Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

im Bachelorstudiengang Wirtschaftsinformatik

Entwicklung einer Schnittstelle für die Anbindung von austauschbaren Datenquellen an KI-Algorithmen

von

Laurenz Anton Dilba

Erstprüfer:	Prof. Dr. Matthias Bertram
Zweitprüfer:	Prof. Dr. Wolfgang Heiden
Unternehmen:	CONET Solutions GmbH
Eingereicht am:	9. Januar 2023

Eidesstattliche Erklärung

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

Inhaltsverzeichnis

Abbildungsverzeichnis v

Tabellenverzeichnis vi

Code Listings vii

Abkürzungsverzeichnis viii

1 Einleitung 1
- 1.1 Motivation und Hintergrund . . . 1
- 1.2 Problemstellung . . . 1
- 1.3 Zielsetzung . . . 2
- 1.4 CONET Solutions GmbH . . . 2
- 1.5 Vorgehen . . . 2

2 Grundlagen 4
- 2.1 Schnittstelle . . . 4
- 2.2 REST und RabbitMQ . . . 4
  - 2.2.1 Application-Programming-Interface . . . 4
  - 2.2.2 Representational-State-Transfer . . . 6
  - 2.2.3 RabbitMQ . . . 7
- 2.3 Microservice-Architekturen . . . 11
- 2.4 Künstliche Intelligenz . . . 11
- 2.5 Verwendete Werkzeuge . . . 13
  - 2.5.1 Python-API mit Flask . . . 13
  - 2.5.2 REDIS und MySQL-Datenbanken . . . 14
  - 2.5.3 Angular-Frontend . . . 14
  - 2.5.4 Logging durch Grafana . . . 15
  - 2.5.5 Deployment über Docker . . . 15

3 Methodik 17
- 3.1 Anforderungen definieren . . . 18
- 3.2 Konzept entwickeln . . . 18
- 3.3 Prototypische Implementierung . . . 19
- 3.4 Umsetzung evaluieren . . . 19
- 3.5 Anpassen . . . 20

4 Konzeption und prototypische Umsetzung 21
- 4.1 Anforderungen . . . 21
- 4.2 Konzeption . . . 25
  - 4.2.1 Softwarearchitektur . . . 25
  - 4.2.2 Programmablauf . . . 27
- 4.3 Prototypische Umsetzung . . . 31
  - 4.3.1 Implementierung der REST-API . . . 31
  - 4.3.2 Nutzeridentifizierung mit JWT . . . 32
  - 4.3.3 Caching mit Redis-Datenbank . . . 34
  - 4.3.4 Kommunikation zwischen Backend und Services mit RabbitMQ . . . 35
  - 4.3.5 Implementierung des KI-Services . . . 38
  - 4.3.6 Management der Services . . . 41

4.3.7 Automatisierte Transformation des Inputs 41
4.3.8 Fehlerbehandlung 43
4.3.9 Event Logging 44
4.3.10 Website mit Angular 46
4.4 Deployment der Software mit Docker 52

5 Evaluation **55**

6 Fazit und Ausblick **57**
6.1 Fazit 57
6.2 Ausblick 58

7 Literaturverzeichnis **59**

Abbildungsverzeichnis

1 UML Sequenzdiagramm einer API 5
2 Grundlegende Kommunikation 8
3 Kommunikation über einen Broker nach Dossot 9
4 Darstellung der AMQP Konzepte nach Dossot 10
5 Use Case Diagramm der Interaktion mit der Schnittstelle 22
6 Softwarearchitekturdiagramm 26
7 UML-Sequenzdiagramm des Programmablaufs 27
8 Mockup Frontend 28
9 Mockup Query 28
10 Mockup Transformation 29
11 Mockup Service 30
12 Kommunikation mit RabbitMQ 38
13 Ablaufdiagramm der Textähnlichkeitssuche 39
14 Logs in Grafana 46
15 Gesamtansicht der Website 47
16 Query Kachel im Frontend 49
17 Transformation Kachel im Frontend 50
18 Service-Kachel im Frontend 52

Tabellenverzeichnis

1 HTTP Statuscodes nach Doglio, 2015 6
2 Schritte des modifizierten Wasserfallmodells 17
3 Implementierte Routen der REST-API 32
4 Log Level des Event Logging Systems 44

Code Listings

1 Aufsetzen einer Flask-Instanz . . . 31
2 Flask im eigenen Thread starten . . . 31
3 Beispiel einer API-Routendefinition über Annotations . . . 31
4 Respond Funktion zur Rückgabe von JSON-Responses . . . 32
5 Generierung der UUID und Erstellung des JWT . . . 33
6 Route zum Upload von Queries mit Nutzung des JWTs . . . 34
7 Aufsetzen der Redis-Verbindung . . . 34
8 Set-Funktion aus Redis . . . 35
9 Get-Funktion aus Redis . . . 35
10 Verbindung zu RabbitMQ und Erstellung eines Channels und einer Queue . 36
11 Aufsetzen und Konsumieren der RabbitMQ-Queue im KI-Service . . . 37
12 Senden eines KI-Ergebnisses an das Backend . . . 37
13 Laden des BERT-Modells . . . 39
14 Starten der Elasticsearch-Instanz . . . 39
15 Auslesen der search_size und Ausführen der KI . . . 40
16 Ähnlichkeitssuche in der Elasticsearch . . . 40
17 Beispiel einer Transformationsanleitung . . . 42
18 Transformationsalgorithmus . . . 42
19 Exception-Handling mithilfe von Wrappern . . . 43
20 Erstellung eines Logs . . . 44
21 Query zur Abfrage der Logs in Grafana . . . 45
22 Definition eines Components . . . 47
23 Erstellung einer Textarea innerhalb der HTML-Datei . . . 48
24 Style-Definition innerhalb einer SCSS-Datei . . . 48
25 Anzeige der Results aus der Transformation . . . 49
26 Abfrage der Services . . . 50
27 Subscriben auf einen Observable . . . 51
28 Dropdown zur Auswahl der Services . . . 51
29 Beispiel einer Dockefile . . . 52
30 Build Befehl zur Erstellung von Docker-Images aus einer Compose-Datei . 53
31 Docker-Compose-Datei zum Bauen der Docker-Images . . . 53
32 Docker-Compose-Datei zum Aufsetzen und Starten der Container . . . 54
33 Starten eines Docker-Containers . . . 54
34 Beenden eines Docker-Containers . . . 54

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
BL	Business Logic
CONET	CONET Solutions GmbH
DSGVO	Datenschutz-Grundverordnung
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikation
IEC	International Electrotechnical Commission
IT	Informationstechnik
JSON	JavaScript Object Notation
JWT	JSON Web Token
KI	Künstliche Intelligenz
MAC	Message Authentication Code
RAM	Random-Access Memory
RDBMS	relationalen Datenbankmanagementsystemen
REST	Representational State Transfer
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtuelle Maschine
WSL	Windows-Subsystem für Linux

1 Einleitung

1.1 Motivation und Hintergrund

Jeden Tag werden weltweit riesige Mengen an Daten - im Jahr 2020 beispielsweise 64,2 Zettabyte - produziert.[1] Dies entspricht 64,2 Billionen Gigabyte. Mithilfe von Datenmanagement und Datenanalyse wird versucht, die Daten so weit aufzubereiten, dass sie durch Menschen und Algorithmen ausgewertet werden können. Je nach Datenmenge und Abweichung der Daten untereinander kann dies ein aufwändiger und damit teurer Prozess sein. Bei größerer Komplexität oder Menge der Daten wird es für Menschen schwerer, Zusammenhänge, Abweichungen und Auffälligkeiten zu erkennen. Dies liegt u. a. daran, dass Muster von neu erfassten Daten nur durch Erinnerungen aus dem Kurzzeitgedächtnis abgeleitet werden können.[2] Um dieses Problem zu lösen, wurden Algorithmen entwickelt, die mit großen Datenmengen trainiert werden, um allgemeine Aussagen über die eingegebenen Daten zu treffen. Je nach Datenquelle und Art der Aussage, die über diese Daten getroffen werden soll, bedarf es an werden unterschiedlichen Algorithmen aus dem Bereich der Künstliche Intelligenz (KI). Durch den Anschluss einer Datenquelle mit einer KI entsteht eine feste Verdrahtung zwischen dem Datenerhebungsalgorithmus und dem KI-gestützten Datenverarbeitungsalgorithmus. Sollte sich entweder die Datenerhebung oder die Auswertung verändern, muss in der Regel die Anbindung der Datenquelle an die KI überarbeitet werden. Dies erfordert spezifisches Know-how im Bereich der eingesetzten KI, der Datenerhebung und der bereits programmierten Schnittstellen.

1.2 Problemstellung

Jede KI benötigt Daten in einem bestimmten Format. KI-basierte Textanalysealgorithmen, beispielsweise der von Google entwickelte BERT-Algorithmus, benötigen reinen ASCII-Text als Input. Ein Entwickler, der eine KI mit gesammelten Daten benutzen möchte, muss diese Daten in das von der KI benötigte Format übertragen. Bei einem Austausch der KI oder des Datenerhebungsalgorithmus hat der Entwickler die Daten auf Kompatibilität und das Ergebnis auf Korrektheit zu prüfen.

Das Thema der Bachelorarbeit ist die Konzeption und Entwicklung einer Schnittstelle, die den Anschluss von Daten an KI-gestützte Datenverarbeitungsalgorithmen vereinfacht. Wie kann ein Entwickler nach Einrichtung der KI die Daten austauschen, ohne die Schnittstelle neu programmieren zu müssen? Wie kann ein Entwickler eine bereits eingesetzte KI mit einer anderen austauschen, ohne die Daten verändern zu müssen?

[1] Statista, 2021.
[2] Snyder, 2000.

1.3 Zielsetzung

Die im Rahmen der Bachelorarbeit entwickelte Schnittstelle bietet diverse Vorteile. Die Schnittstelle ermöglicht, Daten zur Laufzeit der Anwendung einzugeben und diese Daten an einen KI-gestützten Datenverarbeitungsalgorithmus anzuschließen. Ferner ermöglicht die Schnittstelle den dynamischen Austausch der KI. Der Anwender konfiguriert den Anschluss der Daten über eine Transformationsanleitung, um die Anfrage für die KI vorzubereiten. Das aus der KI resultierende Ergebnis wird dem Nutzer angezeigt. Die Konzeption der Schnittstelle wird mit einem Textanalysealgorithmus implementiert, der eine semantische Suche innerhalb eines Textes durchführt. Das Backend ist als REST-API entwickelt. Dadurch können Entwickler die Schnittstelle mit unterschiedlichen oder veränderten Daten und KIs nutzen. Durch die Schnittstelle verringert sich der Arbeitsaufwand, der durch die Anbindung und Nutzung neuer KIs entsteht. Entwickler werden entlastet sowie Personalkosten verringert. Schließlich können alternativen Aufbereitungen der Daten und neue KIs mit geringem Mehraufwand getestet werden.

1.4 CONET Solutions GmbH

Firma:	CONET Solutions GmbH
Muttergesellschaft:	CONET Technologies Holding GmbH
Mitarbeitende:	1.700
Hauptsitz:	Theodor-Heuss-Allee 19, 53773 Hennef (Sieg)
Gründungsjahr:	1987

Das CONET Solutions GmbH (CONET) ist ein IT-Beratungsunternehmen, welches in vielen Bereichen Dienstleistungen anbietet. Das Spektrum der Dienstleistungen umfasst u. a. SAP Consulting, Cyber Security, Cloud Computing, Data Intelligence, Digital Communications & E-Commerce, Critical Communications, Agile Software Development und Management Consulting. CONET ist spezialisiert auf die Entwicklung von Individualsoftware. CONET ist sowohl im privaten als auch im öffentlichen Sektor vertreten. Zu den Kunden gehören u. a. Bundeswehr, Volkswagen AG, Telekom, Deutsche Bahn.[3] CONET ist an 20 Standorten in Europa vertreten.

Thema der Bachelorarbeit, Anforderungserhebung sowie Evaluation wurde gemeinsam mit CONET erarbeitet.

1.5 Vorgehen

Die Bachelorarbeit wird eingeleitet mit der Darstellung von Motivation, Problemstellung und Zielsetzung sowie Vorstellung der die Arbeit betreuenden CONET Solutions GmbH.

[3] CONET, [o. D.], Auszug aus der Kundenliste.

Kapitel 2 erläutert die Grundlagen der Bachelorarbeit. Dargelegt werden Konzepte einer Schnittstelle sowie Methoden zur Kommunikation. APIs, der REST-Architekturstil sowie die Middleware RabbitMQ werden beschrieben. Gleiches gilt für Microservice-Architekturen, auf denen das Konzept der entwickelten Schnittstelle aufbaut. Anschließend wird in das Thema KI eingeführt. Das Kapitel endet mit einer Auflistung und Erläuterung der Einsatzzwecke der für die Bachelorarbeit genutzten Werkzeuge.

Das Kapitel 3 stellt die Methodik dar, mit der die Bachelorarbeit bearbeitet und die Schnittstelle entwickelt wurde. Das Wasserfallmodell wurde als Projektmanagementmethode für die Entwicklung der Schnittstelle beschrieben und mit Begründung ausgewählt.

In Kapitel 4 werden die Ergebnisse der Anforderungserhebung sowie ein Konzept für die Architektur der Schnittstelle beschrieben. Dargestellt werden der Programmablauf und die geplante visuelle Darstellung der Website in Form von Mockups. Abschließend erläutert das Kapitel die technische Umsetzung der einzelnen Bestandteile der Architektur und das Deployment der Software.

Das Kapitel 5 stellt die Evaluation der Schnittstelle dar, einschließlich der mit CONET erarbeiteten Ergebnisse der Code-Reviews.

In Kapitel 6, Fazit und Ausblick, werden etwaige Implikation für die Forschung dargestellt, die aus den entwickelten Komponenten für die Schnittstelle resultieren. Abschließend werden die für einen Einsatz der Schnittstelle in einer Produktivumgebung zwingend erforderlichen sowie sonstige Erweiterungen beschrieben.

2 Grundlagen

2.1 Schnittstelle

Die Schnittstelle in der Informatik beschreibt sowohl den Bereich einer Software, der nach außen hin für andere Programme oder den Nutzer erreichbar ist, als auch eine eigenständig Anwendung, die zwei unabhängige Programme miteinander verbindet. Innerhalb einer Schnittstelle sind Funktionen vorgegeben, durch die der Nutzer mit der Software interagieren kann. Oftmals werden Schnittstellen für eine spezifische Anwendung entwickelt, um die Nutzung dieser Anwendung zu ermöglichen. Das Ziel einer Schnittstelle ist es, die Komplexität von bestehenden Programmen zu verringern. Der Satz der bereitgestellten Funktionen dient zur Vereinfachung der Bedienung komplizierter oder alter Programme.[4]

Es wurden zwei Bereiche innerhalb der Schnittstelle entworfen. Der erste Bereich der Schnittstelle verbindet die Website mit dem Backend. Innerhalb des Backends wurde eine Schnittstelle in Form eines Application Programming Interface (API) implementiert. Der zweite implementierte Bereich der Schnittstelle schließt das Backend mittels RabbitMQ an die einzelnen KI-Services an. In Kapitel 2.2 wird der Aufbau und die Funktionsweise der beiden Schnittstellen genauer erläutert.

2.2 REST und RabbitMQ

2.2.1 Application-Programming-Interface

Eine API ist eine Programmierschnittstelle, die dazu da ist, die Kommunikation zwischen einem Client, oder auch Anwender genannt, und einem Server durch festgelegte Funktionen zu regeln. Der Satz der verfügbaren Funktionen ist durch den Entwickler der API vorgegeben. Eine API sollte nach Möglichkeit selbsterklärend aufgebaut sein.[5] Eine API dient dazu, dem Nutzer Daten bereitzustellen oder dem Server Daten zu senden.

In Abbildung 1 ist die Kommunikation zwischen einem Anwender und der API durch ein Sequenzdiagramm dargestellt. Jede Anfrage an die API findet durch den Aufruf des API Endpunkts durch einen Uniform Resource Locator (URL) statt. Hinter der Grund-URL wird die genaue Ressource innerhalb der API durch den Pfad in der URL angegeben. Jede dieser Funktionen, die über eine bestimmte URL erreichbar sind, kann mehrere Effekte haben. Der Effekt, den eine Funktion hat, ist durch die Hypertext Transfer Protocol (HTTP) Methoden im groben Rahmen vorgegeben. Die Kommunikation zwischen API und Anwender geht immer vom Anwender aus. Die API kann den Anwender von sich aus nicht ansprechen.

[4] Sneed, 2006.
[5] Bloch, 2006.

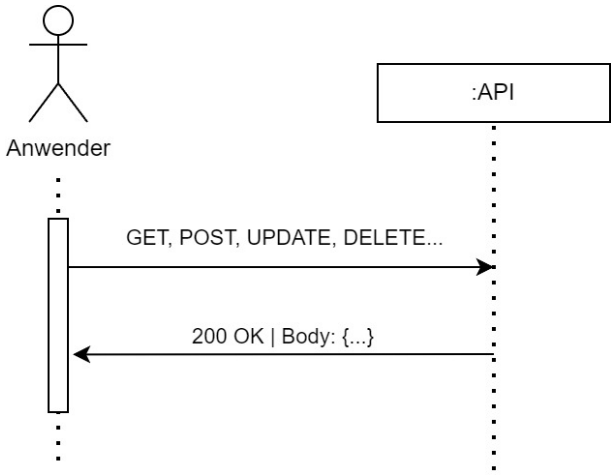

Abbildung 1: UML Sequenzdiagramm einer API

Der gesamte Satz der verfügbaren HTTP-Methoden lautet `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`. Jede der Methoden hat eine bestimmte Aufgabe und bestimmte Rechte, die es vom Entwickler der API einzuhalten gilt.[6] In der Bachelorarbeit werden die Funktionen `GET`, `POST` und `DELETE` verwendet. Routen der API, die mit einer `GET` Funktion aufgerufen werden, sollen lediglich Daten an den Nutzer zurückgeben, ohne Änderungen auf dem Server vorzunehmen. Wenn zweimal hintereinander die gleiche URL mit `GET` aufgerufen wird, sollte auch in beiden Fällen das gleiche Ergebnis von der API zurückgegeben werden.

Die `POST`-Methode liefert Daten vom Nutzer an den Server. Innerhalb des `POST`-Bodys können Daten im JSON-Format an die API gesendet werden. Neben dem JSON-Format gibt es auch noch weitere Möglichkeiten für Formate. Diese werden in der Bachelorarbeit jedoch nicht behandelt. API URLs, die mit einer `POST`-Methode aufgerufen werden, können Veränderungen auf dem Server auslösen. Dies kann Auswirkungen auf die Daten oder den Status des Servers haben, sodass andere Methoden eventuell davon beeinflusst werden. Ein beispielhafter Anwendungsfall für eine `POST`-Methode ist die Registrierung eines neuen Accounts auf einer Website. Dort werden innerhalb des Bodys der Benutzername und das Passwort an die API geschickt. Die API registriert den Account mit den erhaltenen Daten und gibt bei zukünftigen Aufrufen der URL mit den gleichen Daten eine Fehlermeldung zurück, dass der Benutzer bereits registriert ist.

Die `DELETE`-Methode entfernt eine Ressource unter der aufgerufenen URL. Sowohl bei `POST` als auch bei `DELETE`-Methoden ist darauf zu achten, dass der Nutzer die Route nur aufrufen kann, wenn er entsprechende Rechte zur Ausführung verfügt. Ansonsten könnte es zu unkontrollierten Daten- und Statusänderungen innerhalb der API kommen.

[6] MDN, 2022.

Wie in Abbildung 1 zu sehen ist, gibt die API nicht nur die angeforderten Daten zurück, sondern auch einen Statuscode. Ein HTTP-Statuscode besteht aus drei Ziffern. In Tabelle 1 sind die grundlegenden Statuscodes aufgelistet.

Statuscode	Bedeutung
1xx	Informationen, nur unter HTTP 1.1 definiert
2xx	Request war erfolgreich (OK)
3xx	Die Ressource wurde verschoben
4xx	Die Eingabe war fehlerhaft
5xx	Der Server hatte einen Fehler

Tabelle 1: HTTP Statuscodes nach Doglio, 2015

Die Statuscodes, die im Prototyp der Bachelorarbeit verwendet werden, sind Teil der häufiger genutzten Statuscodes für APIs. Der Statuscode `200:OK` ist der am häufigsten auftretende Statuscode. Dieser besagt, dass die Anfrage erfolgreich abgelaufen ist und das Ergebnis zurückgegeben werden konnte. Der Prototyp nutzt ein Authentifizierungssystem. Dadurch kann es dazu kommen, dass bei einer Anfrage mit fehlender Autorisierung der Statuscode `401:Unauthorized` zurückgegeben wird. Weitere Statuscodes, die häufiger auftreten können, sind `404:Not found`, `405:Method not allowed` und `500:Internal server error`. Bei einem Statuscode 404 wurde eine Route aufgerufen, die innerhalb der API nicht definiert ist. Bei dem Statuscode 405 wurde zwar eine vorhandene Route angesprochen, jedoch ist die genutzte HTTP-Methode nicht zulässig. Der letzte verwendete Statuscode 500 beschreibt eine fehlerhafte Ausführung der Anfrage.

Die HTTP-Methoden und Statuscodes dienen dazu, die Entwicklung und Arbeit mit einer API für den Entwickler einfach zu gestalten. Wie die URLs der API aufgebaut sind und welche HTTP-Methoden wann verwendet werden, liegt jedoch vollständig beim Entwickler der API. Um APIs im Allgemeinen etwas zu vereinheitlichen und die Effekte der Funktionen innerhalb der API selbsterklärender werden zu lassen, wurde im Jahr 2000 von Roy Thomas Fielding ein Regelwerk namens Representational State Transfer (REST) veröffentlicht.[7]

2.2.2 Representational-State-Transfer

REST ist ein Architekturstil, die den Aufbau der Kommunikation im World Wide Web. Der Representational-State-Transfer ist kein spezifischer Standard in der Softwareentwicklung. APIs, die versuchen, den Prinzipien dieses Architekturstils zu folgen, werden als „REST-APIs“ oder „RESTful-APIs“bezeichnet. Diese bieten damit einen größtenteils standardisierten Weg, Daten zwischen Client und Server auszutauschen.[8]

[7] R.T. Fielding, 2000.
[8] Richards, 2006.

