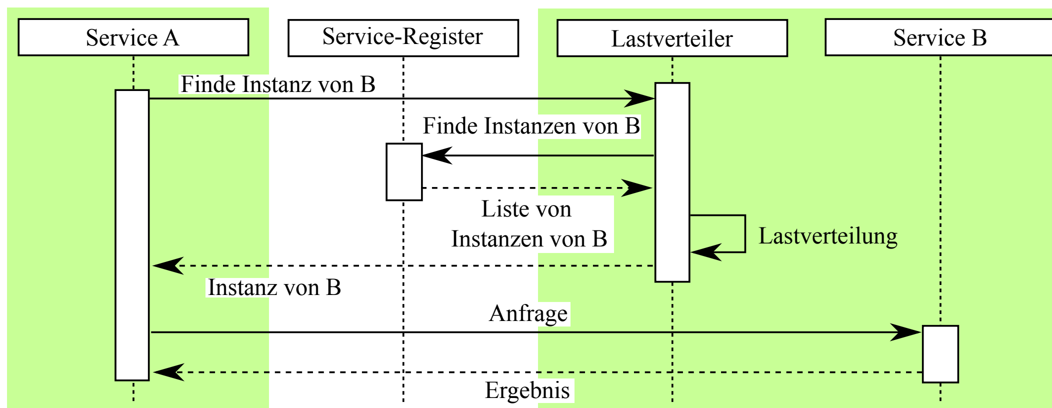
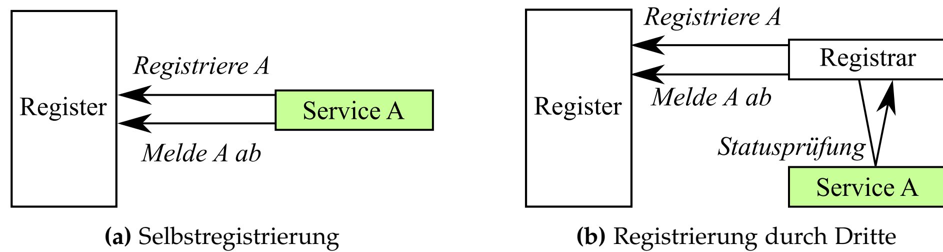


Abbildung 3.6. Serverseitige Entdeckung

Das serverseitige Entdeckungsmuster sieht zusätzlich zu dem Service-Register noch einen Lastverteiler für jeden Microservice vor. Wenn ein Microservice die Instanz eines anderen benötigt, fragt er beim Lastverteiler nach einer Adresse. Dieser verwendet das Service-Register, um Instanzen zu finden, ermittelt daraus eine geeignete und teilt deren Adresse dem fragenden Microservice mit. Abbildung 3.5 zeigt diesen Ansatz. Hier wird ersichtlich, dass Service A nur die Adresse einer Instanz von B erfährt. Bei diesem Ansatz ist die Lastverteilung unabhängig vom aufrufenden Microservice. Dadurch müssen die Microservices diese Logik nicht implementieren, können aber zusätzliches Wissen, das geschicktere Lastverteilung ermöglichen würde, nicht ausnutzen. Zudem gibt es mit dem Lastverteiler eine weitere zentrale Stelle, die hoch verfügbar sein muss.

Registrierungsmuster

Services können sich nach dem Muster Selbstregistrierung selbst bei dem Service-Register anmelden wie in Abbildung 3.7a gezeigt. Dazu wird beim Start eines Services ein entsprechender Aufruf gemacht. Die Netzwerkadresse des Service-Registers kann entweder per Hand konfiguriert oder per DNS gefunden werden. Die Anmeldung muss regelmäßig aktualisiert werden, damit das Register weiß, dass die Service-Instanz noch aktiv und erreichbar ist.



(a) Selbstregistrierung **(b)** Registrierung durch Dritte

Abbildung 3.7. Service-Registrierung

Das alternative Muster ist die Registrierung durch Dritte in Abbildung 3.7b, bei der es einen *Registrar* gibt, der Anhand von Events oder durch Scannen der Betriebsumgebung neue Instanzen findet. Der Registrar meldet neue Instanzen im Register an und führt Überprüfungen durch, ob die bekannten Instanzen noch aktiv sind. Falls eine Instanz inaktiv ist, meldet er sie bei dem Service-Register ab. Wie beim serverseitigen Entdeckungsmuster wird hier Logik aus den Microservices in eine andere Komponente ausgelagert. Dadurch entstehen ähnliche Vor- und Nachteile: Die Logik muss nicht von den Microservices implementiert werden. Dafür entsteht eine zusätzliche Komponente, die betrieben und verwaltet werden muss.

Für unsere Architektur wählen wir das serverseitige Entdeckungsmuster, um die Berechnung der zu verwendenden Instanz jedem Microservice selbst zu überlassen. So kann die Lastverteilung für diesen Microservice geändert werden, ohne andere Services anpassen zu müssen. Die Anwendbarkeit eines Registrars hängt von der Betriebsumgebung ab. Zudem gibt es damit eine weitere zentrale Komponente, von der die Microservices abhängen. Da das Muster der Selbstregistrierung keinen Registrar benötigt und die Microservices so autonomer werden, empfehlen wir dieses Muster.

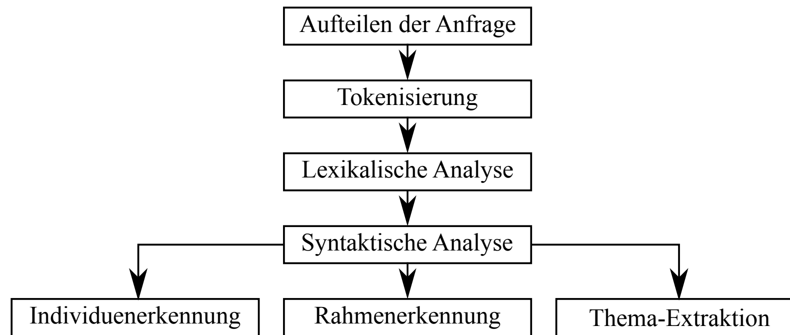


Abbildung 3.8. Möglicher Ablauf einer Textanalyse

3.2.3. Koordination von Arbeitsabläufen

Berechnungen, die durch ein System mit Microservice-Architektur durchgeführt werden sollen, können die Koordination mehrerer Microservices erfordern. Wie in Abschnitt 2.1 auf Seite 7 beschrieben, hängt die semantische Analyse von den Vorbereitungsphasen ab. Wenn die Vorbereitungsphasen als Microservices implementiert sind, muss ein zu analysierender Text diese Services in der richtigen Reihenfolge durchlaufen. Danach muss er an den richtigen Microservice der semantischen Analyse übermittelt werden. In diesem Unterabschnitt erläutern wir Methoden, einen solchen Ablauf zu steuern. Zwei dieser Ansätze arbeiten mit direkten Aufrufen. *Orchestrierung* ist ein zentralisierter Ansatz aus dem SOA-Umfeld und *Choreographie* ein dezentrales Verfahren, das Events verwendet. Abbildung 3.8 zeigt, wie der Ablauf einer Textanalyse aussehen kann. In diesem Beispiel können Individuenerkennung, Thema-Extraktion und Rahmenerkennung parallel ausgeführt werden, benötigen aber die Ergebnisse der Vorbereitungsphasen, die, wie in Abschnitt 3.1.3 beschrieben, auch als Microservices implementiert sind. Am Beispiel wird deutlich, dass eine parallele oder sequentielle Verkettung von Services erforderlich sein kann.

Direkte Aufrufe

Bei der Koordination über direkte Aufrufe entscheidet jeder Microservice, welche anderen er aufruft. Die Koordination geschieht also durch die Entscheidungen der einzelnen Microservices. Wir unterscheiden hier zwischen Aufrufen zur Erfüllung von Abhängigkeiten und Aufrufen zur Weiterleitung von Ergebnissen. Abbildung 3.9 zeigt, wie die Rahmenerkennung nacheinander verschiedene Services aufruft, um die gewünschten syntaktischen Informationen zu erhalten. Eine System-API erhält die Aufgabe, Individuenerkennung, Rahmenerkennung und Thema-Extraktion durchzuführen. Sie ruft parallel die Microservices für diese Methoden auf. Jeder der Microservices ruft wiederum nacheinander die Vorbereitungsphasen auf. Die Vorbereitungsphasen erhalten hierbei jeweils das Ergebnis der vorherigen Phase. Nachdem die System-API die Ergebnisse der drei semantischen Services

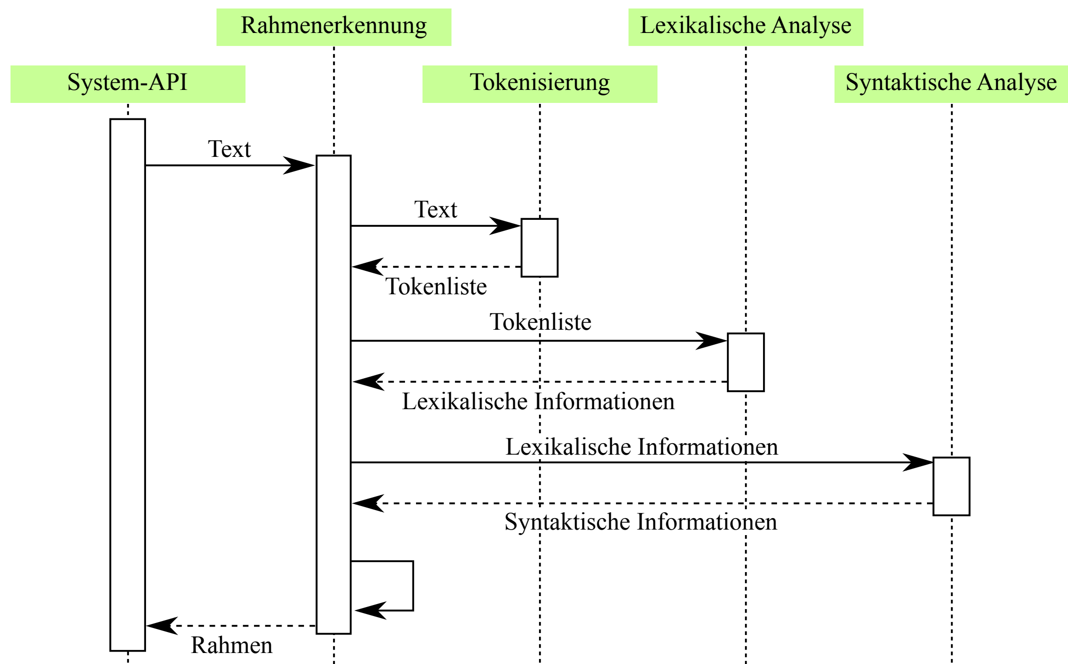


Abbildung 3.9. Koordination durch direkten synchronen Aufruf

hat, wird das Ergebnis kombiniert und an den Aufrufer gegeben. Jeder der Semantik-Microservices orchestriert die Verwendung der Vorbereitungs-Microservices. Newman [2015] bezeichnet diesen Ansatz daher als *Orchestrierung*, obwohl keine zentrale Stelle existiert, die alle Komponenten orchestriert.
Bei diesem Ansatz wird jede Vorbereitungsphase drei mal für den selben Text aufgerufen. Um den Nachteil dadurch zu vermindern, können Ergebnisse in den Vorbereitungsservices zwischengespeichert werden. Außerdem werden die Ergebnisse der Vorbereitungsphasen zwei mal übertragen, obwohl ein mal ausreichen würde. Dafür sind die für die semantische Analysen notwendigen Abläufe an zentraler Stelle festgelegt. Dadurch können diese Services einfach überwacht und geändert werden, da sie selbst ihre Abhängigkeiten auflösen. Die zweite Möglichkeit, direkte Aufrufe zu nutzen, ist asynchron. Hierbei lösen Microservices nicht ihre Abhängigkeiten auf, sondern leiten ihre Ergebnisse weiter. Die Koordinierung geschieht also durch die Berechnung jeden Microservices, an welchen anderen Service er sein Ergebnis weiterleitet.
Im Beispiel in Abbildung 3.10 startet die System-API direkt die Tokenisierung. Wenn diese abgeschlossen ist, sendet der Tokenisierungs-Service das Ergebnis an die lexikalische Analyse. Diese schickt ihr Ergebnis an die syntaktische Analyse und die wiederum übermittelt ihr Ergebnis der System-API. Beim parallelen Aufruf mehrerer Semantik-Services leitet die

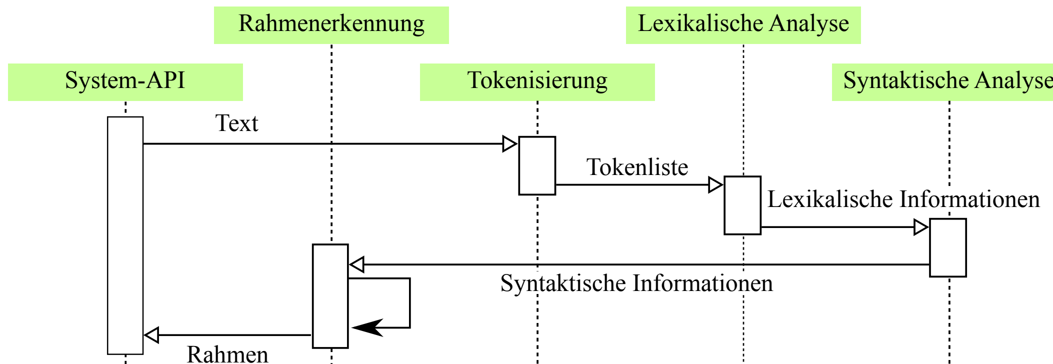


Abbildung 3.10. Koordination durch direkten asynchronen Aufruf

System-API den Text ein mal an die Tokenisierung weiter. Die letzte Vorbereitungsphase ist dafür zuständig, den Text und die Ergebnisse der Vorbereitungsphasen an die Semantik-Services weiterzuleiten und somit die parallele Berechnung zu starten. Die System-API muss dann auf die Nachrichten der Semantik-Services warten und diese vereinen.
Der zweite Ansatz bietet den Vorteil, dass die Koordinierung gleichmäßig auf die Services verteilt wird. Dadurch werden nicht einzelne Komponten, wie die Rahmenerkennung, durch Koordination blockiert und unverhältnismäßig stärker ausgelastet. Außerdem passieren keine doppelten Berechnungen wie bei den synchronen direkten Aufrufen. Dafür lassen sich Abläufe schwerer überwachen und ändern.

Orchestrierung

Orchestrierung [Newcomer und Lomow 2005] ist ein Begriff aus dem SOA-Umfeld (siehe Abschnitt 2.4 auf Seite 12). In diesem Umfeld werden im Rahmen der Enterprise Application Integration (EAI) existierende Anwendungen durch eine zentrale Stelle koordiniert. Diese Stelle wird von uns *Dirigent* [Chappell 2004] genannt.
Abbildung 3.11 zeigt diesen Ansatz. Anstelle des Rahmenerkennungs-Microservices, wie bei den direkten synchronen Aufrufen, ruft der Dirigent die einzelnen Microservices auf. Der Rahmenerkennungs-Microservice bekommt schon beim Aufruf die Ergebnisse, die er benötigt. Parallele Aufrufe führen nicht zu doppelten Aufrufen der Vorbereitungsphasen. Dieser Ansatz erlaubt eine klare Definition der Ablaufreihenfolgen und daher auch leichtes Ändern und Hinzufügen von Arbeitsabläufen. Zudem gibt es neben Modellierungssprachen wie *WS-BPEL* [Weerawarana u. a. 2005] auch *Business-Process-Modeling*-Anwendungen, die Entwickler bei der Orchestrierung unterstützen. Bei diesem Ansatz wird aber viel Logik in den Dirigenten ausgelagert. Wenn sich aber der Arbeitsablauf eines Microservice ändert, weil er zum Beispiel für eine neue Berechnung einen zusätzlichen Microservice benötigt, muss auch der Dirigent angepasst werden. Das führt zu Abhängigkeiten bei der Auslieferung und Entwicklung, die in Microservice-Systemen vermieden werden sollen. Zudem

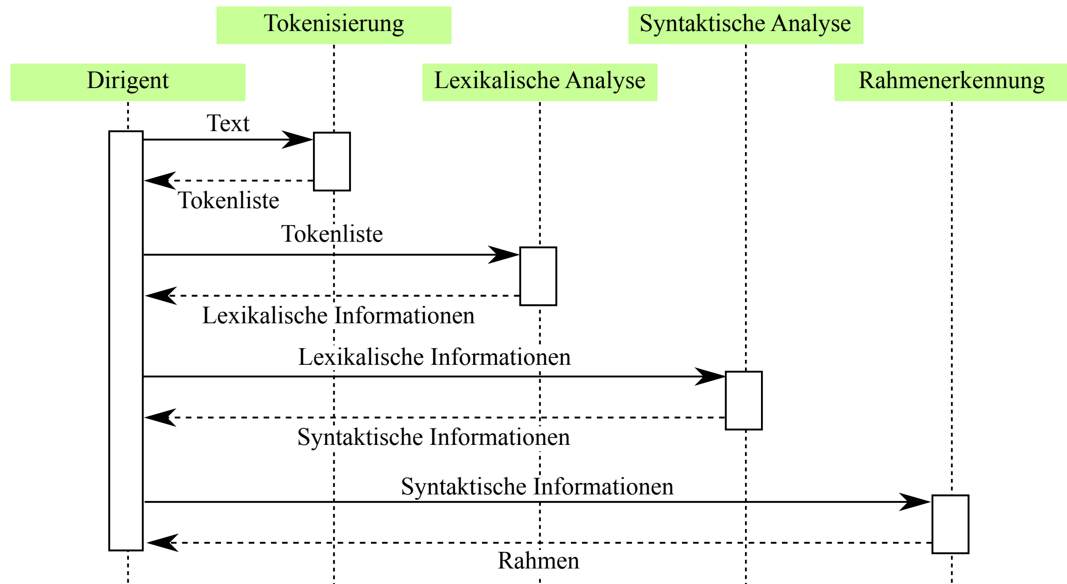


Abbildung 3.11. Koordination im SOA-Orchestrierungsansatz

führt ein Ausfall des Dirigenten dazu, dass der gesamte durch ihn gesteuerte Arbeitsablauf nicht zur Verfügung steht, obwohl die erforderlichen Microservices funktionsfähig sind. Des weiteren bildet der Dirigent einen möglichen Engpass und weitere zu betreibende und wartende Komponente.

Choreographie

Wolff [2016a] und Richardson [2015a] stellen als Alternative zur Orchestrierung und direkten Aufrufen das Konzept der *Event Driven Architecture (EDA)* [Michelson 2006] vor. Newman [2015] beschreibt das selbe Konzept mit dem Begriff *Choreographie* [Peltz 2003]. Bei Choreographie entscheidet jeder Service selbst, zu welchen Zeitpunkten er benötigt wird. Die Koordination erfolgt über Events, bei denen sich Services registrieren können. Die Events können hierbei über ein *Messaging-System* veröffentlicht werden oder jeder Microservice bietet über seine API eigene Möglichkeiten zur Registrierung für andere Microservices.
In Abbildung 3.12 veröffentlicht die syntaktische Analyse ein Event, das besagt, dass die Syntax bestimmt wurde, und der Text bereit ist für die weitere Verarbeitung. Die Microservices für Thema-Extraktion, Individuenerkennung und Rahmenerkennung haben sich vorher für dieses Event registriert und erhalten nun die Benachrichtigung und können mit der Verarbeitung beginnen. Die System-API registriert sich für Events der drei Semanik-Services und wartet bis alle veröffentlicht wurden.

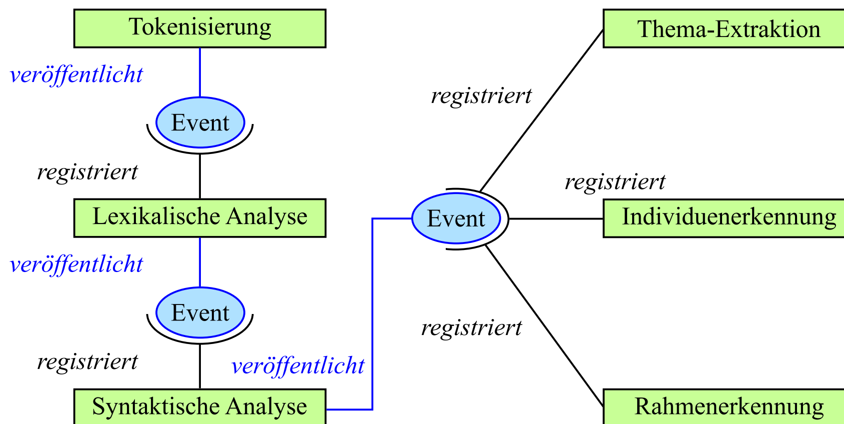


Abbildung 3.12. Koordination mittels Choreographie

Bei diesem Ansatz kann jeder Microservice auf Events reagieren, aber nicht steuern, was auf die von ihm veröffentlichten Events hin geschieht. Es wird mehr Logik in die Microservices verlagert und die Entwickler eines Microservice müssen genau wissen, in welchen Arbeitsabläufen und an welchen Stellen dort der Microservice gebraucht wird. Das führt dazu, dass Änderungen im Ablauf mehr Absprachen zwischen Teams benötigen und diese weniger autonom handeln können. Andererseits folgt aus dem benötigtem gutem Verständnis der Rolle des Microservice, dass es seltener zu Missverständnissen kommt und daher wahrscheinlicher genau die Funktionalität entwickelt wird, die benötigt wird. Da es bei Choreographie keine zentrale Stelle gibt, der die Berechnung eines Microservice überwachen kann, muss dies auf anderen Wegen geschehen. Das Monitoring wird so erschwert und zudem werden Programmabläufe über mehrere Services hinweg verteilt und sind daher nicht so einfach zu überblicken wie bei Orchestrierung. Dafür bietet Choreographie eine losere Kopplung, da die einzelnen Microservices nicht von einem Dirigenten abhängen und auch autark funktionieren.

