HAW HAMBURG

BACHELOR THESIS
Marius Fischmann

Ermittlung eines geeigneten Kommunikationsmusters zwischen Microservices einer Datengenerierungsplattform für künstliche Intelligenz

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Marius Fischmann

Ermittlung eines geeigneten Kommunikationsmusters zwischen Microservices einer Datengenerierungsplattform für künstliche Intelligenz

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 17.08.2023

Marius Fischmann

Thema der Arbeit

Ermittlung eines geeigneten Kommunikationsmusters zwischen Microservices einer Datengenerierungsplattform für künstliche Intelligenz

Stichworte

Inter-Prozess Kommunikation, Kommunikationsmuster, Microservice Architektur, Künstliche Intelligenz, Datengenerierung, Synchrone Kommunikation, Asynchrone Kommunikation

Kurzzusammenfassung

Durch die wachsende Anwendung von künstlicher Intelligenz, in den verschiedensten Bereichen, besteht immer mehr Bedarf für Daten zum Trainieren von Maschine Learning Modellen. Diese Daten sind jedoch häufig nicht oder nur in geringen Mengen vorhanden. DaFne soll eine Plattform bieten, um dieses Problem zu lösen und Möglichkeiten bereitzustellen, solche Daten für die verschiedensten Bereiche zu generieren. Für die Umsetzung dieser Web-Plattform wurde sich für eine Microservice Architektur entschieden. Um diese Architektur zu implementieren, bedarf es eines geeigneten Kommunikationsmusters, für die Kommunikation der einzelnen Services miteinander. In dieser Arbeit wird ein geeignetes Muster ermittelt und Technologien vorgeschlagen, um dieses Muster zu implementieren.

Marius Fischmann

Title of Thesis

Identification of a suitable communication pattern between microservices of a data generation platform for artificial intelligence

Keywords

Inter-Process Communication, Communication Pattern, Microservice Architecture, Artificial Intelligence, Data Generation, Synchronious Communication, Aynchronious Communication

Abstract

Due to the growing application of artificial intelligence, in various fields, there is an increasing need for data to train machine learning models. However, this data is often not available or only available in small amounts. DaFne aims to provide a platform to solve this problem and to provide possibilities to generate such data for various domains. For the implementation of this web platform a microservice architecture was chosen. In order to implement this architecture, a suitable communication pattern is required for the communication of the individual services with each other. In this thesis, a suitable pattern is identified and technologies are proposed to implement this pattern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Allgemeine Einleitung und Einordnung der Arbeit

Im Rahmen des Forschungsprojektes 'Data Fusion Generator für die Künstliche Intelligenz', an der Hochschule für Angewandte Wissenschaften Hamburg (HAW), wird derzeit an der Entwicklung einer Plattform zur Generierung von Daten gearbeitet. Das Projekt befindet sich noch in einer frühen Phase und ist noch nicht abgeschlossen. Die Architektur der Plattform soll auf dem Konzept der Microservices basieren. Hierbei werden die einzelnen Services als eigenständige Container mithilfe von Docker realisiert. Die Verwendung von Containern ermöglicht eine flexible und skalierbare Bereitstellung der Services.

Ein wesentlicher Aspekt bei der Implementierung einer Microservice-Architektur besteht darin, ein geeignetes Kommunikationsmuster zu finden. Im Gegensatz zu monolithischen Anwendungen, bei denen die Kommunikation zwischen den Komponenten hauptsächlich durch Funktionsaufrufe innerhalb der Anwendung erfolgt, müssen bei einer Microservice-Architektur Inter-Process-Communication (IPC)-Mechanismen verwendet werden. Die Wahl des richtigen Kommunikationsmusters ist entscheidend für die Effizienz und Skalierbarkeit der Plattform. Es müssen Mechanismen gefunden werden, die eine effektive Kommunikation zwischen den verschiedenen Services ermöglichen, gleichzeitig aber auch die Flexibilität und Unabhängigkeit der Services erhalten. Die Identifizierung und Implementierung eines geeigneten Kommunikationsmusters ist daher ein wichtiger Schritt in der Entwicklung der DaFne-Plattform und soll mit dieser Arbeit unterstützt werden.

1.2 Problemstellung und Motivation

Eine der Herausforderungen, bei der Verwendung der Microservice-Architektur, besteht in der Kommunikation der einzelnen Komponenten. Um die Vorteile der Microservices-Architektur vollständig zu nutzen, ist es entscheidend, dass die Kommunikation zwischen

den Services effektiv und problemlos erfolgt. Eine fehlerhafte Kommunikation kann zu einer Reihe von Problemen führen, wie z.B. verzögerten Reaktionen, Dateninkonsistenzen und Performance-Einbußen. Daher ist es von großer Bedeutung, geeignete Muster für die Kommunikation der Services zu etablieren, um mögliche Probleme zu vermeiden.

Das Design eines angemessenen Kommunikationsmusters für Microservices hat verschiedene Ziele. Erstens sollte es die Interaktion zwischen den Services erleichtern und sicherstellen, dass die Kommunikation zuverlässig und effizient ist. Zweitens sollte das Muster die Skalierbarkeit der Architektur unterstützen, indem es die Anpassung an die wachsenden Anforderungen und Lasten ermöglicht. Darüber hinaus sollte es die Flexibilität der Architektur fördern, sodass neue Services hinzugefügt oder bestehende Services aktualisiert werden können, ohne die Kommunikation grundlegend zu beeinträchtigen. Um diese Ziele zu erreichen, soll in dieser Arbeit ein geeignetes Kommunikationsmuster für die DaFne-Plattform ermittelt werden.

1.3 Zielsetzung

Im Laufe dieser Arbeit sollen Herausforderungen und Anforderungen der DaFne-Plattform und der Kommunikation zwischen den Microservices identifiziert werden. Um ein geeignetes Kommunikationsmuster zu ermitteln, müssen die dazu nötigen Konzepte überblickt und mögliche Technologien vorgestellt werden. Unter Berücksichtigung dieser Herausforderungen und Anforderungen, soll im Anschluss ein geeignetes Kommunikationsmuster für die Plattform ermittelt werden und zusätzlich eine Empfehlung für geeignete Technologien zur Umsetzung dieses Musters gegeben werden.

1.4 Aufbau der Arbeit

Diese Arbeit beschäftigt sich mit der Betrachtung der DaFne-Architektur, insbesondere im Hinblick auf die Microservices und deren Kommunikation. Zunächst werden grundlegende Konzepte und Prinzipien der Plattform erläutert. Anschließend erfolgt eine Anforderungsanalyse der DaFne-Architektur, wobei insbesondere die Services und deren Kommunikation betrachtet werden. Darauf aufbauend werden im Anschluss ein geeignetes Kommunikationsmuster und Technologien ausgewählt, welche dieses Muster implementieren können. Eine mögliche Lösung für die Kommunikation, wird dann in Bezug auf

die Anforderungen aus dem dritten Kapitel diskutiert. Abschließend werden die Ergebnisse kurz zusammengefasst und ein Ausblick auf die zukünftige Weiterentwicklung der Architektur gegeben. Dabei werden potenzielle Schwierigkeiten und Herausforderungen für die Zukunft identifiziert.

2 Theoretische Grundlagen

Im folgenden Kapitel wird das DaFne-Forschungsprojekt ausführlich vorgestellt, um den Rahmen der Arbeit festzulegen. Es wird außerdem eine Übersicht über die Microservice-Architektur und die Inter-Process Communication (IPC) gegeben.

2.1 Data Fusion Generator für die Künstliche Intelligenz

In den letzten Jahren hat das maschinelle Lernen große Fortschritte gemacht und viel Aufmerksamkeit gewonnen. Allerdings sind die hochwertigen Daten, die zum Trainieren der jeweiligen Machine Learning (ML) Modelle nötig sind, oft nicht in ausreichender Menge vorhanden [vgl. 8]. Die Verwendung von synthetischen Daten, anstelle von oder zusätzlich zu realen Daten, hat das Potenzial, dieses Problem zu überwinden. Diese Generierung ist durch verschiedene Methoden zu erreichen. Um einige dieser Methoden bereitzustellen, soll die DaFne-Plattform dienen. Die Plattform kann daher die Forschung und Entwicklung von Methoden der künstlichen Intelligenz weiter fördern und zur Verbesserung der Genauigkeit von Modellen des maschinellen Lernens beitragen.[vgl. 28]

DaFne soll eine Web-Plattform sein, um einen Bereich zu bieten in dem verschiedene Datengenerierungs- und Evaluationsmethoden gesammelt werden. Diese Plattform soll als Microservice-Architektur mit Docker umgesetzt werden. Es sollen vorerst drei verschiedene Möglichkeiten bestehen Daten zu generieren. Zudem soll es Möglichkeiten geben eigene, sowie öffentliche Daten zu verwenden und geeignet zu erweitern. Für die generierten Daten sollen verschiedene Evaluationsmethoden vorhanden sein. Es sollen zudem verschiedene Funktionalitäten exklusiv für verschiedene Nutzergruppen verfügbar sein. Dies soll dafür sorgen, dass sowohl erfahrene Benutzer, aber auch weniger erfahrene Benutzer die Plattform verwenden können. Im Folgenden wird im Detail auf diese Funktionalitäten eingegangen.

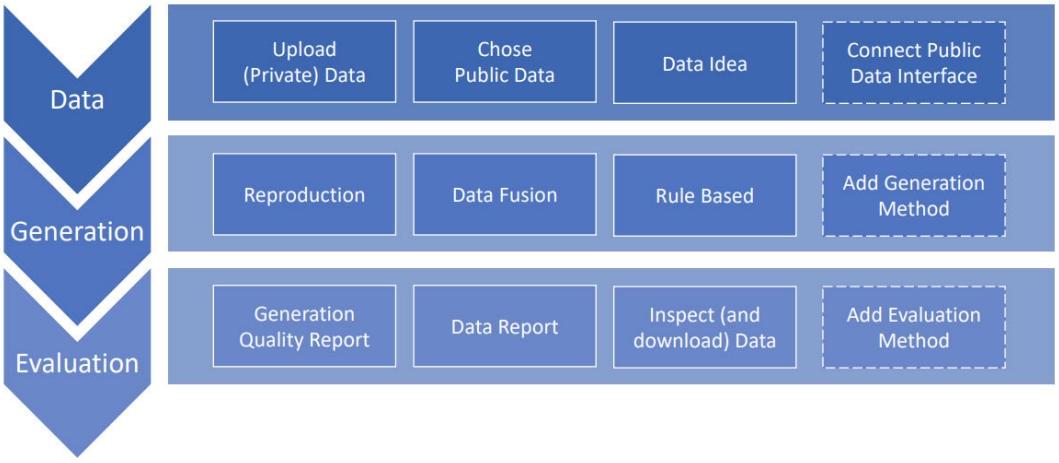

Abbildung 2.1: Übersicht der Funktionalität der DaFne-Plattform [vgl. 24]

Es wird möglich sein, eigene Daten unter dem eigenen Account auf der Plattform hochzuladen und zu speichern. Diese Daten sollen geschützt von anderen Benutzern sein und können angepasst, für das Training von ML-Modellen und zur Generierung von Daten verwendet werden. Zudem soll es eine große Bandbreite an öffentlichen Daten geben, welche zusätzlich zu eigenen Daten für diese Zwecke verwendbar sein sollen.
Grundlegend soll eine Interaktion des Benutzers aus drei Schritten bestehen. Zuerst wird festgelegt, auf welcher Datengrundlage die Generierung erfolgen soll, dies könnte die Auswahl eines öffentlichen Datensatzes sein oder auch beispielsweise der Upload von eigenen privaten Daten. Darauf folgt das Anstoßen eines Generierungsprozesses, dieser kann durch eine der verfügbaren Funktionen realisiert werden. Im letzten Schritt wird die Evaluation der generierten Daten vorgenommen, mit einer oder mehreren verfügbaren Evaluationsmethoden. Abbildung 2.1 zeigt eine Übersicht der geplanten Funktionalitäten in den Bereichen Daten, Generierung und Evaluation. [vgl. 24, S. 4]

Vorerst sollen drei Möglichkeiten zur Datengenerierung bereitgestellt werden. Der Fokus liegt hierbei auf der Erweiterung von bereits bestehenden und der Generierung von neuen tabellarischen Daten. Im Folgenden werden diese drei Möglichkeiten etwas detaillierter beschrieben und in Abbildung 2.2 illustriert.