Innerhalb des REST-Frameworks sind mehrere Aspekte vorgegeben, die eine REST-API ausmachen. Drei der Aspekte sind Simplizität, Skalierbarkeit und Performance. Die Simplizität beschreibt einen allgemeinen, standardisierten Weg, wie der Aufbau und die Kommunikation mit der API ablaufen soll. Dies beinhaltet den Aufbau der Routen und damit der URL, wann Daten an die API geschickt werden sollen, sowie die Form der Daten, die zu versenden sind.

Die Skalierbarkeit beschreibt das Konzept der beliebigen Erweiterung einer API. Das beinhaltet die Entwicklung der API als solche. Routen und damit Möglichkeiten, Daten von der API anzufordern oder Daten an die API zu senden, können während der Entwicklung einfach hinzugefügt und entfernt werden. Das Konzept der Skalierbarkeit beschreibt ebenfalls einen zustandslosen Aufbau der API. Durch die Zustandslosigkeit kann die API horizontal skaliert werden. Eine horizontale Skalierung beschreibt das Hinzufügen neuer Instanzen der API. Jede Instanz ist identisch aufgebaut und kann jede ankommende Anfrage alleinstehend beantworten. Die Anfragen, die an eine REST-API gestellt werden, müssen daher sämtliche für die Bearbeitung notwendigen Informationen bereitstellen.

Der letzte Aspekt des REST-Frameworks beschreibt die Performance. Eine REST-API implementiert ein System zum Zwischenspeichern, Caching gennant, von Responses. Wenn ein Client eine Anfrage mit der HTTP-Methode GET an die API schickt, wird die Antwort, die an den Client zurückgesendet wird, auf dem Server zwischengespeichert. Sollte ein oder mehrere Clients eine identische Anfrage an die API senden, wird das zwischengespeicherte Ergebnis zurückgegeben, statt die dahinter liegende Methode innerhalb der API neu auszuführen. Das ermöglicht eine hohe Performance, auch bei einer großen Anzahl von Anfragen.[9]

2.2.3 RabbitMQ

Kommunikation ist für den Aufbau von komplexen Strukturen essenziell. Bei einer Kommunikation gibt es grundsätzlich zwei Beteiligte: Sender und Empfänger. Im Bereich der Software ist der Sender meist ein Client und der Empfänger ein Server, der öffentlich erreichbar ist. Der Client sendet eine Anfrage, der auch Request genannt wird, an den Server. Anschließend wartet der Client auf eine Antwort. Der Server erhält den Request und verarbeitet ihn. Es wird abhängig vom Request eine Antwort, die als Response bezeichnet wird, generiert und dem Client zurückgesendet. Der Client erhält die Response und schließt damit den Kommunikationsvorgang ab. Dieser Ablauf ist in Abbildung 2 visualisiert.

[9] Masse, 2011.

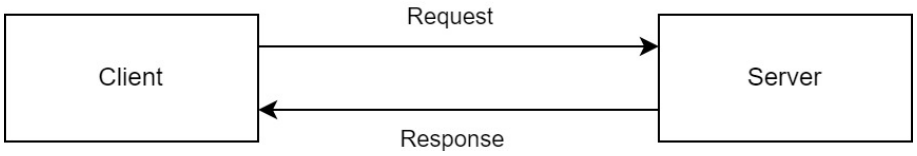

Abbildung 2: Grundlegende Kommunikation

Durch die synchrone Kommunikation sind Client und Server eng miteinander verbunden. Der Client erwartet eine Response von dem Server, an den er den Request geschickt hat. Die synchrone Kommunikation erhöht die Komplexität der Skalierung und Ausfallsicherheit.

Damit eine Kommunikation zwischen unabhängigen Programmen möglich wird, muss es einen Zwischendienst geben, der die Nachrichten von einem Programm zum anderen transportiert. Bei der Kommunikation zwischen einer Website und einer API wird das HTTP verwendet. Dieses stellt sicher, dass die die Nachrichten erfolgreich beim Empfänger ankommen. Sollte eine Nachricht nicht angekommen sein, hat der Absender die Möglichkeit, die Nachricht erneut zu schicken. Über HTTP wird automatisch eine erneute Anfrage geschickt, wenn keine Antwort vom Server zurückkam. Problematisch wird diese Herangehensweise, wenn die Antwortzeit sehr lang wird oder ungewiss ist, ob überhaupt eine Antwort kommen wird. Des Weiteren kann es zu Problemen führen, wenn sehr viele Nachrichten zur gleichen Zeit beim Server eintreffen. Der Server versucht sämtliche eingehenden Nachrichten gleichzeitig anzunehmen und zu verarbeiten. Dadurch entsteht eine sehr hohe temporäre Last. Sollte der Server seine Ressourcen, wie CPU oder RAM nicht dynamisch hochskalieren können, führen die eingehenden Anfragen zu einer Überlastung und damit zum Absturz des gesamten Servers.[10] Um dies zu verhindern, muss ein Zwischendienst implementiert werden, der die eingehenden Nachrichten annimmt, zwischenspeichert und zum passenden Zeitpunkt weiterleitet, wenn der Server die erforderlichen Kapazitäten zum Verarbeiten einer neuen Anfrage hat.

RabbitMQ ist ein eine im Jahr 2007 veröffentlichte, nachrichtenorientierte Middleware, die eine Kommunikation zwischen zwei oder mehreren Programmen durch das Advanced Message Queuing Protocol (AMQP) ermöglicht. RabbitMQ implementiert einen Broker, der Nachrichten von mehreren Clients an mehrere Server vermitteln kann. Im Gegensatz zu einer direkten Kommunikation zwischen Client und Server wie bei HTTP, wird in RabbitMQ eine Queue implementiert, in der sämtliche Anfragen gesammelt werden.[11] In Abbildung 3 ist die Kommunikation zwischen Client und Server mittels eines Brokers abgebildet.

[10] Hoque u. a., 2015.
[11] Johansson u. a., 2020.

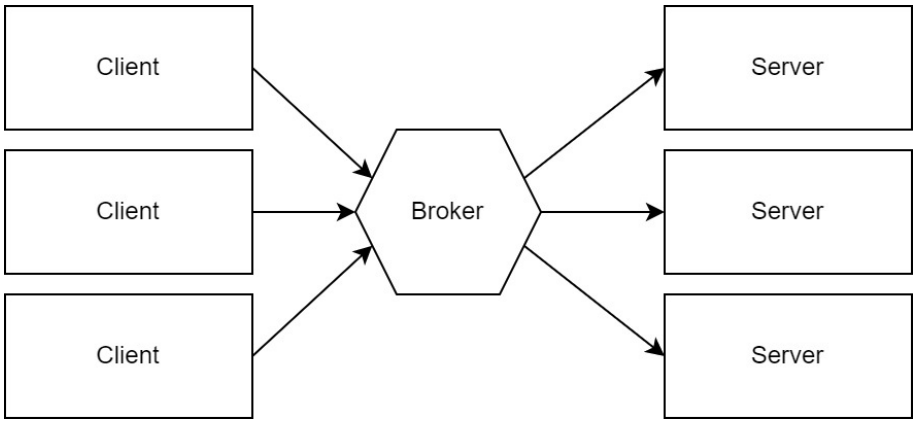

Abbildung 3: Kommunikation über einen Broker nach Dossot

Durch diese Herangehensweise wird eine asynchrone Kommunikation zwischen Client und Server ermöglicht. Sollte ein Server oder ein Client während seiner Laufzeit abstürzen, hat dies keine Auswirkungen auf die anderen Instanzen. Des Weiteren kann die Anzahl der Clients und Server, abhängig von der aktuellen Nutzlast, dynamisch hoch- und runterskaliert werden. Eine weitere Eigenschaft, die durch diese Architektur entsteht, ist, dass der Client keine Information darüber besitzen muss, wo sich der Server befindet, oder mit welcher Technologie er implementiert wurde. Solange der Server die Schnittstelle zum Broker implementiert, können die Nachrichten vom Client empfangen werden.[12]

Das AMQP ist ein offener Standard, der den Nachrichtenaustausch zwischen Produzent und Konsument regelt. AMQP definiert den gesamten Ablauf des Austausches. Innerhalb des Protokolls wird definiert, wie die Transmission Control Protocol (TCP) Verbindung zwischen Client und Broker aufgebaut wird. Ebenfalls Bestandteil des Protokolls ist der softwareseitige Aufbau der Nachrichtenübermittlung. AMQP implementiert mehrere Konzepte, die auch im Prototyp verwendet wurden. Die folgende Aufteilung der Konzepte wurde durch Johansson und Dossot beschrieben.[13]

Der Broker ist das grundlegende Konzept von AMQP. Er empfängt Nachrichten von einer Anwendung und leitet sie an eine andere Anwendung weiter.

Der Virtual-Host, genannt vhost, bietet eine Möglichkeit mit mehreren Anwendungen auf dem gleichen Broker zu arbeiten, jedoch ohne dass die Anwendungen Einfluss aufeinander nehmen können. Der vhost bietet eine logische Trennung der Anwendungen innerhalb der gleichen RabbitMQ Instanz.

Eine Connection beschreibt die physische Verbindung zwischen einer Anwendung und dem Broker. Die Verbindung zwischen Anwendung und Broker wird mittels TCP hergestellt.

[12] Dossot, 2014.
[13] Johansson u. a., 2020.

Ein Channel ist eine virtuelle Verbindung innerhalb einer Connection. Sendet ein Client eine Nachricht an einen Broker, wird dafür eine virtuelle Connection aufgebaut, statt jedes Mal eine neue TCP-Verbindung zu nutzen. Innerhalb einer Connection können mehrere Channels aufgebaut und genutzt werden. Dadurch können auch mehrere Anfragen über die gleiche TCP Verbindung abgearbeitet werden.

Ein Exchange hat die Aufgabe, die Nachrichten zwischen Anwendung und Broker zu vermitteln. Der Exchange stellt sicher, dass die Nachrichten ankommen und in die richtige Queue geschrieben werden. In welcher Queue die Nachrichten landen, ist abhängig von den Regeln, die durch den Exchange Typen definiert werden.

Eine Queue ist eine Datenstruktur, in der mehrere Operationen definiert werden. Die Queue fungiert als Warteschlange, in der Einträge gespeichert und in der Reihenfolge des Eingangs wieder ausgelesen werden. Sie funktioniert nach dem First In - First Out, kurz FIFO, Prinzip. Es wird eine `push` Operation definiert, mit der ein Eintrag an den Anfang der Warteschlange geschrieben wird. Des Weiteren gibt es eine `pop` Operation, die das älteste Element aus der FIFO-Queue ausliest und es aus der Warteschlange entfernt. Jeder Client kann Nachrichten in die Queue schreiben. Diese Nachrichten werden dort so lange gespeichert, bis sie von einem Dienst ausgelesen werden.

Binding beschreibt eine virtuelle Verbindung zwischen einem Exchange und einer Queue innerhalb des Brokers. Diese Verbindung ermöglicht Nachrichtenfluss von einem Exchange zu einer Queue.

In Abbildung 4 sind die Konzepte von AMQP und deren Ablauf visualisiert.

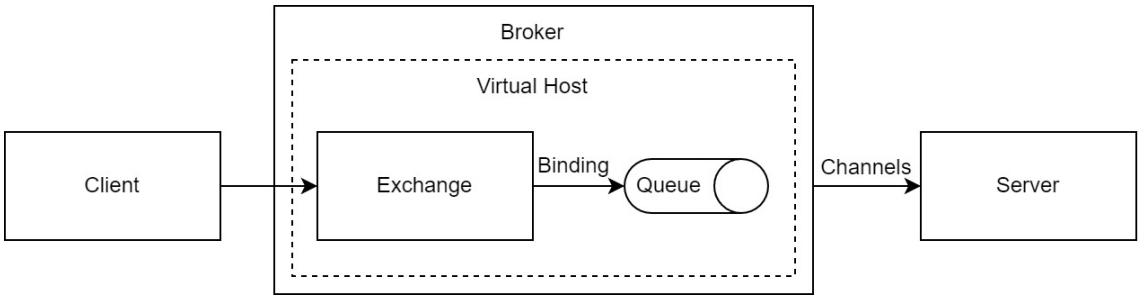

Abbildung 4: Darstellung der AMQP Konzepte nach Dossot

Die Middleware RabbitMQ wird im Prototyp für die Kommunikation zwischen der Flask API und den KI-Services genutzt. Sowohl das Backend, in dem die API implementiert ist, als auch die einzelnen KI-Services sind als Teil einer Microservice-Architektur implementiert. RabbitMQ wird als universelle Schnittstelle für die Kommunikation in der Microservice-Architektur verwendet.

2.3 Microservice-Architekturen

Das Konzept von Microservices beschreibt die Unterteilung einer umfangreichen Anwendung in einzelne Teilanwendungen. Eine Teilanwendung wird dabei Microservice genannt. Die einzelnen Teilanwendungen sollen dabei untereinander kommunizieren können, um die Zusammenarbeit zu gewährleisten. Dazu nutzen die Teilanwendungen eine universelle Schnittstelle. Eine Anwendung soll nur eine Aufgabe erledigen. Der Umfang dieser Aufgabe ist nicht genauer definiert.[14]

Die Idee von Microservices ist es, ein großes System in viele kleine einzelne Systeme aufzuteilen. Dadurch entsteht eine Modularisierung der einzelnen Services. Voneinander unabhängige Softwaremodule können im Gegensatz zu einem Softwaremonolithen unabhängig voneinander deployed werden. Dies ermöglicht es zur Laufzeit bestimmte Teile des Softwaresystems hoch- und runterzufahren. Ein Austausch von fehlerhaften Services oder das Dazuschalten von neuen Services bei mangelnder Leistung ist, während das Softwaresystem in einer Produktivumgebung genutzt wird, möglich.[15]

Jedes Teilsystem kann einzeln deployed werden. Dadurch kann die Verfügbarkeit des gesamten Systems gewährleistet werden, auch wenn bestimmte Bereiche der Software aktualisiert oder verändert werden müssen. Bei einem Monolithen müsste für jede Änderung am System die gesamte Anwendung heruntergefahren und neugestartet werden.

Microservices sind voneinander unabhängige Programme und damit auch eigenständige Prozesse. Die Technologie, mit der ein Service implementiert wird, ist nicht durch die allgemeine Architektur oder das System vorgegeben. Jeder Service kann in der für ihn passenden Technologie und Programmiersprache entwickelt werden.

Da die Technologien der Services unabhängig voneinander sind, muss es eine gemeinsame Schnittstelle zur Kommunikation geben. Über diese Schnittstelle werden Daten und Information ausgetauscht, die die jeweiligen Services zum Bearbeiten ihrer Aufgaben benötigen. Die Kommunikation im Prototyp wird mit RabbitMQ implementiert.

2.4 Künstliche Intelligenz

Der Begriff KI beschreibt eine Software, die das Ziel hat, typisch menschliche Aufgaben, wie Sehen, Hören und Verstehen von Kontext abzubilden.[16] KIs benötigen eine große Menge an Daten als Grundlage, mit denen sie trainiert werden können. Ein nicht trainiertes KI-Modell kann nicht viel mehr als zufällige Entscheidungen treffen. Erst durch das Training bekommen die Entscheidungen der KI eine Struktur. Nachdem eine KI für ihren bestimmten Einsatzzweck mit den entsprechenden Daten trainiert wurde, kann sie verwendet werden. Die Verwendung einer KI folgt den Grundprinzipien eines normalen

[14] Wolff, 2018.
[15] Newman, 2015.
[16] Görz u. a., 2010.

Algorithmus. Es wird ein Input angenommen, wie beispielsweise ein Text oder ein Bild. Dieser wird durch die KI analysiert und in eine für die KI verständliche Form gebracht.[17] Diese Form ist im Falle des Prototyps ein 512-dimensionaler Vektor. Anschließend generiert die KI einen Output. Dieser kann das Endergebnis der Anfrage sein oder für weitere Verarbeitungsschritte genutzt werden.

KIs können von Grund auf selbst entwickelt und trainiert werden. Die Anwendungszwecke einer KI überschneiden sich jedoch oftmals mit bereits vorhandenen Modellen, die es frei oder gegen eine Nutzungsgebühr auf dem Markt gibt. Wenn eine fertig trainierte KI genutzt werden kann, spart dies Entwicklungszeit und Ressourcen. Die vortrainierten KIs werden jedoch meist als Blackbox angeboten. Eine Blackbox ist ein geschlossener Raum, der eine Eingabe annimmt und eine Ausgabe produziert, ohne dass der Nutzer der Blackbox sehen kann, wie der Algorithmus funktioniert, der zu der Ausgabe führt.

Ein KI-Service beschreibt eine Implementierung einer solchen Blackbox. Ein KI-Service ist ein alleinstehendes Programm, das die Aufgabe hat, Nachrichten anzunehmen, sie zu transformieren, zu analysieren und anschließend ein oder mehrere Ergebnisse zurückzugeben.

Um die Nachrichten empfangen und die Ergebnisse zurücksenden zu können, muss in jedem Service eine Schnittstelle implementiert sein, über die die Nachrichten empfangen werden können. Innerhalb des Prototyps wird in den einzelnen KI-Services RabbitMQ als Schnittstelle genutzt.

Im Prototyp zur Anbindung von austauschbaren Datenquellen an KI-Algorithmen ist ein Service zur Textähnlichkeitssuche implementiert. Dieser nutzt das Bidirectional Encoder Representations from Transformers (BERT)-Modell von Google. Dieser Service ermöglicht die semantische Suche in einer Datenbank. Nicht-KI-gestützte Textsuchalgorithmen können in einer Datenbank lediglich nach übereinstimmenden Wörtern oder Wortteilen suchen. Wenn ein Nutzer sämtliche Einträge zu einem bestimmten Thema aus der Datenbank herausfiltern möchte, muss die passenden Schlüsselwörter kennen, die in den Datenbankeinträgen vorhanden sind. Sucht man in einer Datenbank beispielsweise nach „JavaScript“ und es gibt nur Artikel, die die Abkürzung „JS“ beinhalten, liefert die Suche keine Ergebnisse. Eine gewisse Textähnlichkeit kann erreicht werden, wenn zum Beispiel Leerzeichen oder Groß/Kleinschreibung ignoriert werden.

Eine Textänlichkeitssuche, die mittels KI durchgeführt wird, kann zu einer Anfrage wie „JavaScript“ deutlich mehr Ergebnisse liefern. Sie ist nicht durch die in der Suche angegebenen Zeichen beschränkt, sondern versucht den Inhalt der Anfrage zu verstehen. Anschließend kann in der Datenbank nach inhaltlich ähnlichen Artikeln gesucht werden, die zur Anfrage passen. Der „verstandene“ Satz, der für die Suche in der Datenbank genutzt wird, ist abhängig vom verwendeten KI-Modell. Für die Textähnlichkeitssuche im Prototyp wird das BERT Modell von Google verwendet, um die Anfrage in einen 512-

[17] Hamet u. a., 2017.

dimensionalen Vektor zu konvertieren. Jede Dimension des Vektors wirkt sich auf die Interpretation des Vektors aus. Der Vektor wird daher auch als „dense vector“, oder auch dichter Vektor bezeichnet. Damit die Suche über den Vektor funktioniert, wird ein zweiter Vektor benötigt, mit dem der, durch die Suchanfrage erzeugte, Vektor verglichen werden kann. Jeder Eintrag in der Datenbank muss daher einmalig mit dem gleichen KI-Modell analysiert werden. Der jeweils erzeugte Vektor wird in der Datenbank abgespeichert, um ihn für kommende Suchanfragen nutzen zu können.

Der Grundsatz des KI-Modells ist es, dass zwei Sätze, die inhaltlich ähnlich sind, auch zwei ähnliche Vektoren besitzen. Vektoren sind ähnlich, wenn der Abstand der beiden gering und die Richtung ähnlich ist.[18] Diese Abstandsberechnung kann in der Elasticsearch Datenbank seit Version 7.3, welche am 31.07.2019 veröffentlicht wurde, automatisch durchgeführt werden.

2.5 Verwendete Werkzeuge

2.5.1 Python-API mit Flask

Python ist eine um 1991 von Guido van Rossum entwickelte Programmiersprache. Bei der Entwicklung von Python wurde ein besonderer Fokus auf die Lesbarkeit von Code gesetzt. Dank der simplifizierten Syntax im Vergleich zu anderen höheren Programmiersprachen wie Java oder C#, ist Python auch in Bereichen, wie in der Mathematik oder der Wissenschaft ein häufig genutztes Werkzeug. Python bietet ebenfalls die Möglichkeit, von anderen Entwicklern bereitgestellte Bibliotheken in das eigene Projekt zu integrieren.[19]

Flask ist eine der verfügbaren Bibliotheken, die ein Framework für die Implementierung einer webbasierten API bereitstellt. Eine API dient dazu, Funktionen und Routen zu definieren, um die Kommunikation zwischen dem Frontend und dem Backend herzustellen. Das Flask-Framework ist im Gegensatz zu anderen Frameworks sehr klein. Dies ermöglicht ein schnelles aufsetzen und entwickeln. Da Flask nur die nötigsten Grundlagen für eine API mitliefert, ist der Code besser lesbar und damit für andere Entwickler besser wartbar.[20]

Die Flask-API wird für die Anbindung des Frontends an die Datenbank, sowie die Anbindung an die Kommunikationsschnittstelle von RabbitMQ verwendet. Sie nimmt die Daten oder die Eingaben des Nutzers entgegen und vermittelt sie an den richtigen Dienst, damit sie von einer KI-Schnittstelle ausgewertet werden können. Anschließend kann die API angefragt werden, ob es bereits Antworten von einer KI zu der vorher geschickten Anfrage gab. Falls die API die Auswertung der KI erhalten hat, wird diese für das Frontend bereitgestellt, um sie dort anzeigen zu können.