Newman [2015] bezeichnet die direkten Aufrufe zur Auflösung von Abhängigkeiten als *brittle* (spröde) und rät daher zur Choreographie. Auch Wolff [2016a] und Richardson [2015a] raten zur Event Driven Architecture-Lösung. Die SOA-Definition von Orchestrierung widerspricht aufgrund der erläuterten Abhängigkeiten dem Microservice-Stil.
Systeme zur semantischen Analyse können für verschiedene Zwecke genutzt werden, wie die Auflistung der Methoden der semantischen Analyse in Abschnitt 3.1.4 auf Seite 24 zeigt. Events eignen sich aber besonders gut, wenn auf bestimmte Ereignisse immer gleich reagiert wird. Wir schlagen in unserer Architektur daher die Verwendung direkter asynchroner Nachrichten zur Weiterleitung von Ergebnissen vor. Teil des Aufrufs kann so zum Beispiel der Ablaufplan sein, welcher weitergeleitet und von jedem Microservice selbst ausgewertet wird.

3.2.4. Grafische Benutzeroberflächen und Ergebniskomposition

In Abschnitt 3.2.3 schreiben wir, dass verschiedene Microservices an einer Aufgabe beteiligt sein können. Abbildung 3.8 auf Seite 35 zeigt, dass für eine Anfrage mehrere Teilergebnisse erzeugt werden können. Der Nutzer des Systems soll aber in der Regel für eine Anfrage nur eine Antwort bekommen. Daher müssen die Teilergebnisse zusammengesetzt werden. Für die Zusammenführung ist die Repräsentation der Ergebnisse wichtig. Das Ergebnis kann Teil einer GUI oder, wie in Abschnitt 3.2.1 vorgestellt, eine Datei sein. Daher diskutieren wir zunächst, wie eine GUI in Microservice-Systeme integriert werden kann. Anschließend stellen wir einige Möglichkeiten der Ergebniszusammenführung vor.

Integration einer Grafischen Benutzeroberfläche

Grafische Benutzeroberflächen sind meist für Anwendungen, die von Menschen verwendet werden sollen, gewünscht. In Microservice-Architekturen kann ein eigener Service die GUI aus Ergebnisdaten bilden. Alternativ erstellt jeder Service eine eigene Teiloberfläche oder es gibt eine zusätzliche Architekturkomponente, die eine GUI erzeugt. Wolff [2016a] rät dazu, Microservices mit einer eigenen GUI auszustatten. So ist eine weitere Funktionalität, die vom Service abhängt, darin integriert. Das erleichtert das Testen und steigert die Unabhängigkeit des Services gegenüber der Lösung, jede Teil-GUI durch einen eigenen Microservice bereitzustellen. Wenn aber mehrere GUIs zur Verfügung gestellt werden müssen, weil es etwa Geräte gibt, die kein Ausgabeformat teilen, wachsen die Microservices im Funktionsumfang. Zudem müssen dem für den Service zuständigem Team weitere Technologien bekannt sein, was die Komplexität des Services steigert. Die Auslagerung der GUIs kann daher in diesem Fall sinnvoll sein.
Semantische Informationen sind meist zur Verarbeitung durch Computer gedacht. Systeme zur semantischen Analyse sind daher oft Untersysteme von anderen Systemen. Ein Beispiel sind Semantische Content-Management-Systeme. Eine GUI des Semantik-Systems würde daher oft nur zu Demonstrations- und Testzwecken verwendet werden. Dafür genügt eine einzige GUI pro Service, sodass die Nachteile bei der Integration von GUIs in die zugehörigen Microservices sinken. Wir schlagen daher in unserer Architektur optionale, in die Microservices integrierte, GUIs zu Testzwecken vor.

Komposition von Berechungsergebnissen

Die Möglichkeiten zur Zusammensetzung von Berechnungsergebnissen hängen von der Koordination der Services ab. Bei synchronen direkten Aufrufen werden Ergebnisse vom Aufrufer zusammengeführt. Im asynchronen Ansatz hingegen bekommt der Aufrufer die Ergebnisse des Aufgerufenden nicht. Der Aufgerufende muss also seine Ergebnisse mit den bereits vorhandenen zusammenführen. Im Arbeitsablauf in Abbildung 3.8 auf Seite 35 muss die System-API die Ergebnisse der Individuenerkennung, Rahmenerkennung und Thema-Extraktion vor der Ausgabe vereinen.

Bei Orchestrierung mit einem Dirigenten übernimmt dieser die Komposition. Im Fall von Choreographie als Koordinierungsmethode kann es eine zusätzliche Komponente geben, die Anfragen entgegennimmt, entsprechende Events veröffentlicht und auf Ergebnisevents horcht. Wenn ein Ergebnisevent veröffentlicht wird, wird dieses in das Gesamtergebnis integriert und falls das Gesamtergebnis danach komplett ist, ein weiteres Event mit dem Gesamterebnis veröffentlicht. Mit dieser Komponente gibt es eine zentrale Stelle, die aber verglichen mit dem Dirigenten durch die Events loser gekoppelt ist.
Für die Zusammensetzung von GUIs stellen Wolff [2016a], Kraus u. a. [2013] und Newman [2015] verschiedene Ansätze vor. Da wir GUIs nur für Testzwecke der einzelnen Services verwenden wollen, müssen diese nicht zusammengeführt werden. Wir gehen daher hier nicht näher auf diese Ansätze ein.
Da wir in Abschnitt 3.2.3 die Verwendung von direkten asynchronen Aufrufen empfehlen, erfolgt die Ergebniszusammenführung in unserer Architektur durch die Empfänger eines Aufrufs vor der Weiterleitung. Im Beispiel in Abbildung 3.8 auf Seite 35 leitet die Tokenisierung ihre Ergebnisse an die lexikalische Analyse weiter. Diese führt ihre Berechnungen durch und führt die Ergebnisse der Tokenisierung mit ihren eigenen zusammen. Anschließend können die kombinierten Ergebnisse an die syntaktische Analyse gesendet werden.

3.2.5. Datenhaltung

Wie in Abschnitt 3.1.4 auf Seite 22 beschrieben, hat jede Partition der semantischen Analyse und somit jeder Microservice seine eigenen Daten, auf die von anderen Services nur über die API zugegriffen werden kann. Außerdem ist für jede Art von Daten ein Microservice klar als Besitzer definiert. Zusätzlich können Daten für eine optimale Repräsentation in unterschiedlichen Datenbankanwendungen gehalten werden. Diese Unterteilung bezeichnet Lewis und Fowler [2011] als *Polyglot Persistence*, also mehrsprachiger Persistenz. Durch diese Aufteilung der Daten wird sichergestellt, dass die Repräsentation von Daten unabhängig von anderen Microservices geändert werden kann.
In monolithischen Systemen werden meist die so genannten *ACID*-Eigenschaften [Cattell 2011] gefordert: Atomität, Konsistenz, Isolation, Persistenz. Atomität beschreibt die unterbrechungsfreie Ausführung von Befehlen. Konsistenz ist, dass nach außen alle Repliken eines Datums zu jeder Zeit identisch sind. Mit Isolation wird die Sicherheit bezeichnet, dass während Transaktionen keine anderen Transaktionen durchgeführt werden, die die erste Transaktion in einen undefinierten Zustand führen. Persistenz ist die Resistenz der Datenhaltung gegen Software- und Hardwarefehler. Diese Eigenschaften werden durch die Verwendung relationaler Datenbanken sichergestellt. Durch die Nutzung mehrerer Datenbanken in verteilten Systemen können die ACID-Eigenschaften nicht mehr so einfach erreicht werden.

Das *CAP-Theorem* [Brewer 2000] sagt aus, dass sich nur zwei der drei Eigenschaften Partitionstoleranz, Verfügbarkeit und Konsistenz ohne größeren Aufwand erreichen lassen. Wir akzeptieren daher zeitlich beschränkte Inkonsistenzen, um Verfügbarkeit und Partitionstoleranz mit geringerem Aufwand gewährleisten zu können. Vogels [2009] nennt diese zeitweise Einschränkung der Konsistenz *Eventual Consistency*.
Die persistenten Daten in der semantischen Analyse lassen sich in Arbeitsdaten und Ergebnisse unterteilen. Die Arbeitsdaten sind Daten, die für die Extraktion der semantischen Informationen nötig sind und bei der Analyse von Texten aktualisiert werden. Es können zum Beispiel statistische Daten bei heuristischen Verfahren sein oder im Beispiel der Individuenerkennung eine Menge von Vornamen, die der Erkennung genannter Individuen dient. Soweit möglich sollten die Arbeitsdaten nur von dem Microservice genutzt werden, der sie auch erstellt. In diesem Fall muss die Konsistenz nur innerhalb des Microservice sichergestellt werden. Falls die Arbeitsdaten von einem anderen Microservice benötigt werden, kann es zeitweise zu Inkonsistenzen zwischen den Datenbeständen kommen. Das stellt aber nur ein Problem dar, wenn in der Zeit der Inkonsistenz Texte analysiert werden, auf die die inkonsistenten Daten Einfluss haben. Im Beispiel mit den Vornamen für die Individuenerkennung wäre also ein neu hinzugefügter Vorname nur relevant, wenn gerade ein Text analysiert wird, in dem genau dieser Vorname genannt wird. Der Anteil an Texten, in denen ein spezieller Vorname auftaucht, ist in den meisten Fällen gering und daher ist die Wahrscheinlichkeit, dass sich die Inkonsistenz überhaupt auswirkt, ebenfalls niedrig.
Da sich die Ergebnisse der Analysemethoden in Abhängigkeit der Arbeitsdaten und somit auch der bisher analysierten Texte ändert, muss bei einer Verwendung immer davon ausgegangen werden, dass einige semantische Informationen nicht entdeckt werden. Falls sich also eine Inkonsistenz der Arbeitsdaten auswirkt, kann man annehmen, dass fehlende Informationen und daher auch die Inkonsistenzen nicht kritisch sind. Die Inkonsistenzen an Arbeitsdaten durch Eventual Consistency sind also in der semantischen Analyse hinnehmbar.
Ergebnisse von Methoden der semantischen Analyse sind Informationen, die das Textanalyse-System verlassen können. Sie stehen in der Zuständigkeit des erzeugenden Microservice und werden, falls nötig, dort gespeichert. Falls andere Services diese Ergebnisse benötigen, um sie weiter zu verarbeiten, werden sie zum Zeitpunkt der Fertigstellung als Ganzes übermittelt. Da die Ergebnisse vor der Weiterverarbeitung nicht geändert werden, können auch keine Inkonsistenzen auftreten.
Da zeitliche Inkonsistenzen nur eine untergeordnete Rolle spielen, entscheiden wir uns dafür, in unserer Architektur skalierende Datenbanken zu verwenden. So wird Skalierung in X-Richtung durch Duplizieren von Service-Instanzen nicht durch den Zugriff auf die Datenbank limitiert.

3.2.6. Entwicklung

Nach dem Gesetz von Conway [Conway 1968] wird eine Software immer die Struktur des Unternehmens widerspiegeln, das sie entwickelt. Um kleine Microservices zu erreichen, benötigt man also kleine Teams, die jeweils am besten nur einen Microservice entwickeln. Da die Kommunikation zwischen Teams aufwändiger ist als innerhalb eines Teams, vermeiden Entwickler so Abhängigkeiten, die Team-Grenzen überschreiten. Dadurch entsteht implizit eine losere Kopplung des resultierenden Systems. Außerdem ist der Koordinations- und somit Kommunikationsbedarf in kleineren Teams geringer, sodass die Produktivität steigt. Kraus u. a. [2013] schlagen außerdem vor, den Betrieb durch das entwickelnde Team verantworten zu lassen. Dadurch gäbe es einen höheren Anreiz, für Testbarkeit, die Testabdeckung und generell Codequalität zu sorgen. Aus diesen Gründen raten wir dazu, jeden Microservice von nur einem Team entwickeln zu lassen. Falls der Aufwand nicht zu groß ist, kann ein Team aber auch mehrere Microservices entwickeln.

Geteilter Code

Eine Praxis bei der Softwareentwicklung ist Don't Repeat Yourself! (DRY). Damit wird nicht nur beschrieben, dass kein Code doppelt entwickelt werden soll, sondern auch, dass man kein Verhalten und Wissen duplizieren soll. Wolff [2016a] rät, dieses Prinzip innerhalb von Microservices anzuwenden, aber über verschiedene Microservices hinweg möglichst keinen geteilten Code zu verwenden. Dadurch entstehen keine gemeinsamen Abhängigkeiten, bei deren Änderung Kommunikationsbedarf besteht. Zudem können so weniger allgemeine und daher performantere oder leichter verständliche Lösungen verwendet werden.
Nach Newman [2015] sind geteilte Bibliotheken erlaubt, sofern sie nur Zugriffe über eine feste API zulassen. Wenn alle Abhängigkeiten über eine klar definierte Schnittstelle laufen, gibt es keine versteckten Abhängigkeiten, die sich bei Änderungen negativ auswirken. Als Möglichkeit, so etwas zu erreichen, schlagen Kraus u. a. [2013] vor, solche Bibliotheken mit den Quellen zu veröffentlichen und wie externe Komponenten zu behandeln.
Um die Verwendung eines Services einfacher zu machen, kann eine Bibliothek erstellt werden, die die Verwendung der Schnittstelle in einer anderen Programmiersprache ermöglicht. Newman [2015] rät hierbei, solche Bibliotheken nicht von dem Team entwickeln zu lassen, das den Microservice erstellt. Sonst besteht das Risiko, dass Logik in die Bibliothek ausgelagert wird. Änderungen an dieser Logik können dann nur in Betrieb genommen werden, wenn alle Microservices, die die Bibliothek verwenden, ebenfalls aktualisiert werden. Die dafür nötige Koordination soll aber durch den Einsatz von Microservices vermieden werden, um den Entwicklungsprozess skalierbar zu halten.
In Abschnitt 3.1.3 auf Seite 19 diskutieren wir die Integration der Vorbereitungsphasen der semantischen Analyse als Bibliothek. Mit der Nutzung ähnlich einer Bibliothek von Drittanbietern lässt sich das Risiko ungewollter Abhängigkeiten mindern. Änderungen an den Vorbereitungsphasen würde aber weiterhin die Aktualisierung aller Microservices benötigen, daher raten wir davon ab.

Bibliotheken, die Schnittstellen einzelner Microservices kapseln, haben bei der semantischen Analyse keine speziellen Vor- oder Nachteile. Sie können daher unter den oben genannten Voraussetzungen nach Newman [2015] verwendet werden.
Da wir die Vorbereitungsphasen in unserem Architekturentwurf als Microservices integrieren, entsteht hier kein geteilter Code. Für Funktionen wie die Registierung von Services (siehe Abschnitt 3.2.2 auf Seite 34) gibt es aber oft Drittanbieter-Bibliotheken, welche genutzt werden sollen. Geteilten Code für die Kommunikation der Services empfehlen wir hingegen nur unter den von Kraus u. a. [2013] genannten Bedingungen zu verwenden.

3.2.7. Betrieb

Durch den Betrieb von Microservices als verteilte Systeme entstehen spezielle Anforderungen an die Ausfallsicherheit und die Auslieferungsmethoden. Dafür entstehen neue Möglichkeiten, das System zu skalieren. Diese Anforderungen und Skalierungsmöglichkeiten stellen wir in diesem Kapitel vor.

Skalierbarkeit

Abbott und Fisher [2009] stellen mit dem Skalierungsquader ein grafisches Modell zur Unterteilung von Skalierungsmöglichkeiten vor. Das Volumen symbolisiert die Rechenkapazität und wie das Volumen kann die Rechenkapazität anhand der Seitenlängen des Quaders in drei Richtungen skaliert werden. Auf der X-Achse werden die Instanzen der Anwendungslogik dupliziert. Auf der Y-Achse wird das System nach Funktionalität aufgeteilt. Auf der Z-Achse werden ebenfalls Duplikate der Logik verwendet, die hier aber nur für nach einem anwendungsspezifischem Kriterium festgelegten Untermengen der Anfragen zuständig sind. Der Schnitt eines Systems in Microservices entspricht also der Skalierung in Y-Richtung, da die Funktionalität verteilt wird. Microservice-Systeme können aber auch die anderen Skalierungsrichtungen verwenden, um stark ausgelastete Services zu erweitern.
Abbildung 3.13 zeigt, wie die Vorbereitungsphasen der semantischen Analyse anhand der drei Achsen skaliert werden. Auf der Y-Achse erfolgt die Unterteilung in Spracherkennung, Tokenisierung, lexikalische Analyse und Syntaktische Analyse. Die Tokenisierung wird in Z-Richtung skaliert, indem Texte anhand ihrer Sprache an verschiedene Instanzen weitergeleitet werden. Die Instanzen unterscheiden sich nicht in ihrer Implementierung, da sonst eine weitere funktionale Skalierung in Y-Richtung vorliegen würde. Die lexikalische Analyse wird in diesem Beispiel nicht weiter skaliert, während die syntaktische Analyse auf der X-Achse skaliert wird, indem zusätzliche Instanzen erstellt werden.
In Abschnitt 3.3 sehen wir, dass die Vorbereitungsphasen keine Datenhaltung benötigen. Daher können sie sehr einfach in X-Richtung skaliert werden, während die Skalierung in Z-Richtung nur zusätzlichen Rechenaufwand benötigen würde. Für die im Beispiel genannte Unterteilung nach Sprachen schlagen wir wie in Abschnitt 3.1.3 auf Seite 19 beschrieben die Einführung separater Microservices und somit Skalierung in Y-Richtung vor.

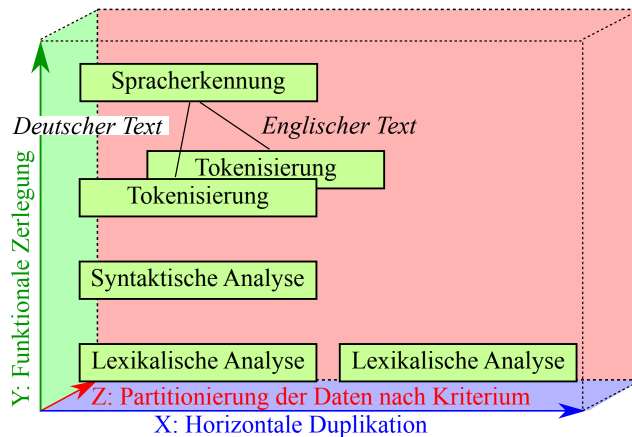


Abbildung 3.13. Skalierungsquader mit Beispielen der Textanalyse

Die Skalierung von Microservices mit Datenhaltung erfordert, dass die Datenhaltung ebenfalls skaliert wird, damit kein Engpass beim Zugriff auf eine gemeinsame Datenbank entsteht. Hierfür empfiehlt Newman [2015] skalierende Datenbanken zu verwenden. Falls es keine geeignete skalierbare Datenbanken für eine bestimmte Repräsentationsform von Wissen gibt, kann Skalierung in Z-Richtung erwogen werden. Dies funktioniert aber nur, falls es ein Kriterium gibt, nach dem sich die Arbeitsdaten so unterteilen lassen, dass Instanzen eines Service immer nur die eigenen Daten benötigen.

Auslieferung

Auslieferungsmuster beschreiben, wie Microservices auf physikalischen oder virtuellen Maschinen betrieben werden. Richardson [2016b] diskutiert als ein Auslieferungsmuster das *Multiple-Services-per-Host*-Muster, bei dem Instanzen verschiedener Services auf einer Maschine betrieben werden. Ein anderes Muster ist das *Service-Instanz-per-Host*-Muster, dass auf jeder Maschine nur einen Service erlaubt. Die dritte Alternative ist eine Server-freie Auslieferung, wie sie zum Beispiel durch *AWS Lambda*[7] ermöglicht wird. Dabei werden Microservices in einer gemeinsamen Umgebung auf einer Cloud-Struktur betrieben. Dies fällt allerdings nach Newman [2015] eher in den Bereich der in Abschnitt 2.3 auf Seite 12 beschriebenen Nanoservices. Auf die Vor- und Nachteile der Muster gehen wir hier nicht weiter ein, da sich diese vor allem auf den Betrieb des Systems und weniger auf seine Architektur auswirken.
Serviceinstanzen manuell zu installieren und einzurichten verhindert nicht nur automatisierte Skalierung, sondern bedeutet auch, dass der Installateur immer genau wissen muss, was die einzelnen Microservices für Technologien benötigen und wie sie aufgesetzt werden.

[7] `https://aws.amazon.com/de/lambda/details/`

Der Prozess kann erleichtert werden, indem Instanzen mit allen benötigten Artefakten zur Auslieferung gekapselt werden. Dafür eignen sich Abbilder virtueller Maschinen und Container, die Richardson [2016b] ebenfalls gegeneinander abwägt.