(a) **Rule-based Data Generation:** Bei der regelbasierten Datengenerierung können Benutzer ihre eigenen Spalten mit bestimmten Datentypen, Verteilungen und Merk-

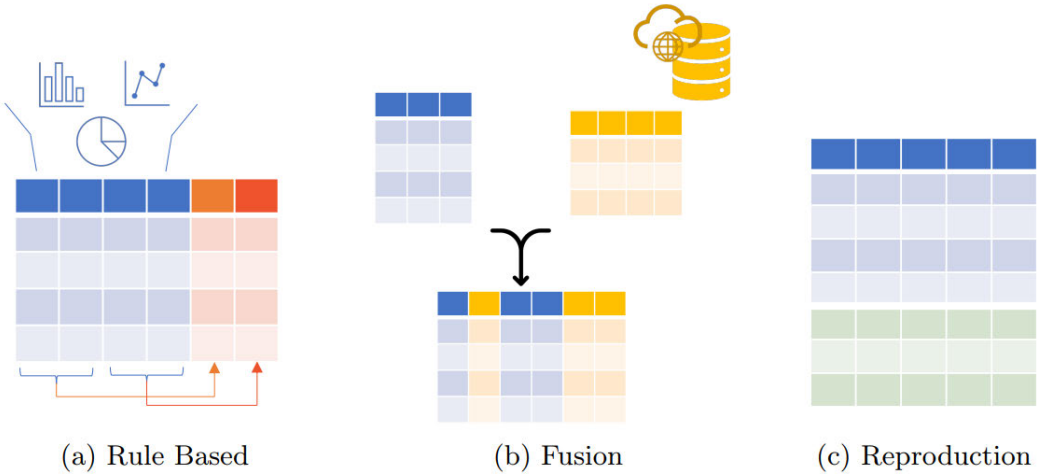

Abbildung 2.2: Varianten der Datengenerierung auf der DaFne-Plattform [vgl. 24]

malsausprägungen definieren, sowie Spalten mit Merkmalen erzeugen, die bestimmten Regeln folgen. Auf diese Weise können tabellarische Daten generiert werden, ohne auf bestehende Daten zurückzugreifen. Darüber hinaus sollte es dem Nutzer möglich sein, diese regelbasierte Generierung von Spalten auf bestehende Spalten anzuwenden, um einen bestehenden Datensatz (spaltenweise) zu erweitern. [vgl. 24, S. 4]

(b) **Data Fusion:** Bei der Datenfusion sollen mehrere Datensätze miteinander verbunden werden. Dazu wäre es vor allem denkbar, den Datensatz eines Nutzers mit öffentlich verfügbaren Daten zu kombinieren. Beispielsweise könnten Wetterdaten oder Daten über Verkehrsströme an einen privaten Datensatz angehängt werden, um den Informationsgehalt des privaten Datensatzes zu erweitern. [vgl. 24, S. 4-5]

(c) **Reproduction:** Bei der Reproduktion sollen auf der Grundlage existierender Daten neue Zeilen erstellt werden, um einen Datensatz zu erweitern. Die generierten Zeilen sollen hierbei den bereits bestehenden Zeilen des Datensatzes in ihren Eigenschaften und Spalten übergreifenden Beziehungen entsprechen. Um dies zu gewährleisten, soll eine große Bandbreite an Evaluationsmethoden auf der Plattform bereitgestellt werden. [vgl. 24, S. 5]

2.2 Microservice Architektur

In der Vergangenheit wurden die meisten Softwareanwendungen als Monolith entwickelt. Hierbei sind alle Funktionen und Komponenten in einer einzigen Anwendung zusammengefasst. Dies bedeutet, dass jede Änderung oder Aktualisierung an der Anwendung den gesamten Code beeinflussen kann, was zu einer erhöhten Komplexität und Schwierigkeiten bei der Wartung führen kann. Diese Art der Architektur ist zudem schwer skalierbar und erweiterbar. [vgl. 11, S. 1-7] Dies sind grundlegende Anforderungen an viele heutige Softwareanwendungen. Für die DaFne-Plattform wurde sich aufgrund der benötigten Anpassbarkeit, Flexibilität, Skalierbarkeit und Erweiterbarkeit gegen eine monolithische Architektur entschieden. Der für die DaFne-Plattform gewählte Ansatz ist die Microservice-Architektur.

Bei der Microservices Architektur wird, für die Entwicklung von Softwareanwendungen, eine Anwendung in kleinere, eigenständige Services oder Komponenten aufgeteilt, die jeweils eine spezifische Funktion erfüllen. Jeder Dienst wird als separate Einheit entwickelt, bereitgestellt und betrieben. [vgl. 25, S. 27] Diese Einheiten werden als Microservices bezeichnet.

2.2.1 Microservices

Wie bereits beschrieben, sind Microservices voneinander unabhängig bereitgestellte Einheiten. Diese sind in der Regel eng gekapselt und haben eine klare Schnittstelle, über die sie mit anderen Services und Anwendungen kommunizieren. Dadurch können Entwickler und Teams unabhängig voneinander an verschiedenen Services arbeiten und Änderungen schneller und sicherer bereitstellen. [vgl. 22, S. 15]

Ein wichtiger Aspekt von Microservices, ist ihre Skalierbarkeit. Da jeder Service unabhängig bereitgestellt wird, kann jeder Service entsprechend seinen spezifischen Anforderungen skaliert werden, um eine höhere Last oder ein höheres Datenvolumen zu bewältigen. Dies bedeutet, dass Ressourcen effizienter genutzt werden können, da nur die Services skaliert werden, die es benötigen. [vgl. 25, S. 27]

Allerdings gibt es auch einige Herausforderungen bei der Verwendung von Microservices. Eine wichtige Herausforderung besteht darin, die Komplexität der Integration zwischen den Services zu managen. Die Kommunikation zwischen den Services muss gut gestaltet

und dokumentiert werden, um sicherzustellen, dass die Services effektiv zusammenarbeiten. Auch das Management und Monitoring von vielen einzelnen Services kann herausfordernd sein. [vgl. 25, S. 32]

2.3 Kommunikation zwischen Microservices

2.3.1 Von In-Prozess zu Inter-Prozess Kommunikation

In-Prozess Kommunikation bezieht sich auf die Kommunikation zwischen verschiedenen Komponenten innerhalb einer monolithischen Anwendung. Dies wird durch direkte Methodenaufrufe, gemeinsame Speicherbereiche oder andere interne Mechanismen erreicht. Häufig ist dieses Vorgehen realisiert, indem ein Objekt ein anderes aufruft.

Im Gegensatz dazu sind Microservices eigenständige Anwendungen, die unabhängig voneinander laufen und miteinander kommunizieren müssen. Diese Kommunikation zwischen verschiedenen Microservices wird als Inter-Process Kommunikation (IPC) bezeichnet und kann über verschiedene Mechanismen erfolgen, wie z.B. RESTful APIs, Messaging-Protokolle oder gemeinsam genutzte Datenbanken. [vgl. 23, S. 89-117] Auf diese Mechanismen wird im Folgenden weiter eingegangen.

2.3.2 Synchrone Kommunikation

Sam Newman definiert die synchrone Kommunikation zwischen Microservices in [vgl. 23, S. 95-97] wie folgt. Bei einem synchronen blockierenden Aufruf sendet ein Microservice einen Aufruf irgendeiner Art an einen nachgelagerten Prozess (wahrscheinlich einen anderen Microservice) und blockiert, bis der Aufruf abgeschlossen ist und möglicherweise bis eine Antwort empfangen wurde. Diese Art der Kommunikation verursacht eine enge Kopplung zwischen den Microservices, da sie direkt miteinander kommunizieren müssen. Ein Vorteil der synchronen Kommunikation besteht darin, dass sie einfacher zu verstehen und zu implementieren ist. Dabei wird häufig HTTP als RESTful API verwendet. Jedoch kann die synchrone Kommunikation auch zu längeren Antwortzeiten und Engpässen führen, insbesondere wenn viele Anfragen gleichzeitig an einen Service gestellt werden. [vgl. 17, S. 6]

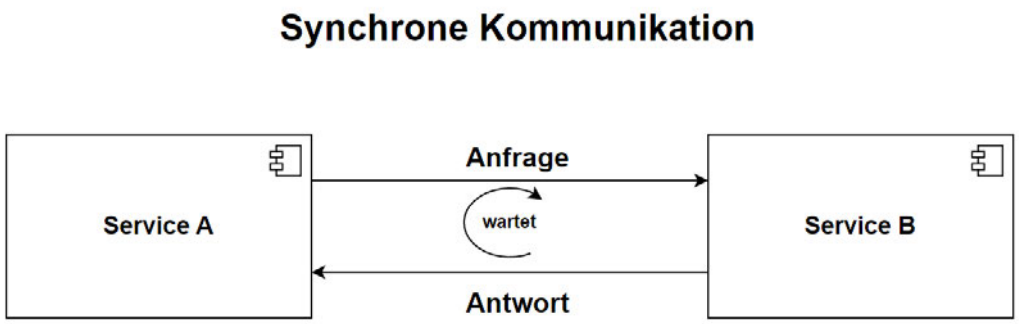

Abbildung 2.3: Beispielhaftes Schaubild der synchronen Kommunikation [1]

2.3.3 Asynchrone Kommunikation

Die asynchrone Kommunikation von Microservices wird von Newman in [vgl. 23, S. 98-100] wie folgt definiert. Bei einem asynchronen nicht-blockierenden Aufruf blockiert der Microservice, welcher den Aufruf tätigt, nicht. Dieser Microservice kann in der Zwischenzeit andere Aufgaben bearbeiten, ohne auf die Antwort seines Aufrufs zu warten. Die Kommunikation erfolgt normalerweise über Messaging-Systeme wie RabbitMQ oder Apache Kafka. Die Nachrichten werden in der Regel in einem Nachrichtenpuffer (Message Broker) zwischengespeichert, bis sie vom Empfänger abgerufen werden. Asynchrone Kommunikation kann dazu beitragen, die Skalierbarkeit und Flexibilität eines Systems zu verbessern, da die Microservices unabhängig voneinander arbeiten und keine direkte Kopplung zwischen ihnen besteht. Die Services müssen lediglich mit dem Broker interagieren. [vgl. 17, S. 6]

2.3.4 Gemeinsame Daten

In [vgl. 23, S. 101-104] wurde eine weitere Möglichkeit erwähnt, welche auch dazu verwendet werden kann, die direkte Kommunikation zwischen einzelnen Microservices komplett zu ersetzen, um sich dabei den Aufwand zu sparen. Newman beschreibt dieses Konzept in [vgl. 23, S. 101-104] wie folgt: Es ist eine Art der Kommunikation, die sich über eine Vielzahl von Implementierungen erstreckt. Dieses Muster wird verwendet, wenn ein Microservice Daten an einem bestimmten Ort ablegt und ein anderer Microservice (oder möglicherweise mehrere Microservices) diese Daten dann nutzt. Das kann so einfach sein,

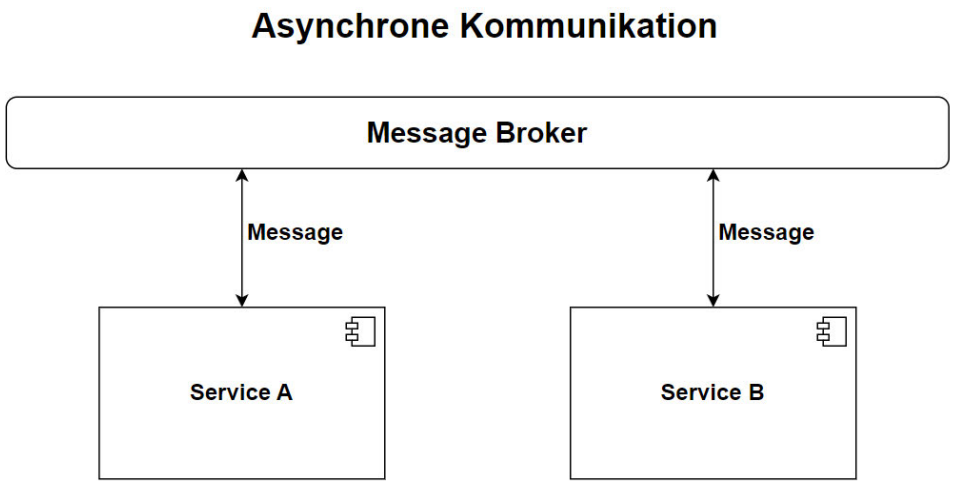

Abbildung 2.4: Beispielhaftes Schaubild der asynchronen Kommunikation, über einen Message Broker [1]

dass ein Microservice eine Datei an einem bestimmten Ort ablegt und ein anderer Microservice diese Datei zu einem späteren Zeitpunkt abholt und Daten daraus nutzt. Es wird außerdem angemerkt, dass ein großer Nutzen in diesem Pattern besteht, wenn es darum geht große Datenmengen zwischen Microservices zu teilen. Vor allem bei Dateien, die mehrere Gigabyte groß sind, ist dieses Pattern laut Newman die richtige Vorgehensweise.

Als Vorteil gilt, dass mit bekannten Technologien gearbeitet wird und beim Versenden von großen Dateien, lediglich der sendende Service blockiert, da dieser direkt mit dem Datenbanksystem interagiert. Als Nachteil wird angemerkt, dass der andere Service erst einmal merken muss, dass dieser neue Daten zum Bearbeiten hat und es deswegen möglicherweise zu einer gewissen Latenz kommen kann. Jedoch sei es unwahrscheinlich, dass bei großen Datenmengen eine kleine Latenz hoch in der Anforderungsliste stehe. [vgl. 23, S. 101-104]

Die Abbildung 2.5 zeigt eine Übersicht der Kommunikationsarten, mit Einordnung zur synchronen und asynchronen Kommunikation.