[18] Rahutomo u. a., 2012.
[19] Josheph, 2021.
[20] Grinberg, 2018.

2.5.2 REDIS und MySQL-Datenbanken

Redis ist eine In-Memory Key-Value-Datenbank. Im Gegensatz zu relationalen Datenbankmanagementsystemen (RDBMS) wie MySQL oder PostgreSQL werden in Redis keine festen Tabellenstrukturen hinterlegt. Redis gehört damit zur Kategorie der NoSQL-Datenbanken (Not Only SQL). Key-Value-Stores sind kein Ersatz für eine relationale Datenbank, bieten aber für bestimmte Bereiche große Vorteile. Durch das Fehlen von komplexen Strukturen innerhalb der Datenbank kann Redis Anfragen weitaus schneller als andere Datenbanksysteme bearbeiten. Da Redis im Random-Access Memory (RAM) ausgeführt wird, werden die Daten grundsätzlich nicht persistent gespeichert. ACID (Atomicity, Consistency, Durability and Isolation) Konformität wird mit Redis ebenfalls nicht gewährleistet. Für den Einsatzzweck als Cache in einer Cloud-Umgebung ist Redis allerdings sehr gut geeignet.[21]

Innerhalb des durch Redis implementierten Key-Value-Stores werden sämtliche relevanten Daten gespeichert, die ein Nutzer während seiner Benutzung der Software produziert. Dort werden ebenfalls die Zwischenergebnisse abgespeichert, die die KI während der Analyse erstellt. MySQL ist ein um 1995 erschienenes Open-Source RDBMS. MySQL ist eines der weitverbreitetsten und schnellsten Datenbanksysteme in seiner Kategorie.[22]

In relationalen Datenbanken werden Daten strukturiert in Tabellenform abgespeichert. Einzelne Tabellen können Verlinkungen und Referenzen auf andere Tabellen haben, damit die Zusammengehörigkeit der Daten beschrieben werden kann, ohne Daten redundant speichern zu müssen. In MySQL, wie auch anderen RDBMS, werden Tabellenstrukturen und Daten persistent abgespeichert. In-Memory-Datenbanken wie Redis können Daten über Umwege auch persistent speichern, jedoch müssen dafür größere Anpassung an der Konfiguration von Redis vorgenommen werden.

Das RDBMS MySQL wird u. a. für die Speicherung der Logs, die der Flask-Server während der Verarbeitung von Requests oder Nachrichten an die KI produziert, verwendet. Ein weiterer Einsatzzweck der MySQL-Datenbank ist die Speicherung der im System registrieren KI-Services. Ein Dienst kann über die Flask-API im System registriert oder deregistriert werden. Das Frontend kann im Anschluss eine Auflistung der verfügbaren Services beim Backend anfragen.

2.5.3 Angular-Frontend

Eine Website wird üblicherweise mit Hypertext Markup Language (HTML) und JavaScript erstellt. Um eine moderne Website zu entwickeln, die ihren Inhalt nicht beim ersten Aufrufen lädt, sondern erst dann, wenn er benötigt wird, müssen Konzepte wie Asynchronous JavaScript and XML (AJAX) verwendet werden. Angular ist ein von Google entwickeltes

[21] Paksula, 2010.
[22] DuBois, 2008.

und gepflegtes Open Source-Framework, zur Vereinfachung des Entwickelns von komplexen webbasierten Anwendungen. Angular bietet im Gegensatz zu anderen Webframeworks wie React und Vue.js eine vollumfängliche Bibliothek, mit der nahezu sämtliche Aspekte in der Web-Entwicklung abgedeckt werden können.[23]

In Angular wird die Programmiersprache TypeScript verwendet. Diese ist eine Erweiterung der Programmiersprache JavaScript und implementiert Konzepte wie feste Typisierung von Variablen. Weitere Konzepte wie Dependency-Injection oder die Trennung von Business Logic (BL) und User Interface (UI) ermöglichen eine schnelle Entwicklung von komplexen Systemen.

Das Frontend wird für die Ein- und Ausgabe der Daten verwendet. Der Nutzer kann auf der Webseite seine Suchanfrage in ein Textfeld schreiben und anschließend auf den Server hochladen. Im nächsten Schritt wird die Möglichkeit bereitgestellt, die eingegebenen Daten automatisiert zu bearbeiten und zu manipulieren. Im gleichen Zug wird die Eingabe des Nutzers in ein für die KI verständliches Format konvertiert. Im letzten Schritt schickt der Nutzer die Anfrage an das Backend, mit der Analyse der Eingabe zu beginnen. Das Frontend fängt daraufhin an, beim Backend in regelmäßigen Abständen nach Antworten der KI zu fragen. Antworten werden in einer Liste visualisiert.

2.5.4 Logging durch Grafana

Grafana ist ein von Torkel Ödegaard in 2014 entwickeltes Open-Source-Datenvisualisierungsprogramm
Grafana kann zeitbasierte Daten durch verschiedene Arten von Grafen und Diagrammen anzeigen.[24]

Eines der möglichen Panels für ein Dashbaord ist das Log-Panel. Dort werden die Log Nachrichten aus einer Datenbank angezeigt und mit einer Farbe, abhängig vom Schweregrad markiert. Als Datenquelle können u. a. zeitbasierte Datenbanken z. B. InfluxDB und Prometheus oder RDBMS wie MySQL verwendet werden.

Im implementierten Prototyp wurde eine MySQL verwendet, in der die zu loggende Nachricht, der Schweregrad, der Zeitstempel und die User-Identifikation (ID) gespeichert werden. Diese Daten werden verwendet, um die Logs im Log-Panel von Grafana chronologisch anzeigen zu lassen.

2.5.5 Deployment über Docker

Docker ist eine Software zur Isolierung von Anwendungen mithilfe von Containervirtualisierung. Ein Container beschreibt eine in sich geschlossene Umgebung, in der ein Programm ausgeführt werden kann. Die benötigten Dateien, Parameter und Umgebungsva-

[23] Moiseev u. a., 2018.
[24] Chakraborty u. a., 2021.

riablen werden beim Starten des Containers mitgegeben. Damit wird sichergestellt, dass ein innerhalb eines Docker-Container ausgeführten Programms sich in jeder Umgebung gleich verhält. Hierdurch wird die Unabhängigkeit vom Host-Betriebsystem gewährleistet. Im Gegensatz zu einer Virtuelle Maschine (VM) muss für die Ausführung eines Docker-Containers kein komplettes Betriebssystem virtualisiert werden. Das Hochfahren einzelner Container ist schneller und ressourcenschonender als die Implementierung einzelner VMs.[25] Des Weiteren können über das Docker-Compose-Plugin mehrere Container gleichzeitig hochgefahren werden, sodass mit einer einzigen Kommandozeileneingabe eine komplette Softwarearchitektur hochgefahren werden kann.

Docker wird für das Deployment der einzelnen Komponenten des Prototyps verwendet. Für Redis, MySQL, RabbitMQ, Grafana und Elasticsearch können die benötigten Images, die eine Bauanleitung darstellen, aus dem Docker-Hub heruntergeladen und genutzt werden. In einem Docker-Image sind sämtliche für die Ausführung des Programms benötigten Dateien gepackt. Docker-Hub ist eine Plattform zur Verteilung von offiziellen Docker-Images, von der automatisch die Images runtergeladen werden, die lokal nicht vorhanden sind.

Für das Angular-Frontend und das Flask-Backend müssen die Images manuell gebaut werden, bevor sie als Container gestartet werden können. Dafür bietet Docker sogenannte Docker-Files an, in der die benötigten Konfigurationen hinterlegt werden.

[25] Anderson, 2015.

3 Methodik

Die Schnittstelle für die Anbindung von austauschbaren Datenquellen an KI-Algorithmen wurde auf Grundlage eines modifizierten Wasserfallmodells entwickelt. Das Wasserfallmodell ist ein im Jahre 1970 von Winston W. Royce erstmals beschriebener Designprozess zur Entwicklung von Softwareanwendungen.[26] Das Modell beschreibt einen sequenziellen Prozess der Softwareentwicklung. Dieser Prozess besteht aus fünf Unterpunkte:[27]

1. Requirement Analysis,
2. Design,
3. Implementation,
4. Testing,
5. Operation and Maintenance.

Da das Wasserfallmodell einen sequenziellen Prozess beschreibt, muss jeder Schritt vollständig abgeschlossen sein, bevor der nächste Schritt gestartet werden kann. Rückschritte sind in diesem Modell nicht vorgesehen. Ähnlich wie bei einem Wasserfall kann der Projektfluss nur von oben nach unten ablaufen.[28]

Das Wasserfallmodell wurde für den Prototyp modifiziert. Nicht übernommen wurde der vierte Schritt „Testing“ und der fünfte Schritt „Operation and Maintenance“. Für den Prototyp wurden stattdessen die Schritte „Evaluation“ und „Anpassen“ implementiert.

Das angepasste Wasserfallmodell mit den dazugehörigen Kapiteln, in denen die einzelnen Schritte umgesetzt werden, ist in Tabelle 2 aufgeführt:

Schritt	Bezeichnung	Kapitel
1	Anforderungen definieren	4.1 Anforderungen
2	Konzept entwickeln	4.2 Konzeption
3	Prototypische Implementierung	4.3 Prototypische Umsetzung
4	Evaluation	5 Evaluation
5	Anpassen	6.3 Ausblick

Tabelle 2: Schritte des modifizierten Wasserfallmodells

Für die Schnittstelle wurde sich gegen ein alternatives Projektmanagementmodell wie Scrum entschieden. Scrum bietet viele Vorteile bei Entwicklerteams, die gemeinsam an einer Software arbeiten. Laut der Untersuchung von Harwardt wirkt Scrum einigen Risiken der Softwareentwicklung entgegen. Es ist innerhalb des Entwicklungszeitraums möglich Änderungen vorzunehmen und diese regelmäßig mit Kunden abzusprechen. Ebenso steigert es die Mitarbeiterzufriedenheit und Motivation im Gegensatz zur Arbeit mit

[26] eduNitas, 2015.
[27] Adenowo u. a., 2013.
[28] Porath, 2020.

dem Wasserfallmodell.[29] Trotz dieser Vorteile von Scrum wurde für die Entwicklung der Schnittstelle das Wasserfallmodell gewählt. Für die Schnittstelle, gibt es keine Kunden. Es waren keine weiteren Entwickler an der Software beteiligt. Ein Framework zur regelmäßigen Absprache mit Kunden und anderen Mitarbeitern ist demnach nicht notwendig. Der Entwicklungszeitraum betrug sechs Wochen. Eine Aufteilung dieser Zeit in mehrere Sprint-Zyklen, wie es mit dem Scrum-Modell notwendig wäre, bietet ebenfalls keinen Mehrwert. Scrum hat im Vergleich zum Wasserfallmodell einen größeren Verwaltungsaufwand zur Folge, da die zu entwickelnden Funktionalitäten der Schnittstelle in User-Stories formuliert und der Arbeitsaufwand mit Story-Points abgeschätzt werden muss.[30]

3.1 Anforderungen definieren

Im ersten Schritt des Wasserfallmodells werden die Anforderungen an die zu entwickelnde Software erhoben nach den Vorgaben der IEC 62304 Kapitel 5.2[31] zur Anforderungserhebung für Softwaresysteme. Die Anforderungserhebung findet in Zusammenarbeit mit der CONET statt. Die Anforderungen werden mit einem Experten aus dem Bereich der Softwareentwicklung mit dem Spezialgebiet KI ausgearbeitet. CONET stellt die Ideenbeschreibung für die Schnittstelle zur Verfügung. Auf Basis dieser Beschreibung wird eine Anforderungsliste entworfen, die die folgenden fünf Punkte der IEC 62304 abdeckt:

- Benutzerschnittstellen,
- die GUI,
- Verhalten des Systems auf Benutzeraktionen im Normal- und Fehlerfall,
- die Geschwindigkeit, mit der diese Reaktionen geschehen,
- auf welchen Umgebungen, sich die Software installieren lassen soll.

In Kapitel 4.1 ist eine schriftliche Ausarbeitung der Liste dokumentiert.

3.2 Konzept entwickeln

Im zweiten Schritt des Wasserfallmodells wird das Konzept für die Software erstellt. Die zuvor definierten Anforderungen bieten dafür eine Leitlinie. Für die verwendeten Technologien wird ein Konzept entworfen. Innerhalb dieses Konzepts wird die Architektur der Software beschrieben. In Kapitel 4.2.1 „Softwarearchitektur“ ist das Zusammenspiel und die Abhängigkeiten der einzelnen Bestandteile des Systems aufgeführt. Bei der zu entwickelnden Anwendung handelt es sich um eine Full-Stack-Architektur, die sowohl ein Frontend als auch ein Backend mit den dazugehörigen Komponenten beinhaltet. Daher

[29] Harwardt u. a., 2012.
[30] Wirdemann, 2022.
[31] Daniel, 2018.

ist die Planung der Architektur für den Entwicklungsprozess entscheidend.[32] Änderungen der Architektur nach dem Aufsetzen der einzelnen Systeme sind aufwändig und zeitintensiv.

Das Konzept beinhaltet neben der Planung der Architektur einen Programmablaufplan. Dieser Plan ist wird in Kapitel 4.2.2 „Programmablauf“ beschrieben. Innerhalb des Programmablaufplans wird die Nutzerinteraktion mit dem System und die dadurch ausgelösten Ereignisse im Backend erläutert. In diesem Schritt werden Rand- und Fehlerfälle nicht behandelt.

Zur Visualisierung des Konzepts werden Mockups für die Website entworfen. Da die Ereignisse innerhalb des Prototyps durch einen Anwender ausgelöst werden, ist die Darstellung der Eingabefelder, Buttons und Ergebnisanzeigen zum Verständnis des Programmablaufs hilfreich. Mockups verbessern den Prozess der Erhebung und Bestätigung der Anforderungen.[33]

3.3 Prototypische Implementierung

Der dritte Schritt des Wasserfallmodells umfasst die prototypische Umsetzung der Schnittstelle. Die dazu verwendeten Werkzeuge werden in Kapitel 2.5 „Werkzeuge“ beschrieben. Bei der Auswahl der Werkzeuge für die Erstellung der Schnittstelle wird darauf geachtet, moderne und in der Praxis genutzte Frameworks und Technologien zu verwenden. Die Auswahl basiert auf Expertenmeinungen von CONET, eigener Recherche und Technologien, die bei CONET im Einsatz sind. Das zuvor definierte Konzept dient als Anleitung für den Entwurf und die Implementierung der Softwarearchitektur. Die Mockups werden als visuelle Referenz für das Design des Frontends verwendet. Der Artikel von Tibshirani zur Implementierung einer Textähnlichkeitssuche in Elasticsearch dient als Grundlage für den KI-Service.[34] In den KI-Service wird eine implementierte Version des BERT-Modells von Tibshirani integriert.[35] Die Implementierung der Software wird in Kapitel 4.3 „Prototypische Umsetzung“ beschrieben.

3.4 Umsetzung evaluieren

Die prototypischen Implementierung wird mit den zu Beginn der Arbeit erhobenen Anforderungen in regelmäßigen Code-Reviews verglichen. Die Durchführung der Code-Reviews findet mit einem Experten von CONET statt. Innerhalb dieser Code-Reviews liegt der Fokus auf der Fertigstellung des Prototyps sowie der Qualität des Codes. Die Art der Implementierung der in den Anforderungen erhobenen Features ist ebenfalls Be-

[32] Taivalsaari u. a., 2021.
[33] Rivero u. a., 2010.
[34] Tibshirani, 2019.
[35] Tibshirani; Oak, 2020.

standteil des Code-Reviews. Des Weiteren wird auf die Modularität, Skalierbarkeit und Performance des Codes geachtet.

Die Ergebnisse der Code-Reviews sind in Kapitel 5 „Evaluation“ festgehalten.

3.5 Anpassen

Im letzten Schritt des Wasserfallmodells „Anpassen“ wird der Prototyp Mitarbeitenden und Experten der CONET im Rahmen einer vorgestellt. Das nach der Präsentation gesammelte Feedback wird in Kapitel 6.3 „Ausblick“ aufgeführt.

4 Konzeption und prototypische Umsetzung

4.1 Anforderungen

Die Anforderungen an die Schnittstelle und den daraus resultierenden Prototyp wurden in Zusammenarbeit mit CONET erhoben. Die Projektidee entstand im Kontext eines Projektes zu Trendscouting-Systemen. Das Ziel des Systems sollte es sein, eine Architektur bereitzustellen, die vom Nutzer gestellte Anfragen annimmt und diese mit einer KI analysiert, um Aussagen darüber treffen zu können, welche Technologien in der Zukunft relevant werden könnten. Aus dieser Projektidee ergab sich als Zielsetzung die Entwicklung einer Architektur, die ein solches System unterstützen könnte. Die genaue Anforderungserhebung wurde nach dem Leitfaden der von der International Electrotechnical Commission (IEC) veröffentlichten Norm IEC 62304 Kapitel 5.2 entworfen.[36] Diese Norm spezifiziert, wie Anforderungen an eine Software aufbereitet und dokumentiert werden sollen.

Auf der Grundlage dieser Norm wurden die Benutzerschnittstellen für den Prototyp spezifiziert. Die grundlegende Interaktion mit dem System findet über die Website statt. Dort hat der Nutzer die Möglichkeit, seine Suchanfrage in Form eines ausgeschriebenen Satzes oder über einzelne Stichpunkte in ein Textfeld einzugeben. Die nächste Benutzerschnittstelle ist die Transformation der zuvor eingegebenen Suchanfrage. Der Nutzer soll dort mittels JSON-Syntax die Suchanfrage modifizieren können. Dies ist im Hinblick auf eine eventuell automatisierte Nutzung des Systems erforderlich. Im dritten Schritt soll der Nutzer die Option haben, die Suchparameter für die KI anzupassen. Insbesondere die Anzahl der Ergebnisse ist relevant, die bei einer gestellten Suchanfrage zurückgegeben werden. An dieser Stelle sollen auch weitere Parameter eingegeben werden können. Anschließend ist eine Möglichkeit vorgesehen die Anfrage inklusive der Parameter abzuschicken. Abbildung 5 stellt die Interaktion mit dem System in einem Use-Case-Diagramm dar.

[36] Daniel, 2018.

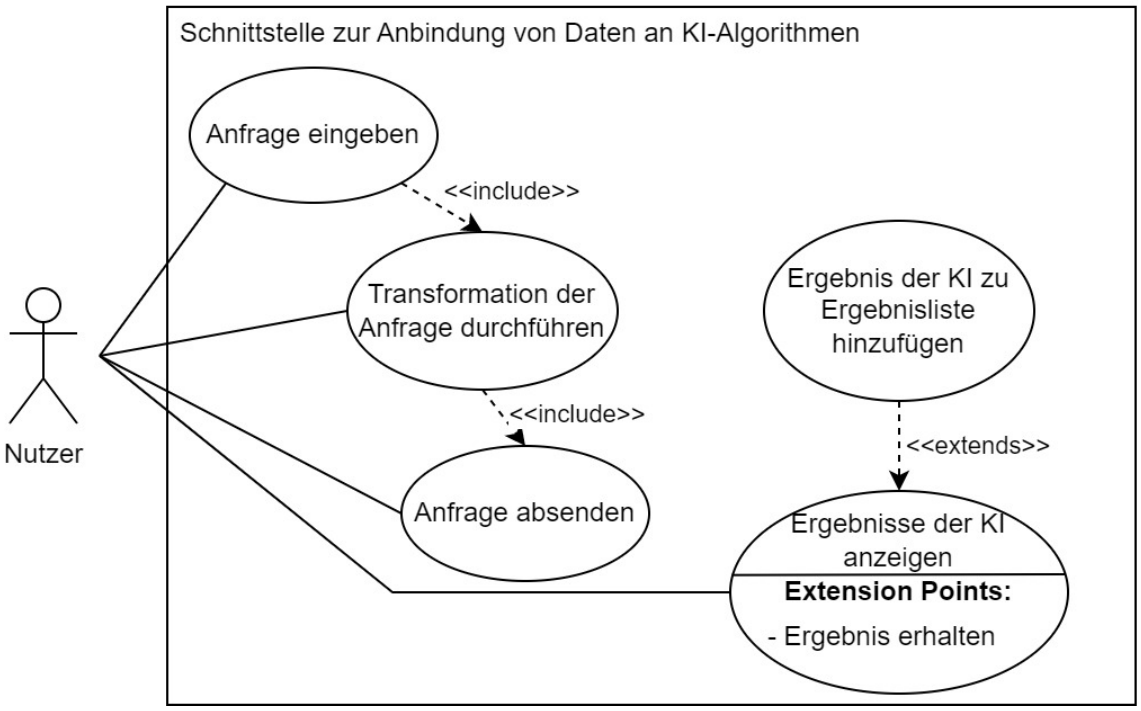

Abbildung 5: Use Case Diagramm der Interaktion mit der Schnittstelle

Die letzte Möglichkeit, die der Nutzer hat, mit der Schnittstelle zu interagieren, ist über die API selbst. Dort gibt es eine Route, über die die aktiven KI-Services verwaltet werden können. Abhängig von der gewählten HTTP-Methode sollen verschiedene Funktionen angeboten werden. Die drei Hauptfunktionen sind die Auflistung sämtlicher aktiven Services, das Erstellen eines neuen Services und das Löschen eines Services.