Ausfallsicherheit

Microservices sind nach Wolff [2016a] eigene Prozesse und müssen daher über andere Wege kommunizieren als Komponenten monolithischer Anwendungen. Was für Möglichkeiten es gibt, beschreiben wir in Abschnitt 3.2.1. Unabhängig von der genutzten Methode kann es zu Teilausfällen des Systems kommen, wenn zum Beispiel einzelne Microservices gewartet werden, überladen sind, noch gestartet werden müssen oder durch einen Fehler ausfallen. Daher müssen Maßnahmen getroffen werden, damit das Fehlschlagen von Anfragen erkannt wird und nicht dazu führt, dass weitere Services ausfallen.
Christensen [2012] schlägt verschiedene Maßnahmen vor, um Ausfallsicherheit zu gewährleisten: Alle Anfrage mit Timeouts zu versehen gewährleistet, dass Services nur begrenzte Zeit aufeinander warten. Fehlgeschlagene Abfragen können so erkannt und geeignet verarbeitet werden, ohne dass der wartende Service zu lange von einem nicht arbeitendem Service blockiert wird. Wenn bereits eine gewisse Anzahl von Anfragen offen ist, ist es wahrscheinlich, dass der andere Service die Anfrage ohnehin nicht bearbeiten kann. Daher sollte die Anzahl der offenen Anfragen an einen anderen Service limitiert werden. Wenn der Empfänger zu sehr belastet ist, würden weitere Anfragen den Zustand eher verschlechtern und das Versagen dieses Service fördern. Als weitere Maßnahme werden *Circuit-Breaker*-Muster [Nygard 2007] vorgeschlagen. Hierbei wird der Anteil der erfolgreich beantworteten Anfragen gemessen und ein bestimmter Anteil als Grenzwert festgelegt. Wenn die Zahl unter diesen Grenzwert sinkt, werden für eine Zeit lang alle Anfragen direkt als fehlerhaft gewertet. Dadurch wird nicht auf Anfragen gewartet, die wahrscheinlich einen Fehler produzieren würden. Das hält das System reaktiver.
Diese Methoden können die Fehlererkennung verbessern, aber keine Fehler verhindern. Daher müssen Microservices auf das Fehlschlagen ihrer Anfragen reagieren können und entweder auch einen Fehler melden oder Standardwerte oder Werte aus einem Zwischenspeicher verwenden. Alternativ kann die Anfrage an eine andere Instanz des Services gesendet werden, falls das Client-seitige Entdeckungsmuster (siehe Abschnitt 3.2.2) verwendet wird.
Da Systeme zur semantischen Analyse wie in Abschnitt 3.2.4 beschrieben häufig in andere Systeme integriert und nicht direkt von Menschen genutzt werden, können Fehler auf der Ebene des aufrufenden Systems behandelt werden. Das erlaubt dem Microservice-System, Anfragen mit Fehlern zu beantworten und somit den Umgang mit Fehlern dem nutzenden System zu überlassen. So können verschiedene Anwendungen, die das selbe Microservice-System nutzen, auf eine jeweils passende Art reagieren.

3.3. Architekturbeschreibung

In diesem Abschnitt stellen wir den Architekturentwurf vor, der aus den Schlüssen der vorangegangenen Abschnitte resultiert. Wir zeigen in Abschnitt 3.3.1 die Gesamtarchitektur ohne Details der einzelnen Services, gehen dann in Abschnitt 3.3.2 auf den internen Aufbau der Services ein und erklären dann die Koordination und Kommunikation der Microservices.
In der folgenden Liste befinden sich die Architekturentscheidungen zusammengefasst:

- ▻ **Integration der Vorbereitungsphasen**: Diverse Vorbereitungs-Microservices siehe Abschnitt 3.1.3 auf Seite 19
- ▻ **Partitionierungsmethode**: Nach Methoden der semantischen Analyse siehe Abschnitt 3.1.4 auf Seite 28
- ▻ **Kommunikationsart**: Asynchrone Notifications ohne Messaging-System (One-to-One) siehe Abschnitt 3.2.1 auf Seite 29
- ▻ **Nachrichtenformat**: Textbasiert, siehe Abschnitt 3.2.1 auf Seite 31
- ▻ **Schnittstellendefinition**: Durch Schemata, siehe Abschnitt 3.2.1 auf Seite 31
- ▻ **Koordination**: Asynchrone direkte Aufrufe, siehe Abschnitt 3.2.3 auf Seite 35
- ▻ **Entdeckung**: Serverseitig, siehe Abschnitt 3.2.2 auf Seite 32
- ▻ **Registrierung**: Selbstregistrierung, siehe Abschnitt 3.2.2 auf Seite 34
- ▻ **GUI**: Nur zum Testen, siehe Abschnitt 3.2.4 auf Seite 40
- ▻ **Ergebniszusammenführung**: Durch den Empfänger des asynchronen Aufrufs, siehe Abschnitt 3.2.4 auf Seite 40
- ▻ **Datenhaltung**: Skalierbare Datenbanken, die von mehreren Instanzen geteilt werden, siehe Abschnitt 3.2.5 auf Seite 41
- ▻ **Geteilter Code**: Nur innerhalb Microservices, Drittanbieter- oder öffentliche Bibliotheken für Fehlertoleranz, Registrierung und Entdeckung, siehe Abschnitt 3.2.6 auf Seite 43
- ▻ **Skalierung**:
  - ▻ **Duplizierung (X)**: Durch Datenhaltung ermöglicht
  - ▻ **Funktionale Zerlegung (Y)**: Microservices
  - ▻ **Partitionierung nach Kriterium (Z)**: Zum Beispiel durch Spracherkennung (siehe Abschnitt 3.3.1 und Abschnitt 3.3.2)

  Erläuterungen in Abschnitt 3.2.7 auf Seite 44

3.3.1. Gesamtbild der Architektur

In Abschnitt 3.1 stellen wir verschiedene Ansätze zur Integration der Vorbereitungsphasen der semantischen Analyse und zur Parititonierung der semantischen Analyse selbst vor. Abbildung 3.14 zeigt, wie die von uns präferierten Ansätze eine Microservice-Architektur bilden können.

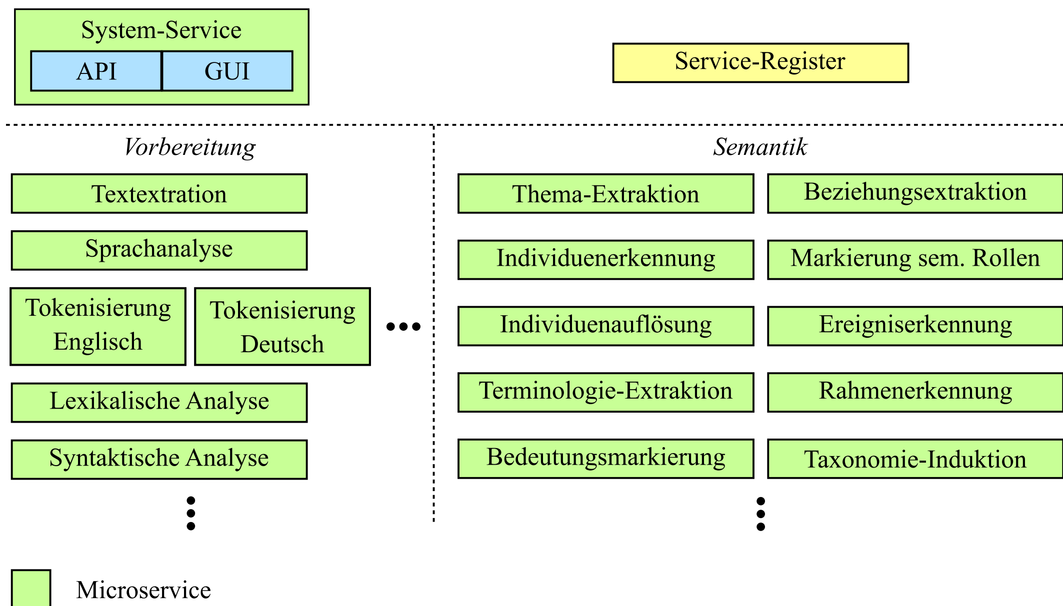


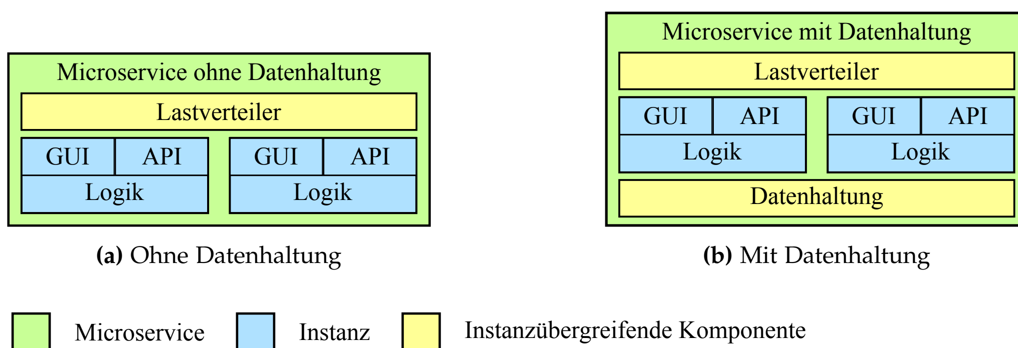
Abbildung 3.14. Microservice-Architektur für die semantische Analyse

Es gibt eine Verwaltungs- und Zugriffsschicht, die im oberen Teil der Abbildung dargestellt ist. Sie beinhaltet einen *System-Service*, der das System für die Integration in andere Systeme kapselt. Er stellt eine API sowie eine optionale GUI bereit. Über die API nimmt der System-Service Anfragen an das Textanalyse-System entgegen, sendet diese asynchron an den ersten Microservice und wartet auf das fertige Ergebnis. Zusätzlich zu diesem Service befindet sich das Service-Register auf dieser Ebene.
Die Funktionalität des Systems teilt sich in zwei Gruppen von Microservices, die links und rechts dargestellt und durch die senkrechte Mittellinie getrennt sind. Eine Gruppe umfasst Services, die die Vorbereitungsphasen bilden. Dazu können neben den von Dale [2010] vorgestellten Vorbereitungsphasen auch weitere Services, wie Spracherkennung und Textextraktion, gehören. Die Punkte symbolisieren hier mögliche weitere Services in dieser Kategorie. Wenn die Implementierung einzelner Funktionen stark sprachabhängig ist, können die Phasen noch weiter nach den jeweils zugehörigen Sprachen unterteilt werden, wie in der Grafik am Beispiel der Tokenisierung gezeigt.

Auf der rechten Seite der Grafik befinden sich die Microservices für die semantische Analyse. Hierbei gibt es, wie in Abschnitt 3.1.4 vorgeschlagen, je einen Service pro Methode der semantischen Analyse. Da die Liste der Methoden der semantischen Analyse von Gangemi [2013] nicht vollständig sein muss und nicht jedes System alle Funktionen enthält, sind auch hier durch die Punkte weitere mögliche Services angedeutet.

3.3.2. Interner Aufbau der Microservices

Abbildung 3.15 zeigt den internen Aufbau der Microservices. In Abbildung 3.15a ist ein Beispiel für einen Microservice dargstellt, der keine Datenhaltung benötigt. Der Microservice in Abbildung 3.15b hingegen hängt von seinen Arbeitsdaten ab. Es gibt in jedem Service Komponenten, die bei der Duplizierung der Instanzen nicht mit verdoppelt werden sollen. Daher bezeichnen wir die Instanzen der zu duplizierenden Komponenten als Instanzen des Microservice. Jeder Service ist mit zwei Instanzen dargestellt, um die Skalierung durch Duplizierung von Instanzen der Anwendungslogik zu verdeutlichen.



(a) Ohne Datenhaltung

(b) Mit Datenhaltung

Abbildung 3.15. Interner Aufbau der Microservices der Textanalyse

Um serverseitige Entdeckung zu ermöglichen, benutzen wir Lastverteiler. Wir verwenden einen Lastverteiler pro Microservice und schließen den Lastverteiler in den Service ein. Auf diese Weise wird verdeutlicht, dass die Lastverteilung Aufgabe des Services ist. Die Lastverteiler verwenden das gemeinsame Service-Register, um Instanzen der Services zu entdecken.

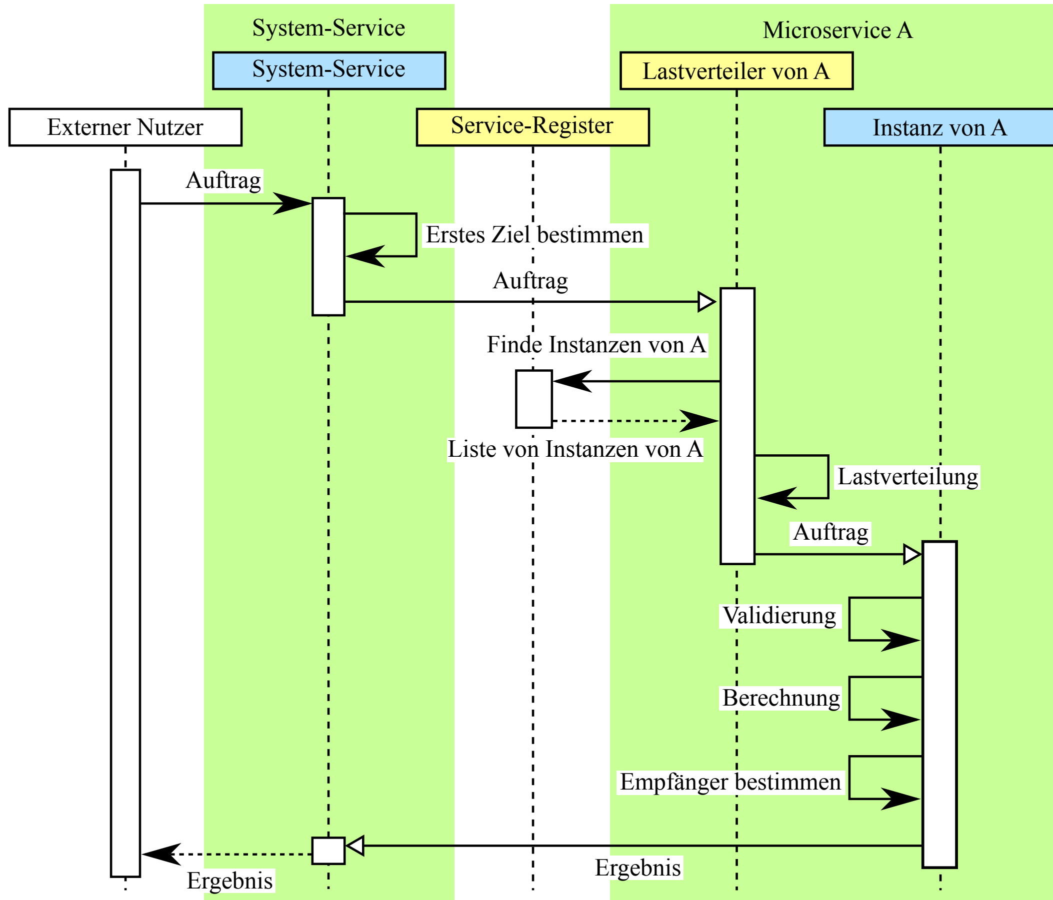
Innerhalb der Service-Instanzen ist die oberste Schicht eine Zugriffsschicht. Sie wird durch eine API und eine optionalen GUI gebildet, durch die auf die Daten und Funktionen des Microservice zugegriffen werden kann. Die eigentliche Programmlogik befindet sich in der darunterliegenden Schicht. Diese wird verwendet, um die Ergebnisse der Methode der semantischen Analyse zu berechnen.

Die Datenhaltung ist eine weitere instanzübergreifende Komponente. Sie sollte wie in Abschnitt 3.2.7 beschrieben durch eine skalierbare Datenbank realisiert werden.

3.3.3. Koordination und Kommunikation

Die Koordination der Microservices geschieht wie in Abschnitt 3.2.3 auf Seite 35 beschrieben durch direkte asynchrone Aufrufe. Nachrichten zwischen den Services enthalten neben dem Text auch den ursprünglichen Auftrag. Anhand des Auftrags kann jeder Service selbst bestimmen, an welchen anderen Service er seine Ergebnisse weiterleitet. Als Format für die Nachrichten wählen wir textbasierte Formate.
Abbildung 3.16a zeigt ein Beispiel für die Koordination und Kommunikation in unserer Architektur. Im Vorfeld hat sich die Instanz von Microservice A nach dem Muster Selbstregistrierung beim Service-Register angemeldet. Der Lastverteiler von A und das Service-Register sind durch zum Beispiel DNS erreichbar.
Zu Beginn startet ein externer Nutzer einen synchronen Aufruf des Analyse-Systems. Diesen Aufruf nennen wir Auftrag. Er enthält den Text, sowie die auszuführende Analyse. Der System-Service berechnet aus der auszuführenden Analyse den ersten benötigten Schritt und sendet den Auftrag an die Adresse des zugehörigen Microservice. Unter dieser Adresse ist der Lastverteiler von Microservice A erreichbar. Er nimmt die Nachricht entgegen und ruft beim Service-Register Instanzen vom Microservice A ab. Anschließend führt er Berechnungen zur Lastverteilung durch und findet so eine verfügbare Instanz. An diese Instanz leitet er den Auftrag anschließend weiter. Die Instanz validiert den Auftrag mittels Schema und führt ihre Berechnungen durch. Danach ermittelt die Instanz von Service A den nächsten Empfänger. Das ist in diesem Fall wieder der System-Service, da die einzige angeforderte Berechnung abgeschlossen wurde. Der System-Service nimmt das Ergebnis entgegen, ordnet es dem externen Nutzer zu und antwortet dem Nutzer mit dem Ergebnis.

In diesem Kapitel haben wir Grundlagen zur semantischen Analyse und Microservice-Architekturen erläutert. Auf dieser Basis haben wir unsere Microservice-Architektur für die semantische Analyse entwickelt. Aufgrund der Designentscheidungen sollen die enthaltenen Microservices autonom sein und möglichst unabhängig voneinander geändert und ausgeliefert werden können. Durch diese Unabhängigkeit ist die Kommunikation und Abstimmung bei der Entwicklung gegenüber monolithischen Ansätzen geringer. Daher skaliert der Entwicklungs- und Wartungsprozess besser mit steigendem Funktionsumfang. Im nächsten Kapitel 4 präsentieren wir die anhand der Architektur erstellte Implementierung StanbolµS. Diese verwenden wir anschließend in Kapitel 5 auf Seite 61, um die Einflüsse der Microservice-Architekturstil auf das Verhalten zur Laufzeit zu analysieren.



(a) Sequenzdiagramm des Ablaufs, Beispiel für A ist Spracherkennung

```
{
 "Auftrag":"Spracherkennung",
 "Text":"Dies ist ein Beispieltext"
}
```

(b) Beispiel für *Auftrag* in JSON

```
{
 "Auftrag":"Spracherkennung",
 "Text":"Dies ist ein Beispieltext",
 "Sprache":"DE"
}
```

(c) Beispiel für *Ergebnis* in JSON

Abbildung 3.16. Beispielhafter Ablauf eines Aufrufs des Systems zur Textanalyse: Über die System-API wird Service A aufgerufen

Kapitel 4

Die Implementierung *StanbolµS*

In diesem Kapitel stellen wir die Architektur von StanbolµS vor. StanbolµS ist die von uns erstellte Implementierung nach den Empfehlungen in Kapitel 3 ab Seite 17. Der Name setzt sich aus Stanbol und µs als Abkürzung für Microservice zusammen.
Als Grundlage für die Funktionen der Textverarbeitung benutzen wir das in Abschnitt 2.6 auf Seite 13 vorgestellte Apache Stanbol. Wir geben im Folgenden einen Überblick über die Architektur und gehen auf die Umsetzung der Konzepte wie Entdeckungs- und Registrierungsmuster ein. Details wie die Ordnerstruktur sowie Installations- und Betriebsanleitung befinden sich in Anhang A.

4.1. Architektur von StanbolµS

Apache Stanbols Funktionen zur semantischen Analyse von Texten ist, wie in Abschnitt 2.6 auf Seite 13 beschrieben, in den Enhancement-Engines enthalten. Nach der Partitionierungsmöglichkeit nach Methoden der semantischen Analyse aus Abschnitt 3.1 auf Seite 17 nutzen wir diese Funktionen als Kriterium zur Erstellung unserer Partitionen.

Abbildung 4.1 zeigt die Komponenten der Implementierung. Sie enthält unter anderem fünf Microservices, deren Funktionalität Enhancement-Engines entsprechen. Diese nennen wir Enhancement-Services:

1. *LangDetect* erkennt die Sprache eines Texts.

2. *OpenNLP-Sentence* findet Satzgrenzen.

3. *OpenNLP-Token* tokenisiert den Text.

4. *OpenNLP-POS* reichert den Text um Informationen zu den Wortarten an.

5. *OpenNLP-NER* erkennt im Text genannte Individuen.

Diese Services rufen sich in der genannten Reihenfolge sequentiell auf (siehe Abschnitt 4.1.1). Zusätzlich gibt es den Service *System*, der Schnittstellen für externe Nutzer bereitstellt. Ein Service-Register dient der dynamischen Entdeckung der Services.

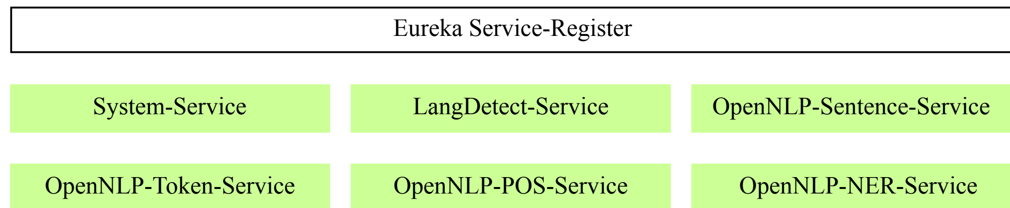


Abbildung 4.1. Komponenten der Implementierung

Der System-Service besteht, wie in Abbildung 4.2a dargestellt, aus einer angepassten Stanbol-Version. Der *Sender* ist eine von uns erstellte Enhancement-Engine, welche eingehende Texte an einen anderen Service übermittelt und anschließend darauf wartet, dass eine verarbeitete Form des Texts mit semantischen Informationen eintrifft. Es können mehrere Aufrufe des Senders parallel erfolgen. Um eingehende Texte entgegen zu nehmen, enthält der System-Service außerdem die Komponente *Response-Listener*, welche einen Empfangspunkt für eingehende Texte bereitstellt und diese an darauf wartende Sender-Komponenten weiterleitet. Diese orange gekennzeichneten Komponenten sind als OSGi-Module in Stanbol integriert, um das vorhandene System zur Modularisierung zu nutzen.