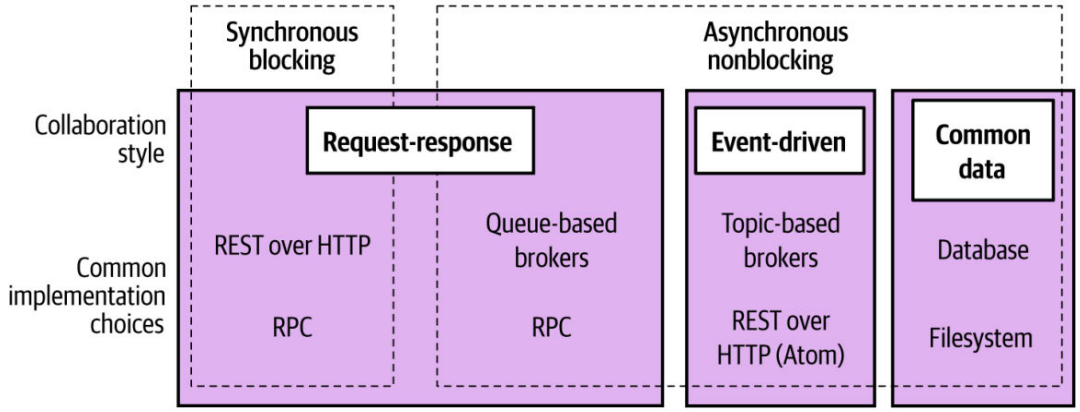

Abbildung 2.5: Übersicht der Kommunikationsarten und möglichen Umsetzungen [vgl. 23, S. 94]

2.3.5 Nachrichtenformat

Bei der IPC werden Messages zwischen den Microservices versendet. Diese Messages enthalten alle Informationen, die für die Kommunikation und für die Verrichtung der Aufgaben der Microservices wichtig sind. Um ein geeignetes Muster für die Kommunikation der Microservices zu erstellen, gilt es neben der Auswahl der IPC-Methode, ein dazu passendes Nachrichtenformat auszuwählen, da dieses ebenfalls einen großen Einfluss auf die Effizienz und Nutzbarkeit einer API haben kann. [vgl. 26, S. 71] Es gibt zwei grundlegende Kategorien, diese sind auf Text basierende und binäre Nachrichtenformate. [vgl. 26, S. 71] Bei den auf Text basierenden können Nachrichten zum Beispiel als XML oder JSON dargestellt werden. [vgl. 30, S. 178] Bei den binären Formaten bietet sich beispielsweise Protocol Buffer an. [vgl. 30, S. 178] Die verschiedenen Formate haben jeweils Vor- und Nachteile, auf diese wird im Kapitel 4 eingegangen, wenn es um die Auswahl eines Formats für die DaFne-Plattform geht.

3 Anforderungsanalyse

In diesem Kapitel geht es um die Identifikation der nicht-funktionalen Anforderungen. Außerdem werden in diesem Kapitel die geplanten Services der DaFne Plattform definiert und ermittelt, welche Services miteinander kommunizieren müssen. Als Ergebnis soll am Ende dieses Kapitels eine Übersicht entstehen, welche Services für das Kommunikationsmuster berücksichtigt werden.

3.1 Nicht-funktionale Anforderungen

Im Folgenden wird eine Übersicht über die ermittelten Anforderungen an die Plattform gegeben, sowie zu jeder Anforderung eine kurze Aussicht, wie sich diese Anforderung auf das Kommunikationsmuster auswirken kann und gegebenenfalls worauf zu achten ist.

Im Paper [24] wurden folgende allgemeine Anforderungen herausgearbeitet:

- Skalierbarkeit: Die Plattformarchitektur sollte skalierbar sein, um eine Parallelisierung von verschiedenen Prozessen zu ermöglichen, auch bei steigender Auslastung. Sie sollte auf typischen Cloud-Plattformen skalieren, sowie die GPU und CPU nutzen. Für die zu wählende Kommunikationstechnologie folgt daraus, dass diese ebenfalls gut skalierbar sein sollte.

- Erweiterbarkeit: Die Architektur sollte so entwickelt werden, dass weitere Services, ohne großen Aufwand hinzugefügt werden können. Bei dem Kommunikationsmuster, sollten demnach auf Lösungen gesetzt werden, welche so komplex wie nötig, jedoch so simpel wie möglich sind.

- Flexibilität: Die Flexibilität bezieht sich auf die Wahl der Hardware-Nutzung sowie auf verschiedene Benutzerrollen, die unterschiedliche Kenntnisse bezüglich der Parametrisierung von Generierungsmethoden haben. Die Kommunikation hat darauf kaum Einfluss.

- Monitoring: Eine weitere allgemeine Anforderung an die Plattformarchitektur besteht darin, das Protokollieren zu ermöglichen. Dies soll Transparenz über die Zustände der einzelnen Services bieten. Darüber hinaus kann dies das Benutzererlebnis in Bezug auf die Plattform verbessern. Für diese Anforderung bietet es sich an Kommunikationstechnologien zu wählen, welche Monitoring-Funktionalitäten bieten. RabbitMQ ist ein Beispiel hierfür. [2]

- Sicherheit: Um die Plattform vor externer Manipulation zu schützen, ist Sicherheit eine weitere allgemeine Anforderung. Dazu gehören beispielsweise die Authentifizierung von Benutzern, sowie die Daten vor unbefugtem Zugriff zu schützen. Um die Sicherheit auch bei der Kommunikation zu gewährleisten, könnte es hilfreich sein, auf bereits länger bekannte und weit verbreitete Technologien zurückzugreifen. Diese haben wahrscheinlich weniger unerkannte Schwachstellen.

Spezifische Anforderungen, welche sich aus der Anforderungsanalyse ergeben haben, sind:

- Kleines Entwicklerteam: Da dieses Projekt in einem Team von ca. 5 Personen entwickelt werden soll, sollte auf vermeidbare Komplexität und Kompliziertheit verzichtet werden, um den Entwicklungsaufwand für das bestehende Team tragbar zu gestalten.

- Umgang mit großen Datenmengen: Die Plattform muss in der Lage sein, mit große Datenmengen zu arbeiten. Diese Daten können von den Benutzern in die Plattform geladen sein, um ML-Modelle zu trainieren, um diese Daten durch generative Methoden erweitern zu lassen oder in der Plattform selbst, durch die verschiedenen generativen Methoden generiert worden sein. Diese Daten müssen von verschiedenen Services genutzt und verfügbar gemacht werden.

3.2 Identifizierte Services

Diese Arbeit basiert auf dem Paper [24] und nimmt die dort vorgestellte Architektur als Grundlage für das zu ermittelnde Kommunikationsmuster. Da sich das Projekt, während der Erstellung dieser Arbeit noch in der Entwicklung befindet, kann es in Zukunft zu Änderungen an der Architektur kommen, welche nicht in dieser Arbeit berücksichtigt, werden können.

Es wurden in der Konzeption einer geeigneten Architektur die folgenden Komponenten für die Umsetzung der Plattform als Microservice-Architektur ermittelt:

GUI: Die grafische Benutzeroberfläche ermöglicht die Interaktion der Benutzer mit der Plattform. Den Benutzern sollen hierrüber Funktionen zur Verfügung stehen, wie Login, Besichtigung der Daten, Parametrisierung der Algorithmen und Modelle.

Evaluation Manager: Die Evaluationskomponente in der Architektur ermöglicht die Beurteilung der Ergebnisse der Datengenerierung. Mögliche Metriken zur Bewertung des synthetischen Datensatzes sind z.B. Korrelationen, bivariate Verteilungen oder Genauigkeit. Die Ergebnisse werden in einem Qualitätssicherungsbericht zusammengefasst.

Data Manager: Das Datenmanagement umfasst Datenintegration, Qualitätsbewertung, Harmonisierung von Daten und Metadatenmanagement.

Generation Manager: Die Generation-Management-Komponente ermöglicht das Hinzufügen von Maschine-Learning-Modellen. Außerdem sollen einzelne generative Modelle parametrisiert, weiterentwickelt und trainiert werden können. Diese werden dann entweder in einer dem Benutzer zugehörigen Datenbank oder einer für alle Benutzer öffentlichen Datenbank gespeichert und können später wiederverwendet werden.

Orchestrator: Die Orchestrierungskomponente sorgt dafür, dass die einzelnen Dienste unter Berücksichtigung der bestehenden Abhängigkeiten in der gewünschten Reihenfolge ausgeführt werden. Sie soll die einzelnen Schritte der zuvor beschriebenen Komponenten (z.B. Datenmanagement, Modellmanagement) so koordinieren, dass die Generierung der Daten reibungslos abläuft.

Authentication: Die Authentifizierungskomponente ist für die Überprüfung der Identität der registrierten Benutzer zuständig.

Personalisation: Die Personalisierungskomponente soll Aufgaben übernehmen, welche die einzelnen Benutzer betreffen. Beispielsweise sollen hiermit die verschiedenen Benutzerrollen abgebildet werden und auch über diese Komponente die Daten der einzelnen Benutzer verwaltet werden. Unter diesen Daten können eigene ML-Modelle, Trainingsdaten usw. sein.

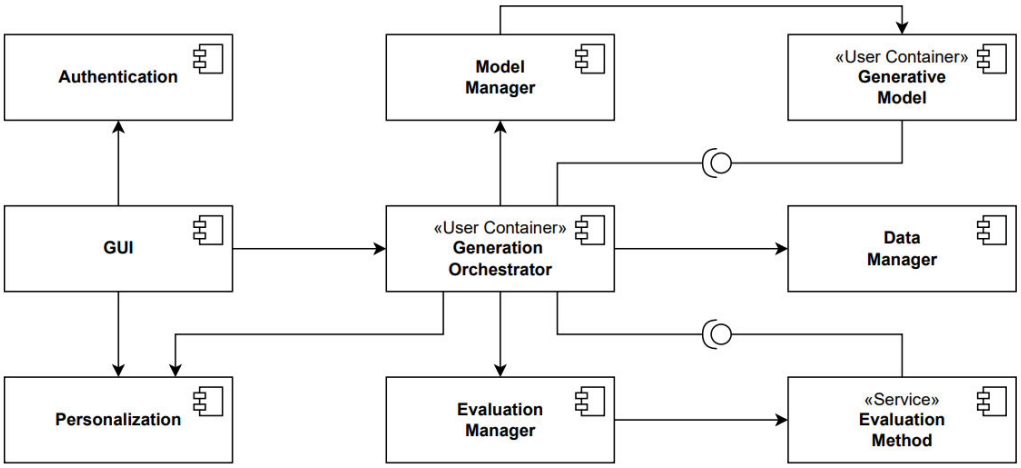

Abbildung 3.1: Komponenten der DaFne Plattform, welche den prototypischen Anwendungsfall für die Reproduktion von tabellarischen Daten implementieren [vgl. 24]

3.3 Identifikation der IPC

Die Services könnten auf viele verschiedene Arten miteinander arbeiten und dazu kommunizieren. Im Paper wurde eine beispielhafte Umsetzung mit REST APIs beschrieben. Dies dient unter anderem als Grundlage zur Identifikation der nötigen Kommunikation zwischen den Services.

3.3.1 Beispielhafter Ablauf einer Datengenerierung aus dem DaFne Paper

Der Benutzer muss sich zunächst über die GUI einloggen oder ggf. registrieren. Hierfür wird der GUI Service mit dem Authentication Service kommunizieren müssen.

Der GUI Service holt die gespeicherten Benutzerdaten von dem Personalisation Service.

Nun kann der Benutzer über die GUI, Eingaben tätigen, um z.B. einen Datengenerierungsprozess zu starten. Für diesen Prozess werden verschiedene Eingaben erforderlich sein, z.B. wird eine Generierungsmethode und gegebenenfalls mehrere Evaluationsmethoden ausgewählt werden müssen, sowie Daten für die jeweilige Generierungsmethode.

Wenn der Benutzer einen Generierungsprozess startet, wird der Orchestrator Service vom GUI Service informiert.

Der Orchestrator Service erhält dann über den Data Manager Service die Position der vom Benutzer ausgewählten Daten in der Datenbank.

Der Orchestrator Service benötigt ebenfalls die Daten für die gewählte Generierungsmethode vom Generation Manager Service. Diese Daten sind z.B. die Dockerimages, um Container zu starten, welche die gewünschte Methode implementieren. Diese Daten müssen dann zu dem Orchestrator Service gelangen.

Der Orchestrator Service kann nun einen Container mit dem Dockerimage starten. Dieser Generation Container benötigt nun die erforderlichen Daten aus der Datenbank und kann diese dann mit der entsprechenden Methode anreichern, bzw. auch ggf. neue generieren.

Zusätzlich erlangt der Orchestrator Service die benötigten Daten zu den Evaluationsmethoden vom Evaluation Manager und startet einen Evaluation Container mit dem jeweiligen Dockerimage und übergibt diesem die Position in der Datenbank, an der die generierten Daten vom Generation Container gespeichert werden.

Der Evaluation Container holt sich, wenn der Generation Container mit der Generierung fertig ist, die Daten und führt die Evaluation an diesen durch. Nach der Evaluation legt der Evaluation Container seinen Bericht in der Datenbank ab. Dafür muss der Evaluation Container informiert werden, dass der Generierungsprozess abgeschlossen ist.

Der Data Manager Service informiert nun den Orchestrator Service über den Abschluss des Prozesses und der Orchestrator Service liefert dann die Daten und den dazugehörigen Bericht über den GUI Service an den Benutzer.

Der GUI Service speichert nun die Benutzerdaten in den Personalisation Service ab, bspw. ML Modelldaten, wie Gewichte und Parameter.

3.3.2 Servicekommunikation

Im Folgenden wird begründet wieso welcher Service mit den anderen Services kommunizieren muss. Eine Auflistung der Kommunikationen zwischen Services befindet sich in der Tabelle 3.1.

GUI und Authentification: Die GUI muss mit dem Authentification Service für die Authentifikation der Benutzer kommunizieren.

GUI und Personalisation: Die GUI muss mit dem Personalisation Service kommunizieren, um die privaten Daten der Benutzer zu erhalten und um die benutzerbezogenen Daten zu erhalten.