Der nächste Punkt der Anforderungsanalyse gemäß IEC 62304 ist die Festlegung des Graphical User Interface (GUI). Die grafische Oberfläche sollte sich auf eine Seite beschränken und in drei Bereiche eingeteilt sein. Jeder der Bereiche beinhaltet eine Kachel mit einer Überschrift. Die drei Überschriften „Query“, „Transformation“ und „Service“ sollen die Aktionen innerhalb der Kachel verdeutlichen. Der linke Bereich der Seite für die Query beinhaltet die Eingabe durch den Nutzer, welche durch ein Textfeld realisiert wird. Des Weiteren gibt es einen Button zum Hochladen der Anfrage an den Server.

Im mittleren Bereich der Seite befindet sich die Kachel für die Transformation der Eingabe. Der Inhalt der Kachel ist in drei Bereiche aufgeteilt. Der erste Bereich zeigt die vom Nutzer hochgeladene Eingabe. In diesem Feld kann der Nutzer keine Änderungen vornehmen. Das zweite Feld enthält die Eingabe für die Transformation. Das Textfeld enthält einen vorgefertigten Text in JSON-Syntax, den der Nutzer anpassen kann. Unter dem Textfeld gibt es einen Button, der die Funktion hat, die Transformationsanleitung an den Server zu senden. Das Ergebnis der Transformation wird im dritten Feld angezeigt, damit der Nutzer überprüfen kann, ob das erzeugte Ergebnis seinen Erwartungen entspricht.

Der rechte Bereich der Seite ist für die Auswahl und Nutzung der KI-Services vorgesehen. Der gewünschte Service soll über ein Dropdown ausgewählt werden. Um den Services Anfrage-Parameter zu übergeben, wurde eine Tabellenstruktur implementiert. Die Tabelle hat zwei Spalten und beliebig viele Reihen. Innerhalb der Tabelle können Schlüssel-Wert Paare eingegeben werden, wobei der Schlüssel in der ersten Spalte und der Wert in der zweiten Spalte steht. Unter der Tabelle muss es einen Button geben, der die Anfrage an den KI-Service sendet. Daraufhin startet der Abfrageprozess der die Antworten der KI-Services zusammenträgt. Die Ergebnisse des KI-Services werden in Listenform angezeigt. Zunächst lediglich als Überschrift, erst bei einem Klick auf die Überschrift wird der Inhalt ausgeklappt.

Der nächste Punkt gemäß IEC 62304 ist die Dokumentation des Verhaltens des Systems. Besonderer Fokus liegt hierbei auf dem Verhalten gegenüber Nutzerinteraktionen. Beim ersten Besuch der Website wird dem Nutzer ein Token zugesendet, welches für die weiteren Anfragen zur Authentifizierung an der API genutzt werden soll. Innerhalb des Tokens ist eine eindeutige User-ID hinterlegt. Nachdem der Nutzer auf der linken Seite der Website in der Kachel „Query“ den Upload Button gedrückt hat, sendet die Website den eingegebenen Text an die API. Dabei soll im Authorization Header der Token stehen und im Request Body der eingegebene Text in JSON-Syntax.

Ist eine Anfrage beim Server eingegangen, wird sie in einem Redis Cache hinterlegt und anschließend an das Frontend zur Bestätigung zurückgesendet. Sollte bei der Anfrage ein Fehler aufgetreten sein, wird der Statuscode 500 an die Website zurückgegeben und das Ergebnis des Uploads wird nicht im ersten Feld der Transformation angezeigt. Ist der Upload erfolgreich, wird der Text auf der Website angezeigt. Der Nutzer hat die Möglichkeit, eine Transformationsanleitung in JSON-Syntax einzugeben und diese über den Transform-Button an das Backend zur Auswertung zu schicken. Die Transformationsanleitung wird, nachdem sie im Backend eingetroffen ist, ebenfalls im Redis Cache gespeichert. Das Backend führt eine Transformation mit der gegebenen Anleitung durch und sendet sie zurück an die Website. Das Ergebnis der Transformation wird im dritten Feld der Transformation-Kachel angezeigt. Bei einem Fehler während der Transformation wird der Statuscode 500 zurückgesendet und es werden keine Ergebnisse angezeigt.

In der Service-Kachel im rechten Bereich der Webseite kann der Nutzer über ein Dropdown einen der aktiven Services auswählen. Die Liste der Services wird beim Laden der Seite von der API angefragt. Die API führt dazu eine SQL-Anfrage in einer MySQL Datenbank aus, in der sämtliche aktiven Services hinterlegt sind. Anschließend definiert der Nutzer einen oder mehrere Parameter in einer Tabelle. Der ausgewählte Service und die eingegebenen Parameter werden durch den Klick auf einen Execute-Button unterhalb der Tabelle an die API gesendet. Das Backend soll die transformierten Anfragen über RabbitMQ an den dafür vorgesehenen Service schicken. Zur Bestimmung des Services wird der von der API mitgesendete Service-Name genutzt, der durch die Dropdown-Auswahl

ermittelt wird. Jede einzelne Anfrage hat eine Anzahl an erwarteten Antworten. Das Backend sendet die Gesamtzahl der erwarteten Antworten an die Website. Dort wird die Antwortzahl für eine Fortschrittsanzeige verwendet werden.

Der zu implementierende Service beinhaltet eine Textähnlichkeitssuche. Dieser Service erhält die vom Backend gesendeten Anfragen. Mittels des BERT-Modells werden die semantisch ähnlichsten Texte aus einer Elasticsearch-Datenbank herausgefiltert. RabbitMQ sendet die ähnlichsten Texte im Anschluss an das Backend. Dort werden sie im Redis-Cache zwischengespeichert. Die Website soll nach dem Absenden der Anfrage durch den Nutzer in regelmäßigen Intervallen selbständig Anfragen an die API stellen, ob Antworten vom KI-Service im Backend eingetroffen sind. Bei einer solchen Anfrage überprüft das Backend im Redis-Cache, ob eine oder mehrere Antworten eingetroffen sind und sendet diese an die Website. Liegen noch keine Antworten im Redis-Cache, sendet die API die Nachricht `successful:False`. Die Website stellt so lange Anfragen an die API, bis die erwartete Zahl der Antworten eingetroffen ist.

Der nächste Punkt in der IEC 62304 ist die Festlegung der Geschwindigkeit, in der das System bestimmte Aufgaben abarbeiten soll. Hierzu gab es bezüglich der KI-Services keine genaue Vorgabe seitens CONET. Das Backend und die Kommunikation zu den einzelnen Services hatte jedoch gewisse Maximallaufzeiten. Eine Anfrage, die durch einen HTTP-Request in der API ankommt, soll maximal 0,5 Sekunden benötigen, um an den richtigen Service weitergeleitet zu werden. Die Rückrichtung vom Service zum Backend soll ebenfalls maximal 0,5 in Anspruch nehmen. Die Gesamtzeit der Kommunikation zwischen Backend und einem Service darf demnach nicht mehr als eine Sekunde betragen. Da der KI-Algorithmus keine festgelegte Laufzeit hat, soll durch die Architektur, mit der die Systeme aufgesetzt sind, kein großer zusätzlicher Overhead entstehen. Antwortzeiten innerhalb einer Sekunde gelten laut CONET als responsiv und praxistauglich.

In der Spezifikation IEC 62304 wird ebenfalls gefordert, die Umgebung, in der die Software installiert werden soll, zu definieren. Die Hauptanforderung CONET war hierbei, dass das System möglichst plattformunabhängig entwickelt werden soll. Für die Installation der einzelnen Komponenten ist Docker vorgesehen.

Der letzte für die Anforderungsanalyse relevante Punkt aus der IEC 62304 ist die Beschreibung, wie das System auf Benutzerfehler und Überlast reagieren soll. Grundsätzlich sollen Benutzerfehler im Backend abgefangen werden und nicht zum Serverabsturz führen. Dazu soll ein System implementiert werden, das potentielle Fehler abfängt und anschließend in eine MySQL-Datenbank logged. Des Weiteren sollen die aufgerufenen Routen und Vorgänge innerhalb des Backends sowie den einzelnen Services geloggt werden. Dadurch kann nachvollzogen werden, ob das System ordnungsgemäß funktioniert. Die Architektur soll horizontal skalierbar sein, wodurch mehrere Instanzen der API und der einzelnen Services parallel betrieben werden können. Jede dieser Instanzen muss für sich allein voll funktionsfähig sein.

4.2 Konzeption

4.2.1 Softwarearchitektur

Die Erstellung eines simplen Dienstes, der Daten mit einer KI verbindet, kann mithilfe eines einzigen Python-Scripts erstellt werden. Die Herausforderung an eine praxistauglichen Anwendung, die gleichzeitig von mehreren Usern genutzt werden kann, geht über den Anschluss der KI hinaus. Eine praxistaugliche Anwendung muss neben den funktionalen Anforderungen auch noch weitere nicht funktionale Anforderungen erfüllen. Laut CONET sind die drei wichtigsten nicht funktionalen Anforderungen Performance, Skalierbarkeit und Verfügbarkeit. Diese drei Anforderungen können mit einem lokal ausgeführten Skript nicht erfüllt werden.

Damit Nutzer mit der Software interagieren können, wird ein Frontend benötigt. Ein zentral gehostetes, webbasiertes Frontend kann von einem Nutzer über eine einfache URL im Webbrowser aufgerufen werden. Für die anzuzeigenden Daten im Frontend wird eine Verbindung zum Backend benötigt. Diese wird über eine HTTP Verbindung zur mit Flask gehosteten REST API bereitgestellt.

Um die Anforderung der Skalierbarkeit erfüllen zu können, ist die REST-API komplett zustandslos implementiert worden. Eine API ohne Zustände speichert keine Zwischenstände zu den Anfragen einzelner Nutzer. Bei jeder Anfrage an die API müssen sämtliche Informationen im Request bereitgestellt werden, die die API zum Bearbeiten der Anfrage benötigt. Dies bietet die Möglichkeit bei steigender Nutzerzahl mehrere parallel betriebene Instanzen der API hochzufahren. Dadurch ist eine horizontale Skalierung gewährleistet. Horizontal skalierbare Instanzen innerhalb der Softwarearchitektur sind in Abbildung 6 mit zwei hintereinander gestapelten Rechtecken visualisiert.

Da die Kommunikation zwischen dem Frontend, der API und den KI-Services asynchron läuft, muss das Flask-Backend trotz seiner Zustandslosigkeit Transformationsanleitungen und Ergebnisse der KI-Services zwischenspeichern, bis sie im Frontend benötigt werden. Um die Performanceanforderungen erfüllen zu können, können nicht sämtliche Zwischenstände in einer MySQL-Datenbank gespeichert werden. Die Lese- und Schreibgeschwindigkeit kann bei steigender Nutzerzahl problematisch werden. Um dem entgegenzuwirken, wird mit Redis ein Key-Value-Store als Cache betrieben. Die zwischengespeicherten Daten werden nach dem ersten Aufruf wieder gelöscht. Eine persistente Speicherung ist daher nicht notwendig. In-Memory Datenbanken speichern und führen ihre Queries direkt im RAM aus, wodurch Anfragen im Vergleich zu einer MySQL-Datenbank deutlich schneller ausgeführt werden.

Im Flask-Backend werden sämtliche Routen und die meisten Funktionen abgekapselt in einem Funktion-Wrapper ausgeführt. Dieser fungiert als eine Art Sandbox, in der auftretende Fehler nicht zum Programmabsturz führen, sondern behandelt und geloggt werden

können. Die Logs werden persistent in einer MySQL-Datenbank gespeichert. Mit dem Dienst Grafana können diese Logs angezeigt werden.

Die Laufzeit von KI-Services hängt stark vom verwendeten KI-Modell, der zu durchsuchenden Datenmenge und der vom Nutzer gesendeten Eingabe ab. Bei einer synchronen Kommunikation zwischen dem Flask-Backend und dem Service können sehr lange Wartezeiten entstehen. Wenn der KI-Service ebenfalls eine REST-Schnittstelle implementieren würde, könnten es bei einem HTTP-Request zum Timeout der Anfrage führen. Aufgrund der schwanken Laufzeit muss eine asynchrone Kommunikationsstruktur wie RabbitMQ mit dem AMQP implementiert werden.

Die einzelnen Services können mit einem Eintrag in der MySQL-Datenbank registriert werden. Für die Registrierung muss lediglich der Name und der im Frontend anzuzeigende Name des Services hinterlegt werden. Die Registrierung eines Dienstes kann durch den Aufruf einer Route in der API durchgeführt werden.

Der im Prototyp implementierte KI-Service nutzt das BERT-Modell von Google zum Konvertieren der Nutzereingaben in semantische Vektoren. Es wird ebenfalls eine Elasticsearch-Datenbank betrieben, in der die zu durchsuchenden Einträge gespeichert sind. Im Gegensatz zu einer MySQL-Datenbank, kann in einer Elasticsearch-Datenbank zu jedem Eintrag ein semantischer Vektor gespeichert werden. Der KI-Service kann mithilfe der Kosinusähnlichkeitssuche den semantischen Vektor der Eingabe mit den Vektoren der Datenbank vergleichen und dadurch die semantisch ähnlichsten Texte herausfiltern. Die gefunden Einträge werden über RabbitMQ im Anschluss wieder an das Flask-Backend geschickt, damit sie dort vom Frontend ausgelesen werden können.

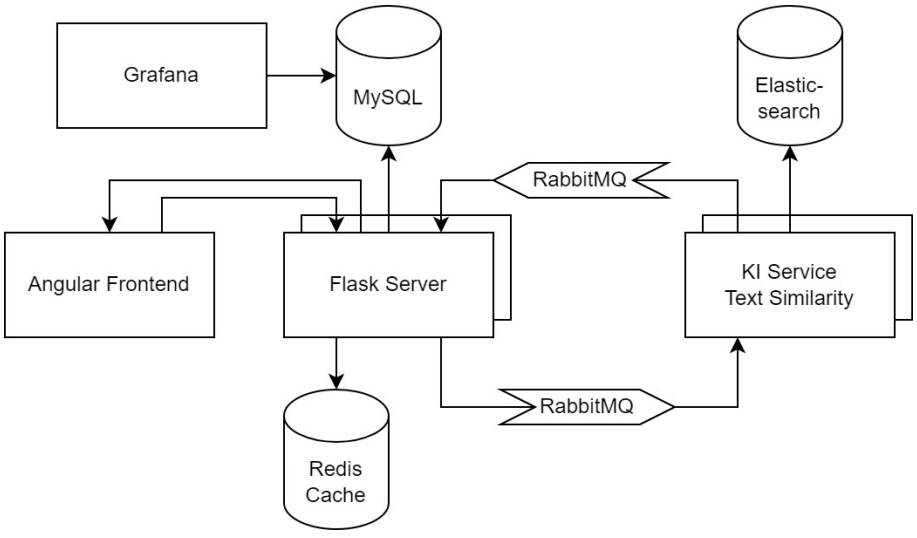

Abbildung 6: Softwarearchitekturdiagramm

4.2.2 Programmablauf

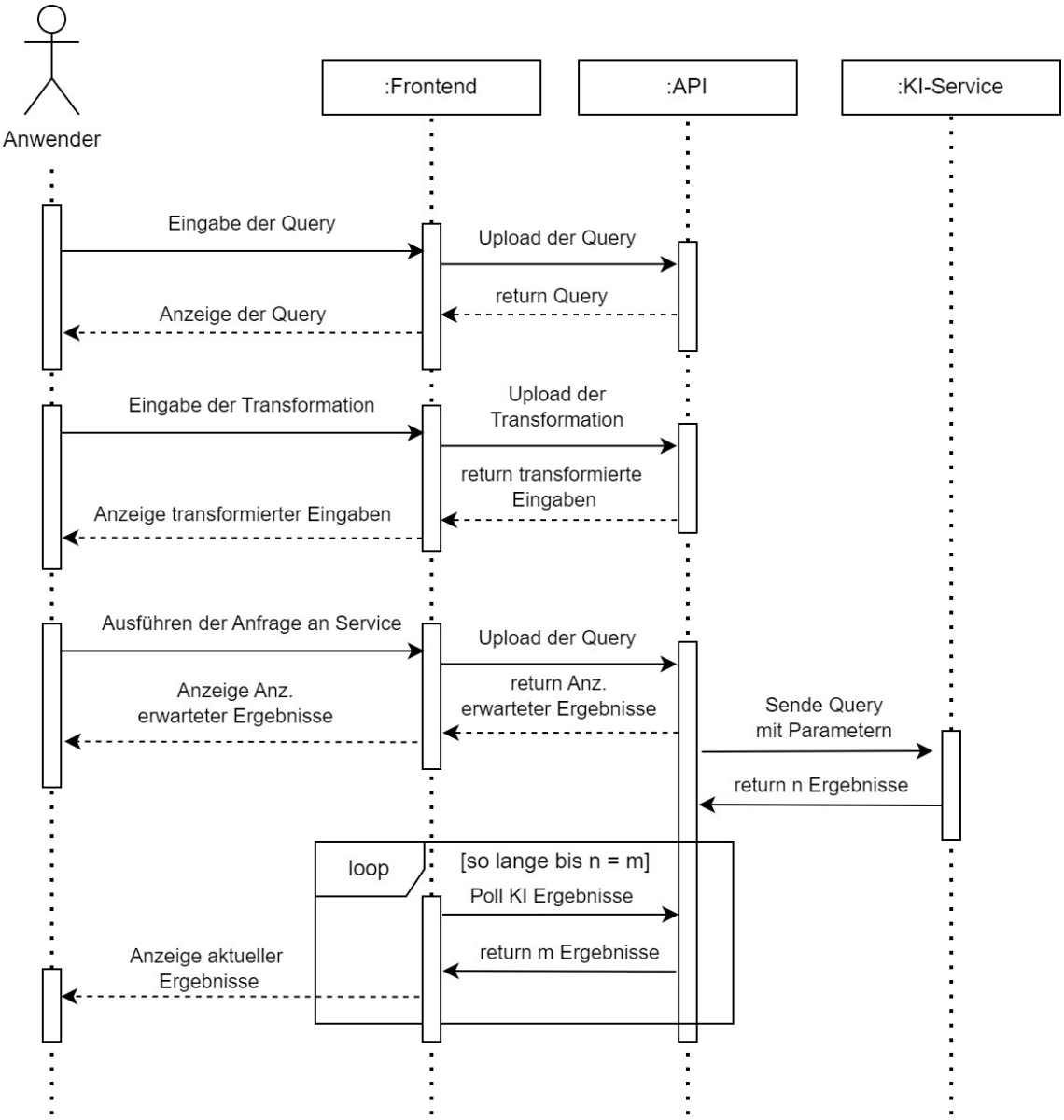

Abbildung 7: UML-Sequenzdiagramm des Programmablaufs

Das Ziel der Bachelorarbeit ist es, eine Schnittstelle zu entwickeln, die Daten an KI-Algorithmen anbindet. Dazu wird im Prototyp ein System implementiert, welches die Anbindung in drei Schritten durchführt. In Abbildung 7 ist der grundlegende Programmablauf in einem UML-Sequenzdiagramm dargestellt. In Abbildung 8 ist ein Mockup der gesamten Website abgebildet. Die Interaktionen, die der Anwender auf der Website tätigen kann, sind hier beispielhaft abgebildet. Die Mockups dienen der Veranschaulichung des entwickelten Konzepts für die Nutzerinteraktion mit der Schnittstelle.

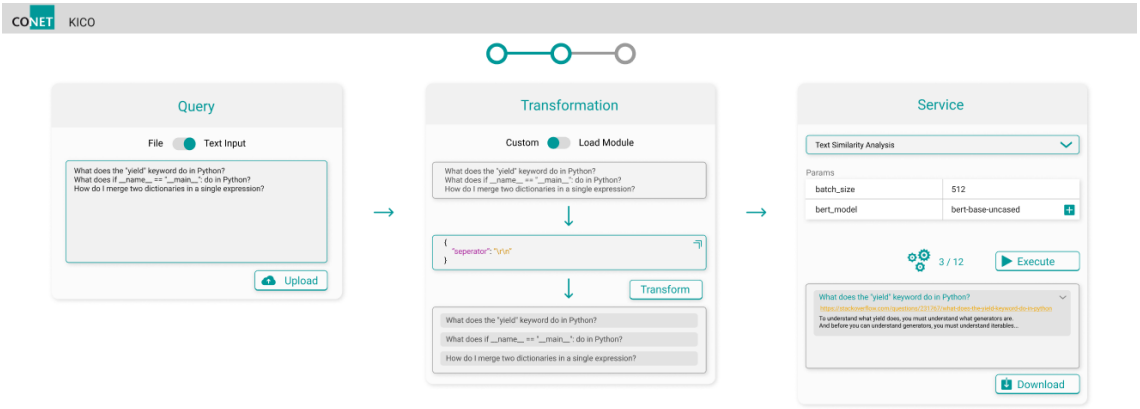

Abbildung 8: Mockup Frontend

Der erste Schritt ist, wie auch in den Anforderungen definiert, die Eingabe der Query durch den Anwender. Auf der Website gibt es dafür ein Textfeld, in dem diese Eingabe stattfindet. Eine Darstellung der Query-Kachel ist in Abbildung 9 gegeben. Die Eingabe kann durch einen Klick auf den Upload-Button an die API per HTTP-POST-Request geschickt werden. Innerhalb des Bodys befindet sich die Eingabe. Zusätzlich zu dem Body wird ein Authorization Header mitgeschickt, der die User-ID des Anwenders enthält. Innerhalb der API wird die User-ID als Schlüssel und die Query als Wert für einen Eintrag in den Redis-Cache genutzt. Die Eingabe wird im Backend zunächst nur zwischengespeichert. Als Bestätigung für den Nutzer sendet die API die erhaltene Query wieder an die Website zurück. Dort wird sie im oberen Bereich der Transformation-Kachel angezeigt.