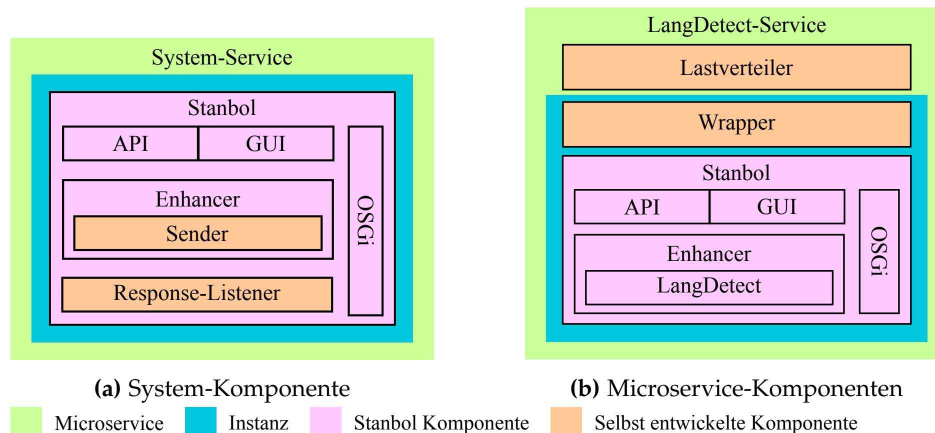


Abbildung 4.2. Interner Aufbau der Microservices von StanbolµS

Die anderen Microservices bestehen aus den in Abbildung 4.2b vorgestellten drei Komponenten: Lastverteiler, Wrapper und einer Stanbol-Instanz. Jede Instanz eines Service besteht aus einem Wrapper und einer Stanbol-Instanz. Der Lastverteiler verteilt Anfragen an die Wrapper seiner Instanzen. Der Wrapper übernimmt die Registrierung bei dem Service-Register und kapselt die Nutzung von Stanbol: Er nimmt alle Aufrufe sei-

ner Microservice-Instanz entgegen, leitet sie dann an seine Stanbol-Instanz weiter und übermittelt anschließend den angereicherten Text an den Lastverteiler eines anderen Microservice. Auf diese Weise kann Stanbol leicht gegen eine andere Implementierung der gleichen Methode der semantischen Analyse ausgetauscht werden. Die Stanbol-Instanzen sind so angepasst, dass sie nur eine einzelne Enhancement-Engine enthalten, welche der Funktionalität des Microservice entspricht.

4.1.1. Umsetzung der Konzepte in StanbolµS

In den folgenden Unterabschnitten beschreiben wir die Umsetzung der Konzepte aus Abschnitt 3.2 ab Seite 29.

Partitionierung

Die Aufteilung des Systems entspricht der Partitionierung nach Methoden der semantischen Analyse. Die Phasen LangDetect, OpenNLP-Sentence, OpenNLP-Token und OpenNLP-POS gehören hierbei zu den Vorbereitungsphasen, die nicht Teil der eigentlichen semantischen Analyse sind. Die Vorbereitungsphasen sind also ebenfalls als verschiedene Microservices implementiert, wie wir auf Seite 20 in Abbildung 3.1c illustrieren. Der OpenNLP-NER-Service ist der einzige Service der kein Vorbereitungsservice ist und somit zu den Semantik-Microservices gehört (siehe Abbildung 3.14 auf Seite 48).

Nachrichtenformat

Als Kommunikationsprotokoll verwenden wir HTTP. Die Nachrichten zwischen den Microservices und zwischen Lastverteiler und Wrapper eines Microservices sind POST-Nachrichten. Sie beinhalten HTTP-Header für die Ausführungsreihenfolge und einen Identifikator für den Text. Im Nachrichtenrumpf befindet sich ein serialisiertes Stanbol-Objekt, welches den Text und errechnete Informationen zu der Struktur und Semantik des Texts enthält. Die genaue Form ist durch den in Stanbol enthaltenen Serialisierer bestimmt. Um die Ergebnisse der Phasen OpenNLP-Sentence, OpenNLP-Token und OpenNLP-POS serialisieren und wieder zu einem Stanbol-Objekt umwandeln zu können, haben wir die dafür zuständige Komponente von Stanbol angepasst. Die Anpassung ist in Anhang A dokumentiert.

Schnittstellendefinition

Die Schnittstellendefinition ist durch die im vorherigen Unterabschnitt geschilderte Beschreibung des Nachrichtenformats und die Adressen der Lastverteiler gegeben. Eine explizite Definition durch dafür vorgesehene Sprache oder Schemata ist von uns nicht umgesetzt, da sie für die Entwicklung und Evaluierung nicht benötigt wird.

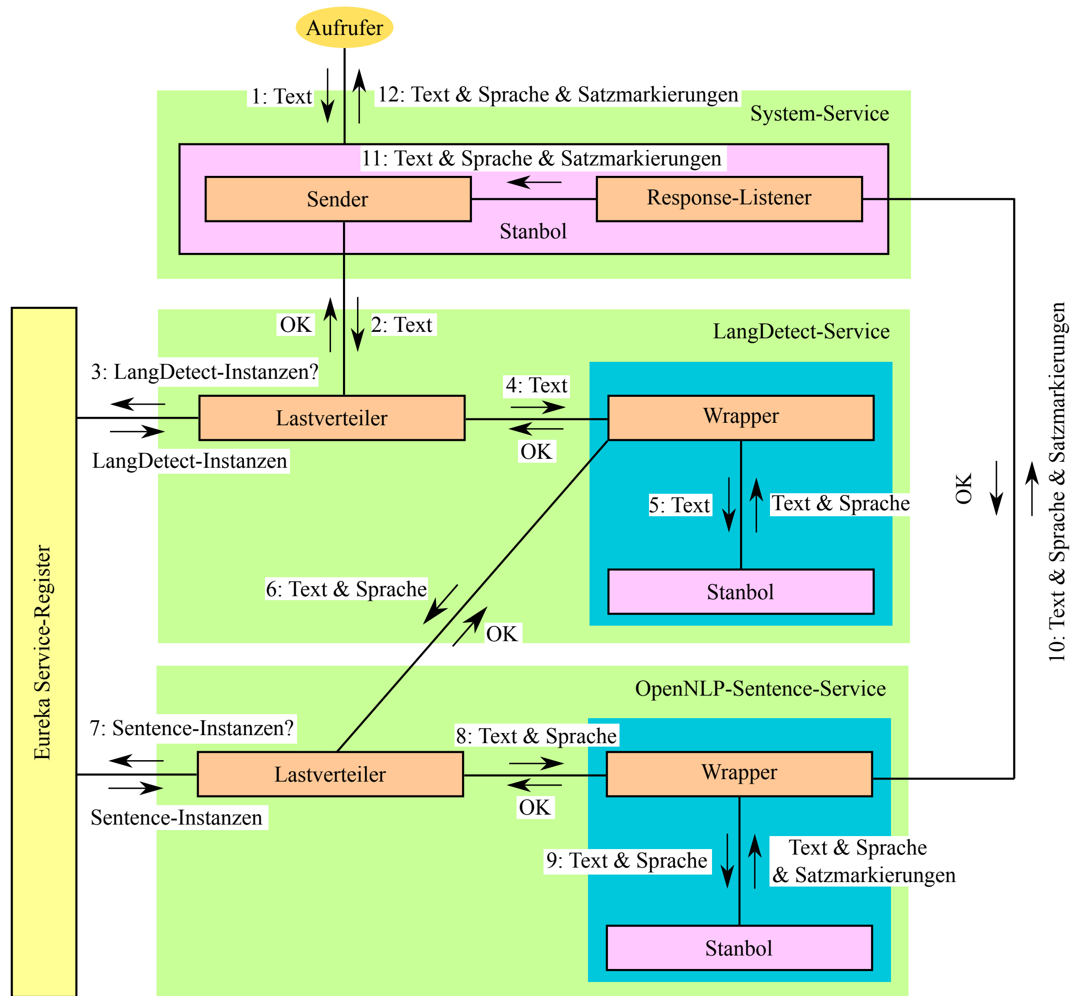


Abbildung 4.3. Kommunikation der Komponenten in StanbolμS

Kommunikation

Abbildung 4.3 zeigt die Kommunikation der Komponenten für den Fall, dass der LangDetect-Service zur Spracherkennung und der OpenNLP-Sentence-Service zum Erzeugen von Satzmarkierungen genutzt werden. Die Nachrichten sind anhand ihrer zeitlichen Abfolge nummeriert. Eine Nachricht ohne Nummerierung ist die synchrone Antwort auf die auf dem anderen Ende der Linie stehenden Nachricht.
Der Aufrufer übermittelt Stanbol über die REST-API oder die GUI einen Text. Die Sender-Enhancement-Engine registriert sich daraufhin bei dem Response-Listener für Nachrichten

in Bezug auf diesen Text. Hierfür wird eine von Stanbol erstellte URI als Identifikator des Texts verwendet. Die Enhancement-Engine übermittelt den Text dann in Nachricht 2 mittels HTTP an den Lastverteiler des zuständigen Microservice. Das ist in diesem Fall der LangDetect-Service. Der Identifikator des Texts wird als HTTP-Header in die Nachricht integriert. Der Lastverteiler bestätigt die Antwort mit dem Statuscode 200 (OK) und verwendet Eurekas REST-Schnittstelle, um Instanzen des LangDetect-Services zu finden. Eureka antwortet mit einer Liste von Instanzen. Der Lastverteiler ermittelt eine geeignete Instanz und sendet den Text ebenfalls mittels HTTP als Nachricht 4 an den Wrapper der Instanz. Dieser beantwortet den Empfang mit 200 (OK) und erstellt aus der erhaltenen Anfrage eine HTTP-Nachricht, die er an die REST-Schnittstelle seiner Stanbol-Instanz sendet. Stanbol beantwortet diese Anfrage mit dem Text und den errechneten Zusatzinformationen. Der Wrapper erstellt aus dieser Antwort die neue HTTP-Nachricht 6, welche an den nächsten Service weitergeleitet wird. Der ist in diesem Fall der Lastverteiler des OpenNLP-Sentence-Services.
Dieser erfragt mit Nachricht 7 Instanzen für seinen Service beim Eureka-Service-Register und leitet die Anfrage an den Wrapper einer der Instanzen weiter. Der Wrapper generiert wiederum die Nachricht 9 für seine Stanbol-Instanz. Die Stanbol-Instanz erzeugt die Satzmarkierungen und sendet die gesamten Daten zurück an den Wrapper. Der Wrapper schickt mit Nachricht 10 den Text, die erkannte Sprache und die Satzmarkierungen an den Response-Listener des System-Services.
Der Response-Listener bestätigt den Empfang des Texts mit 200 (OK), liest den Identifikator des Textes aus dem HTTP-Header und aktiviert den für diesen Identifikator registrierten Sender. Die Sender-Enhancement-Engine beantwortet nun die Anfrage des ursprünglichen Aufrufers.
Die Übermittlung des Textes wird also innerhalb des Systems direkt beantwortet und die Weiterverarbeitung oder Vermittlung des Texts erfolgt im Anschluss. Dadurch bilden wir mit dem synchronen Protokoll HTTP ein asynchrone Notifications nach. Zu den Ausnahmen gehört die Anfrage des Aufrufers, welche blockiert, bis das Ergebnis vorliegt. Die Nutzung des Service-Registers durch den Lastverteilers ist ebenfalls synchron. Der Aufruf von Stanbol durch den Wrapper erfolgt synchron in einem eigenen Thread, sodass mehrere Anfragen parallel bearbeitet werden können.

Koordination

Um die Reihenfolge der Verarbeitung eines Textes durch die Microservices festzulegen, wird der System-Service mit einer System-Variable gestartet, die eine Komma-separierte Liste der Adressen der Lastverteiler enthält. Die Adressen können hierbei IP-Adressen oder Domains sein, die über das DNS aufgelöst werden. Die Verwendung von Domains ermöglicht es, auch Lastverteiler im Betrieb ändern zu können, ohne die Konfiguration der System-Service Instanzen erneuern zu müssen. Der letzte Eintrag der Adressliste ist der Zugangspunkt des Response-Listeners.

Mit dieser Liste bestimmt der System-Service die Ausführungsreihenfolge. Die anderen Services wissen nur anhand dieser Liste, wohin die Nachricht als nächstes weitergeleitet wird. Dadurch sind verschiedene Konfigurationen möglich, ohne die Enhancement-Services anzupassen und das gesamte System ist universeller Einsetzbar.
Der Text wird vom System-Service an den ersten Eintrag der Liste übermittelt. Dabei wird die Liste als Header in die Nachricht integriert. Der empfangende Lastverteiler übermittelt die Nachricht inklusive des Headers an einen zugehörigen Wrapper. Dieser entfernt den ersten Eintrag der Adressliste und lässt den Text von seiner Stanbol-Instanz verarbeiten. Dann schickt er das Ergebnis mit der gekürzten Adressliste an den neuen ersten Eintrag der Liste, welcher auf die gleiche Weise vorgeht. Da der letzte Eintrag in der Liste der Response-Listener ist, wird die letzte Nachricht an diesen gesendet. Er liest den Identifikator des Texts aus dem Identifikator-Header der Nachricht und leitet die mit semantischen Informationen angereicherte Form des Texts an die Sender-Enhancement-Engine weiter, die die Verarbeitung gestartet hat.

Registrierung

Der System-Service registriert sich nicht beim Service-Register. Die Instanzen der anderen Microservices werden durch ihre Wrapper angemeldet. Diese Anmeldung geschieht über die REST-Schnittstelle des Service-Registers. Da sich jeder Service so selbst anmeldet, ist das Muster der Selbstregistrierung genutzt. Die Lastverteiler registrieren sich nicht, da sie über ihre Domain oder IP-Adresse gefunden werden.

Entdeckung

Nach dem serverseitigen Entdeckungsmuster finden sich Microservices gegenseitig nur über ihre Lastverteiler. Da nur eine Instanz des Service-Registers verwendet wird, könnten Instanzen der Microservices sich darüber auch direkt finden. Um clientseitige Lastverteilung und damit ein clientseitiges Entdeckungsmuster zu vermeiden, verwenden wir diese direkte Entdeckung der Service-Instanzen untereinander nicht.

Grafische Benutzeroberfläche

Der System-Service und die Stanbol-Instanzen der anderen Microservices bieten die in Stanbol enthaltene GUI. Die Oberfläche des System-Services reicht mit den unten beschriebenen Maßnahmen zur Fehlerbehandlung zum Testen aus. Daher haben wir für die Wrapper und Lastverteiler keine GUI implementiert.

Ergebniszusammenführung

Die Ergebniszusammenführung erfolgt durch die Stanbol-Instanzen. Jede Instanz ergänzt die errechneten Informationen zu den bereits vorhandenen. Da die Ausführung der Implementierung sequentiell ist, wird so keine weitere Zusammenführung benötigt.

Datenhaltung

Bei Nutzung der REST-Schnittstelle von Apache Stanbol werden keine Daten serverseitig gespeichert.[1] Die Stanbol-Instanzen des System-Services und des LangDetect-Service enthalten daher keine Datenhaltung. Die OpenNLP-Services hingegen benötigen Stanbols Entityhub. Dieser wird dort genutzt, um Arbeitsdaten abzulegen. Diese Daten werden bei dem Erstellen der Stanbol-Instanz in die Datenhaltung integriert. Im Betrieb wird die Datenhaltung jedoch ausschließlich lesend verwendet. Daher treten zur Laufzeit keine Veränderungen auf, die bei getrennter Datenhaltung zwischen den Instanzen zu verschiedenen Ergebnissen führen würden. Eine gemeinsame Datenhaltung für alle Instanzen eines Microservice ist somit auch für die OpenNLP-Services nicht nötig und von uns nicht umgesetzt.

Skalierung

In Abschnitt 3.2.7 auf Seite 44 stellen wir anhand des Skalierungsquaders die drei Skalierungsrichtungen vor. Auf der X-Achse liegt dabei die Horizontale Duplikation. Der Systen-Service und die Instanzen der anderen Services haben keine Datenhaltung und werden dynamisch über das Service-Register gefunden. Daher ist die Duplikation dieser Komponenten ohne weitere Anpassungen möglich. Die Lastverteiler müssen über eine bestimmte Adresse erreichbar sein und können so nur unter Verwendung eines weiteren Lastverteilers erfolgen. Die Y-Achse des Quaders ist die funktionale Zerlegung, welche durch die Aufteilung des Systems in verschiedene, funktional unterschiedliche Services gegeben ist. Die Partitionierung der Daten nach Kriterium auf der Z-Achse nutzen wir nicht.

Fehlerbehandlung

Stanbol gibt bei der Benutzung der grafischen Oberfläche Fehlermeldungen über die Oberfläche aus. Bei Verwendung der REST-API werden HTTP-Statuscodes als Antwort verwendet, welche den Fehlergrund beschreiben. Eine detailliertere Fehlerbeschreibung befindet sich im Nachrichtenrumpf. Wenn die Enhancement-Engine Sender einen Fehler feststellt, wird dieses Standard-Verhalten von Stanbol ausgelöst. Wenn der Response-Listener über eine Nachricht einen Fehler zu einem Text erhält, leitet er diesen Fehler an dafür registrierte Aufrufe des Senders weiter. Der Sender wartet außerdem nur eine bestimmte Zeit auf Antworten. Nach Ablauf der Zeit wird ein Fehler angenommen. So ist sichergestellt, dass Anfragen nach außen auch im Fehlerfall beantwortet werden.
Falls bei der Übermittlung eines Texts über den Lastverteiler an einen Wrapper ein Fehler auftritt, weil zum Beispiel benötigte HTTP-Header fehlen, wird die Anfrage mit HTTP-Statuscode 400 (Bad Request) und einer Fehlerbeschreibung beantwortet.

[1] `https://stanbol.apache.org/docs/trunk/components/enhancer/enhancerrest.html`

Wenn ein Wrapper von seiner Stanbol-Instanz einen Fehler gemeldet bekommt oder eine Anfrage nicht weiterleiten kann, versucht der Wrapper, dieses an den letzten Eintrag der Adressliste zu melden. Da dieser Eintrag der Response-Listener des System-Services ist, erkennt die Stanbol-Instanz des System-Services so auch Fehler, die bei der Verarbeitung des Texts auftreten. Der Aufrufer erhält so genaue Informationen, warum die Verarbeitung fehlgeschlagen ist. Dies hilft, Fehler in der Anfrage oder in der Anwendung zu entdecken.

In diesem Kapitel haben wir mit StanbolµS die Implementierung der in Kapitel 3 vorgeschlagenen Architektur vorgestellt. StanbolµS wird im nächsten Kapitel 5 zur Evaluierung der Architektur verwendet und evaluiert.

Kapitel 5

Evaluierung

In unserer Zielsetzung in Abschnitt 1.2 auf Seite 4 schreiben wir, dass wir neben der Effizienz auch Wartbarkeit und Übertragbarkeit evaluieren wollen, falls möglich. Die Evaluierung von Wartbarkeit und Übertragbarkeit würde umfangreichere Experimente erfordern, die den Rahmen dieser Arbeit überschreiten würden. Wir konzentrieren uns daher auf die Effizienz.
Wie ebenfalls in der Zielsetzung beschrieben, benutzen wir das Goal-Question-Metric-Verfahren [Basili 1992] zum Planen unserer Evaluierung. Bei diesem Verfahren werden zunächst die Ziele der Evaluierung festgelegt. Zu jedem Ziel werden Fragen erfasst, die bei dem Erreichen des Ziels helfen. Die Fragen definieren, welche Eigenschaften gemessen werden sollen, beziehungsweise welche Frage durch eine Messung beantwortet werden soll. Zu den Fragen werden anschließend Metriken entwickelt. Diese Metriken sind Messgrößen, um die Fragen beantworten zu können.
Die für dieses Kapitel erstellten Messergebnisse befinden sich auf dem beiliegenden Datenträger. Pfade in diesem Kapitel sind relativ zum Hauptverzeichnis des Datenträgers angegeben.

5.1. Ziel 1: Analyse der Performance von StanbolµS

Das erste Ziel der Evaluierung ist, die Performance von StanbolµS zu analysieren. Da diese von der verwendeten Hardware abhängt, vergleichen wir die Werte mit denen einer monolithischen Stanbol-Version mit gleicher Funktionalität auf der selben Hardware. Die folgende Liste enthält die ausformulierte Version des ersten Ziels, die nach dem GQM-Verfahren gebildeten Fragen und in der zweiten Unterebene die zugehörigen Metriken. Da StanbolµS und der Monolith Systeme für die semantische Analyse sind, bezeichnen wir sie im folgenden auch abkürzend als *Systeme*.

Z1: Analysiere die Performance von StanbolµS im Vergleich zu einem Stanbol-Monolithen mit gleicher Funktionalität aus Sicht eines Anwenders.

- F1.1: Welche Zeit benötigen die Systeme zur Verarbeitung von Texten?
  - M1.1.1: Berechnungszeit der einzelnen Enhancement-Schritte
  - M1.1.2: Gesamte Berechnungszeit
- F1.2: Wie stark werden die Ressourcen einer Maschine durch die Systeme ausgelastet?
  - M1.2.1: Durchschnittliche Prozessorauslastung in %
  - M1.2.2: Durchschnittlicher belegter Arbeitsspeicher in %
  - M1.2.3: Verbrauchter Festplattenspeicher in Megabyte

5.1.1. F1.1: Vergleich der Zeit für die Verarbeitung von Texten

Zur Beantwortung der ersten Frage führen wir zwei Messdurchläufe auf der gleichen Maschine durch. Bei dem ersten Messdurchlauf wird ein Stanbol-Monolith mit den Komponenten verwendet, die auch in StanbolµS enthalten sind. Bei der zweiten Messung verwenden wir StanbolµS mit einer Instanz jedes Enhancement-Services.
Wir rufen ein Python-Skript [Python] auf, welches für einen Text aus den unten beschrieben Testdaten 100.000 mal einen Aufruf an den Service startet. Wir verwenden nur einen Text, da sonst Unregelmäßigkeiten in den gemessenen Laufzeiten durch Unterschiede in der Länge der Texte auftreten könnten. Da wir keine Ergebnisse zwischenspeichern, wirkt sich dies nicht auf die Messungen aus. Bei der Analyse des 2600 Zeichen langen Texts werden acht Personen, vier Organisationen und die Sprache Englisch erkannt. Der Text besteht aus 450 Wörtern in 20 Sätzen.
Die Messung erfolgt durch das Speichern der aktuellen Systemzeit an verschiedenen Messpunkten, um die Zeiten zwischen diesen Punkten errechnen zu können. Das Python-Skript speichert die aktuelle Systemzeit direkt vor und nach jedem Aufruf. Für die Aufrufe wird das Modul *requests* verwendet.
In den StanbolµS-Komponenten und im Monolithen verwenden wir den selben Java-Code zum Speichern der Zeitpunkte. Zugriffe auf Datenträger benötigen im allgemeinen mehr Zeit und sind weniger konstant als Zugriffe auf den Arbeitsspeicher [Jacobs 2009]. Um einen möglichst konstanten Einfluss der Messung auf das zeitliche Verhalten zu erreichen, halten wir daher alle Zeitstempel zunächst im Arbeitsspeicher. Wenn die Anzahl einen bestimmten Grenzwert erreicht, werden sie dann als Datei abgelegt. So wird der Datenträger erst nach dem Ende der Messungen belastet.
Der Code wird bei den Enhancement-Services im Lastverteiler bei Erhalt eines Texts und im Wrapper direkt vor dem Weiterleiten des Ergebnisses aufgerufen. So wird die Zeit gemessen, die die Microservices benötigen, um den Text zu verarbeiten. Im Monolithen ist die Klasse *EnhancementJobHandler* dafür zuständig, die Enhancement-Engines zu starten. Diesen erweitern wir vor und nach dem Aufruf von *enhancementEngine.computeEnhancements()*, mit dem die Enhancement-Engines aufgerufen werden.

Testdaten

Wir verwenden einen der 2225 Texte von Greene und Cunningham [2006]. Jeder dieser Texte enthält einen englischsprachigen Nachrichtenartikel. Es sind Artikel zu den Themen *Business* (510), *Unterhaltung* (386), *Politik* (417), *Sport* (511) und *Technik* (401) enthalten.
Die englische Sprache ist notwendig, da die OpenNLP-NER-Komponente nicht mit deutschsprachigen Texten funktioniert. Des weiteren eigenen sich diese Texte besonders, da typischerweise mehrere Sätze enthalten sind, sodass auch die OpenNLP-Sentence-Komponente zur Satzerkennung benötigt wird. Nachrichtentexte enthalten außerdem häufig bekannte Orte oder Personen, welche von der Individuenerkennung entdeckt werden können.

Hardware Spezifikation

Als Testgerät dient eine Maschine der Arbeitsgruppe Software Engineering[1] der Christian-Albrechts-Universität zu Kiel mit folgender Konfiguration:

- ▻ Linux version 3.16.0-4-amd64 (debian-kernel@lists.debian.org) (gcc version 4.8.4 (Debian 4.8.4-1)) #1 SMP Debian 3.16.36-1+deb8u1 (2016-09-03)
- ▻ 126 GB RAM
- ▻ 32x Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz mit 8 Kernen
- ▻ Python 2.7.9
- ▻ Java Version "1.8.0_101"
  Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
  Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)

Ergebnisse

Die Ergebnisse der Messungen sind unter */evaluation/results/sequential* auf dem beiliegenden Datenträger zu finden. Tabelle 5.1 listet Durchschnittswerte der 100.000 Messungen.

Tabelle 5.1. Durchschnittszeit der Phasen in Millisekunden

	Monolith	StanbolµS
Berechnungszeit Langdetect	3,2	119
Berechnungszeit OpenNLP-Sentence	0,4	120,9
Berechnungszeit OpenNLP-Token	11,3	164,7
Berechnungszeit OpenNLP-POS	33,9	209,4
Berechnungszeit OpenNLP-Ner	73,4	255,8
Rest	72,6	140
Gesamtzeit	194,8	1009,8

[1] https://www.se.informatik.uni-kiel.de/en

Der zeitliche Ablauf eines Aufrufs wird in Abbildung 5.1 anhand der Durchschnittswerte dargestellt. Auf der X-Achse verläuft die Zeit vom Start der Anfrage an. Das rechte Ende der Balken ist der Zeitpunkt, an dem die Antwort vom System beim Python-Skript vollständig empfangen wurde. Der oberste Balken zeigt den zeitlichen Verlauf einer Anfrage bei StanbolµS. Der darunter den maßstabsgetreuen Verlauf einer Anfrage im Monolithen. Unterhalb der Zeitachse ist der Ablauf des Monolithen auf die Länge der Abfrage bei StanbolµS skaliert, um die Anteile vergleichen zu können.

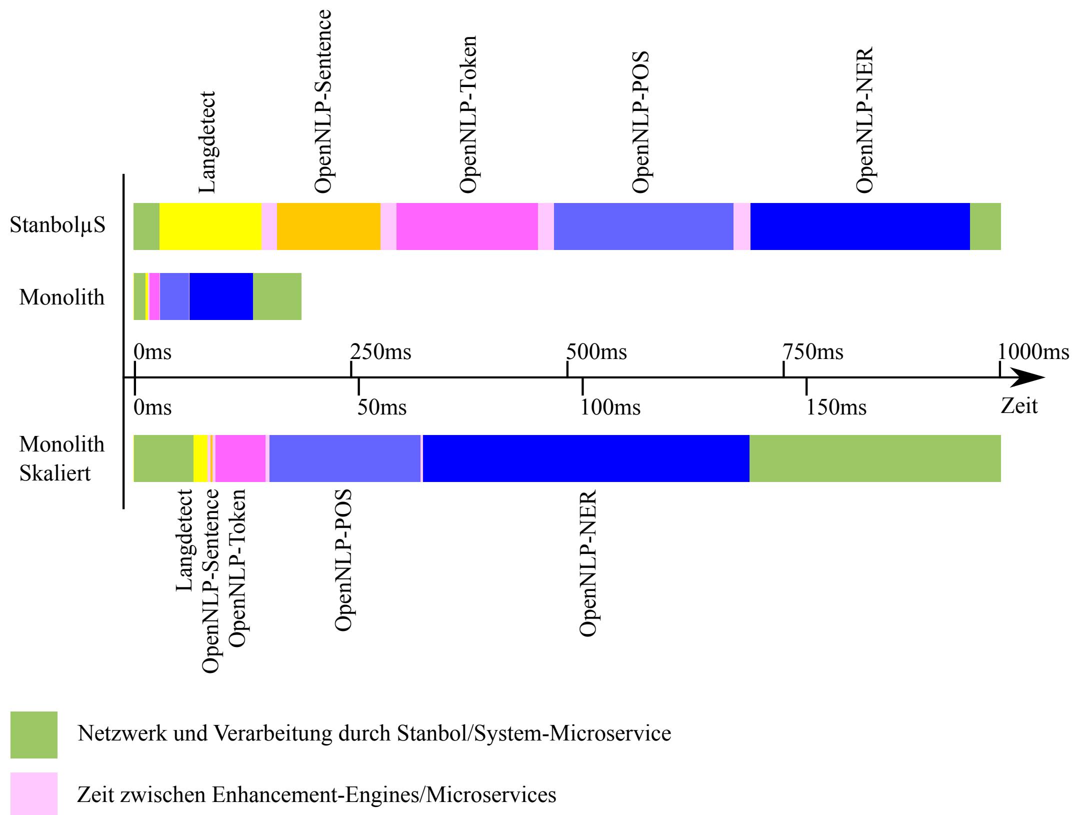


Abbildung 5.1. Durchschnittliche Laufzeiten im Vergleich

Aus den Ergebnissen geht hervor, dass die Gesamtzeit bei StanbolµS im Durchschnitt mit einem Plus von 810ms um ein etwa fünffaches höher ist als beim Monolithen. Dies wird durch den Unterschied der Länge der oberen beiden Balken von Abbildung 5.1 visualisiert. Die Enhancement-Microservices benötigen im Schnitt 150ms länger als die Enhancement-Engines im Monolithen, was der 7,1-fachen Zeit entspricht. Die vorderen grünen Abschnitte stellen den Zeitraum zwischen dem Absenden der Anfrage durch das Python-Skript und dem Empfang bei dem ersten Enhancer dar. Die hinteren grünen Abschnitte symbolisieren

den Zeitraum der Übertragung der Ergebnisse von dem letzten Enhancer zum Python-Skript. Der Anteil dieser Zeiträume am Gesamtzeitraum ist beim Monolithen mit 36% höher als bei StanbolµS mit 6,5%. Die Differenz macht mit etwa 5ms ungefähr 0,6% des gesamten Unterschiedes der durchschnittlichen Berechnungszeiten aus. Die hellen Flächen zwischen den Abschnitten für die Enhancer symbolisieren die Übermittlung zwischen den Enhancement-Engines beziehungsweise den Microservices. Der Anteil dieser Zeiträume beträgt beim Monolithen mit 2,7ms 1,4% und bei StanbolµS mit 72ms 6,5%.
Ein in *R* [Team 2013] ausgeführter Zweistichproben T-Test [Wilcox 1996] über die Gesamtzeiten der Systeme gibt das folgende Ergebnis aus:

```
Two Sample t-test

data:  monolith_total_times and stanbolus_total_times
t = -3002.9, df = 2e+05, p-value < 2.2e-16
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -814.7219
sample estimates:
mean of x mean of y
194.7562 1009.9246
```

Diskussion der Ergebnisse

Der Zeitraum für die Übermittlung zwischen den Systemen und dem Python-Skript ist etwa gleich lang. Den Unterschied von 7% führen wir auf Messungenauigkeiten zurück. Der übrige Anteil der Abweichung wird auch durch die Übertragungen zwischen den Enhancern verursacht. Die Übertragung zwischen verschiedenen Prozessen erfordert durchschnittlich ungefähr 70ms mehr und damit etwa 35 mal so viel Zeit, wie die Übertragung innerhalb einer Stanbol-Instanz. Außerdem sind die Zeiten innerhalb der Enhancer bei StanbolµS durchschnittlich um das 7,1-fache höher. Das ist dadurch bedingt, dass bei StanbolµS zusätzlich zu der Enhancement-Engine noch der Lastverteiler, der Wrapper und eine vollständige Stanbol-Instanz betrieben werden. Der Lastverteiler benötigt Zeit für die Kommunikation mit dem Service-Register und für die Weiterleitung der Anfrage an den Wrapper. Der Wrapper leitet die Anfrage ebenfalls erst an seine Stanbol-Instanz weiter. Dort muss der Inhalt zusätzlich zu der Berechnung der Enhancement-Engine geparsed und serialisiert werden.
Während der Messungen wurden aus ungeklärter Ursache Anfragen zwischen den Wrappern und Lastverteilern dupliziert. Der Code des Lastverteilers wurde an diesen Stellen zwei mal aufgerufen, obwohl der sendende Code vom Wrapper nur ein mal aufgerufen wurde. Diese Duplikate wurden während der Messung anhand der Anfrage-ID erkannt und entfernt. Da auf 100.000 Anfragen lediglich 13 Verdopplungen auftraten, wurde das Messergebnis dadurch nicht signifikant beeinflusst.

Der Zweistichproben T-Test der Gesamtzeiten bestätigt, dass StanbolµS auf der gleichen Hardware signifikant länger für die selbe Anzahl Berechnungen benötigt. Dies entspricht auch unseren aus den Durchschnitten gefolgertem Resultat, nach dem die gesamte Berechnungszeit bei StanbolµS das fünffache der Berechnungszeit des Monolithen beträgt. Die Enhancement-Schritte benötigen durch die zusätzlichen Komponenten die 7,1-fache Zeit.

5.1.2. F1.2: Messung der Systemauslastung

Der Versuchsaufbau entspricht dem für F1.1. Zusätzlich zum Python-Skript und dem System wird eine weitere Anwendung betrieben, die in einem Zeitintervall von 50ms die Systemauslastung dokumentiert. Dafür verwenden wir die Sigar-Schnittstelle.[2] Das Skript aus F1 wird so konfiguriert, dass es über die gesamte Messzeit Anfragen an das System sendet. Für die drei Fälle Monolith, StanbolµS und Leerlauf werden je 100.000 Stichproben genommen. Jeder gespeicherte Messpunkte enthält die aktuelle Systemzeit, die prozentuale Auslastung des Prozessors und den prozentualen verbrauchten Arbeitsspeicher.
Um den verwendeten Festplattenspeicher für M1.2.3 zu messen, lesen wir die Größe der Ordner über die Betriebssystemoberfläche ab. Da Stanbol zur Laufzeit neue Dateien erzeugt, führen wir vor und nach der Messung für M1.2.1 und M1.2.2 Ablesungen durch.

Ergebnisse

Die Ergebnisse der Messung befinden sich unter *evaluation/results/resources* auf dem Datenträger. Durchschnittswerte zeigen wir tabellarisch in Tabelle 5.2. Wir sehen, dass die Prozessoren im Leerlauf im Durchschnitt zu fast 0% ausgelastet ist. StanbolµS erzeugt durchschnittlich 125% der Prozessorlast des Monolithen.
Der Speicherbedarf im Leerlauf liegt bei etwa 0,5% des Gesamtspeichers. Beim Betrieb von StanbolµS werden durchschnittlich ungefähr 37% des Speichers belegt. Da dieser Bedarf den des Leerlaufs mit einschließt, liegt der Verbrauch von StanbolµS bei etwa 36,4% und somit 45,8 GB. Der Monolith verbraucht abzüglich des Leerlauf-Wertes etwa 1% und damit 1,3 GB des Speichers. Im Durchschnitt benötigt StanbolµS 25 mal so viel Speicher wie der Monolith.

[2] https://support.hyperic.com/display/SIGAR/Home

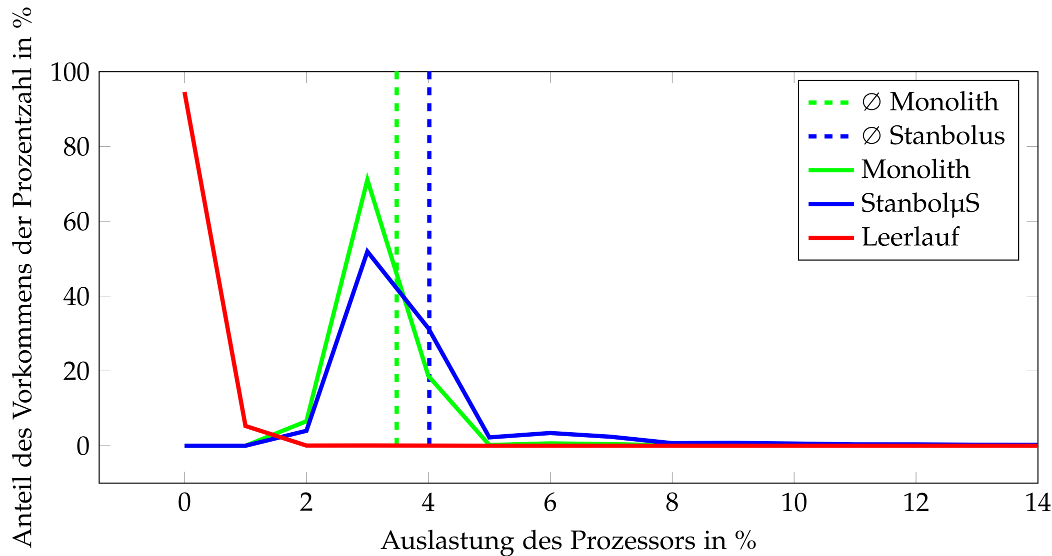


Abbildung 5.2. Verteilung der Prozessorauslastung

Tabelle 5.2. Systemauslastung während der sequentiellen Abarbeitung von Texten

		Monolith	StanbolµS	Leerlauf
Prozessorauslastung	Durchschnitt	3,48%	4,016%	0,037%
	Min	2,424%	1,875%	0%
	Max	45,732%	100%	4,294%
Speicherauslastung	Durchschnitt	1,471%	36,865%	0,465%
	Min	1,396%	15,539%	0,435%
	Max	1,561%	41,068%	0,525%

Abbildung 5.2 zeigt die Verteilung der gemessenen Prozessorauslastungen grafisch. Auf der X-Achse ist die Auslastung des Prozessors in % angegeben, und auf der Y-Achse, wie hoch der Anteil dieser Auslastung an den gesamten Messergebnissen ist. Es sind Graphen für die Prozessorauslastung des Monolithen, StanbolµS und im Leerlauf gegeben. Die senkrechten Linien zeigen die Durchschnitte für die Systeme.
Anhand der Grafik wird ersichtlich, dass die Leerlaufauslastung fast ausschließlich bei 0% liegt. Die häufigste Auslastung durch die beiden Systeme liegt bei etwa 3%. Die Auslastungswerte von StanbolµS weisen eine größere Streuung auf als die des Monolithen.

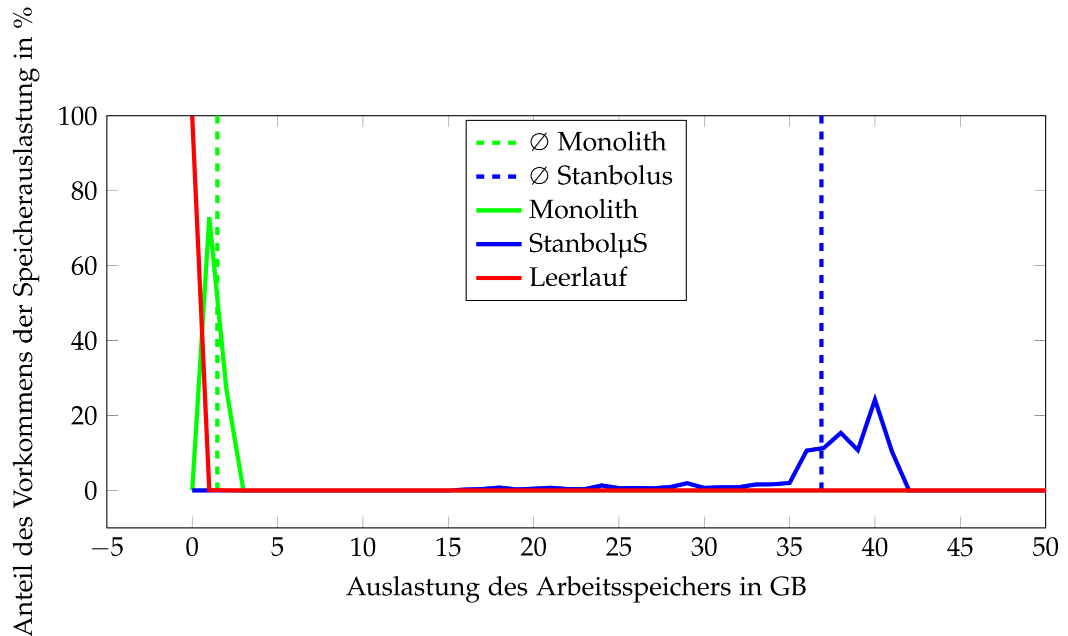


Abbildung 5.3. Verteilung der Speicherauslastung

Wir verwenden einen einseitigen Zweistichproben T-Test in R zur Prüfung der Hypothese, ob die Durchschnittliche Auslastung des Prozessors durch den Monolithen geringer ist als die durch StanbolµS verursachte. Die Ausgabe ist nachfolgend abgebildet:

```
Two Sample t-test

data:  monolith_cpu[["cpu"]] and stanbolus_cpu[["cpu"]]
t = -50.187, df = 2e+05, p-value < 2.2e-16
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -0.5181227
sample estimates:
mean of x mean of y
3.480384  4.016063
```

Die Verteilung der Speicherauslastungswerte ist in Abbildung 5.3 zu sehen. Auf der X-Achse ist die Auslastung des Arbeitsspeichers in Prozent. Die Y-Achse stellt den Anteil dieses Auslastungswertes dar. Auch hier sind Graphen für die beiden Systeme und den Leerlauf eingezeichnet. Die senkrechten Linien sind wieder die Durchschnittswerte.

Tabelle 5.3. Speicherbedarf in MB

System		Vorher	Nachher	Wachstum
StanbolµS	Eureka	36,5	=	0
	Langdetect	216,1 (79.6)	431	+214,9
	OpenNLP-Sentence	238,6 (102,1)	497,3	+258,7
	OpenNLP-Token	238,6 (102,1)	497,3	+258,7
	OpenNLP-POS	238,6 (102,1)	497,3	+258,7
	OpenNLP-NER	238,6 (102,1)	499,2	+260,6
	System	78,3	289,4	+211,1
	Wrapper	68,6	-	-
	Loadbalancer	67,9	-	-
	Gesamt	1.285,3	2.747.9	+1.462,7
Monolith		103,5	435,9	+332,4

Es wird ersichtlich, dass es eine geringe Leerlaufauslastung gibt. Beim Betrieb des Monolithen entsteht ein größtenteils konstanter Speicherbedarf von 1,8 GB, welcher geringer ist als der Speicherbedarf von StanbolµS mit durchschnittlich 46,45 GB. Durch die Grafik wir ersichtlich, dass der Speicherbedarf von StanbolµS nicht so konstant ist wie der des Monolithen. Um die Speicherauslastung der beiden Systeme zu vergleichen verwenden wir erneut einen Zweistichproben T-Test in R:

```
Two Sample t-test

data:  monolith_ram[["ram"]] and stanbolus_ram[["ram"]]
t = -2304.6, df = 2e+05, p-value < 2.2e-16
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -35.36865
sample estimates:
mean of x mean of y
1.470807 36.864719
```

Tabelle 5.3 zeigt den belegten Speicherplatz auf dem Datenträger vor und nach den Messungen der Systemauslastung. Die in Klammern angegebene Zahl ist die Größe der Stanbol-Instanz, zu der noch Wrapper und Loadbalancer für die Gesamtgröße der Enhancer-Services addiert wird.
Es wird ersichtlich, dass StanbolµS ungefähr zusätzliche 2,2 GB und damit 5,56 mal so viel Speicherplatz wie der Monolith mit 436 MB belegt. Die Enhancement-Services sind nach der Berechnung insgesamt etwa 2,3 GB groß, was 88% der Gesamtgröße von StanbolµS entspricht. Die maximale Größe eines Services beträgt fast 500 MB.

Diskussion der Ergebnisse

Die Prozessoren sind im Leerlauf kaum belastet. Wir können daher davon ausgehen, dass die gemessene Auslastung vorrangig durch den Monolithen beziehungsweise StanbolµS verursacht wird. Die Prozessorauslastung durch StanbolµS ist nur 25% höher als durch den Monolithen, da wir nur sequentielle Aufrufe ausführen. Die Komponenten bearbeiten den Text nacheinander. Daher führt jeweils nur eine StanbolµS-Komponente zur Zeit Berechnungen durch. Bei der Übermittlung zwischen den Komponenten sind zwei Komponenten aktiv, sodass mehrere Prozessorkerne parallel belastet werden, daher erzeugt StanbolµS zeitweise höhere Auslastungen als der Monolith.
StanbolµS besteht in der verwendeten Konfiguration aus fünf Lastverteilern, fünf Wrappern, fünf Stanbol-Instanzen, dem System-Service und dem Service-Register. Diese insgesamt 17 Komponenten werden jeweils in einer eigenen JVM betrieben. Diese JVMs benötigen separaten Speicher. Daher benötigt StanbolµS auch ein vielfaches des Speichers des Monolithen. Da sich der Speicherbedarf um mehr als das 17-fache unterscheidet, müssen einige Komponenten von StanbolµS allein mehr Speicher benötigen als der Monolith. Der Speicherbedarf der einzelnen Komponenten wurde von uns nicht ermittelt, daher können wir keine Aussagen über den Speicherbedarf der einzelnen Komponenten treffen.
Das Ergebnis des Zweistichproben T-Tests über die durchschnittlichen Prozessorauslastungen ist, dass StanbolµS die Prozessoren signifikant stärker auslastet. Der T-Test über die Speicherauslastung bestätigt, dass StanbolµS mehr Speicher benötigt. Diese Ergebnisse decken sich mit unseren Interpretationen der Messergebnisse. Die abgelesenen Werte der Dateien auf dem Datenträger zeigen außerdem, dass StanbolµS nach unserem Experiment 5,5 mal so viel Speicherplatz auf dem Datenträger wie der Monolith benötigt.

5.2. Ziel 2: Analyse der Skalierung von StanbolµS

In diesem Teil der Evaluierung analysieren wir das Skalierungsverhalten von StanbolµS im Vergleich zum Monolithen. Dafür erweitern wir das Python-Skript aus dem vorherigen Abschnitt so, dass eine konfigurierbare Anzahl von Threads gleichzeitig Anfragen an das Analyse-System sendet. So wird mit steigender Thread-Zahl die Kapazitäten des Analyse-Systems erschöpft und wir können die maximale Anzahl verarbeiteter Texte pro Sekunde messen.
Im Kontext dieses Abschnittes verwenden wir den Begriff *StanbolµS-Konfiguration*. Damit bezeichnen wir die Anzahl der Komponenten eines StanbolµS-Systems und deren Betriebsumgebung.
Das Ziel dieses Abschnitts lautet durch das Goal-Question-Metric-Verfahren definiert:

Z2: Analysiere das Skalierungsverhalten von StanbolµS aus Sicht eines Anwenders

- F2.1: Wie entwickelt sich der Durchsatz bei der Skalierung von StanbolµS?
  - M2.1.1: Verarbeitete Texte pro Zeit in Abhängigkeit von der Anzahl der Instanzen bei voller Auslastung des Systems
- F2.2: Gibt es eine StanbolµS-Konfiguration, die mehr Texte pro Zeit verarbeiten kann, als ein Stanbol-Monolith mit gleicher Funktion?
  - M2.2.1: Vergleich der Anzahl der verarbeiteten Texte pro Zeit bei voller Auslastung des Systems

5.2.1. Experiment

In diesem Unterabschnitt erklären wir, wie wir durch ein Python-Skript den Durchsatz eines Systems messen. Dann gehen wir auf die Startkonfiguration von StanbolµS für dieses Experiment ein und erläutern, wie sie im Laufe des Experiments durch Skalierung verändert wird.

Bestimmung des Durchsatzes

Um den Durchsatz des Monolithen oder einer StanbolµS Konfiguration zu messen, passen wir das Python-Skript aus Abschnitt 5.1.1 auf Seite 62 an. Das Skript startet wiederholt eine steigende Anzahl von Threads mit der Funktion *startThreads()*, die jeweils sequentiell Anfragen an das Zielsystem senden. Diese Threads nennen wir auch *Producer*. Für jede Anzahl von Threads wird aus der insgesamt benötigten Zeit und der gesamten Anzahl der analysierten Texte der Durchsatz errechnet. So erhalten wir einen Durchsatz in Abhängigkeit von der Nummer der Threads. Die Funktion zum Starten der Threads erstellt erst alle Threads. Dann wird die Startzeit bestimmt. So beeinflusst die Zeit zum Erstellen der Threads nicht die Messergebnisse. Nachdem alle Threads abgearbeitet sind, wird der

Endzeitpunkt bestimmt. Jeder Thread sendet sequentiell 20 Anfragen an das System. Bei niedrigeren Zeiten konnten wir einen zu hohen Einfluss durch das Starten und Beenden der Threads feststellen. Wir verwenden maximal 15 Producer, da dies ausreicht, um die getesteten Systeme zu sättigen. Für jede Producer-Anzahl führen wir 40 Durchläufe durch, um einen aussagekräftigen Mittelwert bilden zu können. Pro System werden so 96.000 Analysen durchgeführt.
Falls mehrere Producer-Maschinen verwendet werden müssen, weil eine einzelne nicht ausreicht, warten diese mit ihrer Ausführung bis zu einem gegebenen Zeitpunkt. So wird sichergestellt, dass die Producer möglichst gleichzeitig anfangen. Dann darf bei der Ausführung des Skripts nur eine feste Anzahl Threads verwendet werden. Die Gesamtzeit über alle Producer-Maschinen berechnet sich dann aus $max(Endzeit) - min(Startzeit)$.

Konfiguration und Skalierung

Anhand der Ergebnisse in Abbildung 5.1 auf Seite 64 sehen wir, dass die Microservices von StanbolµS in etwa gleich lang benötigen. Einzelne Komponenten zu verdoppeln würde also keine signifikante Verbesserung der Performance bedeuten, da die nicht verdoppelten Komponenten weiterhin den Durchsatz auf einem ähnlichen Niveau limitieren würden. Unsere Startkonfiguration von StanbolµS besteht also aus:

- einer System-Service-Instanz
- einer Eureka-Instanz
- je einer Lastverteiler-Instanz für die fünf Enhancement-Services
- je einer Instanz der fünf Enhancement-Services bestehend aus Wrapper und Stanbol-Instanz des Services

Diese 12 Komponenten betreiben wir jeweils auf einer separaten c4.large-VM in der Amazon EC2-Cloud. Die Virtuelle Maschinen (VMs) sind wie folgt konfiguriert:

- Linux version 4.4.19-29.55.amzn1.x86_64 (mockbuild@gobi-build-64012) (gcc version 4.8.3 20140911 (Red Hat 4.8.3-9) (GCC)) #1 SMP Mon Aug 29 23:29:40 UTC 2016
- 3.68GB RAM
- Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz
- Python 2.7.12
- Java Version "1.8.0_101"
  Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
  Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)

Tabelle 5.4. Durchschnittlicher Durchsatz in Texten pro Sekunde

#Producer	Monolith	StanbolµS Konfiguration 1	StanbolµS Konfiguration 2
1	10.276	2.832	1.974
2	13.604	3.287	4.416
3	14.292	5.424	6.957
4	14.615	6.765	9.227
5	15.222	8.068	11.813
6	15.168	9.4	13.837
7	15.108	10.006	15.759
8	15.041	10.802	17.623
9	15.008	11.446	18.588
10	15.013	11.564	19.954
11	14.963	11.792	20.827
12	14.846	11.987	21.127
13	14.765	12.16	21.588
14	14.791	12.097	21.952
15	14.846	12.305	21.745

Aus den gleichen Gründen wie bei der Startkonfiguration würde auch die Skalierung einzelner Enhancement-Services nur unwesentliche Verbesserungen des Durchsatzes bieten. Daher fügen wir bei der Skalierung für jeden Enhancement-Service eine weitere Instanz auf einer eigenen VM hinzu.

5.2.2. Ergebnisse

Bei den Ergebnissen steht StanbolµS Konfiguration 1 für die Startkonfiguration von StanbolµS mit jeweils einer Instanz der Enhancement-Services. StanbolµS Konfiguration 2 ist die ein mal skalierte Konfiguration mit jeweils zwei Instanzen pro Enhancement-Service. Die Messdaten sind unter *evaluation/results/parallel* auf dem Datenträger beigelegt.
Tabelle 5.4 zeigt die durchschnittlichen Durchsätze der Systeme in Abhang von der Producer-Anzahl tabellarisch. Abbildung 5.4 illustriert den Durchschnitt und das Maximum der gemessenen Durchsätze. Auf der X-Achse ist die Anzahl der Producer-Threads aufgetragen. Auf der Y-Achse ist der Durchsatz in Texten pro Sekunde angegeben. Die dickeren Linien markieren den Durchschnitt, die darüber liegenden dünneren das Maximum, die darunterliegenden das Minimum.
Der durchschnittliche Durchsatz des Monolithen steigt von etwa zehn Texten pro Sekunde bei einem Producer zu über 15 Texten pro Sekunde bei fünf Producer-Threads. Danach sinkt er leicht um durchschnittlich 0,0025% pro zusätzlichem Producer.
Der Durchsatz vom unskalierten StanbolµS steigt mit jedem hinzukommenden Producer an. Dieser Zuwachs sinkt mit steigender Anzahl der Producer. Bei einem Producer ist der durchschnittliche Durchsatz von StanbolµS in der Originalkonfiguration etwa 27,5% des

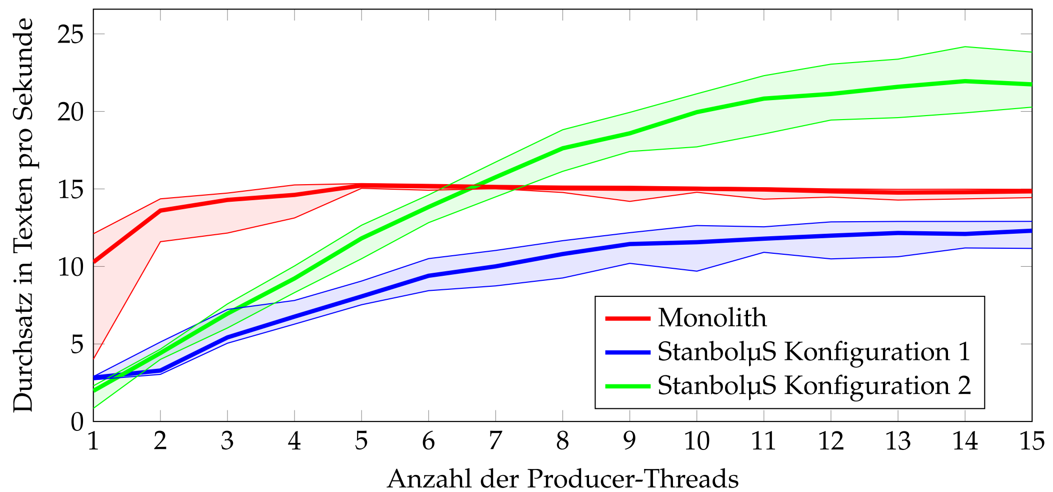


Abbildung 5.4. Durchschnittlicher und Maximaler Durchsatz

durchschnittlichen Durchsatzes des Monolithen und bei 15 Producern etwa 80%. Das Maximum des Durchsatzes ist bei keiner Producer-Zahl höher als der minimale Durchsatz des Monolithen. Ein in R ausgeführter T-Test zum Vergleich der durchschnittlichen Durchsätze von StanbolµS und des Monolithen bei 15 Producern gibt das folgende Ergebnis:

```
Two Sample t-test

data:  monolith_15 and stanbol_configuration_1_15
t = 32.3, df = 78, p-value < 2.2e-16
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
2.40998     Inf
sample estimates:
mean of x mean of y
14.84606  12.30513
```

Abbildung 5.4 zeigt, dass der Durchsatz von StanbolµS nach der Skalierung ebenfalls mit der Zunahme von Producern wächst. Das Wachstum ist jedoch stärker als vor der Skalierung. Bei einem Producer ist der durchschnittliche Durchsatz der unskalierten Version höher. Bei zwei und drei Producern erreichen die beiden Konfigurationen von StanbolµS einen vergleichbaren maximalen Durchsatz. Ab vier Producern liegt jeder gemessene Durchsatz der skalierten Konfiguration über dem maximalen gemessenen Durchsatz der unskalierten Version bei der gleichen Menge von Producern. Ein Zweistichproben T-Test in R zum Vergleich der gemessenen Durchsätze der Konfigurationen bei 15 Producern gibt die folgende Ausgabe:

```
Two Sample t-test

data:  stanbol_configuration_1_15 and stanbol_configuration_2_15
t = -19.998, df = 78, p-value < 2.2e-16
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -8.654228
sample estimates:
mean of x mean of y
12.30513  21.74514
```

StanbolµS erreicht in der skalierten Konfiguration ab sieben Producern einen höheren Durchsatz als der Monolith. Ein T-Test zur Prüfung dieser Hypothese liefert das folgende Ergebnis:

```
Two Sample t-test

data:   monolith_15 and stanbol_configuration_2_15
t = -14.789, df = 78, p-value < 2.2e-16
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -6.122547
sample estimates:
mean of x mean of y
14.84606  21.74514
```

5.2.3. Diskussion der Ergebnisse

Bis zu fünf Producer-Threads steigt der Durchsatz des Monolithen. Dies liegt an der besseren Ausnutzung des Mehrkernprozessors. Danach ist der Durchsatz rückläufig. Dies führen wir darauf zurück, dass die Maschine, auf der der Monolith betrieben wird, dann ausgelastet ist. Durch das Hinzufügen weiterer Producer-Threads wird der Monolith mit zusätzlichem Aufwand für Netzwerk-Operationen belastet. Diese zusätzliche Belastung führt dazu, dass der Durchsatz sinkt, anstatt konstant zu bleiben.
StanbolμS 1 erreicht mit wenigen Producern im Schnitt etwa 24% des Durchsatzes des Monolithen. Dies ist auf die Netzwerkkommunikation sowie die zusätzlichen Aufwände durch die Loadbalancer und Wrapper, und das Sequentialisieren und Parsen der Daten in jeder Stanbol-Instanz zurückzuführen.
In Abschnitt 5.1.1 zeigen wir, dass die Berechnungszeit bei sequentiellen Aufrufen bei StanbolμS wesentlich höher ist als beim Monolithen. Trotzdem erreicht StanbolμS schon mit einer Instanz jeden Services 80% des Durchsatzes des Monolithen. Dies wird dadurch erreicht, dass die Komponenten von StanbolμS eine Pipeline bilden. In dieser Pipeline können mehr Texte zeitgleich bearbeitet werden, als beim Monolithen, da die einzelnen Phasen auf separaten Maschinen laufen. Dadurch kann der Durchsatz höher sein, obwohl die einzelnen Berechnungen länger dauern.
Wie in Abbildung 5.4 ersichtlich wird, sinkt der Zuwachs des Durchsatzes mit steigender Zahl der Producer fast auf null. Daher können wir annehmen, dass der tatsächliche maximale Durchsatz etwa bei dem von uns gemessenen liegt.
Nach der Skalierung verfügt StanbolμS über je zwei Instanzen der Enhancement-Services auf fünf weiteren Maschinen. Abbildung 5.4 zeigt, dass StanbolμS 2 schon ab vier Producern höhere Durchsatzzahlen erreicht als die unskalierte Version. Dies führen wir darauf zurück, dass wir mit der Skalierung die Komponenten mit der längsten Berechnungszeit verdoppelt haben. So können Teile der Pipeline, die bei der unskalierten Version schon ausgelastet waren, auch bei niedrigeren Producer-Zahlen schon zu einem höheren Durchsatz führen.
Die Skalierung führt dazu, dass StanbolμS mit zwei Instanzen pro Enhancement-Service ab sieben Producern einen höheren maximalen Durchsatz erreicht als der Monolith.

F2.1: Wie entwickelt sich der Durchsatz bei der Skalierung von StanbolμS?

Der maximale gemessene Durchsatz steigt mit der ersten Verdopplung der Enhancement-Services von etwa 13 Texten pro Sekunde auf 24 Texte pro Sekunde. Das entspricht einem Geschwindigkeitszuwachs von 84% während die Anzahl der VMs auf das 1,42-fache steigt. Ein einseitiger Zweistichproben T-Test bestätigt, dass zu 95% Konfidenz der Durchsatz nach der Skalierung höher ist.
Aufgrund von Begrenzungen durch Amazon konnten wir keine weiteren Instanzen erzeugen und nicht weiter skalieren. So ist die Anzahl der unterschiedlichen Konfigurationen zu gering, um generelle Aussagen über das Skalierungsverhalten treffen zu können.

F2.2: Gibt es eine StanbolµS-Konfiguration, die mehr Texte pro Zeit verarbeiten kann, als ein Stanbol-Monolith mit gleicher Funktion?

Bei 15 Producer-Threads verarbeitet der Monolith durchschnittlich 14,9 Texte pro Sekunde. StanbolµS mit jeweils einer Instanz der Enhancement-Services erreicht einen Durchsatz von 12,3 Texten pro Sekunde. Ein T-Test ergibt, dass der tatsächliche Durchsatz des Monolithen mit 95% Konfidenz höher ist als der von StanbolµS vor der Skalierung. Obwohl zwölf mal so viele Maschinen verwendet werden, ist der maximale Durchsatz von StanbolµS 17% geringer. Statt des idealen Speedups von 12 erhalten wir 0,83, da die zusätzlichen Rechenkapazitäten nicht für die Textanalyse sondern den Betrieb als Microservice-Architektur verwendet werden.
Nach der Skalierung erreicht StanbolµS einen Durchsatz von maximal 24,2 und durchschnittlich 21,8 Texten pro Sekunde. Ein Zweistichproben T-Test bestätigt hier, dass auch der reale durchschnittliche Durchsatz zu einer Konfidenz von 95% höher ist, als der des Monolithen. Die skalierte Konfiguration von StanbolµS kann also mehr Texte pro Zeit verarbeiten, als ein Monolith mit gleicher Funktion und somit existiert eine solche Konfiguration.

5.3. Zusammenfassung

Anhand der beiden Fragen zu Ziel 1 und den zugehörigen Metriken konnten wir zeigen, dass StanbolµS signifikant mehr Ressourcen benötigt als ein Monolith mit gleicher Funktionalität. Trotz des höheren Verbrauchs ist die Bearbeitungszeit einzelner Texte bei StanbolµS signifikant höher. Das Ergebnis des ersten Evaluierungsziels ist also, dass die Performance von StanbolµS in allen gemessenen Kriterien schlechter ist, also die eines Monolithen mit gleicher Funktionalität. Im Gegenzug sollten Wartbarkeit, Ersetzbarkeit und Anpassbarkeit von StanbolµS besser sein als beim Monolithen. Diese Qualitätsmerkmale konnten wir Rahmen dieser Arbeit jedoch nicht messen.
Die Auswertung vom zweiten Evaluierungsziel zeigt, dass StanbolµS durch den Einsatz von mehr Maschinen so skalieren kann, dass es einen 57% höheren Durchsatz erreicht als ein Monolith mit gleicher Funktionalität. Zudem konnten wir zeigen, dass durch die Verwendung von zwei Instanzen pro Enhancement-Service der maximale Durchsatz um 87% gegenüber einer unskalierten Konfiguration steigen kann.

Kapitel 6

Verwandte Arbeiten

In diesem Kapitel stellen wir thematisch verwandte Veröffentlichungen vor. Die Abschnitte des Kapitels sind wie die Hauptkapitel der Arbeit organisiert. Zunächst nennen wir in Abschnitt 6.1 weitere Beispiele für Microservice-Architekturen. In Abschnitt 6.2 geht es um Anwendungen von Apache Stanbol. Abschließend fassen wir in Abschnitt 6.3 Artikel über den Einfluss von Microservices auf Betriebskosten und Performance zusammen.

6.1. Microservice-Architekturen

In Abschnitt 2.2 auf Seite 9 erläutern wir die Microservice-Charakterisierung mit Vertikalen, welche von Steinacker [2014] und Kraus u. a. [2013] gegeben wird. Hasselbring [2016c], Hasselbring [2016a] und Johanson u. a. [2016] verwenden ebenfalls den Begriff der Vertikalen, die in weitere Microservices unterteilt werden. Im Gegensatz zu den Vertikalen von Kraus u. a. [2013] verwenden verschiedene Microservices bei ihnen aber keine gemeinsamen Datenbanken.
Hasselbring [2016b] betont zusätzlich die Skalierungsmöglichkeiten durch den Microservice-Architekturstil. Der Autor nennt einige Anti-Muster und Muster für Skalierbarkeit. Eins der Anti-Muster ist, eine einzelne Datenbank zu verwenden. Das dazugehörige Muster ist verteilte Datenhaltung mit verschiedenen Datenbanktypen, wie wir es für Microservices vorsehen. In Abschnitt 3.2.5 auf Seite 41 stellen wir Eventual Consistency vor. Hasselbring [2016b] zieht Eventual Consitency verteilten Transaktionen vor, da der Aufwand für Transaktionen mit dem System wächst und so die Skalierbarkeit beschränkt. Außerdem soll asynchrones Messaging verwendet werden, anstatt verschiedene Komponenten über Datenbankschemata zu integrieren.
Johanson u. a. [2016] beschreiben mit *OceanTEA* eine Microservice-Plattform zur Analyse von Ozeanographie-Daten. OceanTEAs funktionale Komponenten sind in Vertikalen organisiert. Zwischen den Microservices gibt es aber auch innerhalb einer Vertikalen keine gemeinsamen Komponenten, sodass die Vertikalen eher eine theoretische Grenze darstellen.
Im Gegensatz zu unserer Architektur und StanbolµS verwendet OceanTEA ein *API-Gateway*, welches die Microservices orchestriert und eine API für Endanwender zur Verfügung stellt. Ein solcher Endanwender ist der *Client*, welcher eine GUI zur Verwendung des Systems

bietet. Im Gegensatz zu unserem Ansatz ist die GUI also nicht in die Microservices integriert. Für den Datenaustausch zeigen Johanson u. a. [2016] eine Schicht zum Datenaustausch.

6.2. Apache Stanbol

Gangemi [2013] vergleicht verschiedene Systeme zur Textanalyse funktional. Dafür stellt er eine Liste mit Methoden der semantischen Analyse vor. Diese verwenden wir in Abschnitt 3.1 auf Seite 17 zur Partitionierung der semantischen Analyse. Auf die Performance der Systeme geht er nicht ein, sondern nur auf die Vollständigkeit und Korrektheit der Ergebnisse.
Kumar [2012] stellt die Stanbol Enhancement-Engine *FORMCEPT Healthcare Engine* vor. Er präsentiert ausführliche Tests, die Speicher- und Rechenzeitbedarf einschließen. Die verwendeten Enhancement-Engines sind:

1. Tika Engine (Extrahiert Texte aus verschiedenen Dateiformaten)
2. Language Identification
3. FORMCEPT Healthcare Engine

Die Healthcare Engine verwendet den externen Service *FORMCEPT Spotter Service* zum Erkennen von genannten medizinischen Begriffen. Die erkannten Begriffe werden dann in die Metadaten des Texts eingefügt. Diese Engine führt also Individuenerkennung für medizinische Begriffen durch. Die Daten für die Tests stammen von *CALBC Corpora* [1]. Es ist nicht angegeben, welche Daten genau verwendet wurden. Die FORMCEPT Healthcare Engine und der Spotter Service sind nicht Open Source und stehen daher nicht öffentlich zur Verfügung. Wir konnten daher keinen Vergleich mit StanbolμS durchführen.

6.3. Performance Evaluierungen

Knoche [2016] stellt einen Ansatz vor, um bei einer Umstellung eines Monolithen auf Microservice-Architektur den Einfluss der Performance durch Simulation vorherzusagen. Dazu werden Ergebnisse von dynamischer Analysen des Monolithen verwendet, um Laufzeiten einzelner Komponenten zu berechnen. Zusätzlich werden Modelle der Architektur des Monolithen und der geplanten modernisierten Architektur erzeugt. Anhand dieser Modelle werden dann Simulationen durchgeführt. Mit diesem Verfahren könnte geprüft werden, ob die von uns ermittelten Ergebnisse den Erwartungen durch die Simulation entsprechen. Außerdem kann der Einfluss von Änderungen der Architektur, zum Beispiel durch den Austausch einer Service-Implementierung, vorher abgeschätzt werden.
Villamizar u. a. [2016] vergleichen die Betriebskosten von Systemen gleicher Funktion

[1] `http://www.ebi.ac.uk/Rebholz-srv/CALBC/corpora/resources.html`

mit ähnlichem Durchsatz in monolithischer Architektur, Microservice-Architektur und Nanoservice-Architektur. Als Plattform für den Betrieb wählen sie dafür Amazon Web Servicess [Amazon 2015].
Um vergleichbare Systeme zu haben, messen sie dazu ebenfalls erst den maximalen Durchsatz des Monolithen. Im Gegensatz zu unserem Experiment in Abschnitt 5.2 auf Seite 71 verwenden sie aber einen skalierenden Monolithen. Die Instanzen des Monolithen greifen auf eine gemeinsame, nicht skalierbare Datenbank zu, wodurch der Durchsatz begrenzt wird. Anhand dieses Durchsatzes erstellen sie vergleichbare Microservice- und Nanoservice-Konfigurationen und messen auf diesen Systemen ebenfalls den Durchsatz.
Die Kosten werden anschließend anhand der Preise von AWS in US-Dollar pro 1 Millionen Anfragen errechnet. Das Ergebnis dieser Berechnung ist, dass der Betrieb der Microservice-Architektur zwischen 9,5% und 13,8% weniger kostet, als der des Monolithen. Das Nanoservice-System kostet zwischen 55,1% und 77% weniger, als der monolitische Ansatz. Mit den Messungen des Durchsatzes haben Villamizar u. a. [2016] außerdem die Antwortzeiten der Anfragen gemessen. Die durchschnittliche Antwortzeit des Microservice-Systems ist je nach Szenario 1,6 mal bis 3 mal so hoch wie die des Monolithen.
Da die Antwortzeit der Microservices höher ist, als die des Monolithen, schließen wir, dass zum Ausgleich mehr VMs verwendet wurden. Die genaue Anzahl der Instanzen ist leider nicht angegeben. Der Faktor bei den Antwortzeiten ist niedriger als der von uns gemessene. Dies führen wir unter anderem darauf zurück, dass die Microservices in diesem Experiment nicht sequentiell aufgerufen werden.
Für die Evaluierung liefern wir StanbolµS direkt auf den VMs aus. In Abschnitt 3.2.7 auf Seite 45 stellen wir Container als Alternative vor. Ein Nachteil beim Einsatz von Containern ist ihr Einfluss auf die Laufzeit. Kratzke [2015, Seite 165] untersucht den Einfluss von Containern, Software-Netzwerken und Verschlüsselung auf die Netzwerkkommunikation zwischen Microservices. Mit Software-Netzwerken ist hierbei zum Beispiel die Nutzung von Service-Registern zur Entdeckung von Services gemeint. Die Ergebnisse sind anteilig an der Laufzeit gemessen. Das Ergebnis der Untersuchung ist, dass die Performance durch Container zwischen 10% und 20% und durch Software-Netzwerke zwischen 30% und 70% beeinflusst wird. Der Einsatz von Verschlüsselung hingegen hat nur geringe Auswirkungen. Aus diesen Ergebnissen folgert Kratzke [2015], dass die Nachrichtengröße möglichst gering gehalten werden soll, da dann der Einfluss auf Anwendungsebene geringer sei. Der Nachteil durch Software-Netzwerke könne weiterhin um 10% bis 20% verringert werden, wenn Router, wie der von uns verwendete Lastverteiler, ohne Container auf dem Host betrieben würden. Dies würde aber erfordern, dass alle Instanzen eines Microservice auf dem selben Host betrieben würden. Dadurch wäre die Skalierung durch Hinzufügen weiterer Instanzen auf die Ressourcen des Hosts beschränkt.

Kapitel 7

Fazit und Ausblick

In dieser Arbeit untersuchen wir, ob sich der Microservice-Architekturstil für die semantische Analyse im Bereich der Computerlinguistik anwenden lässt. Dafür fassen wir zusammen, was Computerlinguistik und semantische Analyse sind. Wir erläutern Anwendungsgebiete der semantischen Analyse und erklären, dass sich Sprache verändert. Daraus folgt ein andauernder Wartungsbedarf für Anwendungen zur Textanalyse. Aus diesem Grund schlagen wir die Anwendung des Microservice-Architekturstils für die semantische Analyse vor.
Den Begriff Microservice haben wir als unabhängig auslieferbare und autonome Komponente einer Anwendung definiert. Außerdem teilen Microservices nach unserer Charakterisierung keine Datenbanken und sind von der Größe so beschaffen, dass sie von einem einzelnen Team entwickelt werden können. Mit Technologien wie HTTP, REST und Apache Stanbol erläutern wir die Grundlagen für die Implementierung der Architektur.
Für unseren Architekturentwurf zeigen wir zunächst, wie man die Textanalyse in Microservices aufteilen kann. Dazu betrachten wir zwei Ansätze: Partitionierung nach Art der Semantik und nach Methoden der semantischen Analyse. Wir entscheiden uns für den zweiten Ansatz.
Wir erläutern, welche Aspekte es neben der Partitionierung in Microservice-Systemen zu beachten gibt. In diesem Rahmen stellen wir verschiedene Ansätze für Kommunikation, Koordination, Datenhaltung, grafische Oberflächen, den Betrieb und die Entwicklung vor und wägen sie gegeneinander ab. Anhand dieser Abwägungen erstellen wir unsere Microservice-Architektur für die semantische Analyse.
Unsere Ziele für die Architektur waren, eine Partitionierung zu erstellen, deren Partitionen unserem Microservice-Begriff entsprechen. Außerdem sollten Skalierungsmöglichkeiten sowie die Koordination der Services beschrieben sein. Durch die Partitionierung nach Methode der semantischen Analyse erreichen wir kleine, autonome Services mit unabhängiger Datenhaltung. Skalierung ist anhand der drei in Abschnitt 3.2.7 auf Seite 44 vorgestellten Richtungen bedacht und die Koordination beschreiben wir allgemein. Somit sind unsere Bedingungen an den Architekturentwurf erfüllt.
Mit StanbolµS stellen wir eine Implementierung der Architektur mit fünf integrierten Enhancement-Engines von Apache Stanbol vor. Wir erläutern, wie die Konzepte der Architektur in StanbolµS umgesetzt werden. StanbolµS verwendet das Spring-Framework für den Wrapper und Lastverteiler. Außerdem bieten alle Komponenten eine REST-Schnittstelle. Die Microservices verwenden Stanbol-Instanzen, welche nur die für den jeweiligen Service

erforderlichen Komponenten enthalten. Somit erfüllt auch die Implementierung die durch Ziel 2 auf Seite 4 gestellten Anforderungen.
Im Rahmen der Evaluierung zeigen wir, dass unsere Implementierung gegenüber dem Monolithen signifikante Nachteile bezüglich Performance- und Ressourcenverbrauch aufweist. Der Performance-Nachteil kann durch Skalierung wieder ausgeglichen werden, wenn der maximale Durchsatz des Monolithen erreicht ist. Die Evaluierung ist nach dem Goal-Question-Metric-Verfahren durchgeführt. Die Ziele der Evaluierung haben wir daher bis auf die Analyse von Wartbarkeit und Übertragbarkeit erreicht.
Mit der Vorstellung verwandter Arbeiten fassen wir relevante Veröffentlichungen zu den Themen Microservice-Architekturen, Apache Stanbol und Performance-Evaluierungen von Microservice-Systemen zusammen.

Im Rahmen dieser Arbeit zeigen wir, dass sich der Microservice-Architekturstil auf die semantische Analyse anwenden lässt. Ein Nachteil dabei ist die Verschlechterung der Performance. Als Vorteil analysieren wir die Skalierbarkeit. Andere Vorteile neben der Skalierbarkeit, die durch Microservices erreicht werden sollen, konnten wir im Rahmen dieser Arbeit nicht evaluieren. Dies schließt insbesondere die Skalierung der Entwicklung, die geringere Code-Komplexität und Ersetzbarkeit ein.
Ein Thema zukünftiger Forschung kann die Evaluierung dieser Vorteile sein. Ein weiteres mögliches Thema ist eine detaillierte Untersuchung, wodurch genau die Performance-Einbußen bei StanbolµS verursacht werden und diese zu optimieren. Dazu gehört auch eine Evaluierung des Overheads verschiedener Technologien zur Implementierung von Microservices. Weitere Maßnahmen zur Verbesserung von StanbolµS sind die Integration weiterer Enhancement-Engines und eine vollwertige REST-Unterstützung mit einem eigenen Dateiformat, da wir bisher nur das von Apache Stanbol verwendete Format kapseln. Mit dem eigenen Dateiformat könnten dann Schemata zur Validierung von Anfragen implementiert werden.

Anhang A

Handbuch der Implementierung

A.1. Inhalt des Datenträgers

Im Ordner *evaluation* sind die Evaluierungsergebnisse so wie die dafür verwendeten Python-Skripte gespeichert. *system-monitor* enthält den Code der Anwendung zum Messen der Systemauslastung. Unter *monolith* befinden sich der für die Evaluierung verwendete Monolith sowie Skripte zum Installieren und Starten des Monolithen. Die von uns erstellen Komponenten von StanbolµS sind sich im Ordner *StanbolµS*. Zusätzlich wird noch der Quellcode von Apache Stanbol benötigt (siehe Abschnitt A.2). Dieser Ordner enthält:

- ▻ einen Ordner **eureka**, welcher den Eureka-Server enthält
- ▻ einen Ordner **microservices**, in dem Maven-Projekte für die Microservices liegen
- ▻ einen Ordner **run**, in dem Windows-Skripte zum installieren und starten von StanbolµS liegen
- ▻ die Maven Projektdatei **pom.xml**, welche alle Komponenten zusammenfasst

Der microservice-Ordner enthält einen Unterordner für jeden Microservice, sowie einen Ordner **shared**, in dem zwischen den Services geteilte Komponenten liegen. Die Ordner der Enhancement-Services enthalten:

- ▻ eine **bundlelist**, welche die für die Stanbol-Instanz dieses Services benötigten Pakete beschreibt
- ▻ einen **launcher**, welcher die Ausführbare Stanbol-Instanz enthält
- ▻ eine Maven Projektdatei um die bundlelist und den launcher zu bauen
- ▻ ein Batch-Skript zum starten des Service inklusive Wrapper und Lastverteiler

Der Ordner **system** enthält zusätzlich zu einer bundlelist und einem launcher noch:

- ▻ eine Konfiguration in **config**, welche eine Enhancement-Chain enthält
- ▻ in **enhancement-engines/remote** die Sender-Enhancement-Engine
- ▻ das **response-listener** bundle

In **shared** befinden sich:

- die **bundles**:
  - **jersey**, welches eine überarbeitete Version des jersey-Artefakts von stanbol enthält, um das Serialisieren und Parsen der Ergebnisse von den OpenNLP-Engines einzubinden
  - **network-tools**, welches Funktionen zum Serialisieren und Parsen von ContentItems enthält
- der **loadbalancer**, welcher Lastverteilung mit Ribbon ermöglicht
- der **microservice-wrapper**, der Bestandteil aller Nicht-System-Microservices ist

A.2. Bauen der Anwendungen

Vorraussetzungen:

- Maven 3.3.9
- Oracle Java 1.8.0.77
- Windows 7 64bit
- Ein Subversion-Client

Andere Versionen könnten auch funktionieren, wurden von uns aber nicht getestet.

A.2.1. Apache Stanbol

1. Über Subversion muss der Quellcode von `https://svn.apache.org/repos/asf/stanbol/trunk/` heruntergeladen werden.
2. Im Stanbol Quellcode befindet sich unter */enhancer/jobmanager/event/src/main/java/org/apache/stanbol/enhancer/jobmanager/event/impl* der EnhancementJobHandler. In diesem müssen drei Anpassungen vorgenommen werden:
   (a) In Zeile 400 muss *float cf = eds/cd;* durch zum Beispiel

   ```
   float cf = cd != 0? eds/cd : eds;
   ```

   ersetzt werden, um Divisionen durch 0 zu verhindern.
   (b) In Zeile 347 muss *Dictionary<String,Object> properties = new Hashtable<String,Object>();* um *final* erweitert werden.
   (c) In Zeile 355 muss *eventAdmin.postEvent(new Event(TOPIC_JOB_MANAGER,properties));* durch

```
Thread t = new Thread(){
        @Override
        public void run() {
                handleEvent(new Event(TOPIC_JOB_MANAGER,properties));
        }
};
t.start();
```

ersetzt werden. Sonst kann es vorkommen, dass das Event-Handling nicht angestoßen wird und die Verarbeitung des Texts nicht beginnt. Die Ursache dafür ist nicht geklärt.

3. Falls ein Proxy verwendet wird, kann dieser in *data/sites/dbpedia/pom.xml* in Zeile 122 hinzugefügt werden:

```
<setproxy proxyhost="host" proxyport="port"/>
```

4. Im Stanbol Hauptverzeichnis kann nun *mvn clean install -DskipTests* aufgerufen werden

Mit *java -Xmx1g -jar stable/target/org.apache.stanbol.launchers.stable-snapshot-version-SNAPSHOT.jar* kann man nun Stanbol starten. Das Webinterface steht unter `http://localhost:8080` bereit.

A.2.2. StanbolµS

Es reicht, im StanbolµS-Verzeichnis *mvn clean install* aufzurufen, um alle Komponenten zu bauen. Alternativ können auch nur Teile über die Projekt-Dateien in den einzelnen Ordnern gebaut werden.

A.2.3. Monolith

Nachdem Apache Stanbol gebaut wurde, kann der Monolith über *mvn clean install* im Hauptverzeichnis gebaut werden. Die ausführbare Jar-Datei befindet sich dann in *monolith/launcher/target*.

A.2.4. System-Monitor

Der System-Monitor ist ein Maven-Projekt und kann ebenfalls über *mvn install* erstellt werden.

A.3. Starten der Anwendungen

A.3.1. StanbolµS

Zum einfachen lokalen Starten der Anwendung können die mitgelieferten Skripte verwendet werden. Um alle Services und benötigten Komponenten zu starten, muss das Skript *StanbolµS/run/start_stanbolus.bat* verwendet werden. Vorraussetzung dafür sind ein Windows-Betriebsystem und eine Java-Installation mit Java 8.
Alternativ müssen die Komponenten über die ausführbaren jar-Dateien in den target-Ordnern der Komponenten wie folgt gestartet werden:

- der Eureka-Server
- der System-Service mit den System-Properties
  - *ResponseListener.host*
  - *ResponseListener.port* Port, auf dem der Response-Listener hört
  - *SenderEnhancementEngine.timeout* Timeout für das Warten auf Ergebnisse
  - *SenderEnhancementEngine.executionPlan* Komma-separierte Liste von Adressen der Lastverteiler, letzter Eintrag muss die Adresse des ResponseListeners sein
  - *-Xmx1g*

  und dem Argument *-p* *PORT*

  ]. Die rot markierten Einträge sind für das Bundle Eureka-Registrator benötigt worden und müssen nur gesetzt werden, wenn dieses verwendet wird.
- Für jeden anderen Service:
  - ein Lastverteiler mit den System-Properties
    - *server.port* Port des Lastverteilers
    - *spring.application.name* Virtueller Name des Lastverteilers

    Und einem Argument *-s SERVICENAME*, wobei SERVICENAME der Name der Service-Instanzen des Lastverteilers ist
  - einen Wrapper mit den System-Properties
    - *spring.application.name* Virtueller Name der Service-Instanzen (wie im -s Argument des Lastverteilers)
    - *server.port* Port des Wrappers
    - *stanbol.url* Url, unter der Stanbol aufgerufen werden soll, muss *?outputContent=*/** enthalten, um alle Informationen auszugeben
  - die zugehörige Stanbol-Instanz mit System-Property *-Xmx1g* und Argument *-p PORT*

Mit den mitgelieferten Skripten wird folgende Konfiguration gestartet:

Port	Komponente	VirtualHostname
12500	Eureka	-
12501	System-Service	-
12502	Response-Listener	-
12520	LangDetect Lastverteiler	-
12521	LangDetect Wrapper	LANGDETECT
12522	LangDetect Stanbol Instanz	-
12530	OpenNLP-Sentence Lastverteiler	-
12531	OpenNLP-Sentence Wrapper	OPENNLP-SENTENCE
12532	OpenNLP-Sentence Stanbol Instanz	-
12540	OpenNLP-Token Lastverteiler	-
12541	OpenNLP-Token Wrapper	OPENNLP-TOKEN
12542	OpenNLP-Token Stanbol Instanz	-
12550	OpenNLP-POS Lastverteiler	-
12551	OpenNLP-POS Wrapper	OPENNLP-POS
12552	OpenNLP-POS Stanbol Instanz	-
12560	OpenNLP-NER Lastverteiler	-
12561	OpenNLP-NER Wrapper	OPENNLP-NER
12562	OpenNLP-NER Stanbol Instanz	-

Nach dem Start von StanbolµS ist die Eureka-Oberfläche unter `http://http://localhost:12500/` erreichbar. Hier sollte geprüft werden, ob alle Loadbalancer und Wrapper registriert sind. Unter `http://localhost:12501/enhancer` ist die Oberfläche des Monolithen bzw. den System-Service zu erreichen. Wenn das System vollständig gestartet ist, sollten bei einer Analyse des Satzes *„Angela Merkel visited the city Paris.“* die Person *Angela Merkel*, die Sprache *Englisch* und der Ort *Paris* erkannt werden.