GUI und Orchestrator: Die GUI muss mit dem Orchestrator Service kommunizieren, um den Genergierungsprozess anzustoßen und um nach dem Abschluss des Prozesses, die Ergebnisse zu erhalten.

Orchestrator und Data Manager: Der Orchestrator muss vom Data Manager die Position der benötigten Daten in der Datenbank erhalten, um diesen an die Container für die Generierung und Evaluation weiterzuleiten.

Orchestrator und Generation Manager: Der Orchestrator muss mit dem Generation Manager Service kommunizieren, um das Container Image zu erhalten, welches die vom Benutzer ausgewählte Generierungsmethode implementiert.

Orchestrator und Evaluation Manager: Der Orchestrator muss mit dem Evaluation Manager Service kommunizieren, um das Container Image zu erhalten, welches die vom Benutzer ausgewählte Evaluierungsmethoden implementiert.

Generation Container und Data Manager: Der Generation Container benötigt die Daten von dem Data Manager für die Generierung und der Data Manager benötigt die generierten Daten für Aufgaben wie Aggregation und Ähnliches.

Generation Container und Evaluation Container: Der Generation Container muss mit dem Evaluation Container für die Übergabe der generierten Daten kommunizieren, damit diese evaluiert werden können.

Evaluation Container und Data Manager: Der Evaluation Container muss die Evaluierungsberichte in der Datenbank speichern. Dafür könnte er die Berichte an den Data Manager schicken, welcher sie dann speichert.

Data Manager und Orchestrator: Der Orchestrator benötigt die Ergebnisse von dem Data Manager, damit dieser die Ergebnisse über die GUI für den Benutzer sichtbar machen kann.

In der folgenden Tabelle ist der benötigte Datenfluss aufgelistet, um eine bessere Übersicht zu ermöglichen. Die Richtung der Pfeile gibt an, ob lediglich Daten von einem zum

anderen Service müssen. Das Symbol <-> bedeutet, dass auf Daten in beide Richtungen versendet werden müssen.

Tabelle 3.1: Übersicht des benötigten Datenverkehr

Service A	Richtung	Service B
GUI	A <-> B	Authentication
GUI	A <-> B	Personalisation
GUI	A -> B	Orchestrator
Orchestrator	A <-> B	Data Manager
Orchestrator	A <-> B	Generation Manager
Orchestrator	A <-> B	Evaluation Manager
Generation Container	A <-> B	Data Manager
Generation Container	A -> B	Evaluation Container
Evaluation Container	A -> B	Data Manager
Data Manager	A -> B	Orchestrator
Orchestrator	A -> B	GUI
GUI	A -> B	Personalisation

4 Konzeption einer Lösung

Die Wahl des Kommunikationsstils ist eine Trade-Off Entscheidung. Für diese Entscheidung gibt es keine allgemeingültigen Regeln, wann welche Art verwendet werden sollte. Jede Art hat ihre eigenen Vor- und Nachteile. Die riesige Menge an verschiedenen Lösungen erschwert die Auswahl zusätzlich. [vgl. 23, S. 93]

Wichtig zu erwähnen ist, dass es sinnvoll sein kann, verschiedene Technologien zu mischen. Da jede Servicekommunikation verschieden ist, sollte in Betracht gezogen werden, verschiedene Technologien in der Architektur bzw. sogar innerhalb eines Services zu implementieren, damit die Kommunikation zwischen allen Services reibungslos abläuft. Dadurch kann auf alle Anforderungen eingegangen werden. Jedoch sollte hier das richtige Maß gefunden werden, da verschiedene Technologien auch zu einer erhöhten Komplexität führen. [vgl. 23, S. 95]

Um für die Kommunikation der einzelnen Services eine geeignete Technologie auszuwählen, wird zuerst bestimmt, ob die jeweilige Kommunikation synchron oder asynchron ablaufen sollte. Im nächsten Schritt wird berücksichtigt, ob der Service einen oder mehrere Services als Empfänger für Nachrichten besitzen sollte. Auf dieser Grundlage kann dann ein geeignetes Muster gefunden werden. Als letztes werden, wenn vorhanden, noch weitere spezifischere Bedingungen berücksichtigt, beispielsweise ob große Datenmengen zwischen den jeweiligen Services versendet werden. Auf Grundlage von diesen Informationen und den Empfehlungen der gesichteten Literatur zu diesem Thema, werden dann geeignete Technologien für die einzelnen Servicekommunikationen ermittelt.

4.1 Vor- und Nachteile synchroner Kommunikation

In [3] wird die synchrone Kommunikation als simples und bekanntes Prinzip aus der Entwicklung von Monolithen bezeichnet. Gördesli und Varol halten die synchrone Kommunikation für erforderlich, wenn die Antwort auf eine Anfrage notwendig für die Weiterarbeit

des anfragenden Service ist. [vgl. 14] Auch im Falle, dass ein direktes Feedback für den Benutzer relevant ist, weil zum Beispiel die Bestätigung einer Aktion, wichtig für das weitere Fortfahren ist, könnte man die synchrone Kommunikation sinnvoll verwenden. Ein weiterer Anwendungsfall für synchrone Kommunikation wird in [vgl. 4] beschrieben. In dieser Arbeit empfehlen die Autoren diese Art zu verwenden, wenn nur eine geringe Anzahl von Anfragen oder geringe Datenmengen zwischen den Microservices übertragen werden, weil dabei weniger Overhead und geringere Latenzzeiten entstehen als bei der asynchronen Kommunikation.

Auf der anderen Seite gibt es einige Gründe auf asynchrone Kommunikation zurückzugreifen. Der wahrscheinlich größte Vorteil besteht in der nicht blockierenden Art dieser Kommunikation. Dadurch kann der Client, welcher eine Anfrage sendet, solange die Anfrage verarbeitet wird, andere Anfragen verarbeiten. Eine weitere interessante Möglichkeit besteht in der Kommunikation mit mehreren Empfängern für Nachrichten. Diese Art ist exklusive nur asynchron möglich. Akbulut und Perros erklären in [3], dass die asynchrone Kommunikation zu präferieren ist, wenn große Datenmengen verarbeitet werden müssen und keine direkte Antwort erforderlich ist. Laut [vgl. 4, S. 6] kann asynchrone Kommunikation zudem, einen positiven Einfluss auf die Verfügbarkeit und Elastizität eines Systems haben.

4.2 Synchronizität der IPC zwischen den Services

1 (GUI -> Authentification): Synchron, weil die Authentifikation eines Benutzers abgeschlossen sein muss, bevor weitergearbeitet werden kann. Hier spielen die anderen Vorteile keine Rolle, da die Reihenfolge essenziell ist.

2 (GUI -> Personalisation): Synchron, da die Antwort für die weiteren Abläufe erforderlich ist und die Anfrage keine langen Prozesse benötigt, also schnell bearbeitet werden kann. Dadurch blockiert der GUI Service nicht lange und die Kommunikation ist trotz Synchronität performant.

3 (GUI -> Orchestrator): Asynchron, weil dieser Prozess einige Zeit in Anspruch nehmen kann. Er sollte deshalb asynchron angestoßen werden, damit in der Zwischenzeit anderen Tätigkeiten auf der Plattform nachgegangen werden kann, z.B. in vorherigen Prozessen generierte Daten begutachten.

4 (Orchestrator -> Data Manager): Synchron, weil der Orchestrator durch eine kurze Interaktion nur eine geringe Datenmenge (Position der vom User ausgewählten Daten in der DB) vom Data Manager erhalten muss und die Kommunikation dem Request/Response Pattern entspricht, was laut [vgl. 15] mit einer synchronen Kommunikation gelöst werden sollte.

5 (Orchestrator -> Generation Manager): Beides möglich. Synchron kann hier die bessere Alternative sein, um es simpel zu halten, da es eine kurze Interaktion ist, mit kleiner Datenmenge, die übertragen werden muss. [vgl. 15] Dafür würde auch sprechen, dass der Orchestrator die Daten des Generation Managers benötigt um den Container mit der Generationsmethode zu starten. Asynchron hätte jedoch den Vorteil, dass die Skalierbarkeit und Elastizität verbessert werden würde, sowie ermöglicht werden würde, dass der Orchestrator parallel etwas anderes bearbeitet, z.B. den Evaluationsmanager anzufragen.

6 (Orchestrator -> Evaluation Manager): Hier gilt das gleiche wie bei der Kommunikation vom Orchestrator mit dem Generation Manager. Es ist beides möglich, jedoch liegt es nahe, hierbei auf die asynchrone Kommunikation zu setzen, da es einige Zeit dauern kann bis der Generation Container seine Arbeit abgeschlossen hat und dadurch kein Zeitdruck besteht, den Evaluation Container schnellstmöglich zu starten. Dieser kann erst anfangen, sobald der Generierungsprozess und die Datenspeicherung abgeschlossen sind.

7 (Data Manager -> Generation Container): Asynchron, da der Data Manager ansonsten blockieren würde, solange der Generation Container den Prozess zur Generierung bearbeitet. Dieser Prozess kann einige Zeit in Anspruch nehmen, zudem werden auch potenziell große Datenmengen an den Data Manager geschickt, was ebenfalls für eine asynchrone Kommunikation spricht.

8 (Generation Container -> Evaluation Container): Der Evaluation Container muss informiert werden, dass die Generierung abgeschlossen ist und mit der Evaluation der Daten gestartet werden kann. Diese Kommunikation sollte asynchron geschehen, da keine Antwort nötig ist und auf diese Weise die Vorteile der asynchronen Kommunikation genutzt werden können.

9 (Evaluation Container -> Data Manager): Asynchron, da hier keine Antwort vom Data Manager erforderlich ist, da der Evaluation Container lediglich die Evaluierungsberichte

in die Datenbank überführen möchte, um diese zugänglich zu machen und diese Berichte gegebenenfalls relativ große Dateien enthalten können, bietet sich hier ebenfalls die asynchrone Kommunikation an.

10 (Data Manager -> Orchestrator): Asynchron, weil lediglich der Ort der Daten in der Datenbank weiter gegeben werden muss, sowie der Orchestrator informiert, dass der Prozess abgeschlossen ist. Es werden also kleine Datenmengen versendet und es ist keine Antwort notwendig, was für eine asynchrone Kommunikation spricht.

11 (Orchestrator -> GUI): Die GUI kann nun asynchron über den Abschluss des Generierungsprozesses informiert werden. Da hier ebenfalls keine Antwort nötig ist und bietet sich eine asynchrone Kommunikation an. Auf diese Weise kann der GUI Service, auf die Information reagieren, sobald es für diesen interessant ist und der informierende Service blockiert nicht .

12 (GUI-> Personalisation): Asynchron, da keine Antwort nötig ist, für die weitere Arbeit des GUI Service. Somit blockiert der GUI Service nicht, bis der Personalisation Service die Anfrage bearbeitet hat.

In Tabelle 4.1 sind die benötigten Kommunikationswege erneut aufgelistet, wobei diesmal die benötigte Synchronität in der vierten Spalte hinzugefügt wurde.

Tabelle 4.1: Übersicht über die benötigte Synchronität der Kommunikationswege

Service A	Richtung	Service B	Synchronität
GUI	A <-> B	Authentication	Synchron
GUI	A <-> B	Personalisation	Synchron
GUI	A -> B	Orchestrator	Asynchron
Orchestrator	A <-> B	Data Manager	Synchron
Orchestrator	A <-> B	Generation Manager	Asynchron
Orchestrator	A <-> B	Evaluation Manager	Asynchron
Generation Container	A <-> B	Data Manager	Asynchron
Generation Container	A -> B	Evaluation Container	Asynchron
Evaluation Container	A -> B	Data Manager	Asynchron
Data Manager	A -> B	Orchestrator	Asynchron
Orchestrator	A -> B	GUI	Asynchron
GUI	A -> B	Personalisation	Asynchron

4.3 Einen oder mehrere Empfänger für Nachrichten

Der Orchestrator Service könnte mehrere Empfänger besitzen, da sich die Manager-Services für die Informationen interessieren, dass beispielsweise ein neuer Generierungsprozess gestartet werden soll. Die Kommunikation könnte nach dem Event Driven Pattern vorgehen und die Kommunikation im Backend über Events steuern. Der Orchestrator könnte ein Event veröffentlichen, dass ein neuer Generierungsprozess gestartet werden soll und die Manager würden, automatisiert auf Grundlage der Daten in diesem Event ihre Container starten und mit dem Data Manager die Position der Daten in der Datenbank klären oder die nötigen Informationen an den Orchestrator senden, damit dieser die Container startet. Bei der Architektur der DaFne Plattform mit wenigen Services und dem Orchestrator, welcher die Informationen sammelt, um den Prozess zu koordinieren, bietet sich das simplere System an, in dem der Orchestrator mit jeweils einem Empfänger kommuniziert. Dafür spricht ebenfalls, dass der Ablauf einer Reihenfolge unterliegt, da beispielsweise die Evaluation erst nach der Generierung starten kann, was dafür sorgt, dass kein großer Vorteil davon zu erwarten ist, wenn die Services gleichzeitig vom Orchestrator angefragt werden, bzw. auf das Event reagieren würden.

Um eine fundierte Entscheidung zu treffen, ob es einen großen Einfluss auf die nichtfunktionalen Anforderungen hat, müsste ein Experiment durchgeführt werden, bei welchem die beiden Methoden verglichen werden würden. Dies ist jedoch nicht Bestandteil dieser Arbeit, deshalb wird in dieser Arbeit davon ausgegangen, dass es keinen großen Einfluss hätte, aufgrund der kleinen Anzahl von Services und der nicht eindeutigen Zuordnungsbarkeit zu einem der beiden Stile. Aus diesem und den im vorherigen Absatz genannten Gründen wird eine Kommunikation mit einem einzelnen Empfänger für Anfragen für alle Services empfohlen.

4.4 Nachrichtenformat

Da es im Kern von IPC um den Austausch von Nachrichten geht, ist ein geeignetes Nachrichtenformat nötig, welches den Anforderungen der Architektur dient und eine effiziente Kommunikation zulässt. Es gibt jedoch auch Technologien, welche das Nachrichtenformat diktieren, wie zum Beispiel gRPC. [vgl. 26, S. 71] Im Folgenden werden die beiden Kategorien der Nachrichtenformate, textbasierend und binär kurz erläutert und auf Grundlage

der jeweiligen Vor- und Nachteile ein geeignetes für die DaFne Plattform Kommunikation ausgewählt.

4.4.1 Textbasierend

Die textbasierenden Nachrichtenformate zeichnen sich durch ihre Lesbarkeit für Menschen aus, welche sie selbstbeschreibend macht. Das bedeutet, dass Menschen die Informationen ablesen können, welche in diesen Formaten enthalten sind, ohne sie vorher in menschliche Sprache zu überführen ablesen können. Vor allem XML neigt jedoch dazu, sehr wortreich zu sein, da jede Nachricht hier alle Namen der Attribute zusätzlich zu den Werten mitführt. Außerdem bringt das Parsen von besonders großen Nachrichten einen gewissen Mehraufwand mit sich. [vgl. 26, S. 71-72]

4.4.2 Binär

Es gibt viele verschiedene binäre Nachrichtenformate, bekannte Vertreter sind Protocol Buffer und Avro. Bei dieser Art von Format, werden die Nachrichten vor dem Absenden serialisiert und beim Empfänger wieder deserialisiert. Diese Schritte bringen einen gewissen Aufwand mit sich, jedoch steigt durch die Verwendung die Performance der IPC. Deshalb eignen sich binäre Formate vor allem, wenn Effizienz und Performance wichtig sind. [vgl. 26, S. 72]

4.5 Entscheidung für DaFne

Es ist vorteilhaft, verschiedene Muster in der Architektur zu mischen und für die jeweiligen Anforderungen, passende Muster auszuwählen. Laut Newman in [vgl. 23, S. 95] ist es die Norm, dass einzelne Services verschiedene Muster implementieren. Da es in der DaFne Architektur Services gibt, welche synchron kommunizieren sollten und welche bei denen es nicht notwendig ist, sollten mindestens 2 Kommunikationsarten implementiert werden, eine synchrone und eine asynchrone. Man sollte den Mehraufwand durch zusätzliche Kommunikationsarten nicht unterschätzen, da die Komplexität durch diese schnell steigen kann. Um die Anforderungen an die Kommunikation zu erfüllen und die Komplexität so niedrig wie möglich zu halten, wird sich in dieser Arbeit für lediglich zwei Kommunikationsarten entschieden. Um für diese eine geeignete Variante zu ermitteln, wird zuerst ein

passendes Kommunikationsmuster für die DaFne Plattform ermittelt und im Anschluss eine geeignete Technologie ausgewählt, um dieses Muster zu implementieren.

Die im Zuge dieser Arbeit durchgeführte Literaturrecherche hat eine Übersicht ergeben, welche dazu dient, die Auswahl der Muster und Technologien, die für die Kommunikation verwendet werden kann, zu beschränken und die am häufigsten behandelten Lösungen zu extrahieren. Im Folgenden wird darauf eingegangen, welche Lösungen extrahiert wurden. Auf dieser Grundlage und auf den Ergebnissen von Experimenten anderer Autoren beruhend, wird dann eine Auswahl für die DaFne Plattform getroffen.

4.5.1 Auswahl des Muster

In der Literaturrecherche hat sich das Muster Request/Response, als besonders verbreitet und häufig empfohlen herausgestellt.

Die andere häufig verwendete Möglichkeit, ist die Umsetzung des Publish/Subscribe Musters in einer Event-basierten Weise. Diese Möglichkeit wurde in [18] ebenfalls als sehr verbreitet erwähnt. Die Autoren der wissenschaftlichen Arbeit [5] erklären, dass Event basierte Kommunikation in den letzten Jahren immer mehr an Interesse gewinnt und ein Wechsel stattfindet, vom Client-Server Denken zum informationszentriertem Denken. In dieser Arbeit wird außerdem auf die verschiedenen Protokolle eingegangen, welche für die Umsetzung verwendet werden können. Diese Protokolle eignen sich für verschiedene Anwendungsfälle, beispielsweise wird das leichtgewichtige MQTT-Protokoll häufig für IoT Anwendungsfälle genutzt. Newman stellt dieses Muster in [23], genauso wie Richardson in [26], vor. Bonér erklärt in [vgl. 7, S. 17], dass die Kommunikation durch asynchrone Weitergabe von Nachrichten gefördert werden sollte, um die Entkopplung der Microservices zu unterstützen. In [vgl. 14] wird die Event basierte Kommunikation empfohlen, für Anwendungsfälle, in denen es mehrere Empfänger für Nachrichten gibt und die Verarbeitungsdauer von Anfragen hoch ist. In [vgl. 29] wird die Event basierte Kommunikation als sehr flexibel und schneller als HTTP bezeichnet. Außerdem wird auf den Vorteil der Persistenz von Nachrichten aufmerksam gemacht. Event basiertes Publish/Subscribe ist jedoch hauptsächlich für Kommunikation geeignet, bei der es mehrere Empfänger für einzelne Nachrichten gibt. Deshalb wurde sich in dieser Arbeit, für die DaFne Plattform vorerst gegen dieses Muster entschieden.

Das Request/Response Muster scheint aus den folgenden Gründen geeigneter zu sein. Dieses Muster lässt sich synchron und asynchron implementieren, wodurch nur ein Muster

implementiert werden muss, trotz des Vorkommens beider Kommunikationsarten zwischen den Services. [vgl. 23, S. 104] In der DaFne Architektur werden hauptsächlich Daten von anderen Services abgefragt. Beispielsweise fragt der GUI Service Daten beim Personalisation Service an. Für diesen Anwendungsfall ist das Request/Response Muster sehr verbreitet laut Newman. [vgl. 23, S. 105]

Bei der synchronen Umsetzung wird für gewöhnlich eine Netzwerkverbindung zu dem angefragten Microservice geöffnet. Diese Verbindung wird dann offengehalten, solange der anfragende Microservice auf die Antwort wartet. Hierbei muss der angefragte Service nichts über den anfragenden Microservice wissen, er schickt lediglich eine Antwort über die eingegangene Verbindung zurück. [vgl. 23, S. 106]

Bei der asynchronen Umsetzung ist das anders. Hierbei muss der angefragte Microservice wissen, wohin die Antwort geschickt werden soll, dies lässt sich explizit oder implizit umsetzen. [vgl. 23, S. 107] Es kann bei der asynchronen Umsetzung Message oriented Middleware (MOM) verwendet werden, welche Warteschlangen für die Messages bereitstellt. Diese hat den Vorteil, dass mehrere Messages gehalten und nacheinander abgearbeitet werden können, wenn der jeweilige Service die Kapazität dafür hat. [vgl. 23, S. 107]

Aus den genannten Gründen und um die Kommunikation so simpel wie möglich zu halten, wurde sich für das Request/Response Muster entschieden, welches synchron und asynchron implementiert werden soll.

4.5.2 Auswahl der synchronen Technologie

In der Literatur wurden REST APIs über HTTP von den synchronen Kommunikationsmöglichkeiten am häufigsten erwähnt. Am zweit häufigsten wurde Apache Thrift erwähnt und am dritt häufigsten gRPC. In [vgl. 6] wurden außerdem die Ansätze vorgestellt, bei denen die Microservices direkt miteinander kommunizieren, was nicht zu empfehlen ist und der Einsatz von API-Gateways zwischen dem Benutzer und den Microservices.

In der systematischen Übersichtsarbeit [18] wurden insgesamt 239 wissenschaftliche Arbeiten gesichtet und für die Beantwortung der Forschungsfragen dieser Arbeit, 38 Arbeiten ausgewählt und genauer betrachtet. In dieser Übersichtsarbeit wurden REST APIs und Apache Thrift als Teil der verbreitetsten Lösungen für synchrone Kommunikation erschlossen. In [17] wurden REST APIs und Thrift vorgestellt. Die Arbeit [32] stellt APIs über HTTP mit Apache Zookeeper als Service Register vor. In dieser Arbeit wurde eine

eigene Lösung entwickelt, um die Vorteile von HTTP APIs mit denen der asynchronen Kommunikation zu vereinen. Diese Lösung wurde als "chain oriented load balancing algorithm (COLBA)"vorgestellt. In [22] wird erwähnt, dass Netflix unter anderem Thrift in ihrer Microservice-Architektur verwenden. [31] beinhaltet ein Experiment zum Vergleich von REST APIs und RabbitMQ. Die beiden Autoren von [14] empfehlen die synchrone Kommunikation über REST APIs, wenn keine komplexe Architektur hinter den Microservices steckt und die Services sofortige Antworten auf ihre Anfragen benötigen. In dem Fachbuch [15] von Indrasiri und Siriwardena, werden Thrift und gRPC als mögliche Technologien für die synchrone Kommunikation vorgestellt. In [20] haben die Autoren ein Experiment zum Vergleich von REST APIs, Thrift und gRPC als synchrone Kommunikationsmöglichkeit durchgeführt und die Ergebnisse vorgestellt. In dem Experiment dieser Autorengruppe, stellte sich Thrift als performanteste Lösung heraus, gefolgt von gRPC und an letzter Stelle die REST APIs. Die Arbeit [3] vergleicht Designmuster für Microservices. In dem Kontext wurden auch REST APIs begutachtet. In [9] werden REST APIs mit RabbitMQ verglichen und REST APIs als weniger performant bei höheren Lasten erklärt. In [27] wurden REST APIs ebenfalls vorgestellt.

Da sich die Kommunikation bei den bisher identifizierten Services der DaFne Plattform hauptsächlich auf leichtgewichtige Anfragen beschränkt, welche ohne große Datenmengen auskommen und schnell verarbeitet werden können, sollten alle drei Technologien performant genug arbeiten. Die im vorherigen Absatz genannten Informationen sollten berücksichtigt werden, wenn sich in Zukunft für weitere synchrone Kommunikation im Backend der DaFne Plattform entschieden werden sollte. REST APIs sind eine bekannte und simple Möglichkeit, synchrone Anfragen an andere Services zu stellen. Dies kommt dem kleinen Entwicklerteam zugute. Außerdem sind sie einfach zu testen und Firewall freundlich. [vgl. 3] Für die Funktionen der DaFne Plattform und angesichts der nichtfunktionalen Anforderungen, eignen sich REST APIs gut als Technologie für die synchrone Kommunikation zwischen den Services. Dafür spricht auch die hohe Verbreitung von REST im Webbereich. Aus diesen Gründen werden in dieser Arbeit REST APIs als Technologie für die synchrone Kommunikation gewählt.

4.5.3 Auswahl der asynchronen Technologie

Bei den asynchronen Kommunikationsmöglichkeiten wurden zwei besonders häufig beschrieben. Die erste ist das Einbauen von MOM, zum Beispiel Message Broker wie RabbitMQ ihn anbietet, um das Request/Response Muster umzusetzen. Diese Möglichkeit

wurde in [18] als eine der verbreitetsten Lösungen dargestellt und in [17, 6, 15] ebenfalls vorgestellt, als Möglichkeit bei Kommunikation mit einzelnen Empfängern für die Nachrichten. In [32] wird als einziges Manko die fehlende Option zur synchronen Kommunikation mit den Message Brokern genannt, weshalb hier die eigene Lösung durch 'COLBA' entwickelt wurde. In [31] wurde in einem Experiment gezeigt, dass die Kommunikation über das AMQP deutlich besser bei großen Datenmengen funktioniert, als über REST APIs. Zu einem ähnlichen Ergebnis kamen die Autoren von [12]. Diese kamen zu dem Schluss, dass es der beste Ansatz ist, RabbitMQ für Systeme zu verwenden, in denen große Datenmengen versendet werden. Sie fanden heraus, dass dieser Ansatz dabei helfen kann, Ressourcen zu sparen, Datenverlusten vorzubeugen und Messages besser zu organisieren.

Für die Umsetzung des Event basierten Publish/Subscribe Musters wird in [15] Apache Kafka empfohlen. Kafka ist ein verteiltes Publish/Subscribe System. Zu diesem Ergebnis kam auch die Umfrage zu Publish/Subscribe Middleware in [vgl. 19]. Hier wurde ebenfalls Kafka für die Kommunikation mit mehreren Empfängern empfohlen. Eine weitere eher exotische Lösung wurde in [vgl. 27] hervorgehoben. In dieser Arbeit wurde das Publish/Subscribe Pattern mit Redis Stream Data Structure implementiert.

Da sich in der DaFne Architektur bisher kein großer Nutzen von Kommunikation mit mehreren Empfängern für Messages gezeigt hat und sich das Request Response Muster für DaFne als sinnvoll herausgestellt hat, wird sich in dieser Arbeit für eine Umsetzung des Request Response Musters über einen Message Broker entschieden. In [vgl. 9] wird RabbitMQ als die am meisten genutzte Lösung zur Implementierung des AMQP erklärt. Aufgrund der großen Popularität von RabbitMQ und der umfangreichen Funktionalität, wird sich in dieser Arbeit für die DaFne Plattform für diese Technologie entschieden, um die asynchrone Kommunikation zu implementieren.

Zur besseren Übersicht, wird in der Tabelle 4.2 aufgelistet, welche Kommunikationsmöglichkeiten in welchen Quellen gesichtet wurden.

4.5.4 Auswahl des Nachrichtenformat

Performance ist keine der priorisierten Anforderungen der DaFne Plattform, zudem lässt sich eine gewisse Wartezeit durch die aufwändigen Datengenerierungsprozesse ohnehin nicht vermeiden. Da die für die Generierung genutzten und daraus entstehenden Daten tabellarisch sein werden und aus zuvor genannten Gründen, sollte für die Nachrichten

Tabelle 4.2: Übersicht der Kommunikationsmöglichkeiten und den Quellen, in denen diese gesichtet wurden

Muster/Technologie	Quellen
REST APIS	[18, 17, 32, 24, 30, 31, 14, 15, 29, 20, 3, 9]
Thrift	[18, 17, 22, 15, 20]
gRPC	[15, 20]
RR mit Message Broker	[17, 6, 32, 23, 26, 31, 3]
Pub/Sub als Event basiert	[18, 5, 23, 26, 7, 14, 15, 29, 9, 19]
Geteilte Datenbank	[25, 23]

ein textbasiertes Format genutzt werden, um die Vorteile dieser Formate zu nutzen. Auf diese Weise sind die Nachrichten für Menschen lesbar und ermöglichen den Konsumenten, lediglich die für sie interessanten Informationen zu extrahieren und den Rest zu ignorieren. In [15] wird vom Autor beschrieben, dass JSON bei REST verbreiteter sei als XML und XML nur in seltenen Fällen besser sei. JSON scheint dadurch als geeignete Wahl für das Nachrichtenformat der DaFne Plattform.

In Tabelle 4.3 werden die Ergebnisse dieses Kapitels zusammengefasst.

Tabelle 4.3: Übersicht der Auswahl für die DaFne Plattform

Kategorie	Auswahl für DaFne
Kommunikationsmuster	Request/Response
Synchrone Technologie	REST APIs
Asynchrone Technologie	RabbitMQ
Nachrichtenformat	JSON

5 Detailbeschreibung und Diskussion der Ergebnisse

Dieses Kapitel hat das Ziel, die ausgewählten Technologien detaillierter zu beschreiben und zu diskutieren, ob die gewählten Technologien die in Kapitel 3 identifizierten Anforderungen erfüllen. Zusätzlich wird auf mögliche Schwierigkeiten der gewählten Lösungen für die Kommunikation eingegangen und ein Ausblick dazu gegeben, wie man in Zukunft mit diesen Schwierigkeiten umgehen könnte.

5.1 Kommunikation über REST APIs und RabbitMQ

5.1.1 REST APIs

APIs sind Schnittstellen, welche eine Sammlung an Daten und Funktionalitäten anbieten, um die Interaktion von Anwendungen zu erleichtern und den Austausch von Informationen zu ermöglichen. Diese werden von Client Anwendungen genutzt, um mit Web Service Anwendungen zu kommunizieren. Web APIs welche dem REST Architekturstil entsprechen, werden REST APIs genannt. [vgl. 21, S. 5] In der Regel hat jeder Service einen eigenen Web Server auf einem bestimmten Port, wie z.B. 8080 oder 443 laufen, wenn in einem System REST APIs für die IPC genutzt werden. Um die Interaktion zu ermöglichen, implementieren die Services verschiedene sogenannte Endpoints. Diese Endpoints besitzen Schnittstellen, in denen eine Reihe an Operationen definiert sind, welche aufgerufen werden können und eine entsprechende Antwort zurück liefern.

Abbildung 5.1 zeigt wie eine Umsetzung des Request/Response Musters aussehen kann, wenn über eine Web API kommuniziert wird.

REST APIs bestehen hauptsächlich aus 3 Komponenten, einer Ressource, einer Repräsentation dieser Ressource und einem Verb. Die Ressource ist eine erreichbare URI und

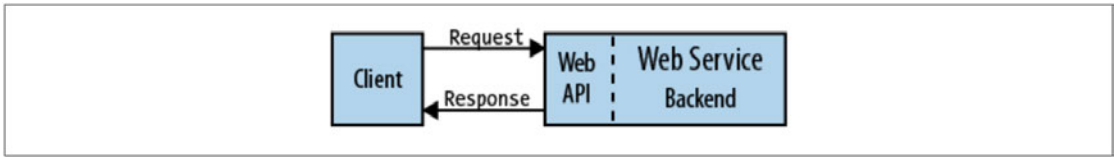

Abbildung 5.1: Web-API [vgl. 21, S. 6]

repräsentiert üblicherweise eine Entität oder eine Sammlung von Entitäten, wie Kunden oder Produkte. Dessen Repräsentation wäre beispielsweise ein JSON Objekt oder eine Datei mit JSON Objekten für alle Entitäten. Das Verb bezieht sich auf die HTTP Methoden, mit denen es möglich ist die CRUD Operationen zu erledigen. Die CRUD Operationen sind Create, Read, Update und Delete. [vgl. 10]

Für die DaFne Plattform sollen die REST APIs als synchrone Kommunikationsmöglichkeit im Request/Response Stil benutzt werden. Hierfür wird ein Service eine Anfrage an einen anderen Service stellen und eine gewisse Zeit darauf warten, dass der Service die Anfrage bearbeitet und eine Antwort zurücksendet. Wenn diese Antwort ausbleibt, kommt es nach einer gewissen Zeit zu einem Timeout. Die Anfrage muss in diesem Fall erneut gestellt werden. Zu diesem Fehler könnte es kommen, wenn die angefragte Serviceinstanz nicht mehr existiert oder keine Kapazität hatte, die Anfrage in der gewünschten Zeit zu beantworten. Der aufrufende Service blockiert normalerweise, während er auf die Antwort wartet, weil die Netzwerkverbindung zwischen den Services offen gehalten wird, bis die Antwort eintrifft oder es zum Timeout kommt. Wenn die Antwort rechtzeitig eintrifft, kann der Service, welcher die Anfrage gestellt hat, weiterarbeiten.

HTTP Anfragen und Antworten folgen einem definierten Muster. Das Folgende ist ein Beispiel für eine Anfrage und die dazu passende Antwort aus [vgl. 21, S. 24]:

Anfrage:
> GET /greeting HTTP/1.1
> User-Agent: curl/7.20.1
> Host: api.example.restapi.org
> Accept: */*

Antwort:
< HTTP/1.1 200 OK
< Date: Sat, 20 Aug 2011 16:02:40 GMT
< Server: Apache
< Expires: Sat, 20 Aug 2011 16:03:40 GMT
< Cache-Control: max-age=60, must-revalidate
< ETag: text/html:hello world
< Content-Length: 130
< Last-Modified: Sat, 20 Aug 2011 16:02:17 GMT
< Vary: Accept-Encoding
< Content-Type: text/html

5.1.2 RabbitMQ

Bei der asynchronen Kommunikation mit RabbitMQ werden Nachrichten von den Services an einen Message Broker gesendet und von diesem an den anzufragenden Service weitergeleitet. Dieser Broker fungiert als MOM und bietet die Funktionalität, die Nachrichten in einer Warteschlange zu speichern, bis diese bearbeitet werden. Die Services können sich bei diesem Broker als Publisher und als Consumer registrieren und auf diese Weise an der Kommunikation teilnehmen. Die Nachrichten, die von einem Service an den Broker gesendet wurden, werden dann von den Consumer Services aus dieser Warteschlange entgegengenommen und verarbeitet, sobald diese die Kapazität dafür haben. [vgl. 16, S. 15] RabbitMQ bietet außerdem eine weite Bandbreite an Konfigurationsmöglichkeiten und weiteren Funktionalitäten, z.B. im Bereich Monitoring. Auf diese Funk-

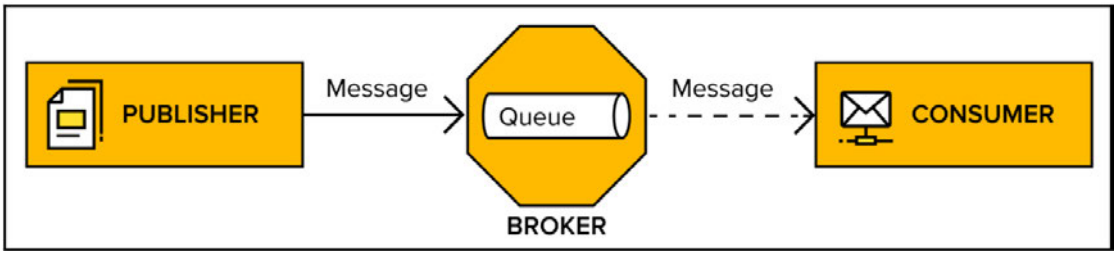

Abbildung 5.2: Grundlegende Komponenten der Kommunikation über einen Message Broker [vgl. 16, S. 9]

tionalitäten wird in dieser Arbeit jedoch nicht weiter eingegangen, da in dieser Arbeit der Fokus auf der Auswahl eines geeigneten Kommunikationsmusters besteht.

In Abbildung 5.2 sind die grundlegenden Komponenten der Kommunikation über einen Message Broker visualisiert.

Queue: In RabbitMQ ist die Queue eine geordnete Sammlung, welche Messages nach dem FIFO-Prinzip speichern und ausgeben kann. Dabei bietet sie den Status der Messages an, diese können beispielsweise konsumiert, abgelehnt oder bestätigt sein. Viele andere Message Broker Lösungen bieten diesen Status der Messages nicht an. [vgl. 3]

Messages: Eine Message ist ein generisches Konzept, welches beschreibt, was ein Message Broker versendet. Eine Message könnte eine Anfrage, eine Antwort oder ein Event sein. [vgl. 23, S. 135]

Messages können aus einem Header und einem Body bestehen. Im Header befinden sich Schlüssel-Wert Paare, welche die Metadaten enthalten. Außerdem kann im Header eine ID und optional eine Rücksendeadresse vorhanden sein. Im Body befinden sich die Daten, welche eigentlich versendet werden sollen. [vgl. 26, S. 86]

5.2 Services mit ausgewählten Technologien

Der GUI Service muss beide Kommunikationsarten implementieren. Dieser wird über die REST APIs mit dem Authentification Service und dem Personalisation Service kommunizieren. Zusätzlich wird der GUI Service mit dem Orchestrator Service über RabbitMQ Messages teilen, um die Aufträge der Benutzer umsetzen zu lassen.

Der Authentification Service wird lediglich die REST APIs anbieten müssen, um die Authentifikation der Benutzer für andere Services bereitzustellen.

Das gleiche gilt beim Personalisation Service. Dieser wird ebenfalls nur REST APIs anbieten, um seine Funktionalitäten für den Benutzer, von der GUI aus, nutzbar zu machen.

Der Orchestrator Service muss wie der GUI Service beide Arten implementieren. Um mit dem Data Manager zu kommunizieren, wird dieser seine REST APIs verwenden. Mit dem GUI Service, dem Generation Manager Service und dem Evaluation Manager Service zu kommunizieren, wird der Orchestrator Service RabbitMQ verwenden.

Der Generation Manager und der Evaluation Manager werden lediglich über RabbitMQ mit dem Orchestrator kommunizieren.

Um hohe Latenzen bei dem Datenverkehr zwischen dem Data Manager Service, dem Generation Container und dem Evaluation Container, trotz der potenziell sehr großen Datensätze, gering zu halten und um das Blockieren dieser Services für eine längere Zeit zu vermeiden, wird die Kommunikation zwischen ihnen über einen gemeinsamen Datenspeicher geschehen. Der Generation Container holt sich die benötigten Daten aus der Datenbank und speichert die von ihm veränderten Daten wieder dort ab. Der Evaluation Container macht das gleiche. Er holt sich selbständig die Daten aus dem Speicher und legt seine Evaluationsdateien nach der Evaluation in der Datenbank ab.

Der Data Manager muss den Orchestrator über die Position der Daten in der Datenbank und den Abschluss des Prozesses informieren. Um die generierten Daten und die Berichte über die Evaluation für den Benutzer sichtbar zu machen, müssen diese nun lediglich über den Orchestrator an die GUI gesendet werden. Durch diesen Ansatz blockiert jeweils nur der beteiligte Service und die Datenbank, statt direkt zwei Services.

5.3 Beispielablauf mit Kommunikationstechnologien

Im Folgenden wird der beispielhafte Ablauf einer Datengenerierung aus dem Kapitel 3 mit den in Kapitel 4 ermittelten Kommunikationsarten vorgestellt:

Am Start muss sich der Benutzer einloggen, dies geschieht über den GUI Service. Der GUI Service schickt dann die Benutzer Logindaten über einen HTTP GET Aufruf an den

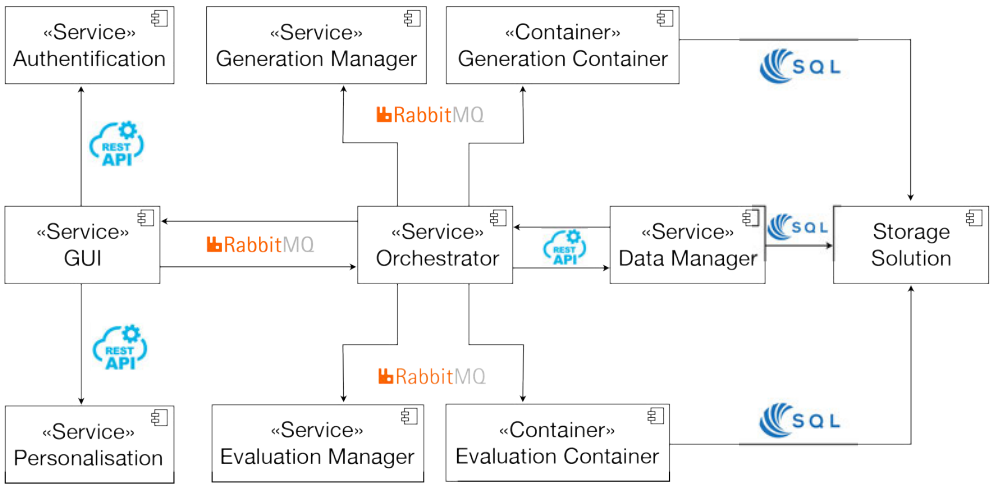

Abbildung 5.3: Übersicht der IPC mit den ausgewählten Technologien [Eigene Darstellung]

Authentifikation Service. Der GUI Service wartet bis er eine Antwort erhalten hat oder die Antwortzeit abgelaufen ist. Der Authentification Service sendet in diesem Beispiel rechtzeitig eine Antwort und der Benutzer ist mit korrekten Logindaten eingeloggt.

Als nächstes fragt der GUI Service ebenfalls über einen HTTP Aufruf bei dem Personalisation Service die benutzerbezogenen Daten an. Wenn dieser antwortet und diese bei dem GUI Service angekommen sind, lädt dieser die GUI für den jeweiligen Benutzer.

Wenn der Benutzer nun über die GUI einen Datengenerierungsprozess gestartet hat, sendet der GUI Service eine Message mit allen nötigen Informationen an den RabbitMQ Messagebroker, bei dem der Orchestrator Service als Consumer registriert ist. Der Orchestrator Service holt sich dann aus der Warteschlange des Brokers die Message, sobald er frei ist und fängt an diese zu verarbeiten.

Der Orchestrator Service fragt dann beim Data Manager Service die Position der Daten in der Datenbank über einen HTTP-Aufruf an und wartet auf die Antwort.

Nach erhaltener Antwort, sendet der Orchestrator Service jeweils eine Message an den Broker, welche als Antwort die benötigten Daten vom Generation Manager und vom Evaluation Manager erwartet.

Sobald der Orchestrator Service die nötigen Daten zum Starten des Generation Container hat, startet dieser den Container und gibt diesem dabei die Position der Daten in der Datenbank mit.

Der gestartete Generation Container lädt sich nun ohne weitere Kommunikation die Daten aus der Datenbank und bearbeitet den Generierungsprozess. Wenn der Generierungsprozess beendet worden ist, speichert der Generation Container die neu generierten Daten in der Datenbank ab, an die Position, welche der Orchestrator Service festgelegt hat. Und sendet bei erfolgreichem Speicherprozess eine Message zum Broker, um den Abschluss der Generierung zu signalisieren.

Der Orchestrator Service startet in der Zwischenzeit den Evaluation Container und gibt diesem die Position der Daten in der gemeinsamen Datenbank mit, an der die zu evaluierenden Daten liegen werden. Sobald der Evaluation Container die Message des Generation Container konsumiert, lädt sich dieser die generierten Daten aus der gemeinsamen Datenbank und fängt mit der Evaluation an.

Wenn die Evaluation abgeschlossen ist, speichert der Evaluation Container seinen Evaluationsbericht in der Datenbank ab und sendet eine Message, welche signalisiert, dass der Evaluationsprozess abgeschlossen ist, sowie die Informationen zu der Position des Evaluationsberichtes in der Datenbank an den Broker.

Da der Orchestrator als einziger Zugangspunkt ins Backend von dem GUI Service geplant ist, sendet der Orchestrator Service nun eine Message mit den nötigen Informationen an den Broker zwischen GUI Service und Orchestrator Service. Der GUI Service kann dann auf die Message reagieren, sobald er verfügbar ist und dem Benutzer ggf. die generierten Daten und den Evaluationsbericht anzeigen.

Sollte der Benutzer benutzerbezogene Daten speichern wollen, beispielsweise den Evaluationsbericht, sendet der GUI Service einen HTTP POST Aufruf an den Personalisation Service, um diesem die Daten zu senden.

5.4 Diskussion der Ergebnisse

Im Folgenden wird ein kritischer Blick auf die ermittelte Lösung geworfen, in dem geprüft wird, ob die ausgewählten Technologien den Anforderungen der DaFne Plattform entsprechen. Im Anschluss daran werden mögliche Schwierigkeiten der ermittelten Lösung aufgezeigt und im letzten Abschnitt wird auf Möglichkeiten eingegangen, diese Schwierigkeiten in zukünftigen Arbeiten anzugehen.

5.4.1 Erfülliung der Anforderungen durch REST-APIs

- Skalierbarkeit: Rest APIs sind nicht optimal zum Skalieren, jedoch ist diese Technologie nur sparsam eingesetzt, weshalb sie vermutlich wenig Einfluss auf die Skalierbarkeit der Plattform haben wird.

- Erweiterbarkeit: Die Anbindung von Services mit der REST API ist einfach möglich, wodurch die Erweiterbarkeit gefördert wird. Es könnte zu Problemen kommen, wenn Services so hinzugefügt werden, dass sich die Anfrageketten vergrößern. Dies kann bei synchroner Kommunikation mit REST APIs zu höheren Latenzen führen.

- Flexibilität: Die Kommunikationstechnologie nimmt kaum Einfluss auf die Flexibilität.

- Monitoring: Für das Monitoring von REST APIs gibt es durch deren hohe Verbreitung im Web Umfeld, einige Tools. Diese machen das Monitoring der APIs einfach und helfen so auch dem gesamten Plattform Monitoring.

- Sicherheit: Mit dem neuen Standard von HTTPS, welcher für die REST APIs verwendet werden sollte, wird zum einen die Kommunikation verschlüsselt, was die Kommunikation für Unbefugte unkenntlich macht und außerdem wird der Webserver authentifiziert, was die Kommunikation ebenfalls sicherer macht.

- Kleines Entwicklerteam: REST APIs sind ideal für kleine Entwicklerteams mit wenig Knowhow, da es eine bekannte Technologie ist und eine API nach der anderen entwickelt werden kann.

Umgang mit großen Datenmengen: REST APIs sind in der Lage mit großen Datenmengen umzugehen, jedoch sind diese dabei nicht überaus performant mit diesen. In

DaFne sollte daher darauf geachtet werden, diese eher zwischen den Services zu verwenden, bei denen keine großen Datenmengen versendet werden sollen.

5.4.2 Erfüllung der Anforderungen durch RabbitMQ

- Skalierbarkeit: RabbitMQ hat sich im Experiment der Autoren von [vgl. 31], als besonders gut skalierbar gezeigt.

- Erweiterbarkeit: Das Hinzufügen von neuen Services ist mit RabbitMQ sehr gut möglich. Dafür müssen die Services sich lediglich beim Message Broker als Publisher oder Consumer registrieren.

- Flexibilität: Die Kommunikationstechnologie nimmt kaum Einfluss auf die Flexibilität.

- Monitoring: RabbitMQ bietet eigene Funktionen für das Monitoring an, dadurch wird das Monitoring der Plattform ebenfalls unterstützt.

- Sicherheit: RabbitMQ bietet verschiedene Möglichkeiten zur Zugriffkontrolle an, darunter verschiedene Mechanismen zu Authentifikation und Autorisierung

- Kleines Entwicklerteam: RabbitMQ lässt sich schwieriger implementieren als REST APIs, jedoch wird deutlich weniger Knowhow benötigt, als beispielsweise für ein Apache Kafka oder ähnliche Lösungen. Es sollte gut umsetzbar sein, auch mit einem kleineren Entwicklerteam.

- Umgang mit großen Datenmengen: RabbitMQ ist ausgezeichnet für den Umgang mit großen Datenmengen geeignet. Durch die Datenverlustprävention und die asynchrone Verwendung, blockieren die Services nicht und der Verlust der Daten, bei Fehlern wird auch vermieden.

In der Tabelle 5.1 sind die ermittelten Anforderungen an die DaFne Plattform aufgelistet und mit einer Bewertung versehen. Bei der Bewertung steht o dafür, dass auf diese Anforderung keinen Einfluss durch die Technologie genommen wird. Ein + steht dafür, dass die Anforderung gut erfüllt wird, - steht dafür, dass eine Anforderung nicht erfüllt wird und -/+ steht für eine mittelmäßige Eignung für die jeweiligen Anforderung.

Tabelle 5.1: Bewertung der Kommunikationstechnologien in Bezug auf die DaFne Anforderungen

Anforderung	REST APIs	RabbitMQ
Skalierbarkeit	-/+	+
Erweiterbarkeit	+	+
Flexibilität	o	o
Monitoring	+	+
Sicherheit	+	+
Kleines Entwicklerteam	+	-/+
Umgang mit großen Datenmengen	-/+	+

5.4.3 Schwierigkeiten der vorgestellten Lösung

- Datenzugang von der GUI aus: Da die einzige Schnittstelle, von der GUI ins Backend, über den Orchestrator geht, kann von der GUI aus nicht so leicht auf die Daten in der Datenbank zugegriffen werden. Dies geschieht umständlicher Weise über den Orchestrator. Die Bereitstellung der Daten wird dadurch unnötig verlangsamt.

- Feststellen der verfügbaren Services: Wenn Serviceinstanzen neu hochgefahren werden oder Instanzen außer Betrieb gehen und durch neue ersetzt werden, müssen diese wieder in das Netzwerk der bestehenden eingepflegt werden, so dass sie sichtbar für die Anfragen der anderen Services sind.

- Load Balancing: Anfragen müssen fair und sinnvoll auf die verfügbaren Instanzen eines Services aufgeteilt werden.

5.4.4 Mögliche Lösungsansätze

Um mit den Schwierigkeiten der in dieser Arbeit ermittelten Lösung in zukünftigen Arbeiten umzugehen, sollte zuerst eine Möglichkeit ermittelt werden, um die Daten einfacher von der GUI aus zugänglich zu machen. Es sollte jedoch berücksichtigt werden, dass in dieser Arbeit davon ausgegangen wurde, dass der Orchestrator Service der einzige Zugang von außen ins Backend ist, um dieses sicherer zu gestalten. Wenn nun durch eine Verbesserung der Lösung, beispielsweise ein direkte Zugriffe vom GUI Service auf die Datenbank getätigt werden könnten, sollte dafür gesorgt werden, dass dieser Kommunikationsweg möglichst nicht für ein hohes Sicherheitsrisiko sorgt.

In der Recherche für diese Arbeit wurden noch weitere Möglichkeiten gesichtet, die ausgewählten Technologien zu verwenden. Diese könnten in der Zukunft ebenfalls berücksichtigt werden.

In [vgl. 15, S. 64] wird erwähnt, dass synchrone Kommunikation nicht automatisch eine blockierende Kommunikation sein muss. Vielmehr sei es die Regel, dass synchrone Kommunikation nicht-blockierend implementiert werde. Dies könnte man in dem Fall der REST APIs der DaFne Plattform umsetzen, indem auf die Anfragen eines Service, eine direkte Antwort erfolgt, welche den Eingang der Anfrage bestätigt. Die benötigten Daten, welche durch die Anfrage angefordert wurden, könnten dann zu einem späteren Zeitpunkt, entweder über eine neue Netzwerkverbindung oder auch über RabbitMQ an den Service gesendet werden. Durch diese Art der Implementierung würde der anfragende Service nicht mehr blockiert sein, bis die eigentliche Antwort eingetroffen ist.

Auch RabbitMQ bietet eine weitere Möglichkeit an, um für die Kommunikation genutzt zu werden. Anstelle des Request/Response Musters könnte auch ein Muster implementiert werde, welches auf Events basiert. Hier für bietet RabbitMQ sogenannte Topics an. Services können sich bei diesen Topics als Observer registrieren und über neue Events informiert werden. [vgl. 23, S. 135] Beispielsweise könnte man den Evaluation Container an ein Topic anbinden, in welches eingespeist wird, wenn sich etwas an dem Status eines Generierungsprozesses ändert. Somit würde der Evaluation Container erkennen können, wenn ein Generierungsprozess beendet wurde und könnte selbstständig auf dieses Event reagieren und die generierten Daten, mit der im Event hinterlegten Methode, evaluieren. Auf diese Weise könnte man eine Kommunikation umsetzen, in der die Services mehrere Empfänger für ihre Messages haben können. Diese Art der Kommunikation ist besonders nützlich, wenn man mehrere Instanzen eines Service hat, welche sich für bestimmte Messages eines anderen Service interessieren. [vgl. 23, S. 135]

Es wäre ebenfalls möglich, die Kommunikation vorerst so simpel wie möglich zu halten und diese in der Zukunft durch eine komplexere Lösung zu ersetzen, welche dann an den Stand der Services angepasst ist, den diese zu einem späteren Zeitpunkt haben. Aufgrund des frühen Entwicklungsstandes des Projektes sind große Änderungen nicht unwahrscheinlich, wodurch sich die aktuelle Auswahl der Technologien möglicherweise unbrauchbar wird. Um dieses Szenario zu verhindern, könnte sich vorerst auf REST APIs zwischen den Services fokussiert werden und zu einem späteren Zeitpunkt, mit den Informationen aus dieser Arbeit, erneut nach einem geeigneten Muster gesucht werden.

6 Ausblick, Zusammenfassung und Fazit

6.1 Ausblick für zukünftige Arbeiten

Nach dieser Arbeit bleiben viele Möglichkeiten, an dem Thema der Microservicekommunikation weiterzuarbeiten. Im Anschluss an diese Arbeit sollten das ausgewählte Kommunikationsmuster und die Technologien zur Umsetzung in der praktischen Anwendung getestet werden. Man könnte hierfür einen Prototyp der Plattform implementieren und auf verschiedene Weisen testen.

Des Weiteren sollten in Zukunft noch weitere Generierungsmethoden genauer mit in die Entscheidung einbezogen werden. Aufgrund des frühen Entwicklungsstandes der Plattform und der Generierungsmethoden, wurde in dieser Arbeit lediglich auf einen sehr generischen Ablauf der Datengenerierung eingegangen.

Ein weiterer Bereich, welcher in zukünftigen Arbeiten behandelt werden sollte, sind die etwaigen Fehlerzustände, welche bei der IPC berücksichtigt werden sollten, beispielsweise wie damit umgegangen wird, wenn eine Message verloren geht oder die Service Instanz stirbt, welche angefragt wurde, bevor die Antwort zurückgesendet wurde.

Auch könnte man die Anforderungen noch differenzierter betrachten. Beispielsweise könnte in Betracht gezogen werden, dass manche Anforderungen eine höhere Priorität besitzen als andere, wodurch sich die Auswahl ebenfalls verändern könnte.

In [13] wird in Kapitel 11 auf die Auswahl und die Unterschiede zwischen dem Orchestration und Choreography Communication Style eingegangen. Für DaFne wurde sich bereits für den Orchetration Style entschieden. Diese Entscheidung könnte in zukünftigen Arbeiten, unter Berücksichtigung der in der Literatur empfohlenen Anwendungsfälle, validiert werden.

Am wichtigsten ist die Validierung der ausgewählten Mittel durch das Testen der implementierten Services, im Zusammenspiel mit den Kommunikationsmethoden. Hierfür

könnten beispielsweise verschiedene Messungen während des Betriebs durchgeführt werden.

6.2 Zusammenfassung

Das Ziel dieser Arbeit war es, ein geeignetes Kommunikationsmuster für die DaFne-Plattform zu ermitteln, welches die Anforderungen, welche ermittelt wurden, erfüllt und der Plattform ermöglicht, ihre Funktionen ohne Hinderungen zur Verfügung zu stellen.

Das Ergebnis dieser Arbeit ist, dass sich die IPC in der DaFne-Plattform durch das Request/Response Kommunikationsmuster und einer Kombination von synchroner Kommunikation über REST APIs und asynchroner Kommunikation mithilfe von RabbitMQ als Message Broker umsetzen lässt. Es wurde bei der Diskussion der Ergebnisse in Kapitel 5 dargelegt, dass die Anforderungen mit dieser Auswahl erfüllt werden. Sie sind eine geeignete Auswahl für die DaFne-Plattform, unter Berücksichtigung des aktuellen Entwicklungsstandes.

Nach einer Einleitung im ersten Kapitel, wurden im zweiten Kapitel theoretische Grundlagen vermittelt, um für diese Arbeit nötiges Fachwissen darzulegen, um im Folgenden auf die Auswahl des Kommunikationsmusters einzugehen. Im dritten Kapitel wurde auf die bestehenden Anforderungen eingegangen, welche, durch im Laufe dieser Arbeit zusätzlich identifizierte Anforderungen, ergänzt wurden. Zudem wurde in diesem Kapitel definiert, welche Services mit welchen anderen Services kommunizieren müssen, um die Funktion der Plattform zu gewährleisten. In Kapitel 4 wurde nun eine Auswahl für ein geeignetes Muster, sowie passenden Technologien getroffen, welche das jeweilige Muster implementieren können. Hierfür wurde, wie in der gesichteten Literatur empfohlen, die Auswahl von zwei Faktoren abhängig gemacht. Der erste Faktor ist die Notwendigkeit von synchroner oder asynchroner Kommunikation und der zweite Faktor besteht in der Anzahl der Empfänger der Nachrichten. Je nach Ausprägung dieser zwei Faktoren wurde dann je nach Service ein Muster gewählt. In Kapitel 5 wurde ein detaillierterer Blick auf die ausgewählten Lösungen gerichtet. Außerdem wurden die ausgewählten Lösungen kritisch hinterfragt und mögliche Alternativen angemerkt.

6.3 Fazit

Mit der in dieser Arbeit ermittelten Lösung für die Kommunikation zwischen den Microservices, lässt sich die Plattform wie gewünscht implementieren, da die Anforderungen an die Plattform bei den Entscheidungen berücksichtigt wurden. Es sollte trotzdem regelmäßig im Laufe der Implementierung der Plattform und den dafür nötigen Entwicklungszyklen überprüft werden, ob die in dieser Arbeit vorgeschlagene Lösung, auch für die weiterentwickelte Architektur und die ggf. veränderten Anforderungen geeignet ist. Auch wenn sich für die genannte Lösung entschieden wurde, gibt es weitere geeignete Lösungen für die IPC dieser Plattform. Es wurde mit dieser Arbeit lediglich eine geeignete Lösung ermittelt. Bevor diese Lösung implementiert wird und langfristig in die Plattform übernommen wird, sollte in folgenden Arbeiten durch Experimente und Messungen ermittelt werden, ob die gezeigte Lösung auch in der praktischen Anwendung überzeugt, da sich in dieser Arbeit lediglich auf die Theorie bezogen wurde, mit Ausnahme von kleineren Experimenten in fremden Arbeiten.

Literaturverzeichnis

[1] *Microservices Communication Frameworks—Part 1.* https://www.eqengineered.com/insights/microservices-communication-frameworks-part-1. – Zuletzt aufgerufen: 01.07.2023

[2] *RabbitMQ Webseite.* https://www.rabbitmq.com/monitoring.html. – Zuletzt aufgerufen: 01.07.2023

[3] Akbulut, Akhan ; Perros, Harry G.: Performance Analysis of Microservice Design Patterns, 2019, S. 19–27. – ISSN 1089-7801

[4] Benyamin Shafabakhsh ; Robert Lagerström ; Simon Hacks: Evaluating the Impact of Inter Process Communication in Microservice Architectures, 2020

[5] Bertelsen, Eirik ; Berthling-Hansen, Gabriel ; Bloebaum, Trude H. ; Duvholt, Christian ; Hov, Einar ; Johnsen, Frank T. ; Morch, Eivind ; Weisethaunet, Andreas H.: Federated Publish/Subscribe Services. In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, IEEE, 2018, S. 1–5. – ISBN 978-1-5386-3662-6

[6] Bhamare, Deval ; Samaka, Mohammed ; Erbad, Aiman ; Jain, Raj ; Gupta, Lav: Exploring microservices for enhancing internet QoS, 2018, S. e3445. – ISSN 21613915

[7] Bonér, Jonas: *Reactive Microservices Architecture.* 1st edition. O'Reilly Media, Inc, 2016. – URL https://permalink.obvsg.at/. – ISBN 9781491957790

[8] Calafiore, Alessia ; Palmer, Gregory ; Comber, Sam ; Arribas-Bel, Daniel ; Singleton, Alex: A geographic data science framework for the functional and contextual analysis of human dynamics within global cities, 2021, S. 101539. – ISSN 01989715

[9] CEBECİ, Kenan ; KORÇAK, Ömer: Design of an Enterprise Level Architecture Based on Microservices, 2020, S. 357–371

[10] Du, Sang G. ; Lee, Jong W. ; Kim, Keecheon: Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN. In: *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA : ACM, 2018, S. 1–6. – ISBN 9781450363853

[11] Familiar, Bob: *Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions*. Berkeley, CA : Apress, 2015. – URL http://gbv.eblib.com/patron/FullRecord.aspx?p=4086842. – ISBN 978-1-4842-1275-2

[12] Fernandes, Joel L. ; Lopes, Ivo C. ; Rodrigues, Joel J. P. C. ; Ullah, Sana: Performance evaluation of RESTful web services and AMQP protocol. In: *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, IEEE, 2013, S. 810–815. – ISBN 978-1-4673-5990-0

[13] Ford, Neal ; Richards, Mark ; Sadalage, Pramod J. ; Dehghani, Zhamak: *Software architecture : the Hard Parts: Modern trade-off analyses for distributed architectures*. Bejing and Boston and Farnham and Sebastopol and Tokyo : O'Reilly, October 2021. – ISBN 9781492086895

[14] Gördesli, Mustafa ; Varol, Asaf: Comparing Interservice Communications of Microservices for E-Commerce Industry. In: *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*, IEEE, 2022, S. 1–4. – ISBN 978-1-6654-9796-1

[15] Indrasiri, Kasun ; Siriwardena, Prabath: *Microservices for the Enterprise*. Berkeley, CA : Apress, 2018. – ISBN 978-1-4842-3857-8

[16] Johansson, Lovisa ; Dossot, David: *RabbitMQ Essentials - Second Edition*. 2nd edition. Erscheinungsort nicht ermittelbar and Boston, MA : Packt Publishing and Safari, 2020. – URL https://learning.oreilly.com/library/view/-/9781789131666/?ar. – ISBN 978-1-78913-166-6

[17] Kapil Bakshi: *Microservices-Based Software Architecture and Approaches*. IEEE, 2017. – URL http://ieeexplore.ieee.org/servlet/opac?punumber=7935773. – ISBN 978-1-5090-1613-6

[18] Karabey Aksakalli, Işıl ; Çelik, Turgay ; Can, Ahmet B. ; Tekinerdoğan, Bedir: Deployment and communication patterns in microservice architectures: A systematic literature review, 2021, S. 111014. – ISSN 01641212

[19] Kul, Seda ; Sayar, Ahmet: A Survey of Publish/Subscribe Middleware Systems for Microservice Communication. In: *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, IEEE, 2021, S. 781–785. – ISBN 978-1-6654-4930-4

[20] Kumar, Prajwal K. ; Agarwal, Radhika ; Shivaprasad, Rahul ; Sitaram, Dinkar ; Kalambur, Subramaniam: Performance Characterization of Communication Protocols in Microservice Applications. In: *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, IEEE, 2021, S. 1–5. – ISBN 978-1-6654-3545-1

[21] Massé, Mark: *REST API design rulebook: Designing consistent RESTful web service interfaces*. Beijing : O'Reilly, 2011. – URL http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10758620. – ISBN 978-1-449-31050-9

[22] Nadareishvili, Irakli ; Mitra, Ronnie ; McLarty, Matt ; Amundsen, Michael: *Microservice architecture: Aligning principles, practices, and culture*. First Edition, Second Release. Beijing and Boston and Farnham and Sebastopol and Tokyo : O´Reilly, 2016. – ISBN 978-1-491-95625-0

[23] Newman, Sam: *Monolith to Microservices*. 1st edition. Erscheinungsort nicht ermittelbar and Boston, MA : Upfront Books and Safari, 2021. – URL https://learning.oreilly.com/library/view/-/1492047848/?ar. – ISBN 9781492047841

[24] Pamela Kunert ; Tom Krause ; Olaf Zukunft ; Ulrike Steffens: A Platform Providing Machine Learning Algorithms for Data Generation and Fusion - An Architectural Approach, 2022

[25] Richards, Mark: *Software architecture patterns: Understanding common architecture patterns and when to use them*. First edition. Sebastopol, CA : O'Reilly Media, 2015. – URL https://permalink.obvsg.at/. – ISBN 978-1-491-92424-2

[26] Richardson, Chris: *Microservices patterns: With examples in Java*. Shelter Island, NY : Manning Publications, 2019. – URL https://learning.oreilly.com/library/view/-/9781617294549/?ar. – ISBN 9781617294549

[27] Sidath Weerasinghe ; Indika Perera: Optimized Strategy for Inter-Service Communication in Microservices, Science and Information (SAI) Organization Limited, 2023

[28] Sixt, Leon ; Wild, Benjamin ; Landgraf, Tim: *RenderGAN: Generating Realistic Labeled Data*

[29] Smid, Antonin ; Wang, Ruolin ; Cerny, Tomas: Case study on data communication in microservice architecture. In: Hung, Chih-Cheng (Hrsg.) ; Chen, Qianbin (Hrsg.) ; Xie, Xianzhong (Hrsg.) ; Esposito, Christian (Hrsg.) ; Huang, Jun (Hrsg.) ; Park, Juw W. (Hrsg.) ; Zhang, Qinghua (Hrsg.): *Proceedings of the Conference on Research in Adaptive and Convergent Systems*. New York, NY, USA : ACM, 2019, S. 261–267. – ISBN 9781450368438

[30] Wolff, Eberhard: *Microservices: Flexible software architekture*. Boston and New York and London and Munich : Addison-Wesley Pearson, 2017 (Always learning). – ISBN 978-0-134-60241-7

[31] Xian Jun Hong ; Hyun Sik Yang ; Young Han Kim: Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)* IEEE (Veranst.), URL http://ieeexplore.ieee.org/servlet/opac?punumber=8509497, 2018, S. 257–259

[32] Yipei Niu ; Fangming Liu ; Zongpeng Li: Load Balancing across Microservices. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications* IEEE (Veranst.), 2018, S. 198–206

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original