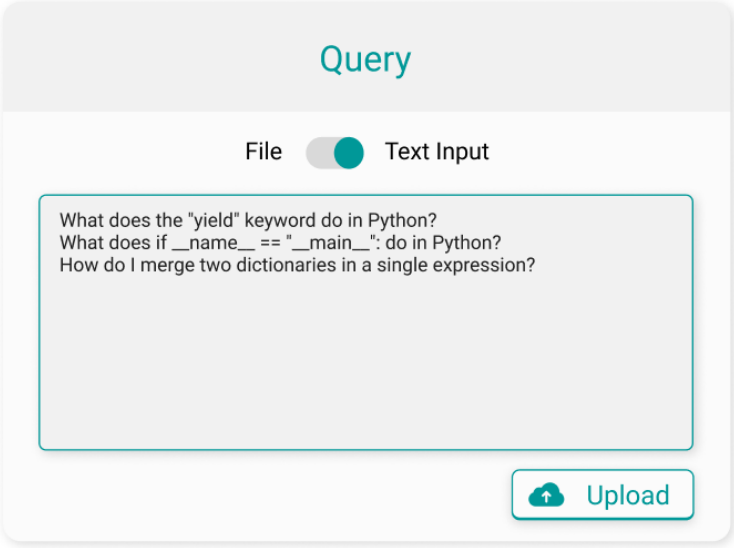

Abbildung 9: Mockup Query

Der zweite Schritt ist die Eingabe der Transformationsanleitung. In Abbildung 10 ist das Mockup für die Transformation-Kachel abgebildet. Im mittleren Textfeld der

Transformation-Kachel ist für diesen Schritt ein Textfeld vorgesehen. Nach Abschluss der Eingabe kann der Anwender auf den Transform-Button klicken, um die Eingabe an die API hochzuladen. Wie auch bei dem Upload der Query wird die User-ID im Header des Requests mitgeschickt. Die API nutzt die User-ID, um die zuvor hochgeladene Query aus dem Redis-Cache zu laden. Die im Request mitgelieferte Transformationsanleitung wird in diesem Schritt genutzt, um die Query zu transformieren. Im Prototyp wird eine Split-Operation für eine beliebige Anzahl an Replace-Operationen implementiert. Die Split-Operation teilt die Query in mehrere einzelne Queries. Dazu wird eine Zeichenkette genutzt, die als Trennzeichen dient. Die Replace-Operation ersetzt eine eingegebene Zeichenkette durch eine andere. Das Ergebnis der Transformation wird innerhalb der Response wieder an die Website gesendet. Sollte eine Split-Operation angegeben worden sein, werden die daraus resultierenden Queries im unteren Bereich der Transformation-Kachel angezeigt.

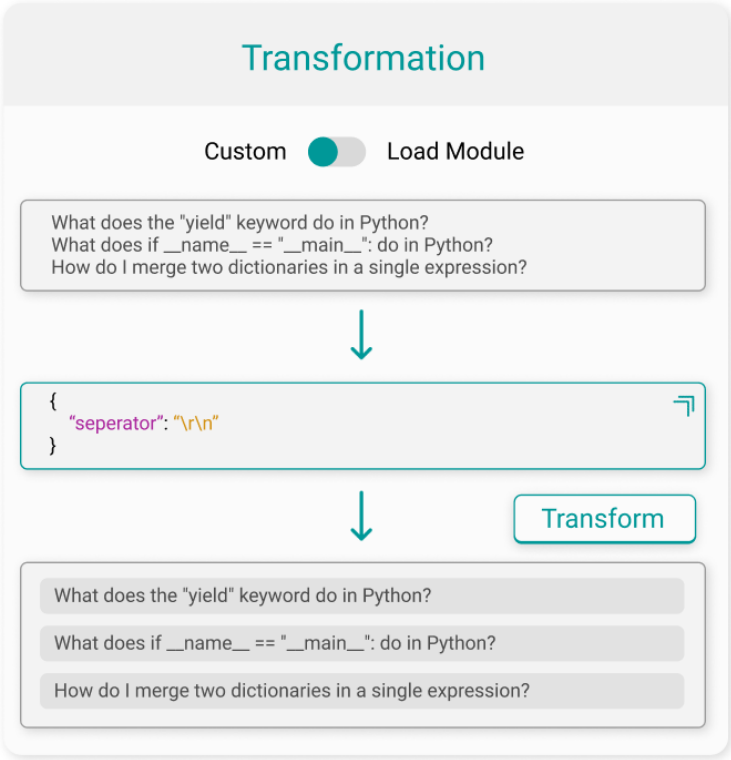

Abbildung 10: Mockup Transformation

Der dritte Schritt ist die Ausführung der Anfrage durch einen KI-Service. In Abbildung 11 ist das Mockup für die Service-Kachel dargestellt. Der Anwender wählt im Dropdown der Serivce-Kachel einen der in der MySQL-Datenbank hinterlegten aktiven Services aus und gibt anschließend einen oder mehrere Parameter in die Tabelle unterhalb des Dropdowns ein. Durch einen Klick auf den Execute-Button wird eine Query im Frontend zusammengebaut und als Request an die API gesendet. Innerhalb der Query befindet sich der Name des KI-Services sowie die in der Tabelle definierten Parameter. Wenn der Request bei der

API eingegangen ist, lädt das Backend sowohl die in Schritt eins eingegebene Query als auch die in Schritt zwei eingegebene Transformationsanleitung. Das Backend führt eine erneute Transformation durch und sendet dem Frontend die Anzahl der erwarteten Antworten des KI-Services zurück. Die Anzahl setzt sich aus der Menge der durch die Split-Operation entstandenen Queries und den eingegebenen Parametern zusammen. Für den KI-Service zur Textähnlichkeitssuche bietet der Parameter `search_size` eine Konfiguration der Suchgröße an. Die Anzahl der erwarteten Antworten wird im Frontend zur Fortschrittsanzeige genutzt. Nach dem Upload der Query startet die Website eine Schleife, in der bei der API regelmäßig nach neuen Ergebnissen des KI-Services gefragt wird. Die Anfrage wird dem KI-Service nach dem Upload der Query über RabbitMQ gesendet. Innerhalb des Services in dem die Textähnlichkeitssuche implementiert wird, wird die eingehende Nachricht angenommen und mithilfe des BERT-Modells in einer Elasticsearch Datenbank nach den semantisch ähnlichsten Einträgen gesucht. Es werden, abhängig von der `search_size`, die ähnlichsten Ergebnisse gesammelt und über RabbitMQ wieder an das Backend gesendet. Im Sequenzdiagramm in Abbildung 7 ist die Anzahl mit der Variable `n` angegeben. Die Ergebnisse werden in einem Redis-Cache zwischengespeichert. Parallel zu der Bearbeitung der Services im Frontend läuft eine Schleife, die die Ergebnisse aus der API abfragt. In einer Schleifeniteration werden sämtliche im Redis-Cache hinterlegten Ergebnisse aus dem KI-Service abgefragt und an das Frontend gesendet. Die maximale Anzahl der Ergebnisse ist dabei `m`. Die Schleife läuft bis die Anzahl der erhaltenen Antworten identisch mit Anzahl der erwarteten Antworten ist, also bis $n = m$. Jedes erhaltene Ergebnis wird im Frontend im unteren Bereich der Service-Kachel angezeigt. Damit ist der Programmablauf abgeschlossen und der Anwender kann eine neue Query eingeben.

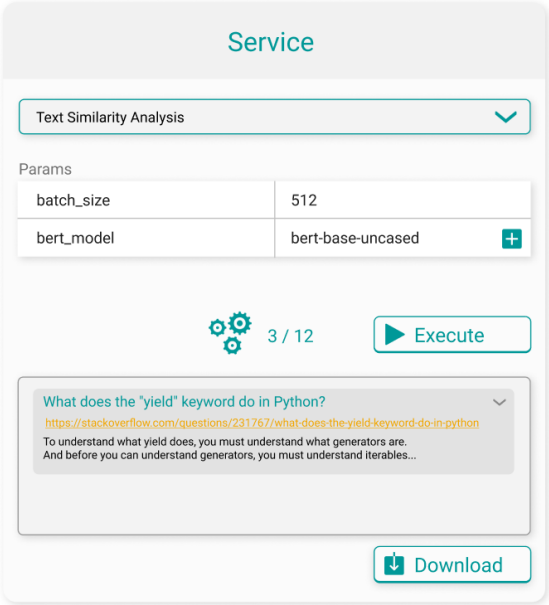

Abbildung 11: Mockup Service

4.3 Prototypische Umsetzung

4.3.1 Implementierung der REST-API

Python bietet mit dem Package Flask die Möglichkeit einen simplen und gut skalierbaren Webserver aufzusetzen. Für den Start einer Flask-Instanz muss das Package Flask in die Python Umgebung importiert werden. Anschließend kann ein Flask-Objekt erzeugt und die Flask-Instanz mit den gewünschten Parametern gestartet werden.

```
from flask import Flask
app = Flask(__name__)
app.run(host="0.0.0.0", port=80, use_reloader=False)
```

Code Listing 1: Aufsetzen einer Flask-Instanz

Damit die API auch automatisiert aus einem Docker-Container heraus gestartet werden kann, muss die Ausführung des Flask-Services in die Main Methode von Python ausgelagert werden. Flask blockiert den Thread, auf dem es ausgeführt wird, wodurch eine asynchrone Kommunikation über RabbitMQ nicht möglich wäre. Der Receiver benötigt seinen eigenen Thread. Daher ist eine Multithreading-Architektur zu implementieren mithilfe des `threading`-Packages. Mit dem Parameter `daemon` kann bei der Erzeugung eines Threads festgelegt werden, dass der Thread im Hintergrund läuft und den Haupt-Thread nicht blockiert.

```
def start_server():
    app.run(host="0.0.0.0", port=80, use_reloader=False)

if __name__ == '__main__':
    thread_server = threading.Thread(target=start_server, daemon=True).
        start()
```

Code Listing 2: Flask im eigenen Thread starten

Eine in der API adressierbare Route kann in Flask über Function-Annotations definiert werden. Die von Flask implementierten Anntotations haben die Form `instanz.route('path', methods=["METHOD"])`. Der Name der Instanz wird am Anfang des Projekts als `app` definiert. Der `path` beschreibt die Route, die vom Frontend aufzurufen ist, um die nachfolgende Funktion auszuführen. Das Array `methods` definiert ein oder mehrere Request-Typen, die von der Funktion akzeptiert werden.

Eine Nutzung der Annotations, um eine Route in der API zu definieren, ist nachfolgend beispielhaft aufgeführt.

```
@app.route('/', methods=["GET"])
def index():
    [..]
```

```
    return r.respond({"token": token}, cookie=f"Authorization={token}")
```

Code Listing 3: Beispiel einer API-Routendefinition über Annotations

Der Inhalt der Methode und deren genaue Funktionsweise wird in den folgenden Kapiteln näher erläutert.

Die Funktion `respond` ist im Skript `api/response_generator.py` definiert. Sie dient als Function-Wrapper, der bei jeder ausgehenden Response die Response-Header, etwaige Cookies und den Reponse-Typen setzt. Der Output der Response wird mithilfe des `json`-Packages in JSON-Syntax konvertiert.

```
import json
from flask import Response
def respond(r, status=200, json_dump=True, cookie=""):
    [...]
    return Response(json.dumps(r), status=status, mimetype='application
        /json', headers=headers)
```

Code Listing 4: Respond Funktion zur Rückgabe von JSON-Responses

In Tabelle 1 sind die in der API verfügbaren Routen aufgeführt. Auf die genaue Funktionalität der einzelnen Funktionen wird in den folgenden Kapiteln eingegangen.

Route	Typ	Funktion
'/'	GET	Erstellung eines JSON-Web-Tokens
'/upload/file'	POST	Hochladen einer Textdatei für den Input der KI
'/upload/text'	POST	Texteingabe für den Input des KI-Services
'/transform'	POST	Festlegen der Tranformationseigenschaften
'/send'	POST	Transformieren und Senden des Inputs an einen KI-Service
'/poll'	GET	Abfrage der vom KI-Service gelieferten Ergebnisse
'/service'	GET	Auflistung der Services
'/service'	POST	Registrieren eines neuen Services
'/service'	DELETE	Löschen eines Services

Tabelle 3: Implementierte Routen der REST-API

4.3.2 Nutzeridentifizierung mit JWT

Innerhalb des Backendes ist es notwendig, einzelne Nutzer voneinander zu unterscheiden. Für jeden Nutzer speichert das Backend den hochgeladenen Text, die Transformationsanleitung und die Antworten des angefragten KI-Services im Redis-Cache. Um Nutzer voneinander unterscheiden zu können, gibt es zwei Möglichkeiten:

1. Identifizierung durch den Nutzer der Software, beispielsweise mittels Registrierung durch E-Mail Adresse und Passwort.
2. Identifizierung durch das Backend der Software; Generierung und Zuweisung einer zufälligen, aber eindeutigen User-ID.

Die Erhebung von personenbezogenen Daten erfordert die Einhaltung der Datenschutz-Grundverordnung (DSGVO). Aufgrund des daraus resultierenden Mehraufwands erhebt die Schnittstelle keine personenbezogenen Daten.

Das Backend nutzt einen Universally Unique Identifier (UUID), der sich durch das Python-Package `uuid` generieren lässt. Eine UUID ist eine 32-Zeichen lange Zahl im Hexadezimalformat. Die importierte Funktion `uuid4()` erzeugt eine zufällige, ohne von Parametern beeinflusste UUID. Der Nutzer bekommt diese UUID mitgeteilt, welche für seine Anfragen aufgrund der zustandslosen Implementierung der API im Authorization-Header mitgeschickt wird. Damit die UUID nicht ausgelesen oder manipuliert werden kann, wird sie nicht als einfacher Text in der Response an den Nutzer geschickt, sondern in ein JSON Token geschrieben und verschlüsselt.

Ein JSON Web Token (JWT) ist ein kompaktes, URL-sicheres Mittel zur Darstellung von Forderungen, die zwischen zwei Parteien übertragen werden sollen. Die Angaben in einem JWT werden als JSON-Objekt kodiert. Der Inhalt des JWT kann digital signiert oder die Integrität mit einem Message Authentication Code (MAC) geschützt und/oder verschlüsselt werden.[37]

Im nachfolgenden Codeausschnitt ist die Generierung der UUID und die Verschlüsselung des JWT dargestellt.

```
def uuid_gen():
    return uuid.uuid4()

def encode_token(param):
    return jwt.encode(param, JWT_PASSWORD, algorithm="HS256")

token = encode_token({'uid': str(uuid_gen())})
```

Code Listing 5: Generierung der UUID und Erstellung des JWT

Für jede Route, ausgenommen die `/service` Routen zum Management der Services, wird der JWT für die Ausführung benötigt. Die Überprüfung und Entschlüsselung des Tokens ist für jede Route gleich, daher ist es sinnvoll diese Funktionalität zu zentralisieren. Damit wird die Fehleranfälligkeit reduziert und und die Wartbarkeit erhöht, sollte sich zum Beispiel der Algorithmus oder das Passwort für die Verschlüsselung ändern. Wie auch Flask-Annotations zum Definieren einer Route verwendet, ist es möglich eigene Annotations zu entwerfen. Für diese Funktion ist das Python Package `functools` mit der Funktion `wraps` zuständig. `wraps` ermöglicht es, Funktionen ineinander zu verschachteln.

Im Prototyp wird die Funktion `token_required(f)` definiert. Diese Funktion dient als eine Umgebung, in der eine weitere Funktion ausgeführt werden. Im Gegensatz zur normalen Ausführung einer Funktion, werden in der `token_required(f)` Funktion vor der Ausführung der eigentlichen Funktion mehrere Rahmenbedingungen geprüft. Der vom Nutzer

[37] Jones u. a., 2015.

gesendete Token muss nach erfolgreicher Entschlüsselung syntaktisch korrektes JSON enthalten. Sollte dies nicht der Fall sein, wird die die Funktion, die zur API Route gehört, nicht ausgeführt. Der Nutzer bekommt direkt eine Response mit dem HTTP-Error-Code 401: Unauthorized gesendet.[38]

Nach erfolgreicher Entschlüsselung des Tokens, wird die innere Funktion ausgeführt. Als Parameter der inneren Funktion wird die im JWT enthaltene UUID übergeben. Durch diesen Aufbau ist der Code für die Verifizierung des Tokens und die Logik der Funktion unter der angesprochenen Route vollständig getrennt.

```
@routes.route('/upload/text', methods=['POST'])
@token_required
def upload_text(uid):
    [...]
```

Code Listing 6: Route zum Upload von Queries mit Nutzung des JWTs

4.3.3 Caching mit Redis-Datenbank

Redis ist ein Key-Value-Store der vollständig im RAM ausgeführt wird. Innerhalb von Redis sind mehrere Datenbanken definiert, die in ihrer Standartkonfiguration über einen Index i, mit $0 \leq i < 16$ aufgerufen werden. Im Backend werden die ersten drei Datenbanken verwendet:

1. Datenbank 0: Cache der hochgeladenen Queries für den Input der KI.
2. Datenbank 1: Cache der Transformationsanleitung.
3. Datenbank 2: Cache der vom KI-Service produzierten Ergebnisse.

Redis wird zum Cachen von Nutzereingaben und für die Zwischenspeicherung von Ergebnissen aus den KI-Services verwendet. In der ersten Datenbank werden die vom Nutzer eingegebenen Queries gespeichert. Redis speichert nur textbasierte Daten. Um die Query in Redis speichern zu können, muss zunächst eine Verbindung zum Redis-Cache aufgebaut werden. Die URL, unter der Redis erreichbar ist, wird aus den Environment-Variablen bezogen. Im ersten Schritt wird ein `ConnectionPool` angelegt. Dieser verbindet sich über einen Host und einen Port mit einer Datenbank. Anschließend wird im zweiten Schritt eine Redis-Instanz in Python erstellt. Diese bekommt den `ConnectionPool` als Argument übergeben. Innerhalb der Redis-Instanz befinden sich die benötigten Funktionen, um mit der Redis-Datenbank arbeiten zu können.

```
redis_host = os.environ.get('REDIS_HOST')
pool_upload = redis.ConnectionPool(host=redis_host, port=6379, db=0)
red_upload = redis.Redis(connection_pool=pool_upload)
```

Code Listing 7: Aufsetzen der Redis-Verbindung

[38] R. Fielding u. a., 1999.

In der Redis-Instanz werden zwei Funktionen definiert, die im Prototyp Verwendung finden. Die `set(k,v)` Methode nimmt einen Schlüssel als ersten Parameter und einen Wert als zweiten Parameter an. Das übergebene Key-Value-Paar wird mit diesem Methodenaufruf entweder neu in der Datenbank angelegt oder, falls bereits ein identischer Schlüssel vorhanden sein sollte, mit den aktuellen Werten überschrieben. Das Zwischenspeichern der hochgeladenen Query ist im nachfolgenden Codeausschnitt abgebildet.

```
red_upload.set(uid, body['text'])
```

Code Listing 8: Set-Funktion aus Redis

Die zweite Funktion der Redis-Instanz ist die `get(k)`-Methode. Diese bekommt einen Schlüssel übergeben und gibt den dazugehörigen Wert zurück. Das nachfolgende Codesegment stammt aus der Transform-Route, in der die zwischengespeicherte Query und die Transformationsanleitung aus dem Redis Cache geladen werden. Die aus dem Redis-Cache geladenen Werte sind in ein Zeichenformat zu konvertiert. Für den Prototyp wurde die Kodierung UTF-8 gewählt worden, da diese die meisten Sonderzeichen unterstützt.

```
messages = transform(red_upload.get(uid).decode('utf-8'), red_transform
    .get(uid))
```

Code Listing 9: Get-Funktion aus Redis

4.3.4 Kommunikation zwischen Backend und Services mit RabbitMQ

Für den Informationsaustausch zwischen dem Backend und den verschiedenen KI-Services ist eine asynchrone Kommunikation implementiert. Je nach Komplexität des Services kann die Verarbeitung einer vom Nutzer gestellten Anfrage mehrere Sekunden bis Minuten dauern. Eine synchrone Kommunikation, in der der Client auf unbestimmte Zeit auf eine Antwort wartet, ist nicht möglich. Wenn nach einer vom Browser definierten Zeit keine Antwort auf den Request kommt, wird der Request mit einem Timeout abgebrochen. Sollte die KI nach der maximal verfügbaren Zeit ihr Ergebnis liefern, wird dieses verworfen und der Nutzer muss eine neue Anfrage stellen. Damit Anfragen nicht verloren gehen und die Antworten dem Server mitgeteilt werden können, wurde der Message Broker RabbitMQ implementiert.

RabbitMQ dient als Middleware, die Anfragen vom Server annimmt und diese in einer Queue zwischenspeichert, um sie anschließend an die Services zu verteilen. Damit die Nachrichten in eine Queue geschrieben werden können, muss im ersten Schritt eine TCP-Verbindung zu RabbitMQ über das Package `pika` aufgebaut werden. Dafür verwendet die Schnittstelle den Host `rabbitmq:`, den Port 5672 und die Login-Credentials.

Innerhalb einer Connection können über einen Channel sowohl der Server als auch die Services eine Verbindung mit dem RabbitMQ Dienst aufbauen. Ein Channel beschreibt

die logische Verbindung zwischen dem Server/Service und dem Broker. Für jede TCP-Verbindung mit RabbitMQ können mehrere Channels erstellt werden.[39]

In einem Channel wird die Queue definiert, in der die Nachrichten zwischengespeichert werden. Im folgenden Codeausschnitt wird gezeigt, wie eine Connection, ein Channel und eine Queue erstellt werden:

```
def produce(uid, service, query, params):
    connection = pika.BlockingConnection(pika.ConnectionParameters(
        rabbit_host, 5672, '/', credentials))
    channel = connection.channel()
    channel.queue_declare(queue=service)
```

Code Listing 10: Verbindung zu RabbitMQ und Erstellung eines Channels und einer Queue

Die Nachrichten, die der Server an die Services schickt, werden mittels der im Channel definierten Methode `basic_publish` in die für den Service zuständige Queue geschrieben werden. Im vorherigen Codeausschnitt wird der Parameter `service` in der `produce` Methode übergeben. Dieser ist abhängig vom Service, an den die Anfrage gerichtet ist. Im Prototyp wurde der Service `text-similarity` implementiert. Dieser kann im Frontend als Ziel ausgewählt werden. Wenn der Nutzer im Frontend die Anfrage an den `text-similarity` Service stellt, wird die Methode `produce` mit dem String „`text-similarity`" aufgerufen.