A.3.2. Monolith

Im Ordner *monolith/run* befinden sich zwei Skripte. *install_monolith.bat* kopiert die Jar-Datei des Monolithen in den aktuellen Ordner. Anschließend kann das Skript *start_monolith.bat* zum Starten verwendet werden. Dieser Startvorgang dauert einige Sekunden und ist beendet, wenn in der Konsole "*INFO * [main] Startup completedërscheint. Der Monolith ist anschließend unter `http://localhost:12501/enhancer` erreichbar.

A.3.3. System-Monitor

Nach dem bauen des System-Monitors befindet sich unter *system-monitor/target* eine ausführbare Jar-Datei. Diese muss mit zwei Argumenten gestartet werden. Das erste Argument ist

die Anzahl der Messungen. Das zweite ist die Zeit zwischen den Messungen in Millisekunden. Wenn also als Argumente 10 und 1000 verwendet werden, erzeugt der System-Monitor in der Datei stats.txt 10 Reihen mit Werten und der Aufruf dauert etwa 10 Sekunden.

A.4. Entwicklung

Die Erstellung eigener Stanbol-Launcher ist unter `http://stanbol.apache.org/docs/trunk/production-mode/your-launcher.html` dokumentiert. Abhängigkeiten der Module werden über Maven aufgelöst. Da einige der mitgelieferten Module von anderen Abhängig sind, sollte StanbolμS nach Abschnitt A.2 gebaut werden, bevor die Projekte in einer IDE wie Eclipse geöffnet werden.

A.5. Python-Skripte

Unter *evaluation/scripts* sind die für die Evaluierung verwendeten Python-Skripte gespeichert. Voraussetzungen zur Anwendung ist Python 2.7.12[1] und das Python-Modul *requests*[2]. Das Skript *test.py* nimmt als einziges Argument eine URL wie zum Beispiel `http://localhost:12501/enhancer` entgegen. Es sendet den Text aus *evaluation/scripts/dataset.txt* ein mal an die URL und prüft, ob die Antwort vom Typ 200 (OK) ist.
Das Skript *producer_sequential.py* nimmt als zusätzlichen Parameter die Anzahl der Iterationen entgegen. Die Laufzeiten werden in *evaluation/scripts/producer_sequential-results* abgelegt. Für die Verwendung von *producer_parallel.py* sind vier Argumente erforderlich: Die URL, die Anzahl der Wiederholungen pro Thread-Anzahl, die maximale Thread-Anzahl und die Anzahl der Aufrufe pro Thread. Die Ergebnisse werden in *evaluation/scripts/producer_parallel-results* gespeichert.

[1] `https://www.python.org/downloads/`
[2] `http://docs.python-requests.org/en/master/`

Literaturverzeichnis

[Abbott und Fisher 2009] M. L. Abbott und M. T. Fisher. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education, 2009. (Siehe Seite 44)

[Amazon 2015] E. Amazon. Amazon web services. *Available in: http://aws. amazon. com/es/ec2/(November 2012)* (2015). (Siehe Seiten viii und 81)

[Basili 1992] V. R. Basili. Software modeling and measurement: the Goal/Question/Metric paradigm (1992). (Siehe Seiten viii und 61)

[Baxter und Vogt 2002] S. Baxter und L. C. Vogt. Content management system. US Patent 6,356,903. März 2002. (Siehe Seite vii)

[Berners-Lee u. a. 2005] T. Berners-Lee, R. Fielding und L. Masinter. RFC 3986: Uniform resource identifier (uri): Generic syntax. *The Internet Society* (2005). (Siehe Seite viii)

[Berry und Castellanos 2007] M. Berry und M. Castellanos. Survey of Text Mining II: Clustering, Classification, and Retrieval. 2007. (Siehe Seite 24)

[Bhattacharya und Getoor 2007] I. Bhattacharya und L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), Seite 5. (Siehe Seite 24)

[Box u. a. 2000] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte und D. Winer. Simple object access protocol (SOAP) 1.1. 2000. (Siehe Seite viii)

[Bray u. a. 1998] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler und F. Yergeau. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210* 16 (1998). (Siehe Seite vii)

[Brewer 2000] E. A. Brewer. Towards robust distributed systems. In: *PODC*. Band 7. 2000. (Siehe Seite 42)

[Cattell 2011] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record* 39.4 (2011), Seiten 12–27. (Siehe Seite 41)

[Chapman 2015] P. Chapman. Microservices with Spring. 14. Juli 2015. URL: `https://spring.io/blog/2015/07/14/microservices-with-spring`. (Siehe Seite 4)

[Chappell 2004] D. Chappell. Enterprise service bus. Ö'Reilly Media, Inc.", 2004. (Siehe Seiten vii und 37)

[Chowdhury 2003] G. G. Chowdhury. Natural language processing. *Annual review of information science and technology* 37.1 (2003), Seiten 51–89. (Siehe Seiten 2 und 7)

[Christ und Nagel 2011] F. Christ und B. Nagel. A Reference Architecture for Semantic Content Management Systems. In: *EMISA*. 2011, Seiten 135–148. URL: https://www.researchgate.net/profile/Robert_Winter/publication/221276398_Towards_a_More_Integrated_EA_Planning_Linking_Transformation_Planning_with_Evolutionary_Change/links/0046352f504177d9c3000000.pdf#page=140. (Siehe Seiten vii und 13)

[Christensen 2012] B. Christensen. Fault Tolerance in a High Volume, Distributed System. 29. Feb. 2012. URL: http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html. (Siehe Seite 46)

[Christensen u. a. 2001] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana u. a. Web services description language (WSDL) 1.1. 2001. (Siehe Seiten vii und 31)

[Ciaramita und Altun 2006] M. Ciaramita und Y. Altun. Broad-coverage sense disambiguation and information extraction with a supersense sequence tagger. In: *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. 2006, Seiten 594–602. (Siehe Seite 24)

[Ciaramita u. a. 2005] M. Ciaramita, A. Gangemi, E. Ratsch, J. Saric und I. Rojas. Unsupervised learning of semantic relations between concepts of a molecular biology ontology. In: *IJCAI*. Citeseer. 2005, Seiten 659–664. (Siehe Seite 24)

[Conway 1968] M. E. Conway. How do committees invent. *Datamation* 14.4 (1968), Seiten 28–31. (Siehe Seite 43)

[Coppola u. a. 2009] B. Coppola, A. Gangemi, A. Gliozzo, D. Picca und V. Presutti. Frame detection over the semantic web. In: *The Semantic Web: Research and Applications*. Springer, 2009, Seiten 126–142. (Siehe Seite 25)

[Crockford 2006] D. Crockford. The application/json media type for javascript object notation (json) (2006). (Siehe Seite vii)

[Dale 2010] R. Dale. Classical Approaches to Natural Language Processing. In: *Handbook of Natural Language Processing*. Band 2. CRC Press, 2010. (Siehe Seiten 7, 8, 22 und 48)

[duden.de 2016] duden.de, Herausgeber. Duden Online. 2016. URL: http://www.duden.de/rechtschreibung. (Siehe Seite 9)

[Evans 2004] E. Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004. (Siehe Seite vii)

[Fielding 2000] R. T. Fielding. Architectural styles and the design of network-based software architectures. Dissertation. University of California, Irvine, 2000. (Siehe Seite 15)

[Fielding u. a. 1999] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee. Hypertext transfer protocol–HTTP/1.1. Technischer Bericht. 1999. (Siehe Seiten vii und 14)

[Foundation 2010] T. A. S. Foundation. stanbol.apache.org. 2010. URL: http://stanbol.apache.org/. (Siehe Seite 13)

[Foundation 2011] T. A. S. Foundation. apache clerezza. 2011. URL: https://clerezza.apache.org/. (Siehe Seite 14)

[Fowler 2005] M. Fowler. Event sourcing. *Online, Dec* (2005). (Siehe Seite 18)

[Fowler 2010] M. Fowler. Richardson Maturity Model. *steps toward the glory of REST*. 18. März 2010. URL: http://martinfowler.com/articles/richardsonMaturityModel.html. (Siehe Seite 15)

[Galiegue und Zyp 2013] F. Galiegue und K. Zyp. JSON Schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)* (2013). (Siehe Seite 32)

[Gangemi 2013] A. Gangemi. A comparison of knowledge extraction tools for the semantic web. In: *The Semantic Web: Semantics and Big Data*. Springer, 2013, Seiten 351–366. (Siehe Seiten 9, 24, 49 und 80)

[Gédéon 2010] W. Gédéon. OSGi and Apache Felix 3.0. *Birmingham: Packt Publishing* (2010). (Siehe Seiten 13, 14)

[Goddard und Schalley 2010] C. Goddard und A. C. Schalley. Semantic Analysis. In: *Handbook of Natural Language Processing*. Band 2. CRC Press, 2010. (Siehe Seiten 8, 23 und 26)

[Gosling 2000] J. Gosling. The Java language specification. Addison-Wesley Professional, 2000. (Siehe Seite 13)

[Greene und Cunningham 2006] D. Greene und P. Cunningham. Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering. In: *Proc. 23rd International Conference on Machine learning (ICML'06)*. ACM Press, 2006, Seiten 377–384. (Siehe Seite 63)

[Hall u. a. 2011] R. Hall, K. Pauls, S. McCulloch und D. Savage. OSGi in action: Creating modular applications in Java. Manning Publications Co., 2011. (Siehe Seite vii)

[Haoyu und Haili 2012] W. Haoyu und Z. Haili. Basic Design Principles in Software Engineering. In: *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*. IEEE. 2012, Seiten 1251–1254. (Siehe Seite vii)

[Hartmann u. a. 2012] S. Hartmann, G. Szarvas und I. Gurevych. Mining multiword terms from Wikipedia. *Semi-Automatic Ontology Development: Processes and Resources* (2012), Seiten 226–258. (Siehe Seite 24)

[Hasselbring 2016a] W. Hasselbring. Innovative Softwareentwicklungskultur und innovative Softwarearchitekturen: DevOps & Microservices. In: *Kieler Woche Digital*. Juni 2016. URL: http://eprints.uni-kiel.de/33213/. (Siehe Seite 79)

[Hasselbring 2016b] W. Hasselbring. Microservices for Scalability (Keynote Presentation). In: *7th ACM/SPEC International Conference on Performance Engineering (ACM/SPEC ICPE 2016)*. März 2016. URL: http://eprints.uni-kiel.de/31711/. (Siehe Seite 79)

[Hasselbring 2016c] W. Hasselbring. Microservices for Scalability: Keynote Talk Abstract. In: *International Conference on Performance Engineering (ICPE 2016)*. Herausgegeben von A. Avritzer und A. Iosup. ACM, März 2016, Seiten 133–134. URL: http://eprints.uni-kiel.de/31829/. (Siehe Seite 79)

[Hesse u. a. 2004] W. Hesse, R. H. Kaschek, H. C. Mayr und B. Thalheim. Ontologien in der und für die Softwaretechnik. In: *Modellierung*. 2004, Seiten 269–270. (Siehe Seite 24)

[Hogenboom u. a. 2011] F. Hogenboom, F. Frasincar, U. Kaymak und F. De Jong. An overview of event extraction from text. In: *Workshop on Detection, Representation, and Exploitation of Events in the Semantic Web (DeRiVE 2011) at Tenth International Semantic Web Conference (ISWC 2011)*. Band 779. Citeseer. 2011, Seiten 48–57. (Siehe Seite 25)

[Hohpe und Woolf 2002] G. Hohpe und B. Woolf. Enterprise integration patterns. In: *9th Conference on Pattern Language of Programs*. 2002, Seiten 1–9. (Siehe Seite 30)

[Indurkhya und Damerau 2010] N. Indurkhya und F. J. Damerau. Handbook of Natural Language Processing. Band 2. CRC Press, 2010. (Siehe Seite vii)

[*ISO 9216*]. ISO 9216. *Information technology – Software product quality – Part 1: Quality model*. Standard. Geneva, CH: International Organization for Standardization, März 2000. (Siehe Seite 4)

[Jacobs 2009] A. Jacobs. The pathologies of big data. *Communications of the ACM* 52.8 (2009), Seiten 36–44. (Siehe Seite 62)

[Johanson u. a. 2016] A. N. Johanson, S. Flögel, W.-C. Dullo und W. Hasselbring. OCEAN-TEA: EXPLORING OCEAN-DERIVED CLIMATE DATA USING MICROSERVICES. In: *Proceedings of the 2016 Climate Informatics Workshop (CI2016)*. 2016. (Siehe Seiten 79, 80)

[Knoche 2016] H. Knoche. Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. In: *International Conference on Performance Engineering (ICPE 2016)*. ACM, März 2016, Seiten 121–124. URL: http://eprints.uni-kiel.de/31830/. (Siehe Seite 80)

[Kratzke 2015] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *Proceedings of CLOUD COMPUTING* 2015 (2015). (Siehe Seite 81)

[Kraus u. a. 2013] S. Kraus, G. Steinacker und O. Wegner. Teile und Herrsche–Kleine Systeme für große Architekturen. *OBJEKTspektrum* 5 (2013), Seiten 8–13. (Siehe Seiten 11, 18, 29, 41, 43, 44 und 79)

[Kumar 2012] A. Kumar. Analyzing Medical Records with Apache Stanbol. 13. Juli 2012. URL: http://blog.iks-project.eu/analyzing-medical-records-with-apache-stanbol/. (Siehe Seite 80)

[Lewis und Fowler 2011] J. Lewis und M. Fowler. PolyglotPersistence. 16. Nov. 2011. URL: http://martinfowler.com/bliki/PolyglotPersistence.html. (Siehe Seite 41)

[Lewis und Fowler 2014] J. Lewis und M. Fowler. Microservices. *a definition of this new architectural term*. 10. März 2014. URL: http://martinfowler.com/articles/microservices.html. (Siehe Seite 9)

[Lindholm u. a. 2014] T. Lindholm, F. Yellin, G. Bracha und A. Buckley. The Java Virtual Machine Specification: Java SE 8 Edition. Pearson Education, 2014. (Siehe Seite viii)

[Linthicum 2000] D. S. Linthicum. Enterprise application integration. Addison-Wesley Professional, 2000. (Siehe Seite vii)

[Liu 2014] D. Liu. Eureka at a glance. 18. Dez. 2014. URL: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance. (Siehe Seite 32)

[Manola u. a. 2004] F. Manola, E. Miller, B. McBride u. a. RDF primer. *W3C recommendation* 10.1-107 (2004), Seite 6. (Siehe Seiten viii und 14)

[Martin 2002] R. C. Martin. The single responsibility principle. *The Principles, Patterns, and Practices of Agile Software Development* (2002), Seiten 149–154. (Siehe Seite vii)

[Masinter u. a. 2005] L. Masinter, T. Berners-Lee und R. T. Fielding. Uniform resource identifier (URI): Generic syntax (2005). (Siehe Seite 24)

[Michelson 2006] B. M. Michelson. Event-driven architecture overview. *Patricia Seybold Group* 2 (2006). (Siehe Seiten vii und 38)

[Mockapetris und Dunlap 1988] P. Mockapetris und K. J. Dunlap. Development of the domain name system. Band 18. 4. ACM, 1988. (Siehe Seiten vii und 32)

[Moschitti u. a. 2008] A. Moschitti, D. Pighin und R. Basili. Tree kernels for semantic role labeling. *Computational Linguistics* 34.2 (2008), Seiten 193–224. (Siehe Seite 24)

[Nadeau und Sekine 2007] D. Nadeau und S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes* 30.1 (2007), Seiten 3–26. (Siehe Seite 24)

[Newcomer und Lomow 2005] E. Newcomer und G. Lomow. Understanding SOA with Web services. Addison-Wesley, 2005. (Siehe Seiten 12 und 37)

[Newman 2015] S. Newman. Building Microservices. Ö'Reilly Media, Inc.", 2015. (Siehe Seiten 10, 11, 19, 29, 36, 38, 39, 41, 43–45)

[Nygard 2007] M. Nygard. Release it!: design and deploy production-ready software. Pragmatic Bookshelf, 2007. (Siehe Seite 46)

[Palmer 2010] D. D. Palmer. Text Preprocessing. In: *Handbook of Natural Language Processing*. Band 2. CRC Press, 2010. (Siehe Seiten 19 und 21)

[Peltz 2003] C. Peltz. Web services orchestration and choreography. *Computer* 10 (2003), Seiten 46–52. (Siehe Seite 38)

[Pereira 2012] J. Pereira. IKS - The Semantic CMS Community. 13. Juli 2012. URL: http://wiki.iks-project.eu/index.php/Main_Page. (Siehe Seite viii)

[Pereira 2013] J. Pereira. iks-project.eu. 11. März 2013. URL: http://wiki.iks-project.eu/index.php/Stanbol. (Siehe Seite 13)

[Perrey und Lycett 2003] R. Perrey und M. Lycett. Service-oriented architecture. In: *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, Seiten 116–119. (Siehe Seite vii)

[Ponzetto und Strube 2011] S. P. Ponzetto und M. Strube. Taxonomy induction based on a collaboratively built knowledge repository. *Artificial Intelligence* 175.9-10 (2011), Seiten 1737–1756. (Siehe Seite 24)

[Postel 1981] J. Postel u. a. RFC 791: Internet protocol (1981). (Siehe Seite vii)

[Python] Python Language Reference, version 2.7. Python Software Foundation, 2010-07-03. URL: http://www.python.org. (Siehe Seite 62)

[Richardson 2014] C. Richardson. Pattern: Microservices Architecture. 2014. URL: http://microservices.io/patterns/microservices.html. (Siehe Seite 22)

[Richardson 2015a] C. Richardson. Event-Driven Data Management for Microservices. 4. Dez. 2015. URL: https://www.nginx.com/blog/event-driven-data-management-microservices/. (Siehe Seiten 18, 38, 39)

[Richardson 2015b] C. Richardson. Inter-Process Communication in a Microservices Architecture. 24. Juli 2015. URL: https://www.nginx.com/blog/building-microservices-inter-process-communication/. (Siehe Seite 30)

[Richardson 2015c] C. Richardson. Introduction to Microservices. 19. Mai 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/. (Siehe Seite 29)

[Richardson 2015d] C. Richardson. Service Discovery in a Microservices Architecture. 12. Okt. 2015. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. (Siehe Seite 32)

[Richardson 2016a] C. Richardson. Choosing a Microservices Deployment Strategy. 10. Feb. 2016. URL: https://www.nginx.com/blog/deploying-microservices/. (Siehe Seite 18)

[Richardson 2016b] C. Richardson. Choosing a Microservices Deployment Strategy. 10. Feb. 2016. URL: https://www.nginx.com/blog/deploying-microservices/. (Siehe Seiten 45, 46)

[Richardson u. a. 2007] L. Richardson, S. Ruby und T. Demmig. Web-Services mit REST. O'Reilly Germany, 2007. (Siehe Seiten vii und 4)

[Steinacker 2014] G. Steinacker. Scaling with Microservices and Vertical Decomposition. 29. Juli 2014. URL: https://dev.otto.de/2014/07/29/scaling-with-microservices-and-vertical-decomposition/. (Siehe Seiten 11 und 79)

[Team 2013] R. C. Team u. a. R: A language and environment for statistical computing (2013). (Siehe Seite 65)

[Thompson u. a. 2004] H. S. Thompson, D. Beech, M. Maloney und N. Mendelsohn. XML schema part 1: structures second edition. *W3C Recommendation (28 October 2004) http://www.w3.org/TR* (2004). (Siehe Seite 32)

[Villamizar u. a. 2016] M. Villamizar, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang u. a. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In: *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE. 2016, Seiten 179–182. (Siehe Seiten 80, 81)

[Vogels 2009] W. Vogels. Eventually consistent. *Communications of the ACM* 52.1 (2009), Seiten 40–44. (Siehe Seite 42)

[Weerawarana u. a. 2005] S. Weerawarana, F. Curbera, F. Leymann, T. Storey und D. F. Ferguson. Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more. Prentice Hall PTR, 2005. (Siehe Seite 37)

[Wilcox 1996] R. R. Wilcox. Statistics for the social sciences. Academic Press, 1996. (Siehe Seite 65)

[Winer 1999] D. Winer u. a. Xml-rpc specification. 1999. (Siehe Seite viii)

[Wolff 2016a] E. Wolff. Microservices. *Grundlagen flexibler Softwarearchitekturen*. dpunkt.verlag, 2016. (Siehe Seiten 8–12, 18, 19, 25, 27, 29, 38–41, 43 und 46)

[Wolff 2016b] E. Wolff. Microservices Primer. 19. Jan. 2016. URL: http://leanpub.com/microservices-primer. (Siehe Seite 18)

[Zimmermann 1980] H. Zimmermann. OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications* 28.4 (1980), Seiten 425–432. (Siehe Seite 14)