RabbitMQ erstellt beim ersten Aufrufen eine Queue, falls diese noch nicht vorhanden ist. Ist sie bereits vorhanden, wird lediglich ein Eintrag in die entsprechende Queue geschrieben.

Die Nachricht, die aus der Anfrage des Nutzers, den eingegebenen Parametern und weiteren im Backend automatisch generierten Parametern, wie dem UUID, besteht, wird anschließend mittels AMQP an den Rabbit Broker gesendet. Dort wird die Nachricht, in die zuständige Queue eingetragen.

Im Serice wird eine Methode implementiert, die die Nachrichten in einer Queue auslesen kann. Zu Beginn muss eine Queue definiert werden, aus der Nachrichten ausgelesen werden. Dies funktioniert analog zum Erstellen einer Queue für das Schreiben von Nachrichten über `queue_declare` gefolgt von dem Namen der Queue. Das Registrieren als Consumer für eine Queue wird mittels der Methode `basic_consume()` durchgeführt. Innerhalb der Parameter ist eine `callback`-Methode definiert, die ausgelöst wird, wenn der Service eine Nachricht erfolgreich aus der Queue ausgelesen hat. Die `callback`-Methode ist der Startpunkt des Services, von dem aus die eingehende Nachricht mithilfe der KI analysiert und verarbeitet wird.

[39] Dossot, 2014.

Um den Consumer zu starten und damit auf neue Nachrichten in der Queue zu warten, wird die Methode `start_consuming()` ausgeführt. Im nachfolgenden Codesegment ist der eben beschriebene Prozess im KI-Service aufgeführt.

```
channel.queue_declare(queue='text-similarity')
channel.basic_consume(queue='text-similarity',
                      auto_ack=True,
                      on_message_callback=callback)
channel.start_consuming()
```

Code Listing 11: Aufsetzen und Konsumieren der RabbitMQ-Queue im KI-Service

Nachdem die KI die eingehende Anfrage verarbeitet hat, schreibt der Service die Response in eine Queue. Es kann jedoch nicht die gleiche Queue für Anfragen und Antworten verwendet werden, da es sonst dazu führen könnte, dass vom Service produzierte Antworten wieder als Anfrage interpretiert werden und es so zur fehlerhaften Ausführung des KI-Algorithmus kommt. Ein weiteres Problem wäre, dass die Antworten dann aus der Queue herausgenommen wurden und der Server keine Antwort mehr zur gestellten Anfrage erhält. Um dem entgegenzuwirken, wurde eine zweite Response-Queue implementiert, in die ausschließlich die Antworten des Services geschrieben werden. Da die Queues über ihren Namen definiert werden, hat jede Response-Queue einen vom Service abhängigen, automatisch generierten Namen. Der Name hat die Form `response-{service}`. Im Falle des `text-similarity` Services, hat die Antwort-Queue den Namen `response-text-similarity`.

```
channel.queue_declare(queue='response-text-similarity')
channel.basic_publish([...], body=f'{{"uid": "{uid}", "service": "{
    service}", "message": {json.dumps(message)}}}'.encode('utf-8'))
```

Code Listing 12: Senden eines KI-Ergebnisses an das Backend

Der Server implementiert, ähnlich wie der Service auch, eine Möglichkeit die Nachrichten in der Response-Queue auszulesen. Innerhalb der Nachricht, die an den Service geht und vom Service wieder zurückkommt, wird die UUID des Nutzers mitgeliefert. Damit ist eine anschließende Zuordnung von Anfrage und Antwort möglich. Das Backend speichert die Antwort des KI-Services im Redis Cache und kann sie dem Frontend bei Bedarf zur Verfügung stellen.

Der Ablauf der Kommunikation zwischen dem Server und den Services mit RabbitMQ ist in Abbildung 12 veranschaulicht:

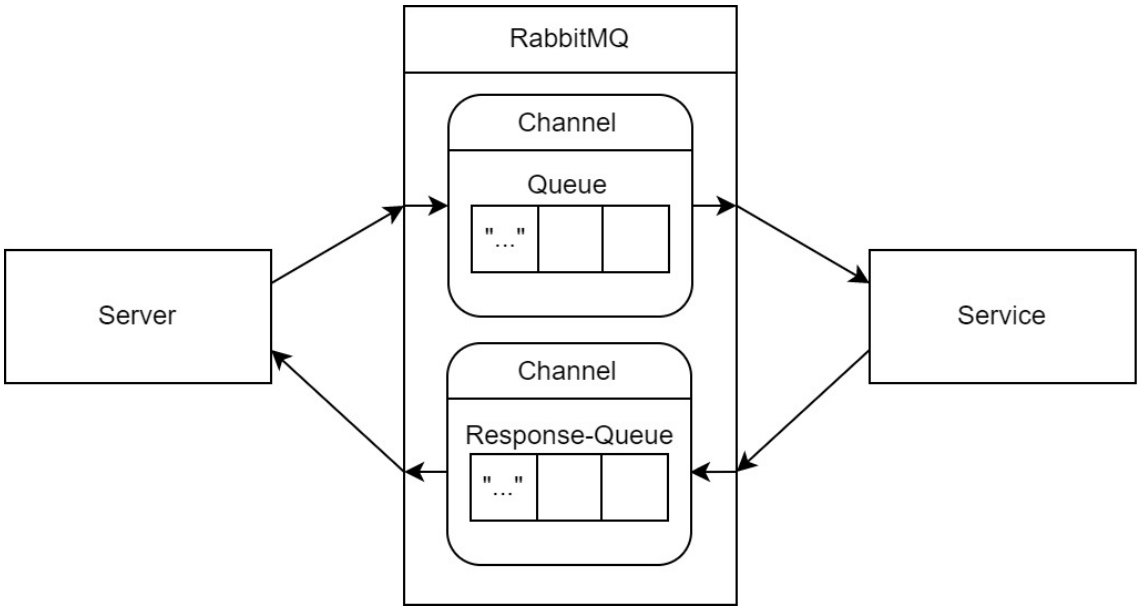

Abbildung 12: Kommunikation mit RabbitMQ

4.3.5 Implementierung des KI-Services

Der implementierte KI-Service lädt im ersten Schritt das KI-Modell, um mit diesem für jeden Eintrag in der Elasticsearch-Datenbank einen semantischen Vektor zu generieren. Dieser Vektor wird ebenfalls in der Datenbank gespeichert und kann für zukünftige Abfragen genutzt werden. Nachdem sämtliche Einträge in der Elasticsearch-Datenbank indiziert wurden, beginnt die Programmablaufschleife. Im ersten Schritt der Schleife wartet der Service eingehende Nachrichten. Diese Nachrichten werden über RabbitMQ aus der für den Service definierten Queue ausgelesen. Falls keine Anfrage an den Service eintrifft, wartet der Dienst auf unbestimmte Zeit. Nach dem erfolgreichen Auslesen einer Nachricht startet der KI-Service mit der Bearbeitung der Anfrage. Im ersten Schritt wird aus dem Satz, den der Nutzer an den Service geschickt hat, ein semantischer Vektor generiert. Dazu wird das gleiche KI-Modell wie bei der Indizierung der Einträge in der Datenbank verwendet. Daraufhin nutzt der Service den generierten Vektor und die `cosineSimilarity()` Funktion der Elasticsearch, um die semantisch ähnlichsten Texte aus der Datenbank herauszufiltern. Die Anzahl der gelieferten Ergebnisse ist abhängig von den übergebenen Parametern. Der Nutzer kann im Frontend das Key-Value-Paar `search_size` eingeben, welches in der Anfrage an den Service mitgeliefert wird. Sollte der Nutzer beispielsweise „`search_size : 3`" eingeben, werden die drei ähnlichsten Artikel zur gestellten Anfrage zurückgegeben. Die Ergebnisse werden in die Response-Queue des Services geschrieben, damit sie im Backend weiterverarbeitet werden können. Nach erfolgreichem Durchlaufen des Prozesses geht der Service wieder zum Zustand „Auf Anfrage warten“ über. Veranschaulicht wird der Programmablauf des Services in Abbildung 13:

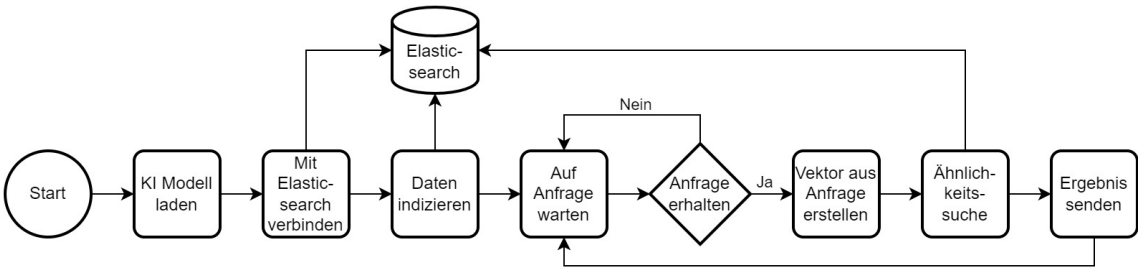

Abbildung 13: Ablaufdiagramm der Textähnlichkeitssuche

Zur Durchführung der Textähnlichkeitssuche importiert der KI-Service das BERT Modell, welches beim Starten automatisch aus dem offiziellen Tensorflow-Hub heruntergeladen wird. Anschließen wird eine Tensorflow-Session erstellt und gestartet.

```
embed = hub.Module("https://tfhub.dev/google/universal-sentence-encoder
    /2")
```

Code Listing 13: Laden des BERT-Modells

Im zweiten Schritt verbindet sich der Service mit der Elasticsearch-Datenbank. Dazu wird die URL, unter der die Elasticsearch erreichbar ist aus den Environment-Variablen geladen. Die Environment-Variablen können entweder beim manuellen Starten des Services mitgegeben werden oder falls der Service mit Docker gestartet wird, in der Docker-Compose File definiert werden. Mithilfe der URL initialisiert der Service eine Elasticsearch-Instanz und speichert sie in der Variable `client` ab.

```
elasticsearch_url = os.environ.get('ELASTIC_URL')
client = Elasticsearch(elasticsearch_url)
```

Code Listing 14: Starten der Elasticsearch-Instanz

Nachdem sich der Service erfolgreich mit der Elasticsearch verbunden hat, beginnt das Indizieren der Daten in der Datenbank. Dazu wird die Methode `index_data()` aufgerufen. Der Datensatz, der für den Prototyp verwendet wurde, ist ein Auszug aus den gestellten Fragen im Forum Stackoverflow. In dem Datensatz sind 18.600 Fragen vorhanden. Jede Frage besteht aus einem Titel und einem Body. Fragen-Body besteht aus einer genauere Erläuterung der Frage, mit eventuellen Codeausschnitten oder Verlinkungen. Für die Indizierung wurde der Titel der Frage genutzt. Das bedeutet im Umkehrschluss, dass der Titel der Frage repräsentativ für die gesamte Frage stehen muss. Nach Möglichkeit soll der Titel den gesamten Inhalt der Frage kurz und bündig zusammenfassen. Aus diesem Grund eignet sich Stackoverflow besonders gut für den Datensatz. In den Guidelines wird „Write a title that summarizes the specific problem“ explizit gefordert.[40]

Nach erfolgreicher Indizierung sämtlicher Fragestellungen geht der Service in die Programmablaufschleife über. Die Schleife beginnt mit der Initialisierung der RabbitMQ-

[40] Stackoverflow, 2022.

Verbindung. Dort verbindet sich der Textähnlichkeitsservice mit der Queue `text-similarity`. Bei einer eingehenden Anfrage ruft die `callback` Funktion die `respond`-Funktion auf, die den KI-Algorithmus zur Verarbeitung der Anfrage und Generierung der Antwort startet. Die Funktion `handle_query()` benötigt als Parameter die vom Nutzer gestellte Query sowie die Anzahl der Ergebnisse, die zurückgegeben werden. Beide Parameter werden aus der Anfrage, die über RabbitMQ an den Service gesendet wurde, ausgelesen. Sollte es vorkommen, dass der Nutzer keine `search_size` angegeben hat, wird der Standardwert 1 gesetzt.

```
search_size = 1
    for p in params:
        if "search_size" in p.values():
            search_size = p["value"]

    message = handle_query(query, search_size)
```

Code Listing 15: Auslesen der search_size und Ausführen der KI

Innerhalb der `handle_query()` Methode wird zunächst der Satz, der vom Nutzer gesendet wurde, durch die Methode `embed_text()` in einen Vektor konvertiert. Dieser wird anschließend in das Feld „`params`" in der Such-Query für die Elasticsearch geschrieben. In dem Feld „`source`" wird die Ähnlichkeitsanalyse definiert. Die Elasticsearch implementiert die Funktion `cosineSimilarity()`, die zwei Vektoren als Parameter annimmt. Der erste Vektor ist der aus der Query, der zweite aus dem Titel der Frage generiert Vektor. Mit der `client.search()` stellt der Service die Anfrage an die Elasticsearch mit der zuvor definierten Query. In der Anfrage wird über das Feld „`size`" die Anzahl der gewünschten Ergebnisse definiert. Wenn die `search_size` beispielsweise 3 beträgt, werden die drei Fragen mit den ähnlichsten Vektoren in die Variable `response` geschrieben.

Im nachfolgenden Codesegment stellt die sind die zuvor beschriebene Erstellung des Query Vektors, der Such-Query für die Elasticsearch und die Anfrage an die Elasticsearch dar:

```
query_vector = embed_text([query])[0]
script_query = {
    [...]
        "source": "cosineSimilarity(params.query_vector, doc['
            title_vector']) + 1.0",
        "params": {"query_vector": query_vector}
}
response = client.search(
    [...]
        body={
            "size": search_size,
            "query": script_query,
            [...]
```

```
        }
)
```

Code Listing 16: Ähnlichkeitssuche in der Elasticsearch

Die Response der Elasticsearch wird nach erfolgreicher Durchführung wieder an die `respond` Funktion übergeben. Dort wird die Nachricht vorbereitet, die als Response auf die Anfrage vom Backend zurückgesendet wird. Die Antwort wird an RabbitMQ in die Queue `response-text-similarity` gesendet.

Der Schleifendurchlauf des Services ist damit abgeschlossen. Der Service nimmt den Zustand „Auf Anfrage warten“ ein, vgl. Abbildung 13.

4.3.6 Management der Services

Die entworfene Softwarearchitektur unterstützt eine beliebige Anzahl an KI-Services. Die Services, die über das System erreichbar sein sollen, müssen über die API registriert werden. Jeder Service implementiert eine Verbindung zur RabbitMQ-Schnittstelle, damit die Anfragen und Antworten asynchron zwischen dem Backend und dem jeweiligen Service gesendet werden können. Eine der Kernanforderungen für eine Architektur, die austauschbare KI-Services unterstützt, ist die dynamische Registrierung und Deregistrierung. Eine Registrierung beschreibt das Hinzufügen eines neuen Services im System, ohne den Quellcode anpassen zu müssen. Ein persistenter Speicher, in dem die zur Verfügung stehenden Services aufgelistet sind, ermöglicht dies. Zur persistenten Speicherung von kleineren Datenmengen eignet sich eine MySQL-Datenbank. Innerhalb der Datenbank werden die aktiven Services in der Tabelle `services` abgespeichert. Jedem Service ist ein Name zugeordnet mit der Bezeichnung `service` und einen Anzeigenamen mit der Bezeichnung `display_name`. Der Name wird für das interne Routing über RabbitMQ genutzt. Der Anzeigename wird im Frontend innerhalb des Dropdowns für die Auswahl des Services verwendet.

4.3.7 Automatisierte Transformation des Inputs

Bei der Nutzung eines KI-Services müssen die Eingaben in einer bestimmten Form vorliegen. Dies ist besonders dann von Bedeutung, wenn KIs von Drittanbietern verwendet werden. Da die Implementation oftmals als Blackbox erfolgt und nur eine Schnittstelle nach außen bereitgestellt wird, kann nicht der Algorithmus an die Daten angepasst werden, sondern die Form der Daten muss an den Algorithmus angepasst werden. Daten kommen jedoch häufiger mit überschüssigen Informationen, je nach Quelle und Methode der Datenerhebung. Die Bereinigung per Hand ist sehr aufwändig und führt schnell zu Fehlern. Sollten Echtzeitdaten an einen KI-Service angeschlossen werden, ist eine händische Bereinigung grundsätzlich nicht möglich.

Um diesem Problem entgegenzuwirken, wurde im Backend ein Algorithmus entwickelt, der Daten mithilfe einer Transformationsanleitung automatisch bereinigen kann. Die Transformationsanleitung wird durch den Anwender eingegeben und hat folgende Form:

```
{
  "seperator": "\n",
  "replace": [
    {
      "old":"\[(\d*:?)*\]",
      "new":""
    }
  ]
}
```

Code Listing 17: Beispiel einer Transformationsanleitung

`seperator` und `replace` sind die im Prototyp unterstützen Transformationsoperationen. `seperator` teilt die vom Nutzer eingegebene Query in mehrere einzelne Queries auf. Dazu nutzt er die `split` Operation aus Python, die für Strings standardmäßig verfügbar ist. In der Transformationsanleitung wird ebenfalls ein Array an `replace` Operationen angegeben. Jeder Eintrag im `replace` Array enthält ein Objekt mit Key-Value Paaren `old` und `new`. Der alte String wird durch den neuen String ersetzt. Dazu wird die `replace` Funktion aus dem Python Package `re` verwendet. Die `replace` Funktion nutzt Regular-Expression als Syntax für das Ersetzen. Dadurch sind auch komplexere Ersetzungsverfahren möglich. Im Beispiel für eine Transformationsanleitung wurde eine Regular-Expression zur Entfernung eines Zeitstempels mit der Form `[12:25:10]` angegeben. Der nachfolgende Code zeigt die Implementierung der `transform`-Methode, die mithilfe der Anleitung die Transformation durchführt. Zunächst wird die Query in einzelne Nachrichten geteilt. Anschließend läuft eine Schleife über sämtliche Nachrichten und wendet die Transformationsschritte an. Der Algorithmus gibt ein Array an transformierten Nachrichten zurück:

```
def transform(file: str, transform_json):
    [...]
    parts = file.split(transform_json['seperator'])
    messages = []
    for m in parts:
        for replacer in transform_json['replace']:
            m = re.sub(replacer['old'].replace("\\\\", "\\"), replacer[
                'new'], m)
        m = m.strip()
        messages.append(m)
    return messages
```

Code Listing 18: Transformationsalgorithmus

4.3.8 Fehlerbehandlung

Während der Laufzeit des Programms können vom Nutzer produzierte Fehler auftreten. Der Nutzer kann im Bereich der Transformation syntaktisch nicht korrekte Texte eingeben, die im Backend verarbeitet werden. Da Nutzereingaben ohne weitere Behandlung ebenfalls ein Sicherheitsrisiko für die Infrastruktur darstellen können, wird im Backend ein System implementiert, um die Verarbeitung der Eingaben in einer isolierten Umgebung ausführen zu können. Ähnlich wie bei der Verifizierung des JWT wurde das System zur Fehlerbehandlung mit Function-Wrappern und Annotations implementiert.

Im Backend wird ein Exception-Handler definiert, der eine Funktion mit zuvor übergebenen Parametern ausführt. Da eine fehlerhafte Ausführung auch dort zum Programmabsturz führen würde, wird die Funktion in einem Try-Except Block gekapselt. Die innerhalb dieses Blocks auftretenden Fehler und die dazugehörige Fehlermeldung werden abgefangen und in einem Exception-Objekt gespeichert. Über `str(e)` kann auf die Fehlermeldung zugegriffen werden. Innerhalb des Except-Blocks wird die Fehlermeldung an den Logger weitergegeben, um diese in einer Datenbank persistent zu speichern. Nach erfolgreichem Log wird dem Frontend in einer HTTP-Response der Statuscode 500 „Internal Server Error“ zurückgegeben.[41]

```
[...]
 @wraps(f)
        def decorator(*args, **kwargs):
            try:
                return f(*args, **kwargs)
            except Exception as e:
                logger.log("error", f"[Server, {msg}]: {str(e)}", "none
                    ")
                return r.respond({"success": False, "error": str(e)},
                    status=500)
            [...]
```

Code Listing 19: Exception-Handling mithilfe von Wrappern

Jede Funktion, die in innerhalb des Exception-Handlers ausgeführt werden soll, wird mit der Annotation `@exception_handler("...")` versehen. Innerhalb der Funktion wird der Ort, an dem die Funktion ausgeführt wird, und damit der Fehler auftritt, als String übergeben. Diese Information wird genutzt, um innerhalb des Fehlerlogs den Ort des Fehlers aufzulisten.

[41] R. Fielding u. a., 1999.

4.3.9 Event Logging

Das Backend implementiert ein Event-Logging-System, mit dem Zustände und Informationen des Systems in einer Datenbank gespeichert werden können. Mithilfe von Logs können Entwickler den Ablauf eines Programms besser nachvollziehen und auftretende Fehler schneller zu ihrer Quelle zurückverfolgen. Es ist ebenfalls möglich, Systeme aufzusetzen, die auf Logs auslesen und beim Auftreten eines Errors die zuständigen Personen alarmieren. Eine Herausforderung bei einem Logging System ist es, dass das System beim Entwickler durch das Loggen keinen erheblichen Mehraufwand produzieren soll. Im Backend wurde ein Logging System entwickelt, welches mit einer einzigen Funktion angesteuert werden kann. Die `log`-Funktion des Loggers wird in den verschiedenen Bereichen der Anwendung über `from logs.logger import log` importiert. Nach erfolgreichem Import steht die `log` Funktion zur Verfügung.

Für einen Log müssen drei Parameter übergeben werden: der Log Level, die Message und die UUID. Die möglichen Ausdrücke für die Log Level sind in Tabelle 4 aufgeführt. Die unterstützten Log Level leiten sich aus der Funktionalität von Grafana ab, welche diese Logs mit einer zugeordneten Farbe visualisiert.

Ausdruck	Log Level	Farbe
emerg	critical	lila
fatal	critical	lila
alert	critical	lila
crit	critical	lila
critical	critical	lila
err	error	rot
eror	error	rot
error	error	rot
warn	warning	gelb
warning	warning	gelb
info	info	grün
information	info	grün
notice	info	grün
dbug	debug	blau
debug	debug	blau
trace	trace	hellblau
*	unknown	grau

Tabelle 4: Log Level des Event Logging Systems

Innerhalb der Log-Funktion wird eine Datenbank-Query mit den drei Parametern und einem aktuellen Zeitstempel erstellt. Der Zeitstempel kann mithilfe des `datetime` Packages erstellt werden. Über `cursor.execute()` wird die erstellte Query in der MySQL-Datenbank ausgeführt. Der Log wird als Eintrag in der Tabelle `logs` gespeichert. Im folgenden Codeausschnitt ist die Log Funktion abgebildet:

```
def log(level, message, uid):
    [...]
    query = f"INSERT INTO logs (`level`, `message`, `timestamp`, `uid`)
        VALUES (%s, %s, %s, %s)"
    cursor.execute(query, (level, message, datetime.utcnow(), str(uid))
       )
    [...]
```

Code Listing 20: Erstellung eines Logs

Die in der MySQL-Datenbank hinterlegten Logs, werden in Grafana visualisiert. Dazu muss ein Dashboard angelegt werden. Innerhalb des Dashboards wird ein durch Grafana bereitgestelltes Logs-Panel benötigt. Grafana ordnet die anzuzeigenden Daten grundsätzlich nach dem Zeitpunkt der Erstellung. Innerhalb des Logs-Panels wird der aktuellste Log oben angezeigt, die anderen chronologisch absteigend. Da in Grafana die MySQL-Datenbank als Datenquelle hinterlegt ist, muss auch die Abfrage der Daten in SQL stattfinden. Innerhalb der SQL-Abfrage ist die Auswahl des timestamps an erster Stelle durch Grafana vorgegeben. Weitere Spalten können beliebig gewählt werden. Eine Spalte mit dem Namen „message“ wird direkt im Logs-Panel angezeigt, ohne den jeweiligen Log ausklappen zu müssen. Der Ausdruck „level“ ist ebenfalls von Grafana reserviert. Sollte sich der Eintrag der „level“ Spalte mit einem der Ausdrücke aus Tabelle 3 decken, wird die entsprechende Farbe für den Log genutzt. Weitere selektierte Spalten werden durch Ausklappen des jeweiligen Log-Eintrags sichtbar. Die genutzte Query zur Abfrage der Logs ist im nachfolgenden Codesegment dargestellt.

```
SELECT timestamp, message, level, uid from db_logs.logs;
```

Code Listing 21: Query zur Abfrage der Logs in Grafana

Das Log Panel mit den aus der Query resultierenden Logs ist in Abbildung 14 dargestellt:

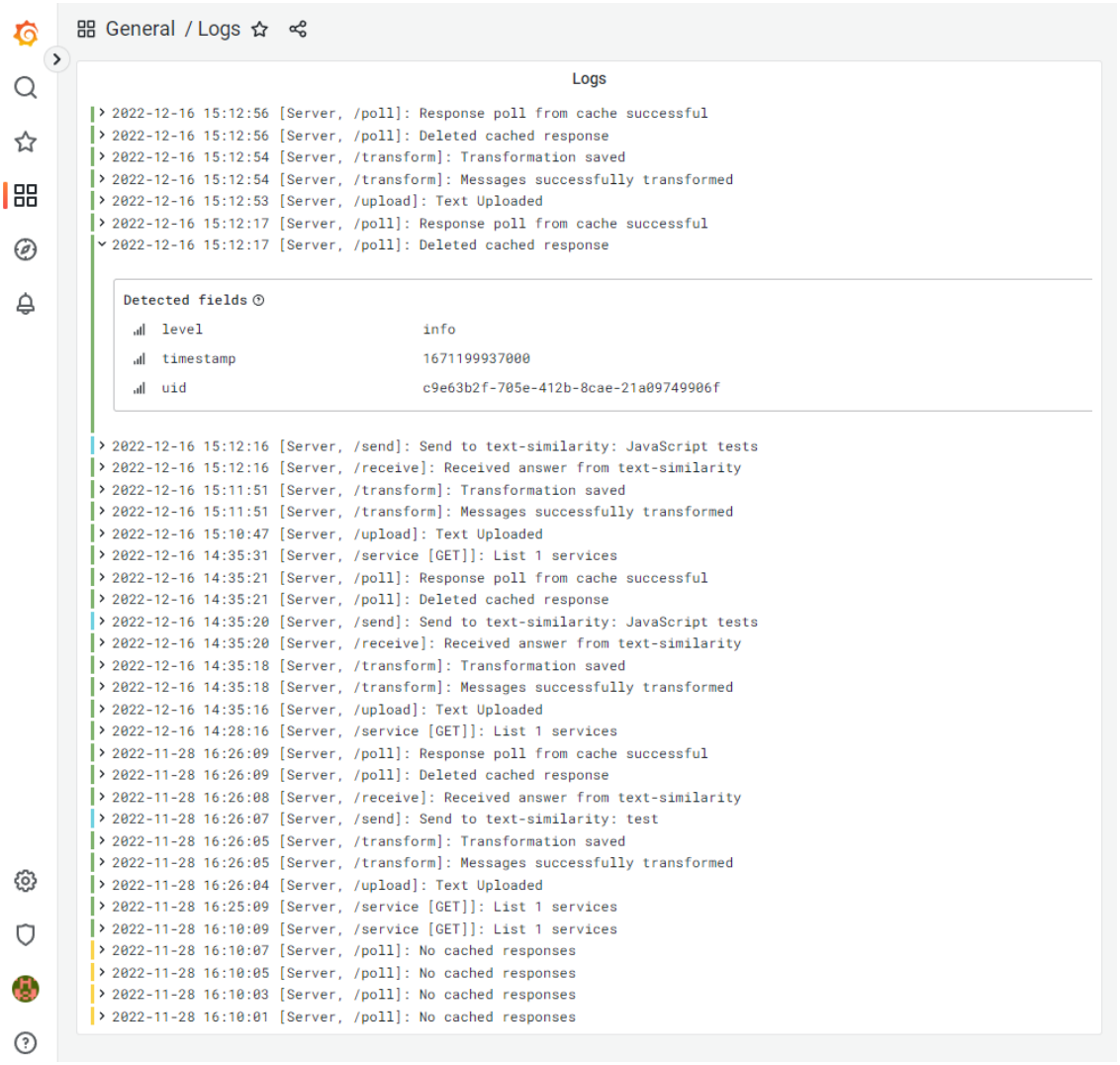

Abbildung 14: Logs in Grafana

4.3.10 Website mit Angular

Damit ein Anwender die Schnittstelle zur Anbindung von Daten an KI-Services nutzen kann, wurde eine Website entworfen und entwickelt. Die grundlegende Funktionsweise der Website ist in den Abschnitten 4.1 Anforderungen und 4.2 Konzeption beschrieben. In diesem Kapitel liegt der Fokus auf der technischen Umsetzung der Website.

Das Frontend wurde mit dem Framework Angular entwickelt. In Abbildung 15 ist der fertiggestellte Prototyp der Website dargestellt:

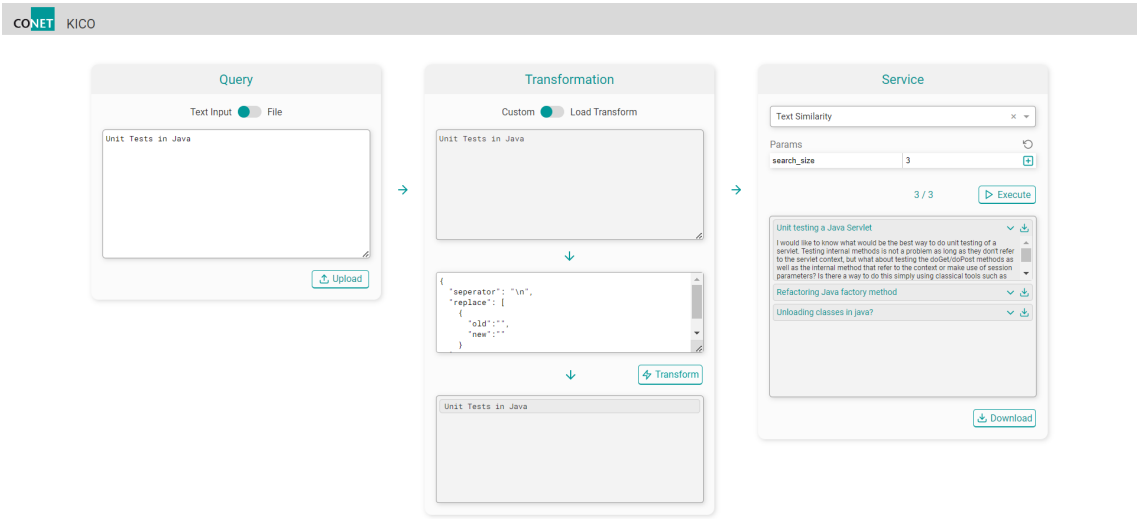

Abbildung 15: Gesamtansicht der Website

Die Website teilt sich in drei Bereiche auf. In Angular ist es möglich, eigene Components zu definieren. Ein Component besteht aus drei Dateien. Die grundlegende Datei ist eine TypeScript Datei. Innerhalb dieser Datei wird der eigentliche Component definiert. Zunächst muss dafür `Component` aus `@anuglar/core` importiert werden. Im Anschluss im Bereich unter `@Component` ein `selector`, eine `templateUrl` und die `styleUrls` definiert. Der `selector` beschreibt, wie der Component im HTML Code aufgerufen werden soll. Jeder Component erfordert einen individuellen Namen. Um die eigenen Components von den Components, die durch Angular bereitgestellt sind, unterscheiden zu können, enthält jeder Namen das Prefix `app-`. Die `templateUrl` gibt den Pfad der dazugehörigen HTML Datei an. Diese Datei ist die zweite Datei, die zur Erstellung eines Components notwendig ist. Innerhalb der `styleUrls` können mehrere Pfade zu verschiedenen Dateien angegeben werden. Die SCSS-Datei ist die dritte Datei, die zur Erstellung eines Component benötigt wird. Im folgenden Codeausschnitt ist die Initialisierung des Query-Imput-Components innerhalb der TypeScript-Datei dargestellt:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-datasource-input',
  templateUrl: './datasource.input.component.html',
  styleUrls: ['./datasource.input.component.scss']
})
```

Code Listing 22: Definition eines Components

Innerhalb der TypeScript-Datei können Variablen und Funktionen definiert werden. Diese werden dazu genutzt, Nutzerinteraktionen mit der Website festzuhalten, oder Ereignisse darzustellen. In der Query-Input-Component wird die Variable `inputStr` definiert.

Innerhalb dieser Variable wird die Eingabe des Nutzers im Textfeld der Query-Kachel gespeichert. Für sich alleinstehend hat die Variable keine Funktion. Diese bekommt sie erst durch den Code aus der HTML Datei.

```
<textarea [(ngModel)]="inputStr" placeholder="Input..." class="text-
    input"></textarea>
```

Code Listing 23: Erstellung einer Textarea innerhalb der HTML-Datei

Innerhalb der HTML Datei für die Query ist eine Textarea definiert. Über den Selector `[(ngModel)]` kann eine Variable angegeben werden, die mit dem Inhalt der Textarea synchronisiert wird. Sobald der Nutzer eine Eingabe in der Textarea tätigt, wird diese sofort in der TypeScript-Datei aktualisiert.

Es ist ebenfalls ein `class` Attribut innerhalb der Textarea angegeben. Dieses setzt eine SCSS-Klasse für die Textarea. In der Definition der Component ist eine SCSS-Datei angegeben, die in diesem Schritt aufgerufen wird. Dort wird die Klasse `text-input` definiert und mit verschiedenen Style-Eigenschaften versehen. Diese Eigenschaften werden auf die Textarea angewandt:

```
.text-input{
  border-radius: 5px;
  border: 1px solid #9A9A9A;
  box-shadow: 2px 2px 8px rgba(0,0,0,.15);
  resize: vertical;
  height: 206px;
  width: 90%;
  font-family: "Roboto Mono", sans-serif;
  padding: 5px;
}
```

Code Listing 24: Style-Definition innerhalb einer SCSS-Datei

Die daraus resultierende Textarea innerhalb der Query-Kachel ist in Abbildung 16 dargestellt:

Abbildung 16: Query Kachel im Frontend

Der Bereich der Transformation besteht aus mehreren Sub-Components. Die obere und mittlere Textarea ist analog zu der Textarea aus der Query implementiert. Da die obere Textarea jedoch ausschließlich zur Bestätigung der Eingabe dient und nicht selbst zu Eingabe genutzt werden soll, wurde die Definition um das Attribut `readonly` erweitert. Der untere Bereich der Transformation-Component teilt sich in zwei Components auf. Der übergeordnete Component ist das `transformation.result`. Dies ist die graue Box, in der eine Liste an transformierten Eingaben dargestellt werden. Jedes Element der Liste ist ein `transformation.result.item` Component. In Angular kann eine Liste im HTML-Code direkt aus einer Variable erzeugt werden. Dazu wird `*ngFor` verwendet. Innerhalb der Schleife ist eine Variable aus der zugehörigen TypeScript-Datei angegeben, über die iteriert wird. Sollte sich die Variable zur Laufzeit ändern, wird auch direkt die HTML Datei angepasst. Der folgende Codeausschnitt zeigt die Implementierung des `transformation.result` Components.

```
<div class="container">
  <li *ngFor="let item of results; index as i" class="list-results">
    <app-transformation-result-item [text]="item"></app-transformation-
       result-item>
  </li>
</div>
```

Code Listing 25: Anzeige der Results aus der Transformation

Abbildung 17 stellt die Transformation der Query, sowie das Ergebnis dar.

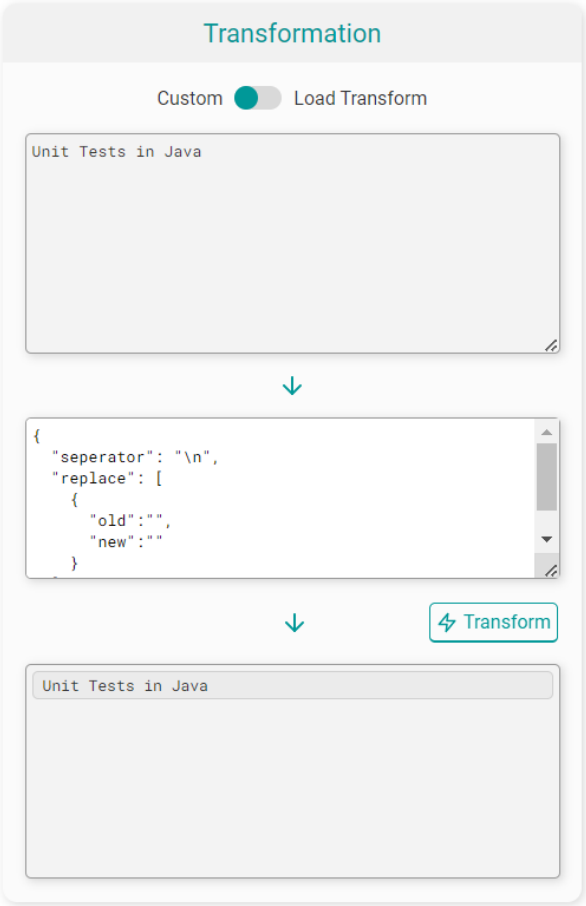

Abbildung 17: Transformation Kachel im Frontend

Der dritte Bereich der Website enthält die Auswahl und Ergebnisauflistung des KI-Services. Die Kommunikation zur API wird durch Services, die in TypeScript-Dateien definiert werden, bereitgestellt. Eine der Funktionen aus dem Service für den Service-Component ist `getService()`-Methode. Diese kann innerhalb der TypeScript-Datei der Component aufgerufen werden. Die `getService()` Methode stellt einen HTTP-GET-Request an die `/service` Route der API. Die genaue URL der API wird aus den Environmentvariablen ausgelesen. Die Methode gibt nicht direkt die Response auf den Request zurück, sondern ein Objekt vom Typ Observable. Innerhalb der TypeScript-Datei des Components kann auf das Observable subscribed werden.

```
getServices() {
  const headers = {'Authorization': environment.token}
  return this.http.get<any>(environment.backend_url + '/service', {
     headers})
}
```

Code Listing 26: Abfrage der Services

Ein asynchroner Aufruf wird an die API gesendet und bei einer Response das Ergebnis verarbeitet. Das Ergebnis ist eine Liste der verfügbaren KI-Services, die in der Variable `services` gespeichert werden:

```
this.serviceService.getServices().subscribe(res => {
  this.services = res['services']
})
```

Code Listing 27: Subscriben auf einen Observable

Im HTML des Service-Components wird die Liste der Services für ein Dropdown verwendet. In Angular werden Events bereitgestellt, die von einem Component ausgelöst werden können. Ein Component reagiert auf ein durch ein Sub-Component ausgelöstes Event. Innerhalb des Events wir die Funktionalität bereitgestellt, TypeScript-Code auszuführen oder vordefinierte Funktionen aufzurufen. Im Dropdown wird das `changeEvent` abgefangen. Bei der Auswahl eines neuen Elements aus dem Dropdown wird das Event ausgelöst. Die Funktion `serviceSelected()` wird durch das Event aufgerufen:

```
<app-dropdown [elements]="services" (changeEvent)="serviceSelected(
    $event)"></app-dropdown>
```

Code Listing 28: Dropdown zur Auswahl der Services

Die Tabelle für die Parameter nutzt ebenfalls `*ngFor`, um die einzelnen Reihen zu generieren.

Der letzte Component innerhalb des Services ist der `service.result`-Component, der ein Ergebnis aus dem KI-Service anzeigt. Das Ergebnis besteht aus einer Überschrift und einem Body. Damit die Ergebnisse beim Frontend eintreffen, muss der Nutzer nach der Auswahl des Services auf den Execute-Button klicken. Dadurch wird im Angular-Service die Methode `sendRequests()` aufgerufen, die Anfrage für den KI-Service an die API sendet. Anschließend wird die Funktion `pollResponses(expectedResponses: number)` mit der Anzahl der erwarteten Ergebnisse aufgerufen. Diese führt in einem 200ms Intervall die `poll()`-Funktion aus, die eine Anfrage an die API stellt, ob bereits neue Ergebnisse des KI-Service im Backend zwischengespeichert wurden. Falls dies der Fall sein sollte, werden die Ergebnisse in der Response mitgeteilt und die Anzahl der erhaltenen Ergebnisse hochgezählt. Die Schleife läuft, bis die Anzahl der erhaltenen Ergebnisse gleich der Anzahl der erwarteten Ergebnisse ist. Für jedes Ergebnis wird ein `service.result`-Component in einer `*ngFor` Schleife erzeugt. Abbildung 18 zeigt die Auflistung der Ergebnisse und die zuvor beschriebenen Elemente der Service-Kachel.

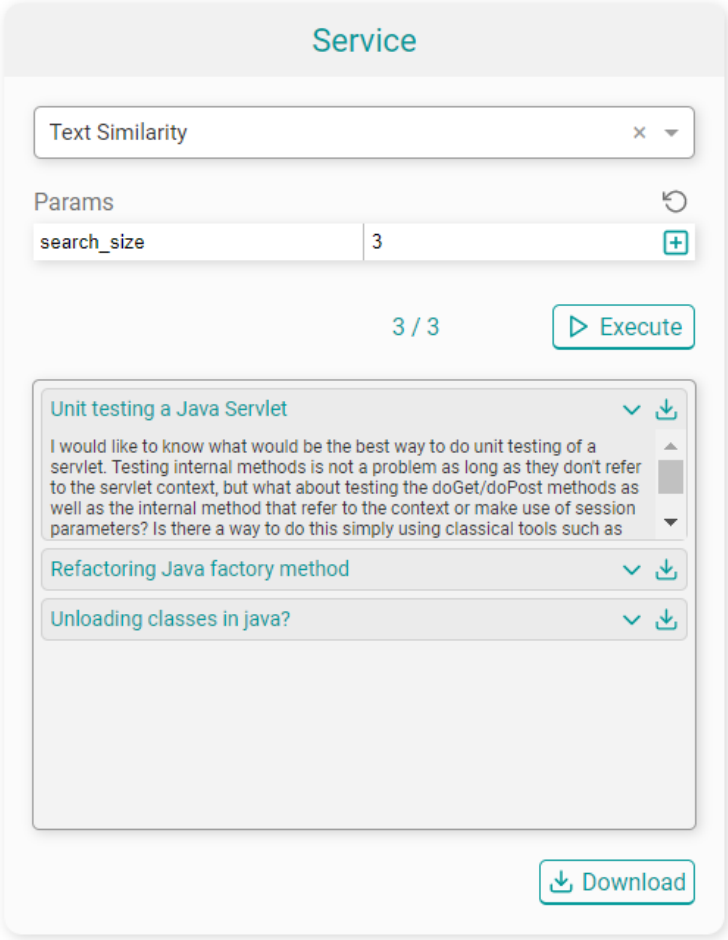

Abbildung 18: Service-Kachel im Frontend

4.4 Deployment der Software mit Docker

Eine der Anforderungen von CONET war die plattformunabhängige Entwicklung der Schnittstelle. Sämtliche Bereiche der Softwarearchitektur sind daher über Docker-Container deploybar. Um eine entwickelte Software in einen Container zu packen, wird ein Docker-Image aus der Software erstellt. Ein Image dient als Bauanleitung für den Container und enthält sämtliche Dateien, die zur Ausführung des Programms benötigt werden. Ein Image kann über eine Docker-File generiert werden. Die Docker-File zur Erstellung des Images für das Backend ist nachfolgend abgebildet:

```
FROM python:3.10-slim
WORKDIR /app
ADD . /app
RUN pip install -r requirements.txt
EXPOSE 80
CMD ["python", "app.py"]
```

Code Listing 29: Beispiel einer Dockefile

Innerhalb der Docker-File können Pakete aus dem Docker-Hub als Abhängigkeit angegeben werden, die beim Bauen des Images heruntergeladen werden. Im Falle des Backends wird das Paket `python:3.10-slim` bezogen. Innerhalb der Docker-File ist die Festlegung der Ordnerstruktur und die Ausführung von Kommandos möglich.

Zum Bauen eines Images mithilfe der zuvor erstellten Docker-File wird der `build`-Befehl benötigt. Für das Backend, den KI-Service sowie das Frontend ist eine Docker-Compose-Datei zuständig, die in einer Linux- oder Windows-Subsystem für Linux (WSL)-Umgebung ausgeführt wird. Der Befehl zum Erstellen der Docker-Images lautet:

```
sudo docker compose -f '/path/to/compose/' build
```

Code Listing 30: Build Befehl zur Erstellung von Docker-Images aus einer Compose-Datei

```
version: '3.3'
services:
    flask-backend:
        build:
            context: ../BA-Backend/
            network: host
    flask-services:
        build:
            context: ../BA-Services/
            network: host
    frontend:
        build:
            context: ../BA-Frontend/
            network: host
```

Code Listing 31: Docker-Compose-Datei zum Bauen der Docker-Images

Nach Ausführung der Compose-Datei sind drei Images in der Docker-Umgebung verfügbar, die zum Bau der Container genutzt werden.

Der Auszug aus der `docker-compose.core.yml`-Datei in Listing 32 zeigt die Bauanleitung für einen Container der MySQL-Datenbank. Innerhalb dieser Compose-Datei wird zunächst der Name des Services definiert. Unter diesem Namen ist der Docker-Container im Netzwerk von anderen Docker-Containern erreichbar. Innerhalb des Services stehen mehrere Optionen zur Festlegung von Eigenschaften des Containers zur Verfügung. Über `ports` können ein oder mehrere Ports freigeschaltet werden, um dem Container Zugriffe von außerhalb zu erlauben. Im Bereich `volumes` sind die Ordner oder Dateien enthalten, die persistent gespeichert werden. Bei MySQL gibt es die Möglichkeit, ein initiales SQL-Skript anzugeben, welches nur bei der ersten Erstellung des Docker-Containers ausgeführt wird. Das SQL-Skript erstellt eine Datenbank und die benötigten Tabellen. Im `image`-Bereich ist das Image zum Bauen des Containers angegeben. Dies ist entweder ein lokal

vorhandenes oder ein im Docker-Hub angebotenes Image. Der Punkt `networks` beinhaltet den Namen des Netzwerks, in dem sich der Docker-Container befindet. Das Netzwerk wird benötigt, damit unterschiedliche Container miteinander kommunizieren können. Sämtliche Container müssen dazu im gleichen Netzwerk sein.

```
services:
    mysql:
        ports:
            - '3333:3306'
        volumes:
            - mysql-conf:/etc/mysql/conf.d
            - mysql-storage:/var/lib/mysql
            - ./setup.sql:/docker-entrypoint-initdb.d/init.sql:r
        [...]
        image: mysql:latest
        networks:
            - custom-network
networks:
    custom-network:
        driver: bridge
        name: custom-network
volumes:
    mysql-storage:
    mysql-conf:
```

Code Listing 32: Docker-Compose-Datei zum Aufsetzen und Starten der Container

Der Container wird mit folgendem Befehl gebaut und gestartet:

```
sudo docker compose -f '/path/to/compose/' up -d
```

Code Listing 33: Starten eines Docker-Containers

Beendet wird der Container mit folgendem Befehl:

```
sudo docker compose -f '/path/to/compose/' down
```

Code Listing 34: Beenden eines Docker-Containers

5 Evaluation

Die Evaluation der Software wurde mit einem Mitarbeiter von CONET während der Entwicklungsphase durchgeführt. Der Prototyp wurde in einem Zeitraum von sechs Wochen implementiert. Die Durchführung eines der Code-Reviews fand in regelmäßigen Abständen statt. Während des Code-Reviews wurde der derzeitige Stand der Software präsentiert, bestehende Probleme und Schwachstellen besprochen und die weiteren Entwicklungsschritte bis zum nächsten Review geplant.

Woche 1: Recherche zu den Bereichen Microservice-Architekturen und Schnittstellen war die hauptsächliche Aufgabe zu Beginn des Projekts. Im ersten Code-Review wurde die geplante Architektur vorgestellt. Das primäre Feedback zur geplanten Architektur war, nicht nur eine allgemeingültige Schnittstelle zu entwickeln, sondern die Implementierung einer Textähnlichkeitssuche in den Vordergrund zu stellen. Anhand dieser KI soll die Funktionalität der Schnittstelle gezeigt werden.

Woche 2: Das Konzept für die Implementierung der Schnittstelle wurde nach einer Anpassung der geplanten Architektur entworfen. Dieses Konzept umfasst ein Architekturdiagramm, welches die Komponenten innerhalb der Software darstellt, ein Prozessablaufdiagramm, in dem die Funktionsweise und Abläufe der Schnittelle visualisiert sind und Mockups für die zu implementierende Website. Die ursprünglich geplante Kommunikation von dem Service zur API sollte ebenfalls mit RabbitMQ umgesetzt werden, statt den API-Endpunkt des Backendes zu nutzen. Dies vereinheitlicht den Programmcode und beschleunigt die Antwortzeit der KI-Services.

Woche 3: Nach Abschluss der dritten Woche, war ein Großteil der REST-API implementiert. Innerhalb dieses Code-Reviews wurde auf den Aufbau der Routen, sowie die Anbindung an die MySQL-Datenbank und den Redis-Cache geachtet. Hierzu gab es seitens CONET keine Kritik oder Verbesserungsvorschläge.

Woche 4: Während der vierten Woche wurde die API fertiggestellt und anschließend im Code-Review vorgestellt. Besonderer Fokus lag dabei auf der Implementierung des Errormanagement- und Loggingsystems. Innerhalb dieses Reviews wurde auch Grafana zur Visualisierung der Logs vorgestellt. Ein Kritikpunkt an der gewählten Kombination aus einer MySQL-Datenbank und Grafana war, dass das System bei der Anzeige einer größeren Anzahl an Logs langsam wird. Die Anzahl der angezeigten Logs in Grafana musste demnach beschränkt werden. Des Weiteren benötigt das System zum Error-Handling für den Einsatz in einer Produktivumgebung eine genauere Aufteilung der Error-Codes. Im aktuellen Zustand des Prototyps wird bei ei-

ner fehlerhaften Eingabe der Error-Code 500 zurückgegeben. Dies ist für den realen Einsatz der Anwendung allerdings nicht ausreichend.

Woche 5: In der fünften Woche wurde das Frontend entwickelt. Dabei wurde sich an den Mockups zur visuellen Gestaltung orientiert. Die Nutzung der Website war laut CONET erst mit einer kleinen Einführung verständlich. Die Website, für sich alleinstehend, bietet zu wenig Erklärungen oder Hilfestellungen, als das sie ein Unbeteiligter nutzen könnte. Die Funktionalität der Website ist jedoch gewährleistet und deckt die in den Anforderungen erhobenen Punkte ab.

Woche 6: In der letzten Woche der Entwicklungsphase wurde der KI-Service zur Textähnlichkeitssuche implementiert. Zur Kommunikation zum Backend wurde der Service an den Dienst RabbitMQ angeschlossen. Anschließend wurden die Teile der Software über Docker in einzelne Container verpackt. Dadurch konnte die Softwarearchitektur auf den Rechnern mehrerer CONET-Mitarbeiter installiert und getestet werden.

Nach dem Abschluss der Code-Reviews wurde der entwickelte Prototyp vor mehreren Mitarbeitern von CONET präsentiert. Im Anschluss an die Präsentation wurde das Ergebnis der Bachelorarbeit mit den zu Beginn erhobenen Anforderungen verglichen. Grundsätzlich erfüllt der Prototyp sämtliche Anforderungen. Die entworfene Architektur wurde für modernen, performant und skalierbar befunden. Sie bietet eine gute Grundlage, um komplexere Systeme darauf aufbauen zu können.

Es gibt jedoch, wie auch in den Code-Reviews angemerkt wurde, einige Bereiche, bei denen es Verbesserungspotential gibt. Diese betreffen jedoch ausschließlich die visuelle Aufbereitung und Nutzerinteraktion mit dem System. Die angemerkten Kritikpunkte sind folgende:

- Die Nutzung der Website ist nicht intuitiv für neue Nutzer
- Grafana wird mit MySQL zu langsam für eine große Anzahl an Logs
- Der Nutzer kann falsche Eingaben auf der Website tätigen, ohne darüber in Kenntnis gesetzt zu werden
- Die Installation der Software ist relativ zeitaufwändig, wenn Docker nicht bereits installiert ist
- Die Elasticsearch-Datenbank im KI-Service kann nicht mit weiteren Daten befüllt werden

6 Fazit und Ausblick

6.1 Fazit

Für diese Bachelorarbeit wurde eine Architektur für eine Schnittstelle zur Anbindung von Daten an KI-Algorithmen mithilfe des Wasserfallmodells entwickelt. Der Prototyp besteht aus drei Hauptkomponenten.

Die erste Komponente ist das mit Angular umgesetzte Frontend. Das Frontend dient als Benutzerschnittstelle für die Eingabe von Anfragen an eine KI. Des Weiteren dient das Frontend zur Eingabe einer Transformationsanleitung, mit der die Eingabe automatisiert auf das von der KI benötigte Format umgewandelt werden kann. Der letzte Einsatzzweck des Frontends ist die Auswahl des KI-Services und der Anzeige der durch die KI produzierten Ergebnisse. Für die Entwicklung einer Website wurde ein Konzept gezeigt und implementiert, welches einen sequenziellen Prozess abbilden kann. Die Website unterstützt das dynamische Hinzufügen und Entfernen von KI-Services, ohne dass der Code der Website angepasst werden muss. Die Website implementiert ebenfalls ein System zum asynchronen Laden von Ergebnissen aus den KI-Services. Während der Entwicklung wurde auf Modularität und Wiederverwertbarkeit von Code geachtet, wodurch, das das System auch für Anfragen genutzt werden kann, die keine KI als Ziel haben.

Die zweite Komponente ist die in Python mit dem Package Flask entwickelte REST-API. Innerhalb der API wurden mehrere Routen definiert, die zur Interaktion mit dem Backend genutzt werden können. Für das Backend wurde im Rahmen der Bachelorarbeit ein Konzept entworfen, Fehler zur Laufzeit abzufangen und behandeln zu können. Innerhalb der API wurde ein System zum Loggen von Systemereignissen konzipiert. Die API wurde nach dem Grundsatz einer Microservice-Architektur entworfen, die eine horizontale Skalierung ermöglicht. Sollte es bei der Nutzung des Systems zu einer hohen Auslastung durch eine große Anzahl an Nutzern kommen, so können beliebig viele Instanzen der API dazugeschaltet werden.

Die dritte Komponente ist die mit RabbitMQ realisierte Kommunikation zu den KI-Services. RabbitMQ unterstützt die entworfene Microservice-Architektur als Nachrichtenaustauschdienst zwischen dem Backend und den Services. Dieser Dienst ermöglicht ist eine asynchrone Kommunikation zwischen den beiden Kommunikationspartnern. RabbitMQ erweitert die im Backend implementierte Skalierbarkeit. Eine horizontale Skalierung der KI-Services ist dadurch erreicht worden.

Die entwickelte Schnittstelle zur Anbindung von austauschbaren Datenquellen an KI-Algorithmen wurde entworfen und in eine praxistaugliche Architektur integriert, die für den Einsatz in einer Produktivumgebung ausgelegt ist.

6.2 Ausblick

Die Bachelorarbeit stellt die Grundlage für die Umsetzung der Projektidee von CONET dar, ein Trendscouting-System zu entwickeln. Dieses System soll zunächst mehrere Datenquellen, wie wissenschaftliche Paper, Auszüge aus Büchern, Webseiten und Nachrichtenfeeds sammeln. Die gesammelten Daten sollen dazu genutzt werden, erkennen zu können, welche Themen im Bereich der IT in den nächsten Jahren relevant werden. Dazu benötigt es eine KI, die textbasierte Daten auf semantische Ähnlichkeit vergleicht. Die grundlegende Funktionalität dazu wurde mit der Eingabe einer Query, dessen Transformation über eine weitere Eingabe im Frontend und die KI-basierte Auswertung bereitgestellt.

Die für den den Einsatz in einer Produktivumgebung noch zu verändernden Aspekte der Software wurden in einem Review des Prototyps mit mehreren Mitarbeitern von CONET festgehalten. Im Prototyp ist das System zur Nutzung mehrerer Anwender nicht implementiert. Im Backend ist die Funktionalität zur Unterscheidung von Nutzern durch eine UUID, die über JWT transportiert wird, bereits implementiert. Im Frontend muss ein System zur Nutzung einer UUID noch implementiert werden, bevor die Architektur in einer Produktivumgebung deployed werden kann.

Im Review wurde für die künftige Weiterentwicklung wurden Erweiterung der Funktionalität mehrfach erwünscht. Für den Prototyp ist die Eingabe einer Query durch den Nutzer ausreichend. Ein etwaiges Trendscouting-System benötigt ein automatisiertes System zur Eingabe von Queries. Das System kann um die Möglichkeit der Eingabe vollständiger Dokumente erweitert werden. Die Dokumente sollten sowohl für die Anfrage an die KI nutzbar sein als auch den Datenbestand der Elasticsearch erweitern.

Der Output der im Prototyp implementierten KI beinhaltet einen Titel und eine Beschreibung. Der Output lässt sich dahingehend erweitern, dass nicht nur textbasierte Ergebnisse angezeigt werden können, sondern auch ganze Dokumente in die Ergebnisliste geladen werden können. Es sollte ebenfalls möglich sein, diese Dokumente anschließend herunterzuladen.

Mit dem Prototyp wurde eine Grundlage für die modulare Anbindung von Daten an KI-Algorithmen geschaffen. Die Softwarearchitektur ermöglicht die Weiterentwicklung für viele Anwendungszwecke.

7 Literaturverzeichnis

ADENOWO, A.A.A.; ADENOWO, B.A., 2013. Software engineering methodologies: a review of the waterfall model and object-oriented approach. *International Journal of Scientific & Engineering Research*. Jg. 4, Nr. 7, S. 427–434.

ANDERSON, C., 2015. Docker [software engineering]. *Ieee Software*. Jg. 32, Nr. 3, S. 102–c3.

BLOCH, J., 2006. How to design a good API and why it matters. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, S. 506–507.

CHAKRABORTY, M.; KUNDAN, A.P., 2021. Grafana. In: *Monitoring Cloud-Native Applications*. Springer, S. 187–240.

CONET, 2022. *Quick Guide für Studierende*. (internes Dokument).

CONET, [o. D.]. *Auszug aus der Kundenliste* [online]. [besucht am 2022-05-11]. Abger. unter: `https://www.conet.de/DE/conet-group/kunden/auszug-aus-der-kundenliste`.

DANIEL, R., 2018. *Schlagwort: Software-Anforderungen IEC 62304 konform dokumentieren* [online]. [besucht am 2022-12-31]. Abger. unter: `https://www.johner-institut.de/blog/tag/software-anforderungen/`.

DOGLIO, F., 2015. *Pro REST API Development with Node. js*. Apress.

DOSSOT, D., 2014. *RabbitMQ essentials*. Packt Publishing Ltd.

DUBOIS, P., 2008. *MySQL*. Pearson Education.

EDUNITAS, 2015. Waterfall model. *Luettavissa: http://www.waterfall-model.com/. Luettu*. Jg. 3.

ENDRES, A., 2000. „Open Source“ und die Zukunft der Software. *Informatik-Spektrum*. Jg. 23, Nr. 5, S. 316–321.

FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T., 1999. *RFC2616: Hypertext Transfer Protocol–HTTP/1.1*. RFC Editor.

FIELDING, R.T., 2000. *Architectural Styles and the Design of Network-based Software Architectures – Dissertation* [online]. [besucht am 2022-12-21]. Abger. unter: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf`.

FRAUCHIGER, D., 2017. Anwendungen von Design Science Research in der Praxis. In: *Wirtschaftsinformatik in Theorie und Praxis*. Wiesbaden: Springer Fachmedien Wiesbaden, S. 107–118.

GÖRZ, G.; SCHNEEBERGER, J., 2010. *Handbuch der künstlichen Intelligenz*. Walter de Gruyter.

GRINBERG, M., 2018. *Flask web development: developing web applications with python*. O‘Reilly Media, Inc.

HAMET, P.; TREMBLAY, J., 2017. Artificial intelligence in medicine. *Metabolism*. Jg. 69, S36–S40.

HARWARDT, Mark; STEIMANN, Friedrich, 2012. Wasserfallmodell versus Scrum. In: *Gesellschaft für Informatik eV (GI) publishes this series in order to make available to a broad public recent findings in informatics (ie computer science and informa-tion systems), to document conferences that are organized in co-operation with GI and to publish the annual GI Award dissertation.* S. 235–237.

HOQUE, N.; BHATTACHARYYA, D.; KALITA, J., 2015. Botnet in DDoS attacks: trends and challenges. *IEEE Communications Surveys & Tutorials*. Jg. 17, Nr. 4, S. 2242–2270.

IONESCU, V.M., 2015. The analysis of the performance of RabbitMQ and ActiveMQ. In: *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*. IEEE, S. 132–137.

JOHANSSON, L.; DOSSOT, D., 2020. *RabbitMQ Essentials*. RabbitMQ Essentials: Build Distributed and Scalable Applications with Message Queuing Using RabbitMQ. Birmingham: Packt Publishing, Limited. ISBN 9781789131666.

JONES, M.; BRADLEY, J.; SAKIMURA, N., 2015. *Json web token (jwt)*. Techn. Ber.

JOSHEPH, T., 2021. Python. *Python Releases for Windows*. Jg. 24.

MASSE, M., 2011. *REST API design rulebook: designing consistent RESTful web service interfaces*. O'Reilly Media, Inc.

MDN, 2022. *HTTP request methods* [online]. [besucht am 2022-12-19]. Abger. unter: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.

MOISEEV, A.; FAIN, Y., 2018. *Angular Development with TypeScript*. Simon und Schuster.

NEWMAN, S., 2015. *Microservices: Konzeption und design*. MITP-Verlags GmbH & Co. KG.

PAKSULA, M., 2010. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science*. Jg. 27.

PORATH, R., 2020. *Internet, Cyber- und IT-Sicherheit Von A-Z*. Internet, Cyber- und IT-Sicherheit Von A-Z: Aktuelle Begriffe Kurz und Einfach Erklärt - Für Beruf, Studium und Privatleben. 2. Aufl. 2020. Berlin, Heidelberg: Springer Berlin / Heidelberg. ISBN 366260910X.

RAHUTOMO, F.; KITASUKA, T.; ARITSUGI, M., 2012. Semantic cosine similarity. In: *The 7th international student conference on advanced science and technology ICAST*. Bd. 4, S. 1. Nr. 1.

RICHARDS, R., 2006. Representational state transfer (rest). In: *Pro PHP XML and web services*. Springer, S. 633–672.

RIVERO, J.M.; ROSSI, G.; GRIGERA, J.; BURELLA, J.; LUNA, E.R.; GORDILLO, S., 2010. From mockups to user interface models: an extensible model driven approach. In: *International Conference on Web Engineering*. Springer, S. 13–24.

SNEED, H.M., 2006. Integrating legacy software into a service oriented architecture. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 11–pp.

SNYDER, B., 2000. *Music and memory: An introduction*. (MIT press).

STACKOVERFLOW, 2022. *How do I ask a good question?* [online]. [besucht am 2022-12-15]. Abger. unter: `https://stackoverflow.com/help/how-to-ask`.

STATISTA, 2021. *Volumen der jährlich generierten digitalen Datenmenge* [online]. [besucht am 2023-01-05]. Abger. unter: `https://de.statista.com/statistik/daten/studie/267974/umfrage/prognose-zum-weltweit-generierten-datenvolumen/`.

TAIVALSAARI, A.; MIKKONEN, T.; PAUTASSO, C.; SYSTÄ, K., 2021. Full Stack Is Not What It Used to Be. In: *International Conference on Web Engineering*. Springer, S. 363–371.

TIBSHIRANI, J., 2019. *Textähnlichkeitssuche mit Vektorfeldern* [online]. [besucht am 2023-01-05]. Abger. unter: `https://www.elastic.co/de/blog/text-similarity-search-with-vectors-in-elasticsearch`.

TIBSHIRANI, J.; OAK, S., 2020. *Text Embeddings in Elasticsearch* [online]. [besucht am 2023-01-05]. Abger. unter: `https://github.com/jtibshirani/text-embeddings`.

WIRDEMANN, R., 2022. *Scrum mit user stories*. Carl Hanser Verlag GmbH Co KG.

WOLFF, E., 2018. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag.