THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Bachelorarbeit

Konzeption, Design und Umsetzung einer Microservices-Architektur auf der Google Cloud Platform

Eine verteilte Machine-Learning-Anwendung basierend auf Serverless-Technologien

---

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik
der Technischen Hochschule Mittelhessen

von

René Gentzen
Matrikelnummer: 5262584

im Oktober 2023

Referent:	Manuel Groh, M.Sc.
Korreferent:	Prof. Dr. Uwe Meyer

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 23.10.2023
Ort, Datum

Unterschrift

Zusammenfassung

Die vorliegende Bachelorarbeit fokussiert sich auf die Transformation eines bestehenden Softwareprototyps in eine Microservices-Architektur, implementiert auf der Google Cloud Platform (GCP). Als Fallbeispiel dient die Applikation *Nachrichtenmanagement*, die Machine-Learning-Dienstleistungen für den Finanzsektor anbietet. Vier zentrale Forschungsfragen leiten die Arbeit: die konzeptionelle Aufteilung der Software in Microservices, die Bewältigung der Herausforderungen einer verteilten Architektur, die Umsetzung einer skalierbaren und erweiterbaren Architektur auf der GCP und die Auswirkungen der Verwendung von Serverless-Technologien.

In der Arbeit wird ein systematisches Dekompositionsverfahren angewendet, unterstützt durch Domain-Driven Design, um die Software in unabhängige Microservices aufzuteilen. Die speziellen Herausforderungen und erhöhte Komplexität, die mit Microservices-Architekturen einhergehen, werden durch Ansätze wie das Saga-Pattern und Infrastructure-as-Code adressiert. Durch die Fokussierung auf Serverless-Technologien und die Auswahl geeigneter GCP-Dienste wird eine Architektur geschaffen, die minimale operationale Tätigkeiten erfordert und hoch skalierbar ist. Dabei arbeitet sie in allen Belastungsszenarien kosteneffizient - von komplett ausbleibendem Nutzerverkehr bis hin zu hunderttausenden täglichen Anfragen.

Die Arbeit identifiziert zudem Herausforderungen bei der Anwendung des Saga-Patterns in Serverless-Architekturen und diskutiert alternative Ansätze für zukünftige Forschungen. Das Nachrichtenmanagement wird zum gegenwärtigen Zeitpunkt aktiv beim Partnerunternehmen weiterentwickelt. Als zukünftige Schritte stehen eine geschlossene Testphase und die darauffolgende Produktivschaltung der Applikation an.

Abstract

This Bachelor's thesis focuses on the transformation of an existing software prototype into a microservices architecture implemented on the Google Cloud Platform (GCP). The application, called *Nachrichtenmanagement* (Message Management), offers machine learning services for the financial sector and serves as a case study. Four central research questions guide the work: the conceptual partitioning of the software into microservices, the tackling of challenges inherent to a distributed architecture, the realization of a scalable and extensible architecture on the GCP, and the implications of employing serverless technologies.

A systematic decomposition method based on Domain-Driven Design is employed to split the software into independent microservices. Various concepts such as the Saga Pattern and Infrastructure-as-Code are introduced to address the challenges and complexity of a microservices architecture. By focusing on serverless technologies and selecting appropriate GCP services, an architecture is created that requires minimal operational activities, while offering excellent scalability. It also shows a high degree of cost efficiency for all levels of user traffic - from no traffic to hundreds of thousands of requests per day.

The thesis also identifies challenges in applying the Saga Pattern in serverless architectures and offers alternative approaches for future research. Nachrichtenmanagement is currently undergoing active development at the partner company. Up next are a closed testing phase and the subsequent production deployment of the application.

Inhaltsverzeichnis

1. Einleitung

Das Thema der vorliegenden Bachelorarbeit ist die Weiterentwicklung eines bestehenden Softwareprototyps hin zu einem skalierbaren, erweiterbaren und kosteneffizienten Produktivsystem zur Bereitstellung von Machine-Learning-Dienstleistungen. Dabei liegt der Fokus auf der Nutzung von Serverless-Produkten der *Google Cloud Platform* (GCP). Die Applikation namens **Nachrichtenmanagement** soll es authentifizierten Geschäftskunden ermöglichen, Nachrichten ihrer Kunden über eine REST-API zu übertragen. Diese Nachrichten werden von mehreren Machine-Learning-Modellen verarbeitet, und die Anwender erhalten eine Liste mit identifizierten Labeln, Entitäten und weiteren Ergebnissen zurück. Basierend auf diesen Ergebnissen sollen Nachrichten automatisch an Fachabteilungen weitergeleitet, oder ganze Geschäftsprozesse automatisiert werden.

1.1. Das Partnerunternehmen

Das Partnerunternehmen dieser Bachelorarbeit ist die Mittelstand.ai GmbH & Co. KG aus Gießen. Die Mittelstand.ai wurde 2020 als Tochterunternehmen der Volksbank Mittelhessen eG gegründet und ist in deren Hauptfirmensitz im Schiffenberger Tal in Gießen ansässig. Sie teilt sich in die zwei Geschäftsbereiche *Finanzinnovation* und *Data Science*, welche zu großen Teilen unterschiedlichen Projekten nachgehen. Diese Bachelorarbeit wurde im Geschäftsbereich Data Science als Anwendungsentwickler verrichtet. Die Zielgruppe der Data-Science-Sparte sind Finanzdienstleister, vorrangig Genossenschaftsbanken. Das Aushängeschild der Produktpalette sind die sogenannten *Bedarfsprognosen*. Diese Dienstleistung richtet sich an Banken, die Kundendaten übermitteln, um darauf basierend maßgeschneiderte Handlungsempfehlungen zu erhalten, die den individuellen Bedarf ihrer Kunden berücksichtigen. Weitere Produkte wie das hier behandelte *Nachrichtenmanagement* und das *Dokumentenmanagement* befinden sich in der Entwicklung. In kompetenzübergreifenden Projektteams entwickeln Data Scientists dafür die nötigen Machine-Learning- und Deep-Learning-Technologien, während Anwendungsentwickler diese in Applikationen umsetzen. Dabei werden Methoden der agilen Softwareentwicklung verwendet.

1.2. Hintergrund und Motivation

Die Motivation für das Projekt Nachrichtenmanagement entspringt dem Bedarf der Zielgruppe der Genossenschaftsbanken nach einer effizienteren Verarbeitung eingehender Kundenkorrespondenz. Derzeit wird diese in ein Sammelpostfach geleitet und von dort manuell weiterverteilt, was erheblichen Arbeitsaufwand und Kosten für die Banken verursacht. Zudem wird jede dritte E-Mail unbearbeitet gelöscht, da es sich um Newsletter, Spam oder sonstige unerwünschte Nachrichten handelt.

Das Nachrichtenmanagement kann von Kunden zur Unterstützung von *intelligentem Routing* genutzt werden. Gemäß einem Bericht von *ccw.eu* ermöglicht intelligentes Routing durch KI-gestützte Ergebnisse „eine deutlich schnellere Bearbeitung von Vorgängen und optimiert sowohl die Reaktionszeit als auch die Erstlösungsquote“ [1]. Außerdem liege der „Mehrwert intelligenter Fachdatenerfassung [...] nicht nur in der erheblichen Zeitersparnis. KI-Systeme machen auch deutlich weniger Fehler als menschliche Mitarbeiter, die bei Routineaufgaben ermüden können“ [1].

Des Weiteren ist die Transformation des Prototyps in eine Microservices-Architektur motiviert durch bestehende Einschränkungen des Prototyps hinsichtlich Performance, Skalierbarkeit, Erweiterbarkeit und Kosteneffizienz. Die Herausforderungen in diesen Bereichen sollen durch die geplante Lösung adressiert werden. Die Microservices-Architektur hat in den letzten Jahren stark an Popularität gewonnen (vgl. Abb. 1.1) und wird in einem Großteil großer Unternehmen verwendet [2]. Gleichwohl gibt es kritische Stimmen, die auf die erhöhte Komplexität und die neuen Herausforderungen einer verteilten Anwendung hinweisen [3]. Daher ist es von essenzieller Bedeutung, bewährte Microservices-Design-Patterns zu implementieren, um die Komplexität zu handhaben und die versprochenen Vorteile zu realisieren.

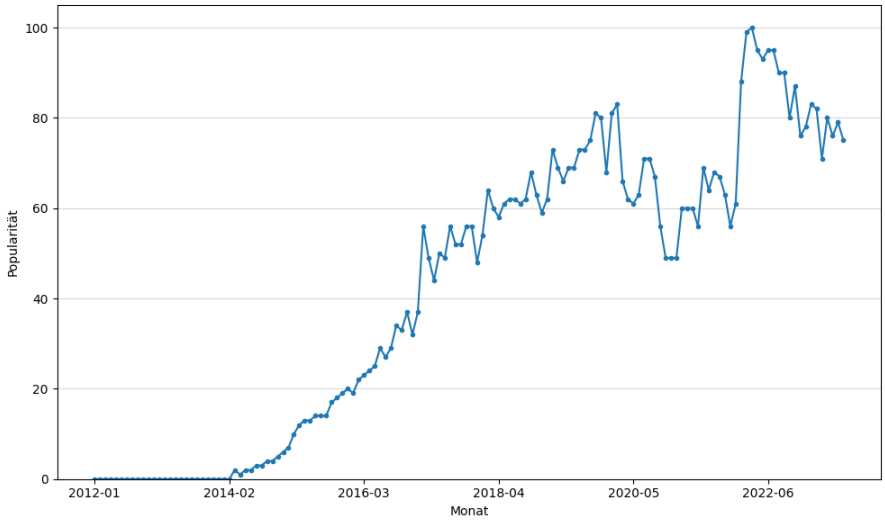


Abbildung 1.1.: Popularität des Suchbegriffs ‘Microservices’ auf Google (weltweit). Quelle: Daten von [4]; eigene Darstellung.

1.3. Zielsetzung der Arbeit

Das Hauptziel dieser Bachelorarbeit besteht darin, eine Lösung für die Weiterentwicklung des bestehenden Softwareprototyps in eine skalierbare Microservices-Architektur zu erarbeiten. Dabei sollen folgende Teilziele erreicht werden:

1. Entwicklung einer skalierbaren Microservices-Architektur: Der Prototyp soll in unabhängige Microservices aufgeteilt werden, um horizontale Skalierung und effiziente Ressourcennutzung zu ermöglichen.

2. Auswahl der Google Cloud-Produkte: Geeignete Dienste der GCP sollen identifiziert und zur Umsetzung benutzt werden, um die Bereitstellung und Skalierung der Anwendung zu optimieren und operationale Tätigkeiten zu minimieren.
3. Gewährleistung der Erweiterbarkeit: Die Architektur soll so gestaltet sein, dass neue Machine-Learning-Dienstleistungen und Funktionen nahtlos in die Anwendung integriert werden können.
4. Analyse von Serverless-Technologien: Die Nutzung von Serverless-Technologien soll in der umgesetzten Lösung bewertet werden, unter Berücksichtigung der Auswirkungen auf die Systemarchitektur.

1.3.1. Forschungsfragen

Die Zielsetzung dieser Arbeit führt zu folgenden Fragestellungen:

1. *Wie lässt sich bestehende Software konzeptionell in Microservices aufteilen?*
2. *Welche Lösungen existieren, um die Schwächen einer verteilten Architektur, wie beispielsweise fehlende atomare Operationen über mehrere Services hinweg, effektiv zu bewältigen?*
3. *Wie kann eine skalierbare und erweiterbare Microservices-Architektur auf der Google Cloud Platform umgesetzt werden, ohne dabei einen signifikanten operationalen Aufwand zu generieren?*
4. *Welche Auswirkungen hat die Verwendung von Serverless-Technologien auf die Anwendbarkeit bewährter Microservices-Design-Patterns?*

1.3.2. Abgrenzung der Problemstellung

Die Mittelstand.ai nutzt ausschließlich die Google Cloud Platform als Deployment-Plattform. Aus diesem Grund werden Cloud-Plattformen anderer Cloud-Anbieter nicht tiefergehend betrachtet. Auch erfolgt keine vollständige Beschreibung aller Implementationsmöglichkeiten einer Microservices-Architektur. Google Cloud Services und andere Technologien, die aufgrund betrieblicher Entscheidungen nicht verwendet werden können, werden nicht näher erläutert. Dazu zählen insbesondere die Containerorchestrierungsplattform Kubernetes und das mannigfaltige Ökosystem aus Microservices-Technologien, die auf dieser und anderen selbst verwalteten Plattformen aufbauen.

Die Umsetzung der Anwendung wird von einem kleinen Projektteam durchgeführt. Während der Bearbeitungszeit dieser Arbeit können nicht alle geplanten Features implementiert werden. Im Kapitel zur Implementierung wird ein Entwicklungsstand der Anwendung behandelt, der als *minimal funktionsfähige Iteration*, auch MVP genannt, gilt. In späteren Iterationszyklen wird dieses MVP um die noch ausstehenden Funktionalitäten erweitert werden, was allerdings nicht Teil dieser Arbeit ist. In der Konzeptionierung der Anwendungsarchitektur sind allerdings alle geplanten Features berücksichtigt.

Der Fokus dieser Arbeit liegt nicht auf dem feingranularen Softwaredesign einzelner Anwendungsteile. Sie soll einen breiteren Blick auf das Gesamtkonzept einer Microservices-Architektur basierend auf Serverless-Technologien geben. Codeausschnitte und Konfigu-

rationsdetails einzelner Services werden daher wenig gezeigt. Auch der agile Softwareentwicklungsprozess dieses Projekts wird nicht näher erläutert. Dazu zählt insbesondere die Übertragung von Entwicklungsschritten in Iterationszyklen.

Es sei außerdem angemerkt, dass die konkrete Nutzung der zurückgespielten der Machine-Learning-Ergebnisse und Machine Learning im Allgemeinen außerhalb des Themenbereichs dieser Arbeit liegen. Es wird keine Bewertung der KI-Vorhersagen oder tiefgehende Diskussion ihrer möglichen Nutzungspotenziale erfolgen.

1.4. Aufbau und Vorgehensweise

Die Bearbeitung der Bachelorarbeit gliedert sich in die folgenden Schritte:

1. Theoretische Grundlagen: Behandlung relevanter Grundlagen zu verteilten Systemen und Microservices-Architekturen, sowie Betrachtung des Stands der Technik.
2. Analyse des Softwareprototyps: Untersuchung der vorhandenen Softwarearchitektur, Abgleich mit den zukünftigen Anforderungen an das Produktivsystem, Identifikation von Schwachstellen und Herausforderungen.
3. Konzeption der Microservices-Architektur: Konzeption einer geeigneten Architektur, die auf Microservices basiert und den Anforderungen an Performance, Skalierbarkeit, Erweiterbarkeit und Kosteneffizienz gerecht wird.
4. Design und Implementierung: Evaluation der Dienste der Google Cloud Platform in Hinblick auf ihre Eignung für das Projekt, Umsetzung eines MVPs der konzeptionierten Architektur unter Verwendung der ausgewählten Google-Cloud-Produkte.
5. Evaluation und Optimierung: Bewertung der Skalierbarkeit, Performance und Kosteneffizienz der entwickelten Lösung. Identifikation möglicher Optimierungspotenziale.

2. Theoretische Grundlagen

2.1. Monolithische Softwarearchitekturen

2.1.1. Grundlagen monolithischer Softwarearchitekturen

Monolithische Softwarearchitekturen stellen einen der klassischen Ansätze zur Organisation von Anwendungscode dar. Zu diesem Modell können sämtliche Softwaresysteme gezählt werden, deren gesamte Funktionalitäten und Komponenten im selben Bereitstellungsprozess (*Deployment*) aufgesetzt werden müssen. Klassischerweise sind sie in einem einzigen Softwareartefakt, dem sogenannten *Monolithen*, gebündelt, aber die Definition kann auf verteilte Architekturen ausgeweitet werden, solange die Charakteristik eines unteilbaren Deployments erfüllt ist [5].

Monolithische Architekturen bieten einige Vorteile, die sie besonders für kleinere Projekte oder Prototypen attraktiv machen. Der initiale Planungsaufwand ist oft geringer als bei verteilten Architekturen, und die technische Komplexität bleibt anfangs überschaubar. Transaktionale Anforderungen und Datenkonsistenz lassen sich in der Regel einfach durch den Einsatz einer einzigen Datenbank erfüllen. Zudem sind die Entwicklungs- und Testprozesse weniger komplex, da alle Abhängigkeiten in der gleichen Codebasis verwaltet werden [5], [6].

Trotz ihrer Einfachheit und Beliebtheit sind monolithische Architekturen nicht ohne Herausforderungen, die mit wachsender Komplexität einer Anwendung immer mehr ins Gewicht fallen. Richardson bezeichnet diesen Umstand als „Monolithic Hell“, der er unter anderem folgende Eigenschaften zuweist [6]:

- Die Komplexität der Anwendung erreicht einen Punkt, an dem sie die Fähigkeit eines einzelnen Menschen übersteigt, das System in seiner Gesamtheit zu verstehen.
- Die Codebasis wächst immer weiter an und kann sehr groß werden, was die lokale Entwicklung verlangsamt und mehr Hardwareressourcen erfordert.
- Der Build- und Deployment-Prozess einer großen Codebasis ist zeitaufwendig, was schnelle und häufige Updates erschwert.
- Hohe Kopplung zwischen Komponenten kann die Skalierbarkeit und Erweiterbarkeit der Anwendung einschränken.
- Es besteht die Gefahr der Technologiebindung. Im schlimmsten Fall kann eine veraltete Technologie den gesamten Monolithen obsolet machen, was einen *Big-Bang-Rewrite* erfordert. Dabei muss die gesamte Anwendung in einem Stück neu entwickelt werden, was mit erheblichen Kosten und Risiken verbunden ist [7, S. 430 f.].

- Die Wartbarkeit eines Monolithen kann im Laufe der Zeit abnehmen, insbesondere wenn keine klare Struktur oder Modularisierung vorhanden ist. Ein häufiges Anti-Pattern in diesem Kontext ist der *Big Ball of Mud*, der in Abschnitt 2.1.2 beschrieben wird.

2.1.2. Der Big Ball of Mud

Das Phänomen des *Big Ball of Mud* ist eine weitverbreitete Softwarearchitektur, die als *Anti-Pattern* klassifiziert wird. Sie entsteht, wenn eine Softwareanwendung ohne klare Struktur oder Designprinzipien entwickelt wird. Die Hauptursachen für die Entstehung eines Big Ball of Mud sind Zeitdruck, mangelndes Fachwissen und die Abwesenheit einer klaren Architekturvision. Die Software wird schrittweise und ohne ausreichende Planung erweitert, was zu einer komplexen und schwer wartbaren Codebasis führt. Der Big Ball of Mud weist in der Regel alle Nachteile einer monolithischen Softwarearchitektur in sehr ausgeprägter Form auf [8].

Die Folgen eines Big Ball of Mud für den Entwicklungsprozess und den Lebenszyklus einer Anwendung sind schwerwiegend. Sie umfassen erhöhte Fehleranfälligkeit, erschwerte Wartung und eine geringe Erweiterbarkeit des Systems. Darüber hinaus steigen die Kosten für die Weiterentwicklung exponentiell an. Der Entstehung eines Big Ball of Mud muss durch proaktive Maßnahmen entgegengewirkt werden. Dazu gehören eine sorgfältige Planung der Software und die Einhaltung von Designprinzipien. Auch die Einteilung eines Systems in kleinere, unabhängige Services (siehe Abschnitt 2.3) kann der Entstehung dieser Architektur Einhalt gebieten. Allerdings besteht dann die Gefahr, den noch folgenschwereren Distributed Big Ball of Mud zu erschaffen [8], [9].

2.2. Verteilte Softwarearchitekturen

Verteilte Softwarearchitekturen repräsentieren einen Paradigmenwechsel in der Organisation und Strukturierung von Softwareprojekten, indem sie die Modularisierung und Verteilung von Komponenten über verschiedene physische oder virtuelle Maschinen nutzen, siehe Abb. 2.1. Im Gegensatz zu monolithischen Architekturen bieten verteilte Systeme unter anderem eine höhere Performance, Skalierbarkeit und Verfügbarkeit [8]. Dabei gehen diese Vorteile oft mit einer erhöhten Komplexität in Bezug auf Entwicklung, Deployment und Betrieb einher [6].

Die Hauptvorteile verteilter Architekturen liegen in ihrer Skalierbarkeit, Performance und Flexibilität. Durch die Entkopplung von Komponenten können einzelne Module unabhängig voneinander skaliert und aktualisiert werden. Dies ermöglicht eine effizientere Ressourcennutzung und erleichtert die Anpassung an veränderte Lastbedingungen und Geschäftsanforderungen. Darüber hinaus fördert die modulare Struktur die Wiederverwendbarkeit von Code und erleichtert die Integration neuer Technologien [5].

Zu den Herausforderungen verteilter Systeme gehören insbesondere die Koordination und Kommunikation zwischen den verteilten Komponenten, die Handhabung von Netzwerklatenzen und die Gewährleistung von Datenkonsistenz in einem verteilten Datenmodell [8]. Die Kommunikation zwischen den verteilten Komponenten erfolgt in der Regel über Netzwerkprotokolle, was eine zusätzliche Latenz in Funktionsaufrufe einbringt.

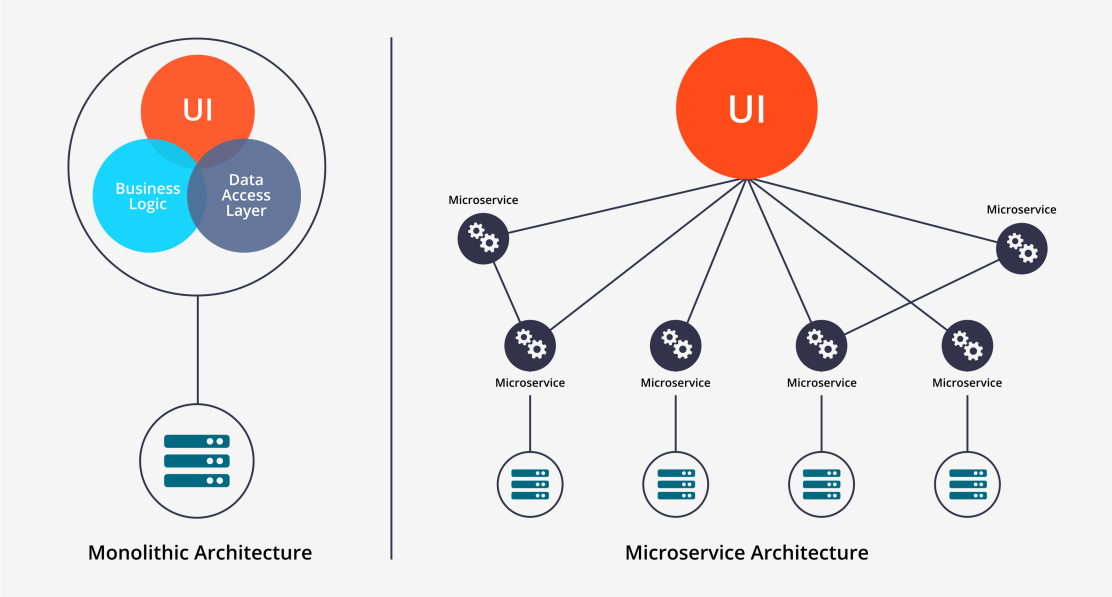


Abbildung 2.1.: Monolithische Architektur und Microservices. Quelle: [10]

Diese Netzwerkabhängigkeit erfordert robuste Fehlerbehandlungsmechanismen, um die Systemverfügbarkeit und -zuverlässigkeit sicherzustellen. In verteilten Architekturen ist zudem die Aufrechterhaltung der Datenkonsistenz eine nicht-triviale Herausforderung. Traditionelle ACID-Transaktionen sind nicht direkt anwendbar und Entwickler müssen alternative Ansätze wie *Eventual Consistency* und spezialisierte Transaktionsmodelle wie *Sagas* in Betracht ziehen [8], die in Abschnitt 2.3.4 erklärt werden. Ein weiterer Faktor, der bei der Entscheidung für eine verteilte Architektur berücksichtigt werden muss, ist erhöhte Betriebskomplexität. Monitoring, Logging und Tracing sind wesentlich komplexer, als in monolithischen Systemen und erfordern oft den Einsatz spezialisierter Tools und Plattformen [8].

2.3. Die Microservices-Architektur

Die *Microservices-Architektur* stellt eine spezialisierte Form einer verteilten Softwarearchitektur dar und kann als Weiterentwicklung der *Service-Oriented Architecture (SOA)* betrachtet werden [5]. Laut Fowler und Lewis wurde der Begriff *Microservice* im Jahr 2011 geprägt [11] und seitdem hat sich ein Architekturstil herauskristallisiert, der unter anderem durch die folgenden charakteristischen Eigenschaften definiert ist [5], [11]:

- Feingranulare Zerlegung in eigenständige Services.
- Organisation der Services entlang von Geschäftsfunktionen oder Domänen.
- Jeder Service übernimmt genau eine Domänenfunktionalität (Implementierung eines *Bounded Contexts*, siehe Abschnitt 5.1).
- Kapselung der Implementationsdetails hinter wohldefinierten APIs.
- Kommunikation über leichtgewichtige Protokolle wie REST und gRPC.

- Minimale geteilte Abhängigkeiten zwischen den Services.
- Unabhängige Bereitstellung, Skalierung und Ersetzung der einzelnen Services.
- Isolierte Ausführung in eigenen Umgebungen, beispielsweise in Containern, virtuellen Maschinen oder auf dedizierten Servern.

Die Microservices-Architektur setzt einen besonderen Schwerpunkt auf die Gewährleistung von Wartbarkeit (*Maintainability*), Testbarkeit (*Testability*) und Bereitstellbarkeit (*Deployability*). Diese Aspekte werden von Microservices besonders effektiv adressiert und ergänzen die klassischen Qualitätsanforderungen wie Skalierbarkeit (*Scalability*), Zuverlässigkeit (*Reliability*) und Sicherheit (*Security*) [9].

Fowler erklärt in [12], dass die Microservices-Architektur trotz ihrer Vorteile auch eine Reihe von Herausforderungen mit sich bringt, die er als „MicroservicesPremium“ bezeichnet. Diese umfassen unter anderem die Komplexität der automatischen Bereitstellung (CI/CD), Monitoring, Fehlerbehandlung und den Umgang mit Eventual Consistency in einem verteilten System. Für diese Herausforderungen existieren zwar etablierte Lösungsansätze, jedoch erhöht ihre Implementierung den Entwicklungsaufwand beträchtlich.

2.3.1. Kommunikationsformen in Microservices-Architekturen

In einer Microservices-Architektur müssen die Kommunikationsmechanismen zwischen Softwaremodulen überdacht werden, da diese Module als eigenständige Services realisiert werden. Dies führt zu einer Verschiebung von *In-Process*-Operationen, bei denen Aufrufe innerhalb desselben Systemprozesses erfolgen, zu *Inter-Process*-Operationen, bei denen Aufrufe über ein Netzwerk erfolgen müssen [11], [13].

Diese Transformation ist nicht trivial und bringt eine Reihe von technischen und konzeptionellen Herausforderungen mit sich. Zum Beispiel können Netzwerklatenzen und -ausfälle die Leistung und Zuverlässigkeit eines Systems erheblich beeinträchtigen. Zudem erfordert die Kommunikation über ein Netzwerk zusätzliche Sicherheitsmaßnahmen, um die Integrität und Vertraulichkeit der Daten zu gewährleisten [8].

Darüber hinaus müssen Entwickler auch die Komplexität der Netzwerktopologie und die damit verbundenen administrativen Herausforderungen berücksichtigen. In einer Microservices-Architektur können Services auf verschiedenen Servern oder sogar in verschiedenen Rechenzentren laufen, was die Netzwerktopologie komplex und dynamisch macht. Dies erfordert eine sorgfältige Planung und Überwachung, um sicherzustellen, dass alle Services effizient miteinander kommunizieren können [8].

Es müssen zudem Aspekte wie Datenkonsistenz, Logging und Schnittstellenverwaltung berücksichtigt werden. Um die Konsistenz der Daten zwischen den Services zu gewährleisten, können beispielsweise verteilte Transaktionsmechanismen erforderlich sein. Für das Logging müssen Mechanismen implementiert werden, die es ermöglichen, Logs von verschiedenen Services zu korrelieren, um das Debugging und die Überwachung zu erleichtern. Schließlich muss die Schnittstellenverwaltung sicherstellen, dass alle Microservices mit den richtigen Versionen der APIs kommunizieren, um Inkompatibilitäten und Fehler zu vermeiden. Das bedeutet, dass die Schnittstellen zwischen den Services klare Verträge definieren müssen, die sowohl die API-Versionen als auch die Datenstrukturen spezifizieren, die von den Services erwartet werden [8].

Grundsätzlich lässt sich Inter-Service-Kommunikation in *synchrone* und *asynchrone* Kommunikationsformen unterteilen, wie in Abb. 2.2 dargestellt. Diese wiederum lassen sich weiter klassifizieren: Synchrone Kommunikation folgt immer dem Request/Response-Modell. Asynchrone Kommunikation kann über Request/Response, eventgesteuert, oder über geteilte Daten erfolgen [13].

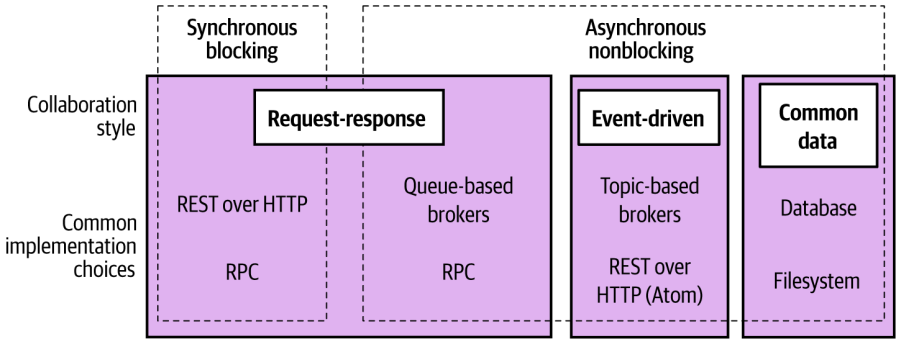


Abbildung 2.2.: Kommunikationsmechanismen in verteilten Systemen inklusive Technologien zur Implementierung. Quelle: [14, Abb. 4.1]

Jede dieser Kommunikationsformen kann mit verschiedenen Technologien umgesetzt werden, wie in Abschnitt 2.3.2 und Abschnitt 2.3.3 erklärt wird. Abschnitt 5.2 behandelt die Auswirkungen der Verwendung moderner Function-as-a-Service- (FaaS) und Serverless-Technologien auf die Anwendbarkeit dieser Pattern und es werden alternative Lösungsansätze für solche Technologie-Stacks vorgestellt.

2.3.2. Synchrone Kommunikationsmechanismen

Die synchrone oder *blockierende* Kommunikation ist eine Kommunikationsform, bei der ein Service eine Anfrage an einen anderen Service sendet und seine Ausführung *blockiert*, bis er eine Antwort erhält. Diese Antwort kann beispielsweise eine Erfolgsmeldung oder Daten enthalten. Übliche Implementierungen verwenden *REST over HTTP* oder *RPC (Remote Procedure Calls)* [15].

Synchrone Kommunikation ist die bekannteste und am leichtesten umzusetzende Kommunikationsform. Fehler in einem angesprochenen Service können vom Aufrufer direkt durch entsprechende Antworten erkannt und behandelt werden. Der Technologie-Stack der gesamten Anwendung wird einfach gehalten, da keine zusätzliche Middleware wie ein Message Broker eingerichtet werden muss [13], [15].

Diese Art der Kommunikation ist allerdings auf das *Request/Response-Modell* und somit auf „One-to-One“-Kommunikation beschränkt [15]. Weiterhin bindet sie mehr Systemressourcen durch die Aufrechterhaltung von Verbindungen, bis eine Antwort erhalten wurde. Ist mit langen Antwortzeiten zu rechnen, kann dies zu beträchtlichem Overhead führen [7, S. 268].

Wiederholungsversuche bei Verbindungsfehlern und Zeitüberschreitungen müssen außerdem in jedem Service implementiert werden. Sie sind essenziell, damit Ressourcen nicht unendlich lange gebunden bleiben und das System eine gewisse Fehlertoleranz aufweist [13]. Darüber hinaus wird die Gesamtverfügbarkeit des Systems erheblich verringert, da

sie sich multiplikativ aus der Verfügbarkeit jedes einzelnen Services zusammensetzt [7, S. 104].

Zuletzt müssen Services die URLs all ihrer Ziele jederzeit kennen. Verändern sich diese häufig, wie es in modernen verteilten Anwendungen durchaus üblich ist, muss eine Lösung für *Service Discovery* implementiert werden [13], [15].

2.3.3. Asynchrone Kommunikationsmechanismen

Die asynchrone oder *nicht-blockierende* Kommunikation setzt auf die Entkopplung von Anfrage und Antwort. Hierbei werden Anfragen an andere Services gesendet, ohne auf eine Antwort zu warten, wodurch ein Service sofort weitere Anfragen bearbeiten kann. Es wird in der Regel angenommen, dass von jedem Service mehr als eine Instanz gleichzeitig existiert und dass die Instanz, die eine asynchrone Anfrage gestellt hat, zu einem späteren Zeitpunkt nicht mehr verfügbar ist. Dies erfordert ein grundlegendes Umdenken bei der Anwendungslogik. Während synchron kommunizierende Systeme Zwischenergebnisse mehrteiliger Arbeitsabläufe (*State*) im Arbeitsspeicher halten können, müssen asynchrone Systeme diese außerhalb der Serviceinstanzen, zum Beispiel in Datenbanken, persistieren und zur Bearbeitung jeder Anfrage von dort wiederherstellen [13].

Request/Response-Modell: Genau wie bei synchroner Kommunikation ist auch bei asynchroner Kommunikation ein Request/Response-Modell umsetzbar. Die Besonderheit gegenüber synchroner Kommunikation liegt dabei darin, dass aus vorher genannten Gründen jede beliebige Instanz eines Services eine Antwort verarbeiten können muss. Dafür wird die Kommunikation oftmals über Messaging-Services wie Message Broker umgesetzt, die Nachrichten puffern können. Asynchrone Request/Response-Systeme sind komplexer als ihre synchronen Äquivalente, können dafür aber ein größeres Anfragevolumen bearbeiten und sind durch die *zeitliche Entkopplung* der Teilsysteme in ihrer Gesamtheit hochverfügbar [13].

Event-Driven Architecture: Neben dem Request/Response-Modell ist die eventgesteuerte Architektur (*Event-Driven Architecture, EDA*) eine gängige asynchrone Kommunikationsform in verteilten Systemen [16]. In seinem Buch *Building Microservices* erläutert Sam Newman, dass EDA eine Architekturform ist, bei der Services auf systeminterne Ereignisse (*Events*) reagieren und ihrerseits Events zur Weitergabe von Informationen erzeugen [13]. Newman betont, dass diese Events bereits abgeschlossene Vorgänge im System repräsentieren. Im Unterschied zum Request/Response-Modell sind in der EDA die Empfänger der Events für den sendenden Service unbekannt. Daher kann ein Service nicht mit Sicherheit sagen, ob und wie auf seine Events reagiert wird.

Newman sieht Events als das Gegenstück zu klassischen Anfragen und hebt hervor, dass Services selbst entscheiden müssen, auf welche Events sie reagieren. Die Arbeitsabläufe in solchen Systemen sind nicht von übergeordneter Stelle aus steuerbar, sondern abhängig von der Kooperationsbereitschaft der nachfolgenden Services. Laut Newman zeichnen sich EDA-Systeme durch eine hohe Entkopplung und Verfügbarkeit aus, da Nachrichten durch Message Broker zwischengespeichert werden können und die Services zeitlich voneinander entkoppelt sind. Zudem sind sie hochgradig erweiterbar, da neue

Event-Empfänger ohne Änderungen am bestehenden System hinzugefügt werden können. Allerdings weist Newman auch auf die erhöhte Komplexität solcher Systeme hin, da die Kommunikationspfade nicht immer eindeutig sind und das Veröffentlichen von Events unerwünschte Nebeneffekte in anderen Services auslösen kann.

2.3.4. Das Saga-Pattern

In Microservices-Architekturen können die ACID-Eigenschaften einer monolithischen Datenhaltung nicht alleine zur Wahrung der Datenkonsistenz des Gesamtsystems genutzt werden. Um eine hohe Kopplung zwischen Services zu vermeiden, ist es üblich, jedem Service eine eigene Datenbank zur Verfügung zu stellen. Innerhalb dieser Datenbank kann der Service ACID-Transaktionen nutzen, sofern die Datenbank diese unterstützt. Geschäftsprozesse laufen aber häufig zwischen mehreren Microservices ab, die zusammen einen konsistenten Gesamtzustand aufrechterhalten müssen [17].

Eine Herangehensweise dafür ist das *Saga-Pattern*. Eine Saga umfasst eine Reihe lokaler ACID-Transaktionen in verschiedenen Microservices. Für die Einzelschritte einer Saga ist eine feste Reihenfolge definiert, wobei Transaktionen auch parallel stattfinden können, sofern sie nicht voneinander abhängig sind. War ein Schritt erfolgreich, wird der nächste Schritt der Saga durch eine asynchron versendete Nachricht ausgelöst. Schlägt ein Schritt fehl, müssen Services alle bereits erfolgten Änderungen in umgekehrter Reihenfolge durch *kompensierende Transaktionen* wieder rückgängig machen. Für jeden Schritt, der Daten manipuliert, muss also eine solche kompensierende Transaktion eingerichtet und zuverlässig im Fehlerfall angestoßen werden. Nicht alle Schritte müssen kompensiert werden. Reine Lesezugriffe brauchen zum Beispiel keine kompensierenden Transaktionen. Einzelschritte sollten aber keine Operationen enthalten, die nicht rückgängig gemacht werden können (*Side Effects*) [18].

Sagas funktionieren nach dem Prinzip der *Eventual Consistency* (schlussendliche / eventuelle Konsistenz). In einem verteilten System kann nicht immer garantiert werden, dass alle Daten in Echtzeit konsistent sind. Dennoch verspricht die Eventual Consistency, dass das System letztendlich einen konsistenten Zustand erreicht, wenn keine neuen Änderungen von außen mehr vorgenommen werden. Dies ist insbesondere wichtig in Systemen, die eine hohe Verfügbarkeit und Fehlertoleranz benötigen [19]. Die kompensierenden Transaktionen in Sagas sind ein Mechanismus, der dazu beiträgt, diese Eventual Consistency zu erreichen. Sie stellen sicher, dass fehlgeschlagene oder halb ausgeführte Prozesse rückgängig gemacht werden können, sodass das System schließlich einen konsistenten Zustand erreicht [18].

Sagas können auf zwei Arten koordiniert werden [18]:

1. Choreografie: Teilnehmer stoßen selbst die nächsten Schritte einer Saga an, indem sie die nötigen Events aussenden.

2. Orchestration: Ein zentraler Orchestrator steuert eine Saga, sendet Befehle an die Teilnehmer und erhält ihre Antworten.

Choreografie

In einer Choreografie sendet ein Service bei jeder Änderung von Anwendungsdaten ein Event aus, das die durchgeführte Änderung widerspiegelt. Services sind theoretisch lose aneinander gekoppelt, da sie nur Message-Kanäle abhören müssen, deren Events für sie wichtig sind. Von anderen Teilnehmern der Anwendung müssen sie keine Kenntnis haben. Für außenstehende Beobachter kann es dadurch aber schwierig sein, den Ablauf choreografiebasierter Sagas nachzuvollziehen und zyklische Abhängigkeiten können leichter auftreten. Auch besteht die Gefahr, dass Sender und Empfänger über die gemeinsam verwendeten Eventtypen effektiv doch stark aneinander gekoppelt sind und zusammen verändert und neu bereitgestellt werden müssen [7].

Orchestration

Orchestrationsbasierte Sagas nutzen einen endlichen Automaten, den zentralen *Orchestrator*, um Arbeitsschritte über Microservices hinweg zu koordinieren. Der Orchestrator sendet asynchron durch Messaging Befehle an Services, erhält Antwortnachrichten und bestimmt darauf basierend die nächsten Schritte einer Saga. Dadurch ist der gesamte Ablauf einer Saga an einem Ort einsehbar und zyklische Abhängigkeiten zwischen Teilnehmern werden vermieden. Anders als beim Choreografie-Pattern sind Services nicht über Eventtypen aneinander gebunden, aber sie müssen sich an ihre Schnittstellendefinition mit dem Orchestrator halten. Die Trennung der Verantwortlichkeiten ist klarer, da Koordination im Orchestrator und Domänenlogik in den Teilnehmern implementiert ist [7].

Herausforderungen von Sagas

Sowohl Choreografie als auch Orchestration erfordern die Behandlung zweier Problemstellungen [18]: Sagas fehlt auf das Gesamtsystem betrachtet die Eigenschaft der *Isolation* – sie bieten nur die *ACD*-Eigenschaften klassischer ACID-Transaktionen (Atomicity, Consistency und Durability). Fehlende Isolation kann dazu führen, dass eine Saga mit Zwischenergebnissen einer parallel laufenden Saga arbeitet, welche später rückgängig gemacht werden. Eine effektive Gegenmaßnahme dafür ist die Verwendung eines *Semantic Locks* (semantische Sperre). Das kann zum Beispiel eine spezielle Spalte in einer Datenbanktabelle sein, die einen Status wie „pending“, „done“ oder „failed“ enthält. Anhand dieses Statusindikators (*Flags*) können parallel lesende Transaktionen unbestätigte Datensätze ignorieren [18].

Die zweite Schwierigkeit bei Sagas ist das atomare Versenden eines Events nach Abschluss einer lokalen Transaktion. Die wenigsten Datenbanksysteme unterstützen dieses als *Transactional Messaging* bekannte Feature von sich aus. Die Lösung ist die Verwendung einer sogenannten *Outbox Table*. Lokal verwendete ACID-Transaktionen werden um einen Schritt erweitert: das Einfügen eines Events in die *Outbox Table*. Diese Tabelle wird von einem externen *Event Publisher* regelmäßig abgefragt, neue Einträge als Events veröffentlicht und dann aus der Tabelle gelöscht. Alternativ zu diesem Polling-Ansatz können neue Events über die Überwachung der Datenbank-Logs (*Transaction Log Tailing*) identifiziert und gleichermaßen veröffentlicht werden [15].

2.3.5. Fehlerbehandlung in Microservices-Architekturen

In einer Microservices-Architektur trägt robuste Fehlerbehandlung dazu bei, die Integrität, Zuverlässigkeit und Resilienz des Systems zu gewährleisten. Im Kontext verteilter Systeme sind Herausforderungen wie Netzwerklatenz, Serviceausfälle und Dateninkonsistenz häufige Problemstellungen. Speziell entworfene Pattern und Techniken zur Fehlerbehandlung ermöglichen es dem System, auf auftretende Probleme angemessen zu reagieren und deren Auswirkungen zu minimieren. Im Folgenden werden einige dieser Schlüsseltechniken erläutert [14, Kap. 12]:

- **Wiederholungsversuche**: In einem verteilten System können Netzwerkfehler auftreten. Durch Implementierung von Wiederholungsversuchen kann das System automatisch versuchen, Operationen erneut durchzuführen, anstatt sofort zu scheitern. Es ist jedoch wichtig, eine geeignete Strategie für Wiederholungsversuche zu wählen, um eine Überlastung des Netzwerks zu vermeiden.
- **Zeitüberschreitungen**: Um zu verhindern, dass ein System auf die Antwort eines Services, der überlastet oder fehlerhaft ist, unendlich lange wartet, sollten Zeitüberschreitungen implementiert werden. Dies stellt sicher, dass Ressourcen nicht unnötig gebunden werden und verbessert die Resilienz des Systems.
- **Idempotenz**: Viele Message Broker garantieren mindestens eine Zustellung jeder Nachricht (*At-least-once-Garantie*). Dies hat jedoch zur Folge, dass im Zweifelsfall eine Nachricht mehr als ein Mal zugestellt werden kann. Daher ist es wichtig, dass Operationen *idempotent* gestaltet werden. Das bedeutet, dass nach wiederholten Ausführungen derselben Operation das Ergebnis unverändert bleibt, ohne Nebenwirkungen zu verursachen. Dies erleichtert die Fehlerbehandlung erheblich und ermöglicht sichere Wiederholungsversuche.

2.3.6. Monitoring und Logging

Genau wie die Fehlerbehandlung sind Monitoring und Logging wesentliche Bestandteile zur Gewährleistung der Leistung und Zuverlässigkeit einer Microservices-Architektur. Dazu zählen unter anderem folgende Aspekte [14, Kap. 10]:

- **Log-Management**: Eine zentrale Plattform für das Log-Management ermöglicht es, Logs von verschiedenen Services zu aggregieren und zu analysieren. Dies erleichtert die Fehlerbehebung und die Erkennung von Mustern oder Anomalien. Moderne Log-Analyse-Tools bieten unter anderem SQL-artige Queries und automatisch generierte Erkenntnisse.
- **Tracing**: In einem Microservices-System können Anfragen durch viele verschiedene Services fließen. Tracing-Tools helfen dabei, den Weg einer Anfrage durch das System zu verfolgen, was die Fehlerbehebung und Optimierung erleichtert.
- **Metriken und Dashboards**: Durch das Sammeln von Metriken aus verschiedenen Teilen des Systems können Dashboards erstellt werden, die einen Überblick über die Leistung und den Gesundheitszustand des Systems bieten. Dies ermöglicht eine proaktive Überwachung und schnelle Reaktion auf mögliche Probleme.

Die Anbieter der großen Cloud-Plattformen bieten für diese Themen tief integrierte Tools wie Googles Operations Suite, die in diesem Projekt verwendet wird.

2.4. Serverless Computing

Serverless Computing, oftmals nur *Serverless*, oder manchmal auch als *serverlos* übersetzt, ist ein Deployment-Modell aus dem Bereich des Cloud-Computings, mit dem Applikationen bereitgestellt werden können, ohne Infrastruktur verwalten zu müssen. Der Begriff ist nur bedingt zutreffend, da die Software letztendlich immer noch auf Servern läuft [5]. Operationale Aufgaben wie die Bereitstellung und Wartung von Servern, Installation von Betriebssystemen und Sicherheitsupdates, sowie die Netzwerkkonfiguration werden aber weitestgehend vom Cloud-Anbieter übernommen. Dies ermöglicht es Entwicklern, sich primär auf die Implementierung der Geschäftslogik zu konzentrieren, was die *Developer Experience* signifikant verbessern kann [20].

Die Plattform bildet eine Abstraktionsschicht zwischen der Laufzeitumgebung einer Applikation und der zugrunde liegenden Infrastruktur. Nutzer erhalten eine fertige Laufzeitumgebung, in die sie ihre Applikation einspielen können [5]. Dafür wird die Applikation in ein Artefakt verpackt, das mit der gewählten Plattform kompatibel ist. Zum Beispiel benötigt Google Cloud Run Container-Images, während FaaS-Angebote wie Google Cloud Functions direkt Quellcode verarbeiten. Auch andere Services wie Message Broker, Datenbanken und Dateisysteme werden als Serverless-Lösungen vertrieben[1].

In Serverless-Diensten ist die Trennung von Speicher und Rechenkapazität (*Separation of Storage and Compute*) ein zentrales Prinzip. Dies ermöglicht eine unabhängige Skalierung der beiden Ressourcen, was zu einer effizienteren Ressourcennutzung führt. Beispielsweise können Daten in einem Cloud-Speicherdienst wie Amazon S3 oder Google Cloud Storage abgelegt werden, während die Rechenoperationen durch kurzlebige Funktionen in AWS Lambda oder Google Cloud Functions ausgeführt werden. Diese Entkopplung erleichtert auch die Wartung und Aktualisierung einzelner Komponenten [20].

Serverless-Plattformen sind darauf ausgelegt, dynamisch und bedarfsgerecht zu skalieren. Neue Instanzen einer Serverless-Anwendung können binnen kurzer Zeit bereitgestellt und bei geringer Auslastung ebenso schnell wieder heruntergefahren werden. Serverless-Dienste bieten zudem eine konstante Performance, unabhängig vom Ausmaß der Skalierung [20].

Das Kostenmodell von Serverless-Plattformen basiert auf einem „Pay-as-You-Go“-Prinzip. Beispielsweise berechnet Google Cloud Run CPU- und Arbeitsspeicher-Nutzung in 100-Millisekunden-Intervallen und fügt einen festen Betrag pro eine Million Anfragen bei Überschreitung des kostenlosen Kontingents hinzu [21]. Dies stellt einen Paradigmenwechsel im Vergleich zu traditionellen Modellen dar, bei denen Kosten auf Basis von *reservierten* statt *tatsächlich genutzten* Ressourcen anfallen. In der Vergangenheit musste entweder eine Überprovisionierung in Kauf genommen oder eigene Skalierungslogik implementiert werden, um Spitzenlasten abzudecken [20].

Serverless-Dienste sind in der Regel zustandslos, das heißt sie speichern keine Daten zwischen den einzelnen Ausführungen, etwa im Arbeitsspeicher oder dem lokalen Datei-

[1] vgl. Serverless auf GCP, AWS und Azure

system. Dies ermöglicht eine einfache horizontale Skalierung, da jede Instanz unabhängig von den anderen ist. Allerdings ist oft der Einsatz von externen Diensten zur globalen Zustandsverwaltung nötig, wie Datenbanken oder Caches, um Zustandsinformationen zwischen den Funktionsaufrufen zu erhalten [22].

Serverless-Plattformen haben häufig strenge Beschränkungen, was die Ausführungsdauer und den Ressourcenverbrauch von Applikationen angeht. Dies ist sowohl eine Sicherheitsmaßnahme als auch eine Einschränkung, die Entwickler bei der Implementierung ihrer Funktionen berücksichtigen müssen. Die Bearbeitung einer Anfrage bei Google Cloud Runs darf beispielsweise maximal 60 Minuten dauern [23].

Kaltstarts sind eine der größten Herausforderungen bei Serverless-Technologien. Ein Kaltstart tritt auf, wenn eine Funktion zum ersten Mal oder nach einer Periode längerer Inaktivität aufgerufen wird. In diesem Fall muss die Laufzeitumgebung der Funktion initialisiert werden, was zusätzliche Latenz in die Bearbeitung einer Anfrage einbringt. Kaltstarts sind besonders problematisch für Anwendungen, die eine niedrige Latenz erfordern, wie beispielsweise Echtzeitanwendungen [20].

Das Gegenteil eines Kaltstarts wird als *Warmstart* bezeichnet. Er tritt auf, wenn eine bereits initialisierte Funktion erneut aufgerufen wird. Da die Laufzeitumgebung bereits vorhanden ist, wird die Funktion schneller ausgeführt. Warmstarts sind jedoch nicht immer garantiert, da Cloud-Anbieter inaktive Funktionsinstanzen nach einer bestimmten Zeitdauer herunterfahren können. Es gibt verschiedene Strategien zur Minimierung der Auswirkungen von Kaltstarts, darunter das „Warmhalten“ von Funktionen - die dauerhafte Bereitstellung einer Minimalanzahl an Instanzen - und die Optimierung des Codes für schnellere Initialisierung [22]. Es ist wichtig zu beachten, dass Warmstarts zwar die Leistung verbessern, aber auch Ressourcen verbrauchen, da die Leerlaufzeiten in Rechnung gestellt werden (siehe Abschnitt 8.2.2). Dies kann insbesondere bei Anwendungen mit vielen, selten genutzten Funktionen die Kostenvorteile einer Serverless-Architektur schmälern.

Ein weiterer Schwachpunkt von Serverless-Technologien kann verringerte Kontrolle sein. Da der Cloud-Anbieter die meiste Infrastruktur verwaltet, haben Entwickler weniger Kontrolle über die Umgebungen, in denen ihre Anwendungen laufen. Dies kann die Anpassungsfähigkeit einschränken und ist besonders problematisch, wenn spezielle Hardware- oder Software-Anforderungen bestehen [22]. Die Nutzung von proprietären APIs und Diensten eines bestimmten Cloud-Anbieters kann außerdem zu einer starken Abhängigkeit führen, die als *Vendor Lock-In* bezeichnet wird. Dies kann den Wechsel zu einem anderen Anbieter kompliziert und kostspielig gestalten [24, S. xiv].

Obwohl das „Pay-as-You-Go“-Modell bei geringer Nutzung kosteneffizient ist, können zudem die Kosten bei Anwendungen mit hohem Durchsatz und kontinuierlicher Nutzung schnell ansteigen. Es ist daher wichtig, das Kosten-Nutzen-Verhältnis sorgfältig zu analysieren. In [25] wird beschrieben, wie Amazon Prime Videos *Audio/Video Monitoring Service* erst als Microservices-Architektur auf Amazons Serverless-Diensten aufgebaut wurde, was initial eine kurze Entwicklungszeit und theoretisch hohe Skalierbarkeit brachte. Als das System skaliert werden sollte, stiegen die Kosten durch das hohe Maß an Kommunikation zwischen den Komponenten allerdings untragbar an. Das Entwicklerteam blieb bei Serverless-Technologien, verringerte aber die Anfragen im System durch Zusammenlegung der Anwendung in einen Monolithen.

3. Stand der Technik

3.1. Von Monolithen zu Microservices und Serverless

Der Umzug von Unternehmen in die Cloud hat in den letzten Jahren erheblich zugenommen, motiviert durch die Versprechen von mehr Agilität und Flexibilität. Allerdings haben viele Unternehmen festgestellt, dass die einfache Portierung ihrer monolithischen Anwendungen in die Cloud nicht die gewünschten Ergebnisse liefert [26].

Monolithische Architekturen waren vor der flächendeckenden Nutzung von Cloud Computing die Norm und sind oft für eine relativ statische On-Premises-Infrastruktur optimiert, bei der Änderungen kostspielig und zeitaufwendig sind [27]. Mit dem Aufkommen von Infrastructure-as-a-Service (IaaS) in der Cloud wurde die Möglichkeit geschaffen, neue Hardware in Sekunden bereitzustellen. Traditionelle Softwarearchitekturen konnten jedoch die dynamische Skalierbarkeit von Cloud-Ressourcen nicht effizient nutzen. Microservices haben sich als die Architektur der Wahl für Cloud-Umgebungen herausgestellt, da sie Skalierbarkeit und Flexibilität bieten, die sich besser mit den Möglichkeiten von Cloud-Infrastrukturen deckt [28].

Während IaaS viel Flexibilität in Bezug auf die Laufzeitkonfiguration und Hardware-Spezifikationen bietet, bleibt die Verantwortung für die Verwaltung der Infrastruktur beim Nutzer [29]. Mit der Adoption von Microservices führte dies zu erhöhter Komplexität und operationalen Aufwand, was den effektiven Gewinn an Agilität durch den Umzug in die Cloud schmälerte.

Die nächste Entwicklungsstufe war Platform-as-a-Service (PaaS), unterstützt durch Containerisierungstools wie Docker und Kubernetes. Es bringt eine noch höhere Abstraktionsebene als IaaS und verringert den operationalen Mehraufwand, der durch die Nutzung von Microservices entsteht. Allerdings verursachen alle Komponenten in einem PaaS-System kontinuierlich Kosten, auch wenn kein Nutzerverkehr vorhanden ist. Sie werden zwar teilweise automatisch skaliert, aber nie komplett heruntergefahren [30].

Serverless Computing, oft gleichgesetzt mit Function-as-a-Service (FaaS), ist aus dem Bedarf nach transparenter Infrastrukturverwaltung und effizienteren Preismodellen entstanden. Es bietet eine vollautomatische, bedarfsgerechte Skalierung von Komponenten bis auf null Instanzen und ein feingranulares Kostenmodell mit nutzungsabhängiger Bepreisung. Dazu ermöglicht es Entwicklern, ihren eigenen Code bereitzustellen, während die Infrastruktur vollständig vom Cloud-Anbieter verwaltet wird [24].

3.2. Aktuelle Entwicklungen und Herausforderungen

Laut der Cloud Native Computing Foundation ist Cloud Native der neue Standard für Unternehmen, und Serverless ist ein wichtiger Bestandteil davon [31], wobei sich Google

Cloud als Marktführer im Serverless-Bereich etabliert hat [32]. In den Anfängen von Serverless Computing war FaaS der meistgenutzte Serverless-Dienst. Mittlerweile tendieren Unternehmen aber immer stärker dazu, stattdessen containerbasierte Serverless-Modelle zu nutzen. Google Cloud Run erreichte im letzten Jahr fast die gleiche Nutzungsrate wie Cloud Functions [32].

Trotz der Vorteile von Serverless-Technologien sehen Unternehmen immer noch Herausforderungen wie unvorhersehbare Kostenspitzen, Vendor Lock-in und mangelnder Unterstützung von *stateful* Serverless-Anwendungen [33], [34]. Die öffentliche Meinung zu Serverless ist generell geteilt: Während einige Anwender es für latenzkritische und benutzerorientierte Anwendungen bevorzugen, halten andere es für eben diese Anwendungsfälle für ungeeignet. Ähnlich spalten sich die Meinungen bei der Kosteneffizienz von Serverless-Anwendungen und der Eignung für datenintensive Aufgaben [35].

3.3. Einordnung der vorliegenden Arbeit

Die vorliegende Bachelorarbeit fügt sich in den aktuellen Stand der Technik im Bereich der Microservices und Serverless-Architekturen ein, indem sie mehrere der identifizierten Herausforderungen und Trends adressiert. Entsprechend der aktuellen Tendenz zur Zerlegung monolithischer Anwendungen in Microservices, konzentriert sie sich auf die Konzeption einer solchen Architektur, um die Vorteile der Cloud-Native-Architekturen voll auszuschöpfen.

Ein weiterer wichtiger Aspekt der Arbeit ist die Fokussierung auf Serverless-Technologien. Dies spiegelt den aktuellen Trend zur Automatisierung von Infrastruktur und zur Minimierung des operativen Aufwands wider. Die Arbeit identifiziert jedoch auch spezifische Herausforderungen bei der Anwendung des Saga-Patterns in Serverless-Architekturen, was einen Beitrag zur aktuellen Diskussion über die Eignung von Serverless-Technologien für verteilte Anwendungen leistet.

4. Anforderungsanalyse

In diesem Abschnitt werden die Ergebnisse der Anforderungsanalyse für das geplante Produktivsystem *Nachrichtenmanagement* dargestellt. Alle Anforderungen wurden durch Gespräche mit dem bisherigen Projektteam und dem Auftraggeber (der Geschäftsleitung) herausgearbeitet. Zusätzlich lag bereits eine Darstellung der Projektvision vor, siehe Anhang A.2.

4.1. Grundlagen der Anforderungsanalyse

Bei Anforderungen an Softwaresysteme wird zwischen *funktionalen* und *nicht-funktionalen* Anforderungen unterschieden. Funktionale Anforderungen beschreiben die erwarteten Funktionalitäten des Systems [36]. Darunter fallen Beschreibungen wie „*Das System soll das Hochladen von Bildern im JPEG-Format unterstützen.*“

Die nicht-funktionalen Anforderungen, auch *technische Anforderungen* oder *Qualitätsanforderungen* genannt, umfassen hingegen Anforderungen an funktionsübergreifende Aspekte wie Skalierbarkeit (*Scalability*), Zuverlässigkeit (*Reliability*) und Sicherheit (*Security*) [36].

Die Auswahl einer Softwarearchitektur ist entscheidend für die Erfüllung der nicht-funktionalen Anforderungen. Während funktionale Anforderungen in der Regel durch nahezu jede Architektur erfüllt werden können, sind nicht-funktionale Anforderungen stärker von der spezifischen Architektur abhängig [9].

Hinzu kommen die *Randbedingungen* (Constraints) eines Softwareprojekts. Das sind organisatorische oder technologische Anforderungen, die die Entwicklung der Anwendung einschränken. Dazu können vorgegebene Programmiersprachen, Kostenrahmen und andere Einschränkungen gehören [37].

4.2. Beschreibung des bestehenden Prototyps

Der praktische Teil dieser Arbeit beruht auf einem bereits bestehenden Prototypen. Abb. 4.1 zeigt das Architekturschaubild des *Proof of Concept* (PoC) des Nachrichtenmanagements, wie es zu Projektbeginn vorlag. Der PoC besteht aus zwei Docker Images, welche in Cloud Runs (siehe Abschnitt 6.1.3) auf der GCP bereitgestellt werden und ist vollständig in Python geschrieben.

Das erste Image fungiert als API Gateway, welches über eine REST-Schnittstelle die Möglichkeit gibt, einzelne Nachrichten analysieren zu lassen und synchron eine Antwort mit Ergebnissen zu erhalten. Zusätzlich übernimmt es noch die Persistierung aller Daten in einer relationalen Datenbank, sowie domänenspezifische Datenverarbeitung wie das

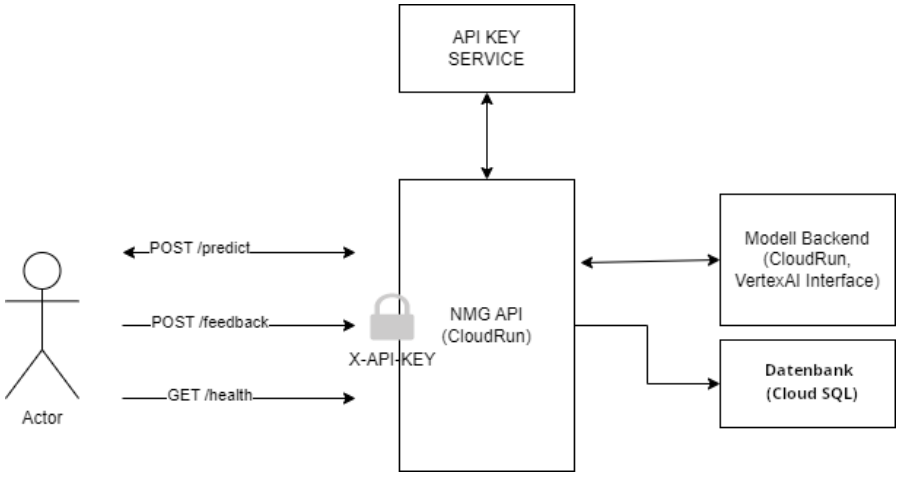


Abbildung 4.1.: Architekturschaubild des Prototyps. Quelle: Internes Wiki der Mittelstand.ai

Filtern von Rückgabewerten. Das zweite Image stellt dem API Gateway über eine REST-Schnittstelle KI-Modelle zur Verfügung. Alle neun Modelle des Nachrichtenmanagements laufen im selben Docker Container und somit auf der selben Maschine.

Es sind zudem externe Systeme an den Prototypen angeschlossen: Eine PostgreSQL-Instanz dient als transaktionale Datenbank und der externe *API Key Service* der hauseigenen Analyse-Plattform validiert API Keys von Kunden und tauscht sie gegen `orgIds` (Organisation IDs) aus, anhand derer Anfragen Geschäftskunden zugeordnet werden können.

4.3. Identifikation von Schwachstellen und Herausforderungen

Beim PoC-System wurden Schwachstellen identifiziert, die die Motivation für das hier behandelte Projekt darstellen. Vor der Produktivschaltung der Anwendung sollen folgende Punkte adressiert werden:

- **Lange Startzeiten:** Die Cloud Runs skalieren aufgrund geringen Nutzerverkehrs immer wieder auf null Instanzen zurück, um in Leerlaufzeiten zwischen Anfragen Kosten einzusparen. Die Zeit bis zum Herunterfahren einer Instanz ist zwischen fünf Sekunden und 15 Minuten variabel [38]. Da alle KI-Modelle bei Systemstart erst einmal in den Arbeitsspeicher geladen werden müssen, führt das zu einer Kaltstartzeit von über zwei Minuten.
- **Geringe Verfügbarkeit:** Während der zwei Minuten Kaltstartzeit werden Anfragen mit generischen HTTP-Fehlercodes abgelehnt - das System ist nicht verfügbar. Um die Schnittstelle nutzen zu können, ist im Zweifelsfall also immer eine Vorlaufzeit von mindestens zwei Minuten einzuplanen.
- **Fehlende GPU-Unterstützung:** Die KI-Modelle laufen nur auf der CPU. Es ist aber vorhersehbar, dass dies mit steigendem Anfragevolumen zu Performanceproblemen führen kann. Auch könnte durch zukünftige Entwicklungen im Bereich der

Large Language Models der Einsatz dedizierter Grafikkarten interessant werden, welche aber nicht für normale Cloud Runs verfügbar sind[1].

- **Fehlende CI/CD:** Es wird keine CI/CD-Infrastruktur verwendet. Images werden manuell gebaut und in der Google Cloud bereitgestellt.
- **Fehlende Test Suite:** Es gibt keine automatisierten Tests. Tests werden manuell durchgeführt, indem drei Container (API Gateway, Model Backend und eine Datenbank) lokal gestartet, über interaktive Terminals eingerichtet und dann händisch nach eigenem Ermessen funktionell getestet werden.
- **Schlechte Beobachtbarkeit:** Es gibt kein spezialisiertes Monitoring. Laufzeitfehler können in Google Cloud Logs eingesehen werden, da Cloud Run diese dort automatisch einspeist.
- **Optimierungsbedarf im Software-Design:** Der aktuelle Anwendungscode weist Spielraum für die Implementierung etablierter Designprinzipien auf. Dies erhöht das Entstehungsrisiko eines Big Ball of Mud (siehe Abschnitt 2.1.2).

4.4. Anforderungen an das zukünftige Produktivsystem

Die Anforderungen an das Produktivsystem wurden erneut in Gesprächen zwischen Projektteam und Auftraggeber ausgearbeitet und den drei Kategorien **funktionale** Anforderungen, **nicht-funktionale** Anforderungen und **Randbedingungen** (siehe Abschnitt 4.1) zugeordnet. Nicht alle der in Abschnitt 4.3 identifizierten Schwachstellen und Herausforderungen sollen im Produktivsystem vollständig behoben werden. Der Auftraggeber möchte einige Kompromisse bei Verfügbarkeit, Performance und Elastizität eingehen, um die in Abschnitt 4.4.3 beschriebenen Randbedingungen einzuhalten. Außerdem sind neue Funktionalitäten wie integrierte Challenger-Modell-Testläufe gewünscht.

4.4.1. Funktionale Anforderungen

Es gibt neue funktionale Anforderungen, die beim PoC noch nicht berücksichtigt wurden. Zusammen mit den Funktionalitäten, die im Schaubild der Projektvision (Anhang A.2) dargestellt sind, wurden folgende funktionale Anforderungen identifiziert:

- **Schnittstelle:** Die Applikation soll eine API anbieten, die sowohl die Anbindung externer Clients, als auch die Integration in verbreitete Bankensoftware ermöglicht. Es soll sich um eine REST/HTTP-Schnittstelle handeln, die Daten im *JSON*-Format annimmt und zurückliefert.
- **KI-gestützte Vorhersagen:** KI-Modelle sind zu *Komponenten* zusammengefasst. So gibt es zum Beispiel eine *Entitätenkomponente*, die mehrere Modelle zur Named Entity Recognition enthält. Geschäftskunden sollen textbasierte Kundenkorrespondenz in *HTML* oder *Plaintext* zusammen mit einer Auswahl der gewünschten KI-Komponenten an die Applikation senden können. Die Nachricht wird an die

[1] Nur für GKE und GCE, siehe: https://cloud.google.com/gpu

angeforderten Komponenten weitergeleitet und der Kunde erhält die aggregierten Klassifikationen und Entitäten zurück. Dabei ist es akzeptabel, wenn der Kunde das Ergebnis asynchron erhält, indem er periodisch die API abfragt.

- **Feedback:** Kunden sollen die Möglichkeit haben, Feedback zu den erhaltenen Vorhersagen abzugeben. Dabei sind vier Level von Feedback geplant, die immer feingranularer werden. Das Feedback kann für jede erfolgreich bearbeitete Nachricht zu einem beliebigen späteren Zeitpunkt über die API eingereicht werden. Zu einer Nachricht kann mehrfach Feedback eingereicht werden, solange das Level des Feedbacks sich unterscheidet.
- **Validierung von Eingaben:** Eingabedaten werden auf das richtige Textformat (HTML oder Plaintext) und andere Bedingungen geprüft.
- **Autorisierung und Authentifizierung:** Anfragen können eindeutig einem Kunden zugeordnet werden. Nur Kunden, die einen gültigen Nutzungsvertrag haben, erhalten Vorhersagen.
- **Weiterentwicklung der KI-Modelle:** Erneutes Training der KI-Modelle auf den anfallenden Daten soll für den Anfang erleichtert werden, etwa indem die Daten zentralisiert abrufbar sind und Modelle sich leicht aktualisieren lassen. Als spätere Weiterentwicklung soll eine Automatisierungsmöglichkeit dieses Prozesses berücksichtigt werden.
- **Evaluation neuer KI-Modelle:** Wurden neue Modelle für eine bereits existierende Komponente trainiert, sollen diese als *Challenger-Modelle* im Parallelbetrieb zu den bisherigen Modellen evaluiert werden können, ohne dass Anwender davon etwas merken (*A/B-Testing*, *Shadow-Testing*). Auch die Erprobung neuer Komponenten soll integriert möglich sein.
- **Modellüberwachung:** Performancemetriken der KI-Modelle sollen erfasst und auf Dashboards dargestellt werden. Es soll konfigurierbare Alarmsysteme bei Abweichungen dieser Metriken geben.
- **Rechnungsstellung:** Kunden soll eine Rechnung basierend auf ihren Anfragen im Abrechnungszeitraum gestellt werden können. Dies kann zuerst händisch erfolgen, aber die Daten sollen leicht abrufbar sein. Später soll die Rechnungsstellung aus der Anwendung heraus automatisiert zu beliebigen Zeitpunkten möglich sein.
- **Persistente Datenhaltung:** Transaktionale Daten sind persistent gespeichert. Zusätzliche Daten wie externe Trainingsdatensätze können in der Datenhaltung abgelegt werden.
- **Individuell trainierte Modelle:** Banken können Datensätze liefern und erhalten damit exklusiven Zugriff auf für sie individuell trainierte Modelle.
- **Geschäftszeiten:** Da gerade Nachts so gut wie keine Nachrichten erwartet werden, denkt der Auftraggeber über die Möglichkeit einer Pufferung von Anfragen nach, um Kosten durch dauerhaft laufende Infrastruktur zu vermeiden. Es soll eventuell die Möglichkeit geben, rund um die Uhr Anfragen an die API zu senden, welche aber nur innerhalb bestimmter Geschäftszeiten abgearbeitet werden. Ergebnisse sollen jederzeit asynchron abgefragt werden können.

4.4.2. Nicht-funktionale Anforderungen

Als nicht-funktionale oder Qualitätsanforderungen wurden mit dem Auftraggeber folgende Punkte erarbeitet:

- **Antwortzeit:** Die Zeit vom Eintreffen einer Nachricht bis zum Erhalt eines Ergebnisses soll (innerhalb der Geschäftszeiten) immer unter zehn Minuten liegen. In der manuellen Nachrichtenverteilung liegt die Bearbeitungszeit meistens bei mehreren Stunden[2], weswegen immer noch ein beachtlicher Performancegewinn erreicht würde.
- **Erweiterbarkeit:** Neue KI-Modelle sollen jederzeit hinzugefügt werden können. Neue Use Cases und neue Zielgruppen (zum Beispiel Versicherungen, mittelständische Unternehmen) sollen zukünftig ebenfalls bedient werden können.
- **Skalierbarkeit:** Skalierbarkeit in realistischem Maße ist erwünscht. Am Anfang wird mit einem sehr geringen Anfragevolumen von ca. 50 Anfragen pro Tag gerechnet. Es wurde aber prognostiziert, dass es innerhalb des ersten Jahres auf viele Tausend Anfragen am Tag steigen kann. Werden neue Banken als Kunden gewonnen, bringen diese manchmal Stämme von mehreren hunderttausend Kunden mit[3], deren Nachrichten dann vom Nachrichtenmanagement bearbeitet werden müssen. Die Anwendung sollte auf diesem gesamten Spektrum die geforderten Antwortzeiten einhalten können und dabei zu jeder Zeit möglichst kosteneffizient sein.
- **Portabilität:** Es ist auf längere Sicht denkbar, die Anwendung auf ein hauseigenes Kubernetes-Cluster (die „Analyseplattform“) umzuziehen. Eine solche Migration sollte möglichst reibungslos möglich sein.
- **Interoperabilität:** Andere Projekte der Mittelstand.ai sollen sich (auch ohne oben genannte Analyseplattform) mit den Diensten des Nachrichtenmanagements integrieren lassen.

4.4.3. Randbedingungen und Kompromisse

Der Auftraggeber möchte einige Kompromisse beim Produktivsystem eingehen, um folgende Randbedingungen einzuhalten:

- **Time to Market:** Der Zeitraum bis zur Markteinführung (*Time to Market*) muss gering gehalten werden. Das System soll im vierten Quartal 2023 testweise erste Kundenanfragen bearbeiten können.
- **Kosten:** Die Betriebskosten sollen gerade in den ersten Monaten so gering wie möglich gehalten werden, da dort mit relativ geringem Anfragevolumen und mit wenigen Einnahmen zu rechnen ist.
- **Operations:** Operationale Tätigkeiten (*Operations*) müssen minimiert werden, da es momentan kein dediziertes Personal dafür gibt und diese Arbeiten von den Entwicklern durchgeführt werden müssten. Das schließt insbesondere den Betrieb eines

[2] Quelle: Vorstudie der Mittelstand.ai zum Nachrichtenmanagement
[3] Siehe bspw. www.vb-mittelhessen.de/wir-fuer-sie/ueber-uns/zahlen-fakten.html

projekteigenen Kubernetes Clusters oder anderer selbstverwalteter Infrastrukturen aus.

- **Deployment-Plattform:** Die Anwendung muss auf der Google Cloud Platform bereitgestellt werden.
- **Programmiersprache:** Alle an der Entwicklung beteiligten Personen haben ausreichende Pythonkenntnisse zur Umsetzung des Projektes. Die KI-Modelle laufen ohnehin in einer Python-Umgebung und für alle Google Cloud Services sind SDKs in Python verfügbar. Daher werden sämtliche Services initial in Python entwickelt.
- **Software Design:** Aufgrund schlechter Erfahrungen mit einem vorherigen Projekt ist die Vermeidung eines *Big Ball of Mud* besonders wichtig.

Die Hauptkompromisse liegen im Bereich der nicht-funktionalen Anforderungen. So kann die Systemverfügbarkeit in Abhängigkeit von Geschäftszeiten und geplanten Wartungsphasen variiert werden. Hinsichtlich der Zuverlässigkeit genügt eine Betriebszeit von 99 Prozent, unter der Annahme, dass die zugrundeliegende Infrastruktur keine katastrophalen Ausfälle erleben wird. Die Zielsetzung einer maximalen Antwortzeit von zehn Minuten stellt ebenfalls einen bewussten Kompromiss dar, um die Komplexität der Architektur zu minimieren und finanzielle Ressourcen zu schonen. Hinsichtlich Skalierbarkeit und Elastizität genügt es, wenn die Anwendung den prognostizierten Nutzerverkehr des ersten Betriebsjahres bewältigen kann und Lastspitzen von bis zu 1000 simultanen Anfragen aushält.

5. Konzeption der Microservices-Architektur

Dieser Abschnitt behandelt die Konzeption einer Softwarearchitektur für das *Nachrichtenmanagement*, die die in Abschnitt 4.4 aufgenommenen Anforderungen bestmöglich erfüllt. Zuerst werden in Abschnitt 5.1 der *Domain-Driven Design*-Ansatz und seine Relevanz für Microservices-Architekturen erklärt. Abschnitt 5.2 beschreibt die Einzelschritte des Konzeptionsprozesses. In Abschnitt 5.3 wird das entworfene Konzept als abstrakte Systemarchitektur visualisiert.

5.1. Domain-Driven Design

Domain-Driven Design (DDD) ist ein Ansatz zur Softwareentwicklung, der den Fokus auf die Geschäftsdomäne und die darin enthaltenen Fachbegriffe legt. Diese Methodik zielt darauf ab, die Komplexität der Geschäftslogik durch eine enge Zusammenarbeit zwischen Entwicklern und Fachexperten zu bewältigen. Dabei wird ein gemeinsames Verständnis der Geschäftsdomäne geschaffen und in den Code integriert, wobei eine Reihe von Pattern und Prinzipien zum Einsatz kommen [7].

Für Microservices-Architekturen gewinnt DDD an Relevanz, weil es die Entkopplung von Softwaremodulen fördert, indem es klare Grenzen um verschiedene Geschäftsbereiche zieht. Diese werden als *Subdomains* bezeichnet. Jede Subdomain kann ihre eigene Definition von Domänenkonzepten wie *Kunde* und *Produkt* haben, siehe Abbildung 5.1. Man sagt, dass eine Subdomain ihr eigenes *Domain Model* besitzt. Der Anwendungsbereich eines Domain Models heißt *Bounded Context*. Die beiden Begriffe werden teilweise synonym verwendet. Jeder Microservice kann als Implementierung eines Bounded Contexts betrachtet werden, was die sogenannten *God-Classes* vermeidet - Klassen mit zu vielen unterschiedlichen Aufgaben, die überall in einer Anwendung verteilt benutzt werden und eine Dekomposition des Systems stark erschweren [7].

Darüber hinaus fördert DDD die interne Kohäsion innerhalb eines Bounded Contexts. Alle Elemente innerhalb eines Kontexts stehen eng miteinander in Zusammenhang, um ein bestimmtes Geschäftsziel zu erreichen. Das macht Microservices effizient und wartbar, da eine veränderte Anforderung an die Funktionalitäten eines Systems erfahrungsgemäß nur Änderungen innerhalb einer Menge von Elementen mit hoher Kohäsion und damit innerhalb eines Services erfordert [7].

Die Kommunikation zwischen Projektbeteiligten ist ein weiterer zentraler Aspekt von DDD. Es legt großen Wert auf die *Ubiquitous Language*, eine gemeinsame Sprache zwischen Entwicklern und Fachexperten, in der Begriffe immer genau eine fest definierte Bedeutung haben. Diese Sprache erleichtert die Kommunikation und das Verständnis,

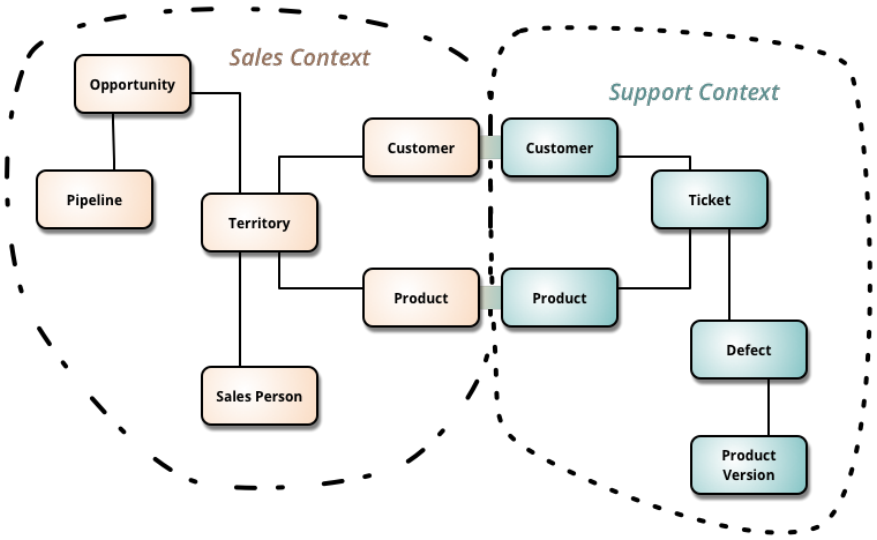


Abbildung 5.1.: Bounded Contexts ermöglichen unterschiedliche Definitionen der gleichen Domänenkonzepte. Quelle: [39]

was in einer verteilten Architektur aufgrund der inhärenten Komplexität besonders wichtig ist [7].

Ein weiterer Vorteil von DDD ist die Förderung der Datenkonsistenz durch Muster wie *Aggregates* und *Entities*, die dabei helfen, die Datenintegrität in einem verteilten System zu wahren. Entities sind die einzelnen Objekte in der Geschäftsdomäne, während Aggregates unterschiedliche Entities zusammenfassen, die im Zuge der Geschäftsfunktionen zusammen modifiziert werden müssen. Daten werden dann nur über Aggregates verändert; ein feingranularer Direktzugriff auf einzelne Entities ist nicht vorgesehen [7].

Insgesamt bietet Domain-Driven Design einen Rahmen für die Modellierung und Implementierung von Geschäftslogik in einer Weise, die gut zu den Prinzipien der Microservices-Architektur passt. Es fördert die Entkopplung, Kohäsion und effektive Kommunikation, die für den Erfolg von Microservices entscheidend sind [40].

5.2. Aufteilung in unabhängige Microservices

Laut Richardson besteht die Hauptaufgabe einer Anwendung im Bearbeiten von Anfragen [41]. Basierend darauf stellt er zwei Prozesse zur Architekturmodellierung vor. Er führt weiter an, dass sie sich in erster Linie darin unterscheiden, ob Services anhand von *Geschäftsfunktionen* oder *Subdomains* im Sinne des Domain-Driven Designs definiert werden. Aufgrund der in Abschnitt 5.1 erläuterten Eignung von DDD zur Modellierung von Microservices-Architekturen wurde letzterer Ansatz gewählt und angewendet, der von Richardson mit folgenden Schritten beschrieben wird:

1. Definition von Systemoperationen
    a) Erstellung eines High-Level Domain Models
    b) Identifikation der Systemoperationen

2. Definition von Services
    a) Identifikation der Subdomains
    b) Übersetzung der Subdomains in Services
3. Definition von Service APIs
    a) Zuweisung von Systemoperationen an Services
    b) Entwurf der Service APIs

5.2.1. Definition von Systemoperationen

Der erste Schritt in der Definition einer Softwarearchitektur ist die Definition der *Systemoperationen*. Eine Systemoperation ist die Abstraktion einer Anfrage, die die Anwendung bearbeiten können muss. Es handelt sich entweder um ein *Command* (Befehl), das Daten verändert, oder um eine *Query* (Abfrage), die Daten abfragt. Jede Systemoperation wird durch ihre Interaktion mit einem abstrakten Domain Model beschrieben. Sowohl das Domain Model als auch die Systemoperationen werden aus den Anforderungen an das System abgeleitet [41].

Erstellung eines High-Level Domain Models

Das Domain Model lässt sich besonders aus den *Nomen* der in Abschnitt 4.4.1 beschriebenen Anforderungen ableiten [41]. User Stories wie „*Kunden sollen die Möglichkeit haben, Feedback zu den erhaltenen Vorhersagen abzugeben.*“ legen die Existenz der Entitäten *Kunde*, *Feedback* und *Vorhersage* nahe. In Abb. 5.2 sind die Entitäten der Geschäftsdomäne und die Beziehungen zwischen ihnen dargestellt.

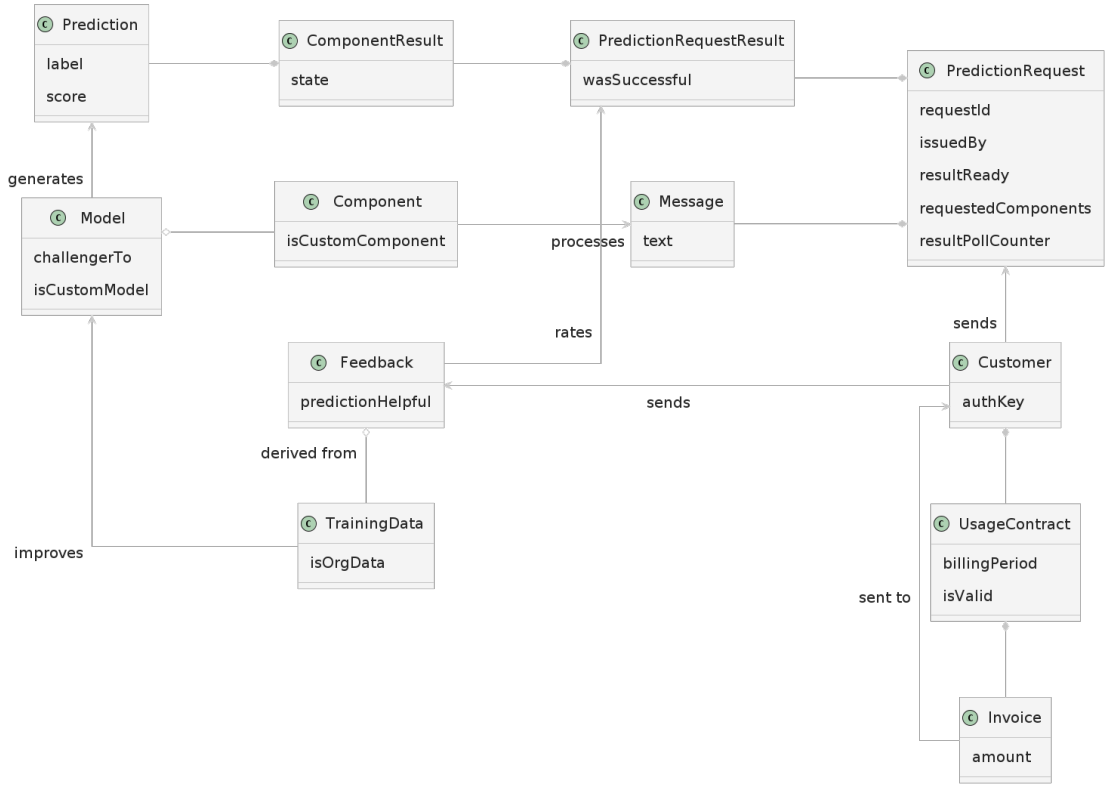


Abbildung 5.2.: Domain Model des Nachrichtenmanagements. Quelle: Eigene Darstellung.

Identifikation der Systemoperationen

Nachdem das Domain Model entworfen wurde, können die Systemoperationen definiert werden, die die Anwendung bearbeiten soll. Der vorherige Entwurf des Domain Models ist wichtig, da Systemoperationen mit den Entitäten der Domäne arbeiten. Zur Bestimmung der Systemoperationen sollten vor allem die *Verben* der in Abschnitt 4.4.1 beschriebenen Anforderungen betrachtet werden [41]. Ein Teil der Systemoperationen des Nachrichtenmanagements ist in Tabelle 5.1 aufgelistet. Diese Operationen ergeben sich direkt aus den wichtigsten User Stories: Ein Kunde soll Kundenkorrespondenz an das Nachrichtenmanagement (NMG) schicken und Ergebnisse dazu abfragen können. Anschließend soll er Feedback zu den erhaltenen Vorhersagen geben können. Der logische Ablauf einer Vorhersageanfrage ist in Abb. 5.3 dargestellt.

Zu jedem Command gehört eine Spezifikation, die seine Parameter, Rückgabewerte und sein Verhalten in Bezug auf die Objekte der Domäne beschreibt. Zum Verhalten gehören die *Vorbedingungen* und die *Nachbedingungen* der Operation. Die Vorbedingungen werden am Beginn der Operation überprüft. Nur wenn sie erfüllt sind, führt die Systemoperation die nötigen Schritte aus, um die Nachbedingungen zu erfüllen [41].

Tabelle 5.2 zeigt die Spezifikation des erfolgreichen Verlaufs der Operation `createPredictionRequest()`, mit der ein Kunde Vorhersagen für eine Nachricht in Auftrag geben kann. Die Vorbedingungen verlangen, dass ein Kunde, identifiziert durch die übermittelte `orgId`, dem System bekannt ist und dass er einen gültigen Nutzungsvertrag besitzt.

Abbildung 5.3.: Logischer Ablauf einer Vorhersageanfrage. Quelle: Eigene Darstellung.

Außerdem müssen die angeforderten Komponenten existieren und der Kunde muss diese anfragen dürfen. Als Rückgabewert erhält er eine eindeutige Kennziffer (`message_id`). Mit dieser Kennziffer kann er später den Bearbeitungsstatus seiner Anfrage abrufen, der sich mit jedem Schritt der Anfrage durch das System verändert. Die Nachbedingungen legen fest, dass die Anfrage im System als `PENDING` gespeichert und die Nachricht an die angefragten Komponenten weitergeleitet wird.

Tabelle 5.1.: Systemoperationen des Nachrichtenmanagements

Actor	Operation	Typ	Beschreibung
Kunde	createPredictionRequest()	Command	Beauftragt die Generierung von Vorhersagen für einen Nachrichtentext.
Kunde	queryResult()	Query	Fragt den Status / das Ergebnis einer Vorhersageanfrage ab.
Kunde	createFeedback()	Command	Übermittelt Feedback zu den erhaltenen Vorhersagen einer Nachricht.

Tabelle 5.2.: Spezifikation der Systemoperation `createPredictionRequest()`

Operation	`createPredictionRequest`(orgId, requestedComponents, message)
Rückgabewert	`messageId`
Vorbedingungen	• Der Kunde ist durch eine `orgId` identifiziert. • Der Kunde hat einen gültigen Nutzungsvertrag. • Alle `requestedComponents` existieren. • Der Kunde darf die `requestedComponents` anfragen.
Nachbedingungen	• Es wurde eine `PredictionRequest` im Status `PENDING` erstellt. • Die Nachricht wurde an die angefragten Komponenten geleitet.

Command Query Responsibility Segregation

Auch Queries können starken Einfluss auf die Konzeption der Systemarchitektur haben [7, Kap. 7]. Zur Bearbeitung der Abfrage `queryResult()` zum Beispiel muss der Bearbeitungsstatus einer Anfrage an einem Punkt aktualisiert werden und jederzeit abrufbar sein. Ist die Bearbeitung einer Anfrage fertig, müssen zudem die Ergebnisse in dieser Operation zurückgegeben werden. Da von Kunden erwartet wird, dass sie für jedes ausstehende Ergebnis jeder von ihnen gesendeten Nachricht die API periodisch abfragen (*pollen*), muss diese Abfrage besonders effizient beantwortet werden können. Außerdem sollen Ergebnisse rund um die Uhr abgefragt werden können, selbst wenn andere Teile der Anwendung Geschäftszeiten unterliegen. Sind die Daten in einem Service aggregiert, muss nur dieser eine Service dauerhaft erreichbar sein. Auch wenn Ergebnisse also konzeptionell zum Objekt *Anfrage* gehören, werden sie aus diesen Gründen einen eigenen Service bekommen, der einen *CQRS View* implementiert.

CQRS steht für *Command Query Responsibility Segregation* und bezeichnet ein Design Pattern, bei dem die Datenmodelle für lesende und schreibende Operationen (Commands und Queries) unterschiedlich sind [42]. Für das Nachrichtenmanagement heißt das, dass die verarbeitenden Services nur jeweils einen Teil der Daten einer Vorhersageanfrage kennen. Es gibt jedoch einen speziellen Service namens *Result Service*, der ausschließlich dafür zuständig ist, eine vollständige Repräsentation jeder Anfrage zu aktualisieren und diese Informationen für spätere Abfragen zur Verfügung zu stellen. Wie später erklärt wird, benutzt der Result Service zur Aktualisierung seiner Modelle asynchrone Events,

die von den anderen Services ausgesendet werden. Um den Unterschied für die Abarbeitung einer Vorhersageanfrage zu zeigen, ist in Abb. A.2 und Abb. A.3 einmal der Ablauf mit und einmal ohne Result Service dargestellt.

5.2.2. Definition von Services

Nachdem ein Domain Model erstellt und die Systemoperationen definiert wurden, besteht der nächste Schritt darin, die Geschäftsdomäne zu unterteilen und diese durch Services abzubilden. Diese Einteilung kann entlang der *geschäftsspezifischen Funktionalitäten* (*Business Capabilities*), oder entlang der *Subdomain* des Domain-Driven Design erfolgen [41]. Eine Business Capability ist etwas, durch das das Unternehmen letztendlich Wert generieren kann, wie *Auftragsabwicklung*. Subdomains hingegen stellen eine technischere Sicht auf Teilaspekte der Geschäftsdomäne dar. Sie werden so gewählt, dass möglichst wenige Abhängigkeiten zwischen Subdomains existieren und die Komplexität gering bleibt [43]. Für das Nachrichtenmanagement wurde die Aufteilung in Subdomains nach dem Ansatz des *Domain-Driven Design* gewählt und damit folgende Domänen identifiziert:

- **Anfragen:** Diese Subdomain beschäftigt sich mit Nutzeranfragen und deren Validierung.
- **Ergebnisse:** Der Bearbeitungsstatus von Anfragen, sowie deren Ergebnisse, werden in dieser Domain behandelt. Streng genommen würden sie besser anderen Domänen untergeordnet sein, wurden aber aus den in Abschnitt 5.2.1 genannten Gründen als eigene Domäne modelliert.
- **Vertrag:** Der Gegenstand dieser Subdomain sind Nutzungsverträge und Rechnungen.
- **Komponenten:** Diese Subdomain enthält die existierenden KI-Komponenten und die Informationen darüber, wie sie anzusprechen sind.
- **Vorhersagen:** In dieser Subdomain sind KI-Modelle und die von ihnen generierten Vorhersagen abgebildet.
- **Feedback:** Diese Subdomain enthält das Feedback zu Vorhersagen und die Regeln des Feedback-Prozesses.

Für jede der festgelegten Subdomains kann im nächsten Schritt der dazugehörige Service beschrieben werden, der die Systemoperationen auf den Objekten seiner Subdomain ausführt. Diese Übertragung ist in Abb. 5.4 dargestellt und wird im Folgenden beschrieben.

- **API Gateway:** Das API Gateway übernimmt die Identifizierung von Clients, ist der Startpunkt für Vorhersageanfragen und leitet diese an andere Services weiter.
- **Result Service:** Der Result Service unterhält *CQRS Views* (siehe Abschnitt 5.2.1) der Bearbeitungsstatus aller Anfragen und macht diese durch Queries zugänglich.
- **Contract Service:** Dieser Service verwaltet Nutzungsverträge, stellt Rechnungen aus und validiert die aus Nutzungsverträgen hervorgehenden Berechtigungen für angeforderte Komponenten.

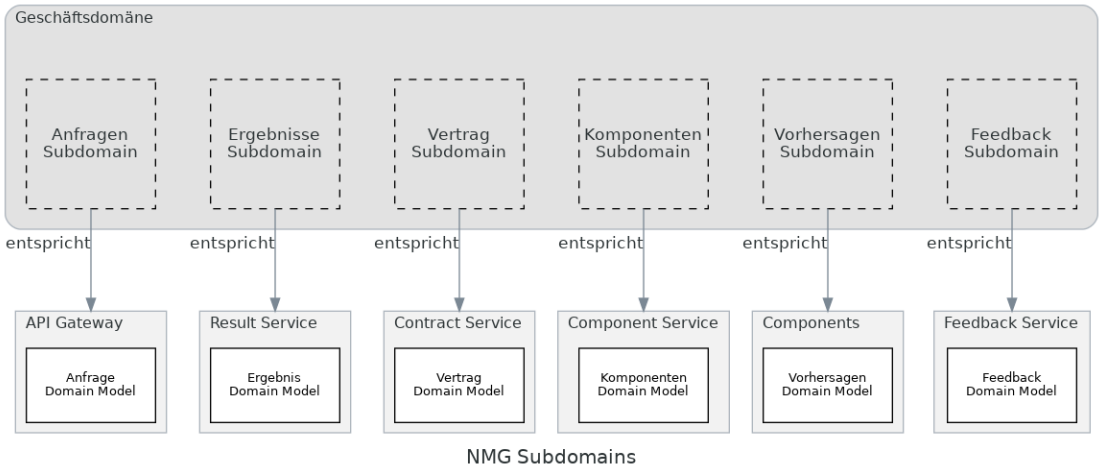


Abbildung 5.4.: Abbildung von Subdomains des Nachrichtenmanagements auf Services. Quelle: Adaptiert von [41, Abb. 2.9].

- **Component Service:** Dieser Service dient als *Service Registry* für KI-Komponenten. Er übersetzt die Bezeichner angeforderter Komponenten in die Aufrufe interner Services.
- **Components:** Jede Komponente ist ein Zusammenschluss eines oder mehrerer KI-Modelle. Innerhalb einer Komponente müssen Modelle in bestimmter Reihenfolge und mit bestimmten Parametern angesprochen werden. Die von den Modellen gelieferten Vorhersagen müssen gegebenenfalls nachbearbeitet werden (*Postprocessing*), etwa durch Filterung duplizierter Vorhersagen ähnlicher Modelle. Jede Komponente produziert zu einer Nachricht ein aggregiertes Komponentenergebnis, das die aufbereiteten Vorhersagen all seiner Modelle enthält. Das Gesamtergebnis und die einzelnen Modell-Outputs müssen zu Trainings- und Monitoring-Zwecken gespeichert werden.
- **Feedback:** Feedback muss auf seine Gültigkeit überprüft werden. Es gibt mehrere mögliche Level von Feedback mit unterschiedlichem Detailgrad zu jeder erfolgreich bearbeiteten Nachricht. Jedes Level darf separat, aber nur ein Mal für eine Nachricht eingesendet werden. Das Feedback muss so an die ursprüngliche Nachricht und die Vorhersagen angeknüpft werden, dass eine einfache Trainingsdatensatzerstellung möglich ist.

5.2.3. Definition von Service APIs

Der letzte Schritt in der Konzeptionierung einer Anwendungsarchitektur ist die Erstellung der Schnittstellenbeschreibungen (API-Definitionen) der einzelnen Services. Die API eines Services umfasst seine ausführbaren *Operationen* (Commands und Queries) und seine ausgesendeten *Events*. Erst werden die Systemoperationen den Services zugeordnet und dann die APIs entworfen, die für die *systeminterne* Kooperation der Services notwendig sind [41]. Die Zuteilung von Systemoperationen zu Services kann Tabelle 5.3 entnommen werden.

Tabelle 5.3.: Zuweisung der Systemoperation an Services

Service	Operation
API Gateway	`createPredictionRequest()`
Result Service	`queryResult()`
Feedback Service	`createFeedback()`
Component Service	• `registerComponent()` • `listComponents()`
Contract Service	• `updateContract()` • `sendInvoice()`

Häufig ist an einer Systemoperation mehr als ein Service beteiligt. Für eine vollständige API-Definition müssen Servicefunktionen ergänzt werden, die nicht von externen Clients, sondern von internen Services zur Kollaboration aufgerufen werden [41]. Das API Gateway hat dabei eine Sonderrolle, da es als einziger Einstiegspunkt in die Applikation dient. Es ist daher streng genommen zuständig für *alle* in Tabelle 5.3 beschriebenen Operationen und ruft die anderen Services nur intern als Kollaborateure auf. Tabelle 5.4 zeigt einen Auszug aus der vollständigen Schnittstellenbeschreibung des Nachrichtenmanagements.

Zu einer vollständigen API-Definition fehlen als Letztes noch die Events, die von Services ausgesendet werden. Events dienen zur asynchronen und unverbindlichen Benachrichtigung anderer Services über geschehene Vorgänge im System, siehe Abschnitt 2.3.3. Beim Nachrichtenmanagement werden Events zur Aktualisierung der Daten im Result Service verwendet. Zu jeder Operation aus Tabelle 5.4 gehört dabei mindestens ein Event, welches beim Abschluss des Vorgangs ausgesendet wird. Für die meisten Operationen gibt es jeweils ein Event für den Erfolgsfall und ein Event für den Fehlerfall. Ein Event enthält genügend Informationen, um die durchgeführten Operationen nachvollziehen zu können, aber nicht unbedingt eine vollständige Abbildung des betroffenen Domänenobjektes. Als Beispiel soll das Domänenobjekt einer *Anfrage* dienen, zu der der Component Service gerade die Operation `makePredictions()` ausgeführt und somit Vorhersagen bei allen geforderten Komponenten angefragt hat. Nachdem die entsprechenden Commands zu den Komponenten geschickt wurden (idealerweise als atomare Operation, siehe Abschnitt 2.3.4), wird vom Component Service das in Listing 5.1 gezeigte Event für jede angefragte Komponente ausgesendet. Der Result Service erkennt daran, welche Komponentenergebnisse zu dieser Anfrage noch fehlen und wann die Bearbeitung abgeschlossen ist. Das Event im Fehlerfall ist in Listing 5.2 abgebildet.

Tabelle 5.4.: Auszüge der API-Definition der NMG Services

Service	Operation	Kollaborateure
API Gateway	• `createPredictionRequest()`	• Contract Service `verifyCustomer()` • Component Service `makePredictions()`
	• `listComponents()`	• Component Service `listComponents()`
Component Service	• `makePredictions()`	• Components `createPrediction()`
	• `registerComponent()` • `listComponents()`	
Contract Service	• `verifyCustomer()` • `updateContract()` • `sendInvoice()`	—
⋮	⋮	⋮
Result Service	• `queryResult()`	—

Listing 5.1: Beispiel eines Events im Erfolgsfall

```
{
  "event_id": "352e5d26-d855-4126-8f79-0922c2e3d60e",
  "payload_schema": "component_service.ComponentRequestedEvent",
  "schema_version": "1.0.0",
  "timestamp": 1696089038,
  "data": {
    "message_id": "aeb9f89a-5aa7-4990-9ec2-d7293508f489",
    "component": "topics"
  }
}
```

Listing 5.2: Beispiel eines Events im Fehlerfall

```
{
  "event_id": "6d221646-7bf2-4102-a195-e9296f1d8611",
  "payload_schema": "component_service.ComponentRequestFailedEvent",
  "schema_version": "1.0.0",
  "timestamp": 1696089038,
  "data": {
    "message_id": "aeb9f89a-5aa7-4990-9ec2-d7293508f489",
    "component": "Lottozahlen",
    "reason": "The requested component 'Lottozahlen' does not exist."
  }
}
```

5.3. Grobentwurf der Anwendungsarchitektur

Um ein klares Verständnis für die Konzeption der Microservices-Anwendung zu schaffen, wurde ein High-Level-Architekturschaubild erstellt, das in Abb. 5.5 zu sehen ist. Dieses Schaubild gibt eine abstrakte Sicht auf die verschiedenen Services und ihre Beziehungen zueinander und dient als Ausgangspunkt für detailliertere Design- und Implementierungsüberlegungen.

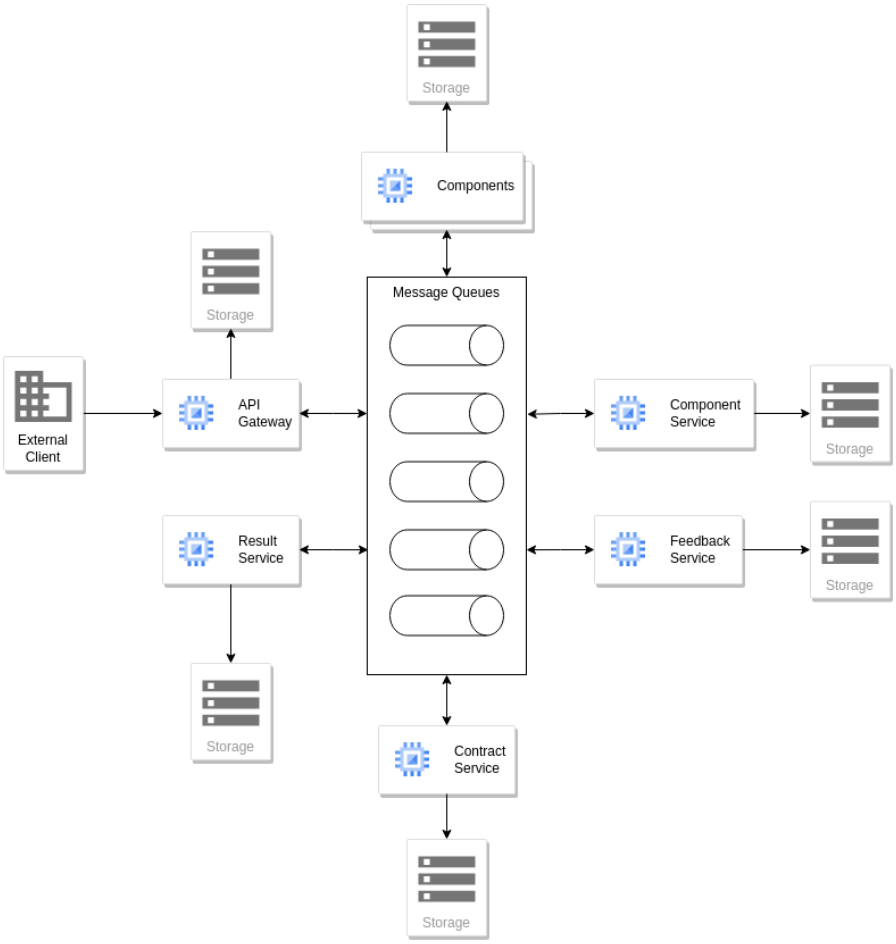


Abbildung 5.5.: Eine erste High-Level Systemarchitektur des Nachrichtenmanagements. Quelle: Eigene Darstellung.

Die Kommunikation zwischen den Services erfolgt möglichst asynchron. Wie in Abschnitt 2.3.3 erläutert, bietet dieser Ansatz den Vorteil zeitlicher Entkopplung und hoher Flexibilität. Jeder Service ist mit einer eigenen Datenbank verbunden, was der Kopplung über gemeinsame Daten entgegenwirkt und die Fähigkeit zur autonomen Entwicklung und Skalierung jedes Services erhöht. Es kann jedoch Herausforderungen in Bezug auf die Datenkonsistenz zwischen den Services geben, die mittels des Saga-Patterns (siehe Abschnitt 2.3.4) gelöst werden müssen. Der *externe Client* stellt eine oder mehrere Anwendungen dar, die mit dem API Gateway interagieren, um Zugang zu den Diensten und Daten des Systems zu erhalten.

Abb. 5.5 dient als Leitbild für die vollständige Version des Nachrichtenmanagements.

Der nächste Schritt ist die Auswahl geeigneter Cloud-Dienste für die Umsetzung der Architektur, die in Kapitel 6 erfolgt.

6. Auswahl geeigneter Cloud-Dienste

In diesem Kapitel wird die Auswahl der Technologien für die Umsetzung der konzipierten High-Level-Architektur aus Abb. 5.5 behandelt. Die Abschnitte 6.1 bis 6.4 führen durch die Auswahl der Google-Cloud-Dienste für die Laufzeitumgebung der Anwendungslogik, Serviceintegration, Datenhaltung und Bereitstellung der KI-Modelle. Als Grundlage dienen die in den Abschnitten 4.4.2 und 4.4.3 definierten Qualitätsanforderungen und Randbedingungen der NMG-Anwendung.

Die GCP bietet eine breite Palette von Serverless-Diensten an, die von Infrastructure-as-a-Service (IaaS) bis hin zu Platform-as-a-Service (PaaS) reichen. In den folgenden Abschnitten wird ein Teil der GCP-Dienste jeweils kurz vorgestellt und die Erkenntnisse des Projektteams bezüglich der Eignung für die Implementierung des Nachrichtenmanagements dargestellt. Alle Informationen zu den evaluierten Services wurden, sofern nicht anders gekennzeichnet, den offiziellen Dokumentationsseiten der Google Cloud entnommen.

6.1. Ausführungsumgebung

6.1.1. Google App Engine

Google App Engine stellt ein vollständig verwaltetes PaaS-Angebot dar, das es Entwicklern ermöglicht, alle Teile einer Microservices-Anwendung ohne Infrastrukturverwaltung und Build-Prozesse zu betreiben. Ganze Applikationen aus mehreren Komponenten werden als Paket aus Source Code und Spezifikationsdatei an App Engine übergeben und die Plattform übernimmt die weiteren Deployment-Schritte. Es wurden folgende Einschränkungen identifiziert:

- Bindung an eine einzige Cloud-Region [44], was für KI-Applikationen mit Bedarf an Grafikkarten ein Problem darstellen kann.
- Beide Versionen des Produkts („Environments“) haben Nachteile: Die *Flexible Environment* hat eine weniger flexible Kostenstruktur ohne Skalierung auf null Instanzen und eine sehr hohe Kaltstartlatenz. Die *Standard Environment* bietet nur vorgefertigte Laufzeitumgebungen mit starker Bindung an die dazugehörigen Google SDKs [45].
- In der Flexible Environment muss von jeder Revision eines jeden Services dauerhaft mindestens eine Instanz laufen.
- Bei Verwendung der Standard Environment wäre die Portabilität der Anwendung nicht gegeben, da sie durch die notwendige Nutzung der integrierten SDKs nicht ohne weiteres auf andere Deployment-Plattformen übertragen werden könnte.

Google App Engine wurde aufgrund der oben genannten Einschränkungen ausgeschlossen. Das Projektteam bräuchte gerade in den frühen Betriebsphasen die Skalierbarkeit und Kostenstruktur der Standard Environment, in der man testweise verschiedene Versionen von Services dynamisch mit Nutzerverkehr versorgen und auf null skalieren kann, kombiniert mit der frei wählbaren Runtime und Portabilität der Flexible Environment, um auf lange Sicht anpassbar zu bleiben.

6.1.2. Google Cloud Functions

Google Cloud Functions ist speziell für die Komposition und Erweiterung ereignisgesteuerter Anwendungen konzipiert. Cloud Functions sind als kurzlebige Integrationshilfen zwischen komplexeren Komponenten einer Anwendung gedacht. Um eine Cloud Function zu erstellen, wird eine Source-Code-Datei zusammen mit einer Spezifikationsdatei für Dependencies verpackt und an den Cloud-Dienst übergeben. Der Code sollte aus einer einzigen Funktion oder Methode bestehen und die Konventionen des zugrunde liegenden Functions Frameworks einhalten. Der Cloud Functions Service übernimmt alle weiteren Schritte, um die Funktion auf einem HTTPS-Endpunkt bereitzustellen.

Cloud Functions sind einfach einzurichten und zu verwenden, da sie den Build- und Deployment-Prozess auf den zugrundeliegenden Cloud Runs vor dem Entwickler verbergen. Man kann sich voll auf die Entwicklung von Anwendungslogik konzentrieren. Sie bieten jedoch nicht die Flexibilität und Kontrolle, die für komplexere Anwendungen erforderlich sind. Man ist, bis auf die Angabe von Package Dependencies, auf die vorgegebenen Runtimes und Ausführungsmodelle beschränkt. Cloud Functions sollten kurze Laufzeiten aufweisen, jeweils nur *eine* klar abgegrenzte Aufgabe erfüllen und keine Hintergrundaktivitäten durchführen. Sie können als die „Microservices der Microservices“ betrachtet werden [46].

Anwendungen wie REST-APIs können zwar mit Cloud Functions aufgebaut werden, aber da jede Cloud Function nur einen Endpunkt (eine Funktion) beinhalten kann, wäre eine solche Architektur schnell unübersichtlich und schwer zu dokumentieren. Weiterhin wird jede Cloud Function im Hintergrund als eigener Cloud Run ausgeführt (und in Rechnung gestellt), was effizient für rechenintensive Aufgaben sein kann. Bei I/O-lastigen Aufgaben, wie sie häufig im Nachrichtenmanagement zu erwarten sind, würden aber die Ressourcen eines Cloud Runs nicht gut ausgenutzt [24]. Um kosteneffizient zu arbeiten, wäre die Ausführung umfangreicherer Anwendungsteile im gleichen Cloud Run sinnvoll.

Cloud Functions wurden aus den genannten Gründen nicht zur Implementierung der NMG-Services benutzt. Ein möglicher Anwendungsfall wird aber in Abschnitt 7.1.2 gezeigt.

6.1.3. Cloud Run

Cloud Run ist eine Serverless-Ausführungsumgebung, die eine Kombination aus den Vorteilen von Serverless-Architekturen und der Flexibilität von Containern darstellt. Die Infrastruktur wird vollständig von Google verwaltet, was es Entwicklern erlaubt, sich auf den Code anstatt auf Infrastrukturmanagement zu konzentrieren.

Einer der größten Vorteile von Cloud Run ist die Fähigkeit zur automatischen Skalierung. Die Anwendung kann nahezu sofort von null auf tausende von Containern skaliert werden, je nachdem, wie der Nutzerverkehr ansteigt oder abnimmt. Im Gegensatz zu traditionellen VM-basierten Diensten, bei denen Ressourcen dauerhaft reserviert werden müssen, wird Cloud Run dabei nur für die tatsächliche Ausführungszeit einer Anwendung in Schritten von 100 Millisekunden abgerechnet.

Cloud Run basiert auf dem offenen Standard *Knative* und daher kann eine Anwendung problemlos auf andere Knative-unterstützende Plattformen wie Kubernetes übertragen werden. Da Anwendungen in Containern laufen, ist man zudem nicht an eine bestimmte Programmiersprache oder ein spezielles Framework gebunden. Die Laufzeitumgebung kann maßgeschneidert eingerichtet werden, solange sie in einem Container ausgeführt werden kann.

Obwohl Cloud Run in gewissem Maße für schnelle Kaltstarts optimiert ist, kann es dennoch zu einer gewissen Latenz kommen, insbesondere wenn eine Anwendung komplexe Initialisierungslogik hat [24]. Das hatte beim ursprünglichen Proof of Concept des Nachrichtenmanagements bereits zu Problemen geführt (siehe Abschnitt 4.3). Wie andere Serverless-Technologien auch, sind Cloud Runs für *stateless* Anwendungslogik konzipiert, was die Anwendbarkeit traditioneller Design Patterns einschränkt, siehe Abschnitt 7.1.

6.1.4. Fazit

Unter Berücksichtigung der in den vorherigen Sektionen definierten Qualitätsanforderungen und Randbedingungen ergibt sich **Cloud Run** als die optimale Wahl für die Implementierung der Services des Nachrichtenmanagements. Es kombiniert die Flexibilität und Kontrolle, die für komplexe Anwendungen erforderlich sind, mit minimalem operationalem Aufwand. Im Vergleich zu App Engine und Cloud Functions bietet es eine überlegene Kostenstruktur und Portabilität. Obendrein hat das Projektteam viel praktische Erfahrung mit der Entwicklung und dem Deployment von containerisierten Anwendungen, weswegen die direkte Arbeit mit den Container-Umgebungen von Cloud Run gewohnter ist, als die „Black Box“-Abstraktionsschichten von Cloud Functions und App Engine.

6.2. Prozessübergreifende Kommunikation

Die Services der geplanten Anwendung müssen, da es sich um eine verteilte Anwendung handelt, über ein Netzwerk miteinander kommunizieren. Die GCP bietet dafür mehrere Serverless-Dienste an, die in den folgenden Abschnitten diskutiert werden.

6.2.1. Cloud Tasks

Cloud Tasks ist ein Message-Queue-Dienst, der es ermöglicht, Anfragen (Tasks) serverlos zwischenzuspeichern, zu verzögern oder zu einem bestimmten Zeitpunkt auszuführen. Tasks entsprechen Nachrichten mit beliebigem Inhalt. Anstatt Anfragen synchron an

andere Services zu schicken, reiht ein Client (Publisher) seine Nachrichten in Queues ein, aus denen Empfänger (Consumer) sie nach dem *Push*-Prinzip erhalten.

Für Queues können Durchsatzratenbegrenzungen (*Rate Limits*) und Wiederholungsversuche eingestellt werden. So werden Consumer nicht überlastet, müssen nicht dauerhaft verfügbar sein und die Anfragebearbeitung ist fehlertolerant. Für jede Task kann der gewünschte Zustellzeitpunkt oder eine Verzögerung festgelegt werden. Cloud Tasks deduplizieren Tasks beim Einfügen in Queues und garantieren mindestens einen erfolgreichen Zustellversuch (*At-least-once Delivery*), wobei die meisten Anfragen genau ein Mal zugestellt werden (*Exactly-once Delivery*). Für die Zustellreihenfolge von Tasks oder eine maximale Zustelldauer werden keine Garantien gegeben [47].

Cloud Tasks wurden ausgiebig für das Nachrichtenmanagement evaluiert und haben sich aus folgenden Gründen als ungeeignet erwiesen:

- Es ist nur *Point-to-Point*-Kommunikation möglich - Tasks können immer nur an einen Empfänger adressiert werden und Absender müssen die URLs ihrer Ziele kennen.
- Deduplikation erfolgt nur anhand einer vom Publisher selbst gesetzten `ID`. Es ist dem Publisher überlassen, für gleiche Nachrichten die gleiche ID zu vergeben, was bei den zustandslosen Cloud Runs zum Beispiel über zusätzliche Datenbankeinträge umgesetzt werden müsste.
- Wiederholungsversuche werden nicht nur bei Laufzeitfehlern, sondern auch bei Nichtverfügbarkeit eines Consumers hoch gezählt. In Kombination mit den zu erwartenden Kaltstarts von Cloud Runs führt das nur bei Konfiguration von unendlich vielen Wiederholungsversuchen zu einem wirklich resilienten System. Das wiederum führt zu Problemen im Umgang mit fehlerhaften Tasks, deren Bearbeitung unmöglich ist, und überlasteten Services, die von den immer wiederkehrenden Tasks weiter unter Last gesetzt werden.
- Man kann weder erfolgreich abgearbeitete, noch endgültig fehlgeschlagene Tasks einsehen. Gerade der letzte Punkt ist kritisch: Man muss entweder Queues mit unendlich vielen Wiederholungsversuchen konfigurieren (mit den oben beschriebenen Problemen), oder hat das Problem still fehlschlagender Operationen [24]. Ein Dead Lettering-Mechanismus wäre für das Monitoring, Tracing und die Feststellung des Fehlschlagens einer Vorhersageanfrage aber essenziell.

6.2.2. Cloud Scheduler

Cloud Scheduler ist ein vollständig verwalteter *Cron Job*-Dienst, der die Ausführung von automatisierten Tasks in festgelegten Zeitintervallen ermöglicht. Das kleinste einstellbare Zeitintervall ist eine Minute. Es ist daher für wiederkehrende, zeitbasierte Aufgaben und nicht als Kommunikationsgrundlage in Event-driven Architekturen geeignet. Ein möglicher Anwendungsfall für Cloud Scheduler im Nachrichtenmanagement wäre aber das Anstoßen von Batch-Exporten zu BigQuery.

6.2.3. Cloud Workflows

Cloud Workflows bietet einen Serverless-Dienst für die Orchestrierung komplexer, mehrstufiger Prozesse, welche mit `YAML`-Dateien oder grafischen Oberflächen beschrieben werden. Es dient als Koordinator für die Zusammenarbeit verschiedener Services, einschließlich Cloud Run, und verfügt über umfassende Mechanismen für Fehlerbehandlung und die Weitergabe von Ergebnissen zwischen den verschiedenen Prozessstufen. Die Möglichkeit, Arbeitsabläufe im Request/Response-Modell zu modellieren, während der Orchestrator in einen Serverless-Dienst ausgelagert ist, macht dieses Tool auf den ersten Blick zu einer geeigneten Lösung für das Nachrichtenmanagement.

Allerdings bringt die Nutzung von Cloud Workflows auch Nachteile mit sich, vor allem im Hinblick auf Datenschutz und Datensicherheit. Eine beträchtliche Schwachstelle ist die vollständige Protokollierung sämtlicher Eingabedaten eines Workflows in Google Cloud Logging, wie Testläufe gezeigt haben. Im Kontext der NMG-Anwendung betrifft das Textnachrichten, die zwischen Kunden und ihren Finanzdienstleistern ausgetauscht werden. Da diese Daten eine hohe Vertraulichkeit aufweisen, ist eine derartige Form der Protokollierung nicht akzeptabel. In internen Gesprächen zwischen dem Projektteam und den Geschäftsberatern von Google konnte keine geplante Änderung hinsichtlich der Eingabeprotokollierung in Aussicht gestellt werden.

Es ist wichtig zu beachten, dass Cloud Logs in der Regel für Monitoring, Tracing und Debugging vorgesehen sind und daher einem breiteren internen Publikum zugänglich gemacht werden. Das erzeugt einen Konflikt zwischen der Notwendigkeit, den Systemzustand zu überwachen und gleichzeitig datenschutzkonform zu handeln. Sollte ein Kunde auf die Löschung seiner Daten bestehen, würde das Löschen der entsprechenden Logs eine Gefährdung für die Observability der Anwendung mit sich bringen. Das bedeutet, dass wichtige Metriken und Ereignisse nicht mehr nachvollziehbar wären, was im schlimmsten Fall zu verminderter Systemstabilität und verlängerten Ausfallzeiten führen könnte.

Darüber hinaus würde ein solcher Schritt zur Löschung der Logs auch administrative und operative Herausforderungen mit sich bringen, da es keinen trivialen Mechanismus gibt, um selektiv Logs zu entfernen, ohne die Integrität der übrigen Daten zu beeinträchtigen.

Daher wurde entschieden, Cloud Workflows nicht für die Umsetzung des Nachrichtenmanagements zu verwenden. Es wird als zu risikoreich erachtet, vertrauliche Daten in einem System zu verarbeiten, das nicht vollständig den Datenschutzanforderungen entspricht. Sollte das Logging von Eingaben zukünftig abstellbar sein, wäre eine erneute Evaluation von Cloud Workflows denkbar.

6.2.4. Cloud Pub/Sub

Cloud Pub/Sub ist ein serverloser Message Broker, der eine enge Integration mit anderen Diensten wie Cloud Run ermöglicht. Im Grundprinzip senden *Publisher* Nachrichten an sogenannte *Topics*, während interessierte Clients, die als *Subscriber* bezeichnet werden, *Subscriptions* zu diesen Topics anlegen, über die sie Nachrichten erhalten (siehe Abb. 6.1).

Ein wesentliches Merkmal von Pub/Sub ist die *temporale Entkopplung* von Publisher

und Subscriber durch Persistierung von Nachrichten. Das bedeutet, dass ein Subscriber eine Nachricht zu einem beliebigen späteren Zeitpunkt abrufen kann, sofern eine Subscription zum Veröffentlichungszeitpunkt der Nachricht vorhanden war. Während viele Messaging-Services diese Funktionalität besitzen, entstehen durch die Serverless-Natur von Pub/Sub in der Zeit bis zum Abruf der Nachrichten keine Kosten[1].

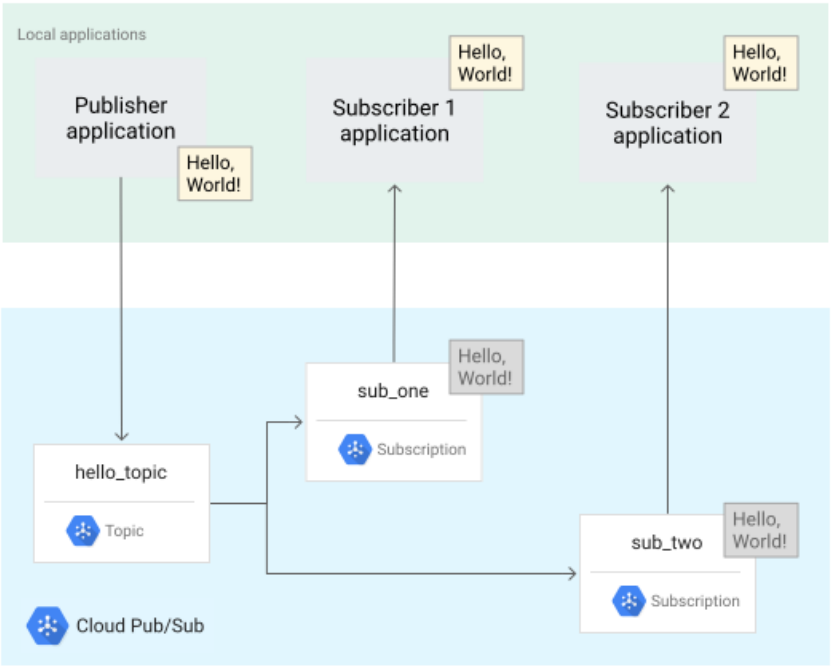


Abbildung 6.1.: Publish/Subscribe mit Google Cloud Pub/Sub. Quelle: cloud.google.com/pubsub/docs/building-pubsub-messaging-system

Für dieses Projekt wurde Pub/Sub vor allem aufgrund folgender Eigenschaften und Funktionen in Erwägung gezogen:

- **Push- und Pull-Subscriptions:** Pub/Sub unterstützt sowohl Push- als auch Pull-Subscriptions. Während bei Push-Subscriptions die Nachrichten aktiv an Subscriber gesendet werden, können bei Pull-Subscriptions die Subscriber Nachrichten nach Bedarf abrufen. Beide Modelle haben ihre eigenen Vorteile und können je nach Anwendungsfall sinnvoll eingesetzt werden. Push-Subscriptions sind für das Nachrichtenmanagement besonders interessant, da sie keine aktiven Cloud-Run-Instanzen zur Nachrichtenverarbeitung benötigen. Instanzen können durch den *Push* einer Nachricht erst gestartet werden.
- **Erneute Zustellversuche:** Auf jeder Subscription lassen sich Wiederholungsrichtlinien mit exponentiell ansteigenden Wartezeiten (*Backoffs*) und maximalen Wiederholungen konfigurieren.
- **Anwendungsseitige Kontrolle:** Subscriber müssen Nachrichten explizit gegenüber Pub/Sub bestätigen. Erfolgt dies nicht innerhalb eines gewissen Zeitfensters,

[1] Für sieben Tage, danach entstehen Speicherkosten: cloud.google.com/pubsub/pricing#storage_costs

oder wird mit einer Ablehnung geantwortet, wird die Nachricht erneut an den Endpunkt (aber nicht zwangsläufig an dieselbe Instanz) zugestellt.

- **Dead Lettering:** Ein weiteres nützliches Feature ist das sogenannte *Dead Lettering*. Falls eine Wiederholungsrichtlinie keinen weiteren Zustellversuch einer Nachricht mehr vorsieht, wird diese auf einem speziellen *Dead Letter Topic* abgelegt. Diese Funktionalität erleichtert die Fehleranalyse gegenüber Cloud Tasks erheblich.
- **Attributfilter:** Pub/Sub erlaubt bei der Erstellung von Subscriptions die Definition von Attributfiltern und Messages können beim Veröffentlichen mit den dazugehörigen Attributen versehen werden. Dadurch können Subscriber für sie irrelevante Nachrichten ignorieren, was die Verarbeitungseffizienz steigert und unnötige Serviceaufrufe verhindert.

In den Punkten Leistungsfähigkeit und Preisstruktur ist Pub/Sub für den Einsatz in globalen Größenordnungen konzipiert. Angesichts dessen, dass der erwartete Datendurchsatz der geplanten Applikation in den ersten Betriebsmonaten relativ niedrig sein wird, fällt die Nutzung von Pub/Sub wahrscheinlich in das kostenlose Kontingent. Dadurch bietet Pub/Sub durch seinen hohen Durchsatz, die geringe Latenz und die effiziente Kostenstruktur eine robuste und skalierbare Lösung für Event-Driven Architectures und stellt somit eine ausgezeichnete Wahl für die Inter-Service-Kommunikation im Nachrichtenmanagement dar.

6.2.5. Fazit

Unter Berücksichtigung der Qualitätsanforderungen und der bereits erlangten Erfahrungen des Teams bietet **Cloud Pub/Sub** die flexibelste und robusteste Lösung für die Serviceintegration im Nachrichtenmanagement. Es hebt sich von den anderen Lösungen durch die Kombination von automatischen Wiederholungsversuchen, Dead Lettering, Performance und Kosteneffizienz ab. Für spezialisierte Anwendungsfälle, wie zeitgesteuerte Aufgaben, könnten jedoch Cloud Tasks oder Cloud Scheduler in Kombination mit Pub/Sub in Betracht gezogen werden.

6.3. Datenhaltung

In diesem Abschnitt werden zwei Dienste der GCP zur Datenhaltung gegeneinander abgewogen: Cloud Firestore und Cloud SQL. Beide Dienste bieten unterschiedliche Modelle und Funktionen zur Datenverwaltung und können, je nach den spezifischen Anforderungen und dem Kontext des Projekts, variabel eingesetzt werden.

6.3.1. Firestore

Cloud Firestore ist eine dokumentenbasierte Serverless-NoSQL-Datenbank, die sich besonders für die schnelle Entwicklung von Web-, Mobil- und Serveranwendungen eignet. Firestore skaliert automatisch mit steigendem Anfragen- und Datenvolumen und unterstützt trotz seiner NoSQL-Natur ACID-Transaktionen auf Dokumentenebene. Da es

sich um eine NoSQL-Datenbank handelt, erfordert Firestore kein festes Schema und bietet die Flexibilität, Datenstrukturen ohne umfangreiche Datenmigrationen zu ändern. Trotzdem gibt es die Möglichkeit komplexer Queries mit Sortierung und Filterung, um gezielt einzelne Dokumente, oder ganze Gruppen von Dokumenten anzufordern. Firestore ist sehr kosteneffizient und bietet, ähnlich wie die anderen Google Cloud-Dienste, ein kostenloses Kontingent für Abfragen und Datenspeicherung.

6.3.2. Cloud SQL

Cloud SQL ist ein vollständig verwalteter (*fully-managed*) Datenbank-Service, aber kein Serverless-Dienst. Datenbankserver werden von Google Cloud bereitgestellt, aufgesetzt und mit Updates versorgt, sowie automatisiert Sicherungskopien (Backups) erstellt. Es gibt aber keine automatische Skalierung oder nutzungsabhängige Kostenstruktur. Festplattenspeicher, Arbeitsspeicher und Rechenleistung werden nach *angeforderter*, nicht nach *genutzter* Kapazität berechnet und es gibt kein kostenloses Kontingent.

Die vertikale Skalierung von Instanzen muss eigenständig über die Veränderung der Instanzparameter angefordert werden. Horizontal kann ebenfalls über die manuelle Einrichtung von *Read Replicas* skaliert werden. Cloud SQL unterstützt MySQL, PostgreSQL und SQL Server, und ist durch sein relationales Datenmodell insbesondere für komplexe Queries und Transaktionsdatenbanken geeignet. Konfigurationen für hohe Verfügbarkeit sind durch *High Availability Replicas* und *automatische Fail-Over* konfigurierbar. Cloud SQL integriert sich sehr gut mit Cloud Run: Bei der Konfiguration einer Cloud-Run-Revision kann direkt eine Cloud-SQL-Instanz als Datenbank angegeben werden.

6.3.3. Fazit

Nach sorgfältiger Abwägung der verschiedenen Aspekte der Datenhaltung hat sich **Firestore** als die am besten geeignete Lösung für die Anforderungen dieses Projekts herausgestellt. Die Serverless-Architektur von Firestore bietet Skalierung ohne jeglichen operationalen Aufwand. Dies ist ein wichtiger Vorteil gegenüber Cloud SQL, welches manuelle Eingriffe erfordert.

Ein weiterer entscheidender Faktor für die Wahl von Firestore ist seine Flexibilität im Umgang mit Datenstrukturen. Da es keine festen Schemata gibt, können Änderungen an der Datenstruktur ohne aufwendige Migrationen vorgenommen werden. Dies ist besonders nützlich in der agilen Entwicklung, wo sich Anforderungen schnell ändern können. Cloud SQL, obwohl leistungsfähig und gut für komplexe Queries, erfordert ein festes Schema und damit verbundene Migrationsprozesse, die zusätzlichen Aufwand und mögliche Ausfallzeiten bedeuten könnten.

Firestore ist für dieses Projekt, insbesondere unter Berücksichtigung des kostenlosen Kontingents, kosteneffizienter als Cloud SQL. Cloud SQL bietet zwar umfangreiche Funktionen, jedoch zu einem relativ hohen Preis gegenüber den anderen gewählten Anwendungskomponenten. Dieser Nachteil seitens Cloud SQL wird verstärkt, wenn für unabhängige Skalierung und Erhöhung der Leistung eine eigene Datenbankinstanz pro Service aufgesetzt werden sollte.

6.4. KI-Modelle

Für das Nachrichtenmanagement wurde, vor allem bewertet durch die projektbeteiligten Data Scientists, Google Clouds **Vertex AI** als Plattform zur Verwaltung und zum Betrieb der KI-Modelle ausgewählt. Vertex AI, früher bekannt als AI Platform, ist ein vollständig verwalteter Dienst, der speziell für Machine-Learning-Workflows entwickelt wurde. Der Dienst bietet eine ganzheitliche Lösung für das Training, die Evaluation und die Bereitstellung von Modellen. Es unterstützt Frameworks wie TensorFlow, Scikit-Learn und XGBoost durch vorgefertigte Ausführungsumgebungen, kann aber auch benutzerdefinierte Container-Images verwenden, was für die spezifischen Anforderungen des Nachrichtenmanagements unerlässlich ist.

Die *Model Registry* von Vertex AI fungiert als zentrales Repository zur Verwaltung von KI-Modellen. Dabei besteht die Möglichkeit, Modelle entweder direkt in Container-Images zu integrieren oder sie durch Angabe einer Google-Cloud-Storage-URI zu importieren, unter der die Modellartefakte gespeichert sind. Diese Modelle können sowohl eigenständig hinzugefügt werden als auch als neue Versionen bereits existierender Modelle.

Ein wichtiges Konzept von Vertex AI sind *Prediction Endpoints*, die als HTTP-Endpunkte der Entkopplung dienen. Modelle können jederzeit auf Endpoints bereitgestellt und wieder abgeschaltet werden. Erst bei der Bereitstellung eines Modells auf einem Endpunkt wird die gewünschte Hardware ausgewählt und bereitgestellt. Darüber hinaus können mehrere Modelle auf einem einzigen Endpoint bereitgestellt werden, wobei der Nutzerverkehr dynamisch zwischen ihnen aufgeteilt werden kann.

Endpoints haben eine stabile URL, selbst wenn die dahinterliegenden Modelle ausgetauscht werden. Dies vereinfacht die Integration in andere Systemkomponenten erheblich. Kosten für die Ausführung eines Modells entstehen nur, wenn es auf einem Endpoint bereitgestellt ist. Allerdings fallen auch Speicherkosten für die Modelle in der Model Registry an und die mit einem Endpoint assoziierten Ressourcen werden nach reservierter, nicht nach genutzter Kapazität berechnet.

Ein kritischer Punkt ist, dass der Bereitstellungsprozess eines Modells auf einem Endpoint bei den NMG-Modellen etwa 15 Minuten dauert. Dies macht eine schnelle, reaktive Skalierung von null, wie sie mit Cloud Runs möglich ist, unpraktikabel. Nichtsdestotrotz bietet Vertex AI automatische Skalierungsfunktionen, die allerdings mit einer gewissen Trägheit einhergehen.

In Anbetracht der beschriebenen Anforderungen und der Funktionalitäten, die Vertex AI bietet, erscheint diese Plattform als äußerst geeignet für den Betrieb und die Verwaltung der KI-Modelle der Anwendung. Die Fähigkeit zur reibungslosen Modellaktualisierung entspricht den Anforderungen für die Weiterentwicklung der KI-Modelle. Gleichzeitig ermöglicht die Abstraktion von Endpoints die Implementierung von A/B- und Shadow-Testing, was für die Evaluation neuer KI-Modelle entscheidend ist.

6.5. Zusammenfassung der Ergebnisse

Dieser Abschnitt hat die Auswahl der geeigneten Google Cloud-Produkte für verschiedene Aspekte des Nachrichtenmanagements diskutiert. Die Entscheidungen wurden auf Grundlage der funktionalen Anforderungen, Qualitätsanforderungen und Randbedingungen des Projektes und unter Berücksichtigung der Erfahrungen des Projektteams und Informationen aus offiziellen Dokumentationen getroffen.

- **Ausführungsumgebung:** Cloud Run wurde für die Implementierung der Services ausgewählt. Es bietet eine überlegene Kombination aus Flexibilität, Kontrolle und Kosteneffizienz im Vergleich zu anderen Lösungen wie App Engine und Cloud Functions.
- **Kommunikation zwischen Services:** Cloud Pub/Sub dient als das Kommunikationsmittel für die Integration der Microservices. Es bietet wichtige Funktionen wie automatische Wiederholungsversuche und Dead Lettering, die es von anderen Lösungen abheben.
- **Datenhaltung:** Firestore wurde als angemessenste Datenhaltungslösung bewertet. Es bietet Vorteile in Bezug auf Skalierbarkeit, Flexibilität der Datenstrukturen und Kosteneffizienz im Vergleich zu Cloud SQL.
- **KI-Modelle:** Vertex AI wurde als Plattform für die Entwicklung und Bereitstellung der KI-Modelle gewählt.

In Kapitel 7 wird eine vollständige Architektur der Anwendung entworfen, auf ein *Minimum Viable Product* heruntergebrochen und unter Verwendung der in diesem Kapitel ausgewählten Technologien umgesetzt.

7. Design und Implementierung

In diesem Kapitel wird die schematische Darstellung aus Abb. 5.5 mit den gewählten Technologien aus Kapitel 6 zu einer umsetzbaren Architektur kombiniert. Besondere Aufmerksamkeit gilt den Folgen der Auswahl von Serverless-Diensten, da diese maßgeblich die Eigenschaften und Funktionsweise des Systems beeinflussen. In Abschnitt 7.1 werden die Herausforderungen erörtert, die sich aus der Verwendung dieser Dienste ergeben. Die konzipierte Architektur des vollständigen Nachrichtenmanagements (NMG) wird in Abschnitt 7.2 dargestellt. Anschließend wird in Abschnitt 7.3 aus dieser Architektur eine erste Iteration der Anwendung abgeleitet und implementiert.

7.1. Orchestration und Choreografie in Serverless-Architekturen

In Abschnitt 2.3.4 wurde die *Saga* als neue Herangehensweise zur Wahrung der Datenkonsistenz in verteilten Systemen vorgestellt. Sagas können als *Orchestration* mit einem zentralen Orchestrator, oder als eventgesteuerte *Choreografie* umgesetzt werden. Beide Pattern sind in der Literatur ausgiebig beschrieben, allerdings verändert die Verwendung von Serverless-Technologien die Anwendbarkeit etablierter Implementierungsansätze.

7.1.1. Orchestration mit Serverless-Technologien

Google Cloud Workflows ist die offizielle Lösung für die Implementierung von Serverless-Orchestration auf der Google Cloud Platform und es konnte fast keine Literatur zu dem Thema gefunden werden, die alternative Dienste betrachtet. Da Workflows aber aufgrund von Datenschutzbedenken nicht zum Einsatz kommen kann (siehe Abschnitt 6.2.3), müssten zur Verwendung des Orchestration-Patterns eigene Konzepte erarbeitet werden.

Bei typischen Implementierungen einer orchestrationsbasierten Microservices-Architektur wird von einem dauerhaft laufenden Orchestrator-Service ausgegangen [48], der mit wenig Latenz (vorzugsweise aus dem Arbeitsspeicher) den momentanen Zustand eines Vorgangs (*State*) abrufen kann.

Durch die zustandslose Natur der in diesem Projekt verwendeten Cloud Runs ist diese grundlegende Eigenschaft des Orchestrators nicht erfüllbar. Eine Instanz für die Gesamtdauer einer Anfrage laufen zu lassen und synchron auf die Antworten kollaborierender Services zu warten, würde die Kosteneffizienz und Verfügbarkeit der Anwendung deutlich verringern, da die Wartezeit als genutzte Rechenzeit gilt. Ein asynchron arbeitender Orchestrator-Service könnte für jede Antwort gestartet werden und vor der Verarbeitung den State aus einer Datenbank rekonstruieren. Dieser Ansatz würde allerdings im

schlimmsten Fall pro Teilschritt die Latenz eines Kaltstarts und eines Datenbankzugriffs zur Bearbeitungszeit einer Anfrage addieren.

Abgesehen von den genannten Möglichkeiten konnte nur eine sehr geringe Anzahl alternativer Technologien zur Umsetzung von Serverless-Orchestration gefunden werden. Diese sind allesamt in einem sehr frühen Entwicklungsstadium und zu diesem Zeitpunkt nicht in Produktionsreife für die Google Cloud implementiert. Sie könnten sich aber in Zukunft zu geeigneten Optionen für Serverless-Orchestration entwickeln:

1. **TriggerFlow** (2020): TriggerFlow wurde nur für AWS und IBM Cloud implementiert [49] und das GitHub Repository ist seit März 2022 archiviert.
2. **ReactiveFnJ** (2023): ReactiveFnJ ist ein sehr neues Framework mit sehr wenig Aktivität im dazugehörigen GitHub Repository und wurde als Proof-of-Concept bisher nur für AWS implementiert [48].
3. **Unum** (2023): Unum ist für das NMG wohl das vielversprechendste der genannten Projekte, da laut [50] ein Prototyp der Unum Runtime nicht nur für AWS, sondern auch für die GCP entwickelt wurde. Im GitHub Repository scheint allerdings hauptsächlich die AWS-Version weiterentwickelt zu werden.

Abschließend lässt sich sagen, dass als einzige produktionsreife Option zur Umsetzung von Serverless-Orchestration auf der Google Cloud Platform Googles eigener Service Cloud Workflows identifiziert werden konnte. Alternativen sind Gegenstand aktiver Forschung und noch in einem sehr frühen Entwicklungsstadium. Für das NMG-Projekt wurde sich daher gegen die Verwendung von Orchestration zur Modellierung der Anwendungslogik entschieden.

7.1.2. Choreografie mit Serverless-Technologien

Der Verwendung des Choreografie-Patterns steht durch den Einsatz von Serverless-Technologien prinzipiell nichts entgegen. Laut Richardson [18] sind bei der Implementation choreografiebasierter Sagas zwei Herausforderungen zu lösen:

1. Services müssen erhaltene Events zu ihren gespeicherten Domänenobjekten zuweisen können.
2. Das Versenden eines Events muss atomar nach Abschluss einer lokalen Datenbanktransaktion erfolgen.

Zuweisung von Events zu Domänenobjekten

Die erste Herausforderung ist leicht durch die Zuweisung einer *Correlation ID* zu lösen. Gelangt eine Anfrage erstmals in das System, vergibt das API Gateway ihr eine `message_id` in Form einer UUID. Jedes Event enthält die `message_id` der Anfrage, auf die es sich bezieht. UUIDs sind als Correlation IDs für den vorliegenden Anwendungsfall ausreichend, könnten aber zum Beispiel nicht zur Bestimmung einer *totalen Ordnung* aller Anfragen verwendet werden [51].

Verlässliches Versenden von Events

Bei der zweiten Herausforderung handelt es sich um das in Abschnitt 2.3.4 angesprochene *Transactional Messaging*. In App Engines Standard Environment (siehe Abschnitt 6.1.1) gäbe es die integrierte Möglichkeit, das Versenden einer Nachricht als Teil einer Datenbanktransaktion in einer Queue einzureihen [52]. Diese Unterstützung von Transactional Messaging durch die Messaging-Middleware existiert für Pub/Sub allerdings nach bestem Wissen nicht [53], [54].

In [55] wird die von Google vorgeschlagene Umsetzung des *Transactional Outbox*-Patterns mit Serverless-Diensten der GCP gezeigt. Die vorgestellte Lösung nutzt Datastore, einen Vorläufer von Cloud Firestore, als Datenbank. Durch eine Transaktion wird bei jeder Veränderung eines Objekts in der Datenbank atomar ein Event in eine *Outbox*-Tabelle eingefügt, wie in Abb. 7.1 dargestellt.

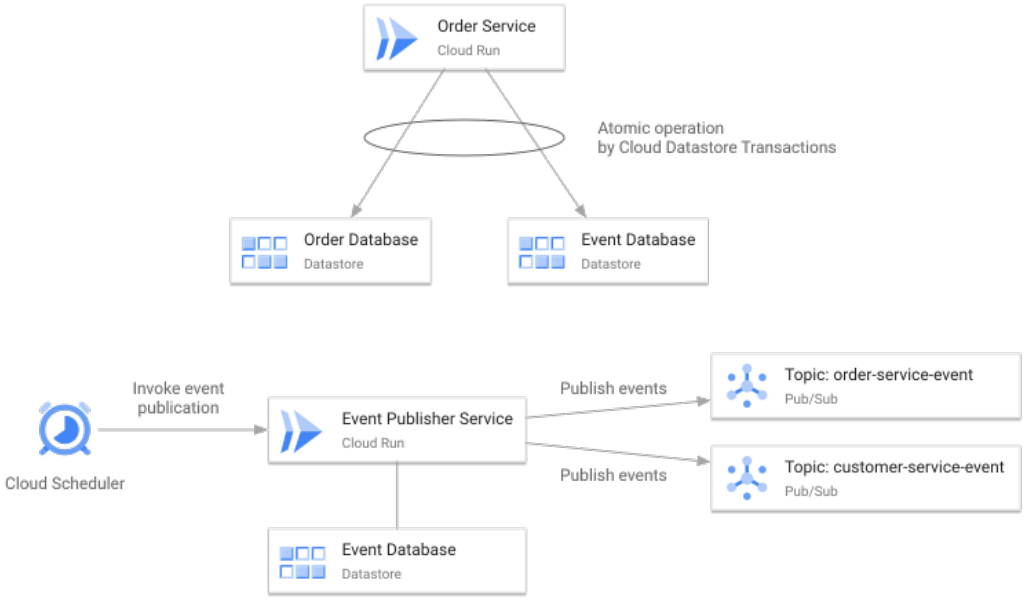


Abbildung 7.1.: Umsetzung von Transactional Messaging auf der Google Cloud Platform. Quelle: Adaptiert von [55].

Die Herausforderung des Outbox-Patterns liegt nun darin, wie die Events aus der Outbox-Tabelle heraus gesendet werden. Es ist dafür immer ein externer Publisher-Service nötig. Ein gängiger Ansatz in Deployments auf „klassischer“ Infrastruktur ist das *Transaction Log Tailing*, bei dem ein dauerhaft laufender Service den Transaktionslog der verwendeten Datenbank überwacht und neue Einträge als Events aussendet [15]. Serverless-Technologien wie Cloud Runs und Cloud Functions sind hingegen nicht für den Dauerbetrieb geeignet, weswegen dieses Pattern nicht umsetzbar ist [56]. Außerdem fehlt der direkte Zugriff auf die Transaktionslogs der Datenbank [54].

Das alternative *Polling-Publisher*-Pattern sieht die Verwendung eines Services vor, der die Outbox-Tabelle periodisch abfragt und hinzugekommene Events versendet [15]. Wie

in Abb. 7.1 zu sehen ist, wurde diese Variante in Googles Beispielarchitektur verwendet. Die periodische Abfrage wird von Cloud Scheduler, dem Serverless-Cron-Job-Dienst der GCP, angestoßen. Die kritische Schwachstelle dieser Implementierung ist Cloud Schedulers kleinstmögliches Wiederholungsintervall von 60 Sekunden. Würden Events nur ein mal pro Minute im System verteilt werden, hätte das gravierende Folgen für die Performance und Skalierbarkeit der Anwendung. Alternativ könnte ein permanent laufender Service mit einer Zeitsteuerung im Anwendungscode das Polling durchführen, was aber aufgrund der vorher genannten Limitationen von Cloud Runs und Cloud Functions nicht umsetzbar ist.

Zum Zeitpunkt der Bearbeitung dieses Projekts konnte keine Möglichkeit identifiziert werden, Transactional Messaging allein mit den in diesem Projekt verwendeten Serverless-Komponenten der Google Cloud Platform umzusetzen. Die offiziell dokumentierte Implementation weist gravierende Probleme in Hinblick auf Performance und Skalierbarkeit auf, während bewährte Pattern aus der Literatur aufgrund der Kurzlebigkeit von Serverless-Prozessen und der fehlenden Integration von Messaging-Middleware und Datenhaltung nicht anwendbar sind. Wie damit in der Implementierung der Anwendung umgegangen wurde, wird in Abschnitt 7.2 behandelt.

Nach Abschluss der Hauptforschungsphase dieser Arbeit und in einem sehr fortgeschrittenen Entwicklungsstadium der in Abschnitt 7.3 behandelten Anwendung wurde eine alternative Methode identifiziert, die potenziell die Implementierung von Transactional Messaging mit Serverless-Diensten der Google Cloud Platform ermöglichen könnte. Wie in Abb. 7.2 gezeigt, nutzt diese Methode Cloud Functions und die noch in Vorschau befindlichen Google Cloud Firestore Trigger, um die zuvor diskutierten Limitierungen zu umgehen. Theoretisch könnte durch diese Methode eine höhere Zuverlässigkeit und bessere Observability erreicht werden, ohne einen permanent laufenden Publisher-Service zu erfordern.

Es sei darauf hingewiesen, dass diese Methode zum Zeitpunkt der Abgabe dieser Arbeit nicht praktisch getestet werden konnte. Daher bleibt die tatsächliche Machbarkeit und Effizienz dieser Lösung ein Thema für zukünftige Forschungen. Die möglichen Vorteile im Hinblick auf die Wahrung der Datenkonsistenz und die Anwendbarkeit etablierter Design-Patterns sind jedoch vielversprechend und könnten eine Lösung für die in dieser Arbeit diskutierten Herausforderungen bieten.

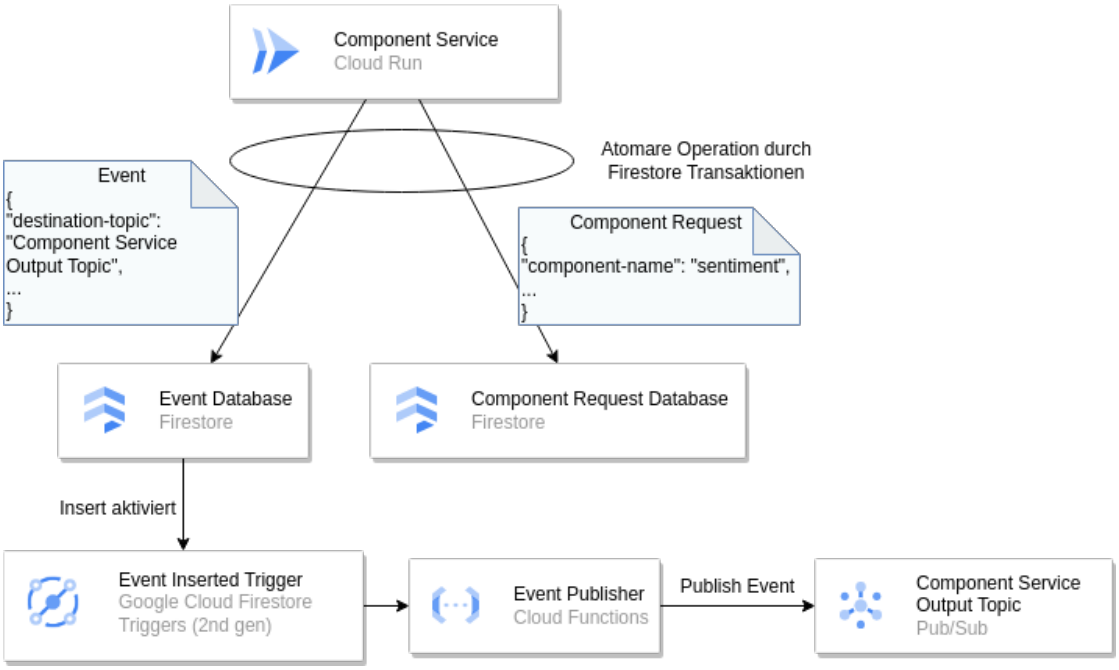


Abbildung 7.2.: Theoretische Umsetzung von Transactional Messaging auf der Google Cloud Platform durch Cloud Function Firestore Trigger. Quelle: Eigene Darstellung

.

7.2. Vollständige Architektur

Mit der in Kapitel 5 beschriebenen Systemarchitektur, den gewählten Technologien aus Kapitel 6 und unter Berücksichtigung der Einschränkungen aus Abschnitt 7.1 wird in diesem Abschnitt das technische Konzept der vollständigen NMG-Anwendung erläutert.

Alle Services der Anwendung werden durch Cloud Runs umgesetzt, die über asynchrones Messaging via Pub/Sub miteinander kommunizieren. Die Laufzeit der Instanzen sollte nicht mit Warten verbracht werden, um kosteneffizient zu sein. Das heißt, dass auf Ergebnisse ausgelöster Vorgänge in anderen Services nicht gewartet werden kann, sondern Instanzen nach Erfüllung ihrer Aufgabe ihre Ausführung direkt beenden. Für die Verarbeitung von Antworten (etwa zur Aggregation) müsste ein Service jedes Mal neu starten und den Zustand der Anfrage aus der Datenbank wiederherstellen.

Das NMG setzt für Vorhersageanfragen stattdessen auf eine Verarbeitung via Choreografie, ohne Request/Response. Bei der Verwendung von Pub/Sub hat das Projektteam drei Kommunikationsmuster betrachtet, die in Abb. 7.3 dargestellt sind. Jeder Service in der Kette kennt, je nach Implementierung, lediglich den vorherigen, den nachfolgenden, oder keinen weiteren Service in der Kette:

1. **Fan-Out** Jeder Service hat ein dediziertes *Output*-Topic, auf das er seine Events und Commands sendet. Er hat keine Kontrolle über die Subscriber (Downstream-Services), die von diesem Topic lesen. Er muss zudem die Output-Topics der Services kennen, an deren Outputs er interessiert ist, um auf ihnen Subscriptions anzulegen.

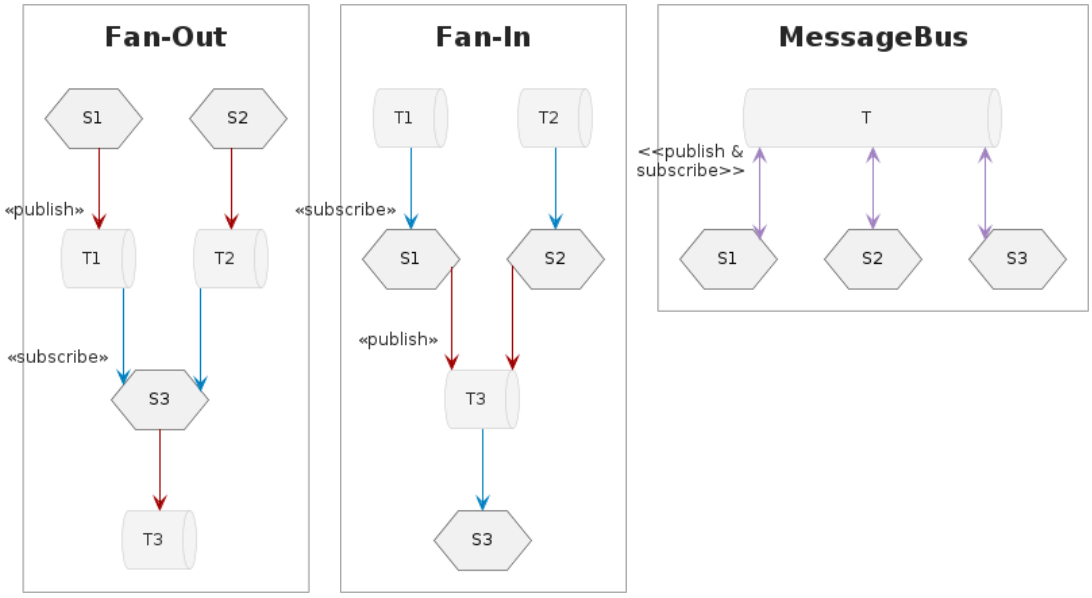


Abbildung 7.3.: Verschiedene Kommunikationsstile choreografiebasierter Sagas. Quelle: Eigene Darstellung.

2. **Fan-In** Jeder Service hat ein dediziertes *Input*-Topic, von dem er Events und Commands erhält. Er hat keine Kontrolle über die Publisher (Upstream-Services), die auf dieses Topic schreiben. Er muss zudem die Input-Topics der Services kennen, denen er seine Outputs schicken will.
3. **Message-Bus** Alle Services sind mit *demselben* Topic verbunden, sowohl lesend, als auch schreibend. Das Topic fungiert als *Message Bus*, bei dem Services keine Kontrolle über andere Publisher und Subscriber haben. Die Messaging-Middleware muss die Filterung von Messages unterstützen, damit Subscriber nicht eine Vielzahl für sie irrelevanter Messages empfangen, was bei Pub/Sub gegeben ist.

Für die NMG-Architektur wurde sich für die Fan-Out-Option entschieden, da sie einen guten Kompromiss aus Anzahl der Komponenten (Topics und Subscriptions) und Observability bietet. Die Ausgaben eines Services leicht an einem Ort auswerten zu können, hat sich gerade in frühen Entwicklungsphasen für simples Debugging als wertvoll erwiesen.

Es gibt zwei Message-Typen, die von allen Services während der Verarbeitung einer Vorhersageanfrage ausgesendet werden:

1. Der Ablauf einer Vorhersageanfrage wird über *Commands* gesteuert. Sie wurden über Message-Attribute und Subscription Filter (siehe Abschnitt 6.2.4) so gestaltet, dass jeweils genau ein anderer Downstream-Service ein Command verarbeitet und potenzielle andere Empfänger es ignorieren.
2. Ergebnisse von Arbeitsschritten werden zusätzlich über *Events* veröffentlicht. Bei den momentan geplanten Funktionalitäten der Anwendung verarbeitet lediglich der Result Service Events, aber es könnten beliebig viele Services (einschließlich keinem) auf ein Event reagieren.

Zwei Operationen der Anwendung finden nicht über Messaging statt:

1. Komponenten rufen ihre KI-Modelle in der Vertex AI mittels des Vertex AI Python SDKs über gRPC auf.
2. Die Ergebnisabfrage zwischen API Gateway und Result Service verwendet REST over HTTP.

Als Datenhaltung wird in der gesamten Anwendung Cloud Firestore verwendet. Jeder Service wird zur Entkopplung strikt auf seine eigenen Dokumente zugriffsbeschränkt. Benötigt ein Service die Daten eines anderen Services, kann er sie nur über dessen API anfragen. Für Monitoring und Analytics werden regelmäßige Batch-Exporte der Daten in BigQuery durchgeführt.

Der generelle Ablauf eines Serviceaufrufs sieht folgendermaßen aus:

1. Der Service erhält von Pub/Sub eine Message, meistens ein Command, als Anhang eines HTTP-*POST*-Requests.
2. Das Command wird verarbeitet.
3. Die neuen oder veränderten Daten werden in Firestore gespeichert.
4. Events und gegebenenfalls Commands werden auf das Output-Topic des Services gesendet.
5. Die Message wird bei Pub/Sub bestätigt.

Da in Abschnitt 7.1 keine mit Serverless-Technologien kompatible Umsetzung für Transactional Messaging identifiziert werden konnte, erfolgen Schritte 3 und 4 nicht atomar. Tritt nach dem Schreiben in die Datenbank ein Fehler auf und es können keine Messages gesendet werden, wurde die Anfrage laut der Datenbank des Services abgeschlossen, aber für den Rest des Systems ist sie verloren gegangen. Dadurch würde die Message allerdings auch Pub/Sub gegenüber nicht bestätigt werden und von diesem erneut zugestellt. Die Message Handler sind *idempotent* gestaltet (siehe Abschnitt 2.3.5), wodurch wiederholte Bearbeitung der gleichen Anfrage problemlos möglich ist.

Um bei kritischen Fehlern Vorhersageanfragen letztendlich als fehlgeschlagen markieren zu können, werden nur endlich viele Zustellversuche unternommen. Ist die maximale Anzahl an Wiederholungsversuchen erreicht, werden Messages auf das zentrale *Dead-Letter*-Topic der Anwendung geleitet. Dieses Topic wird, genau wie alle erfolgreich gesendeten Events, vom Result Service ausgewertet, der als *Single Source of Truth* fungiert. Er konstruiert aus der Kombination von fehlgeschlagenen und erfolgreichen Arbeitsschritten den Bearbeitungsstatus jeder Vorhersageanfrage und gibt diesen auf Anfrage zurück.

Es wurde beschlossen, dass serviceübergreifende Dateninkonsistenz durch das Fehlen einer praktikablen Lösung für Transactional Messaging annehmbar ist, solange der Status einer Vorhersageanfrage an einem Punkt, dem Result Service, verlässlich abgefragt werden kann. Bei den geplanten Funktionalitäten werden die serviceeigenen Datenbanken nur zur Sicherstellung von Idempotenz verwendet. Die Anwendungslogik arbeitet zustandslos und wird nicht auf Basis lokal gespeicherter Daten ausgeführt. Aus diesen Gründen wurde auf die Implementierung kompensierender Transaktionen verzichtet.

Für die weitere Arbeit mit der Anwendung ist zu beachten, dass der beschlossene Kompromiss Auswirkungen auf die Interpretierbarkeit der Daten der einzelnen Services hat. So kann zum Beispiel für die Rechnungsstellung nicht einfach die Zahl der Anfragen

verwendet werden, die im API Gateway registriert wurden. Anfragen könnten in späteren Bearbeitungsschritten fehlgeschlagen sein, ohne dass das API Gateway dies mitbekommt.

Abb. 7.4 zeigt die finale Systemarchitektur des Nachrichtenmanagements, die nach Berücksichtigung der in diesem Abschnitt behandelten Punkte entworfen wurde. Synchrone Kommunikation wird durch durchgezogene Pfeile und asynchrone Kommunikation durch gestrichelte Pfeile dargestellt.

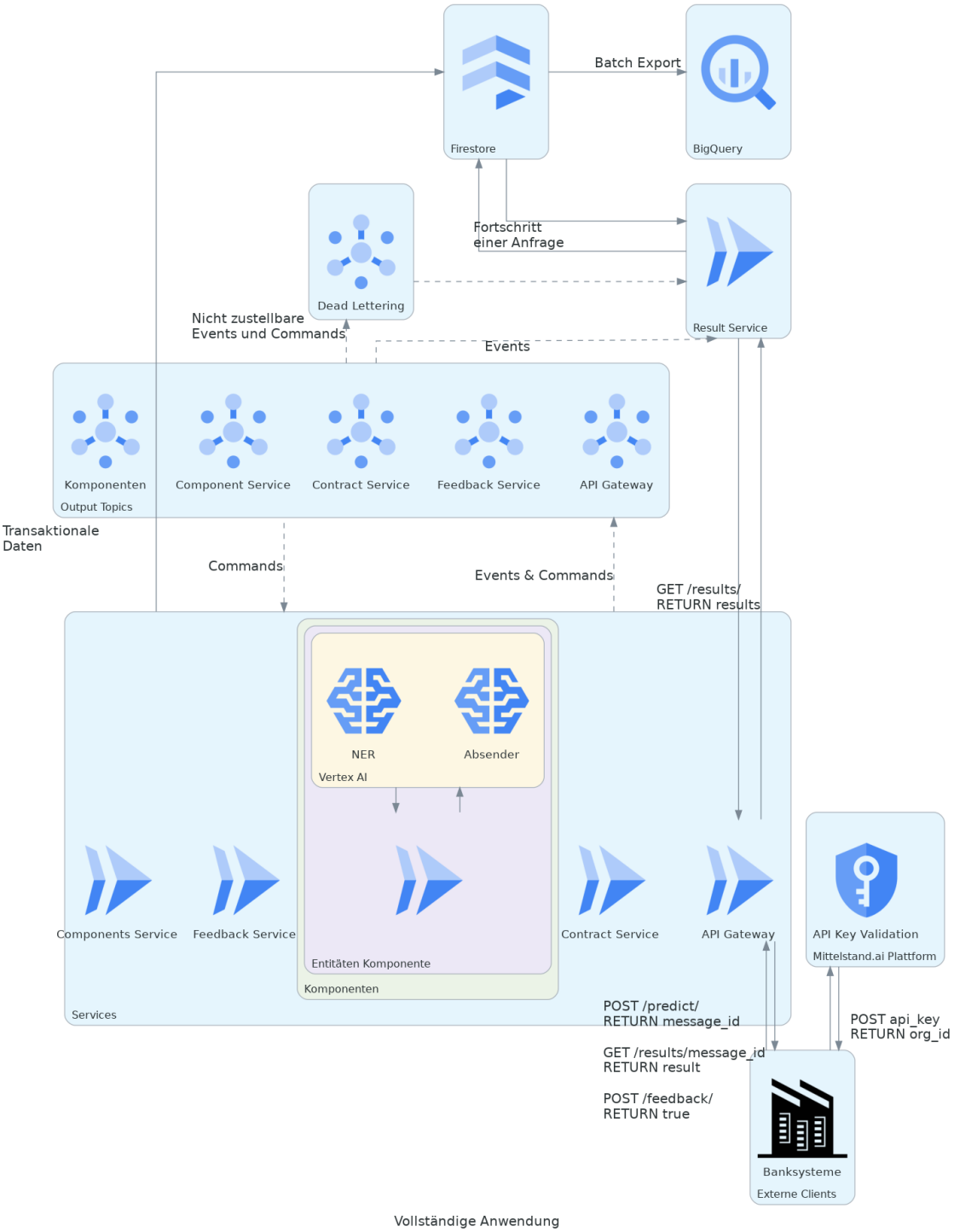


Abbildung 7.4.: Architekturschaubild des vollständigen Produktivsystems. Quelle: Adaptiert von Darstellung im internen Wiki der Mittelstand.ai

7.3. Umsetzung eines MVPs der Anwendung

7.3.1. Design

Die Entwicklung des Nachrichtenmanagements folgt einem iterativen Ansatz mit dem Ziel, zügig ein marktfähiges Produkt zu schaffen. Für den praktischen Teil der vorliegenden Arbeit wurde die in vorherigen Abschnitten konzipierte Systemarchitektur auf ein *Minimum Viable Product* (MVP) reduziert. Das Ergebnis ist die in Abb. 7.5 dargestellte Version der Anwendung mit folgenden Änderungen gegenüber dem finalen System:

1. Kunden zahlen einen monatlichen Festbetrag für die Nutzung aller momentan verfügbaren Komponenten. Der Contract Service wurde aufgrund der Vereinfachung der Bepreisung von Komponenten entfernt. Die Rechnungsstellung erfolgt ebenfalls nicht über integrierte Mechanismen, sondern über externe Prozesse.
2. Die API-Key-Validierung wird durch die externe Analyseplattform der Mittelstand.ai durchgeführt und das API Gateway überprüft die `orgIds` von Anfragen.
3. Feedback-Mechanismen werden in späteren Iterationen implementiert.
4. Monitoring und Analytics mit BigQuery sind nicht enthalten.
5. Für die Modellentwicklung und das Retraining werden keine integrierten Funktionen bereitgestellt.

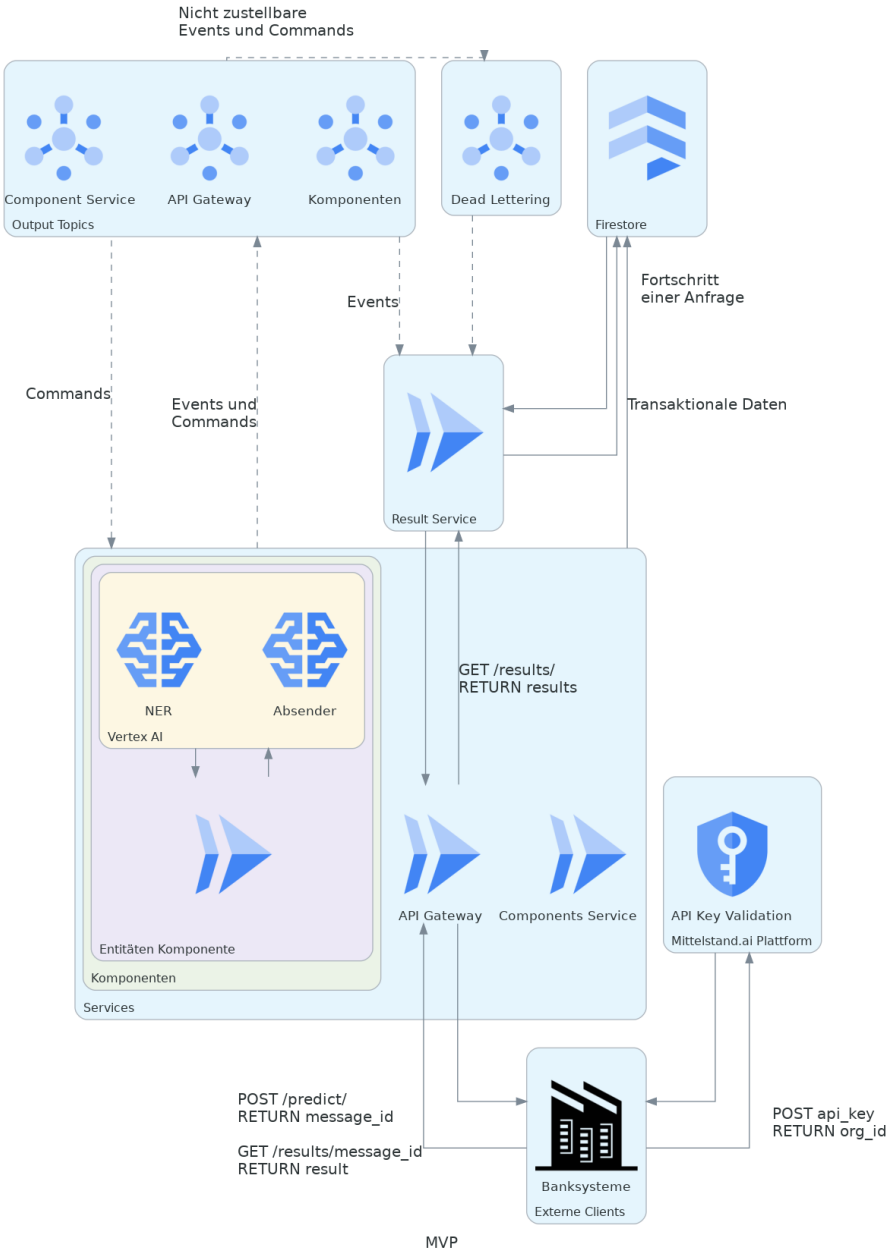


Abbildung 7.5.: Architekturschaubild des Minimum Viable Product des Nachrichtenmanagements. Quelle: Adaptiert von Darstellung im internen Wiki der Mittelstand.ai

7.3.2. Implementierung

Source-Code-Organisation

Für die Organisation des Source Codes in einer Microservices-Anwendung existieren verschiedene Strategien, darunter *Monorepo* und *Multirepo* [57]. Für dieses Projekt wurde der *Multirepo*-Ansatz gewählt, bei dem für jeden Microservice ein eigenes Source Code Repository (Repo) gepflegt wird.

Jedes Repo enthält eine Devcontainer-Spezifikation. Nach dem Klonen des Repos können IDEs wie das bei der Mittelstand.ai verwendete *Visual Studio Code* (VS Code) nach dieser Spezifikation eine containerisierte Entwicklungsumgebung aufsetzen, die alle für das Projekt erforderlichen Programme, Tools und Konfigurationen bereits enthält. Durch das *Dockerfile* in Listing A.3 wird ein Container-Image mit Python in der Version `3.11` und dem in allen Repos verwendeten Dependency-Management-Tool Poetry definiert. Beim Start eines Devcontainers über die Spezifikationsdatei in Listing 7.1 installiert VS Code die angegebenen Extensions, die Google Cloud CLI und ein Plugin für Poetry. Durch diese standardisierte und leicht verständliche „Entwicklungsumgebung-as-Code" wird der Wechsel zwischen Arbeiten an verschiedenen Services erleichtert und Fehlern durch abweichende Konfigurationen von Entwicklungsmaschinen wird vorgebeugt.

Listing 7.1: Beispiel einer Devcontainer-Spezifikation (`devcontainer.json`)

```
{
  "name": "NMG Topic Component",
  "build": {
    "dockerfile": "./Dockerfile"
  },
  "postCreateCommand": "poetry self add poetry-dotenv-plugin",
  "customizations": {
    "vscode": {
      "extensions": [
        "ms-python.python",
        "ms-python.pylint",
        "ms-python.black-formatter",
        "ms-python.isort",
        "tamasfe.even-better-toml"
      ]
    }
  },
  "features": {
    "ghcr.io/dhoeric/features/google-cloud-cli:1": {}
  }
}
```

Aufbau der Services

Die Entwicklung eines allumfassenden *Microservice-Chassis*, das als Grundlage für die Anwendungslogik jedes Services verwendet werden kann, wurde vorerst zugunsten der Entwicklung von Features für das MVP depriorisiert. Zwischen Services wird kein Code über Bibliotheken geteilt, sondern sie werden lose nach den gleichen Prinzipien entwickelt:

- Es wird überall die Validierungsbibliothek Pydantic für Type Hints genutzt.
- Die Konfiguration von Services erfolgt durch `pydantic_settings` über Umgebungsvariablen. Ein Beispiel ist in Listing A.1 gegeben. Durch dieses Package sind alle Konfigurationsvariablen an einem Ort einsehbar und profitieren von Pydantic's starker Validierungslogik. Sie können mit Standardwerten versehen und automatisch in Python-Objekte konvertiert werden, etwa von einem JSON-String in ein *Dictionary*.

- In jedem Service läuft ein Uvicorn Webserver.
- Der Webserver führt eine FastAPI-Applikation aus, die mindestens eine Route zum Verarbeiten von Messages besitzt, siehe Listing 7.2. Der Request Body wird über den Type Hint an der Route in die richtige Klasse umgewandelt. Über ein Union kann FastAPI bei Bedarf automatisch zwischen mehreren Arten von Messages unterscheiden.
- Jedes Repo enthält ein Multistage-Dockerfile, das in der *Build* Stage ein Package aus der Anwendung baut und dieses in der *Runtime* Stage durch Uvicorn ausführt. Das finale Image enthält dadurch keine Abhängigkeiten aus der Build Stage mehr. Ein Beispiel ist in Listing A.4 abgebildet.

Listing 7.2: FastAPI App eines NMG Services

```python
# ....
app = FastAPI()


@app.post(APP_SETTINGS.command_route, summary="Process a command.")
async def process_command(
    command: MakePredictionsCommand,
    publisher_client: Annotated[PublisherClient, Depends(get_publisher_client)],
    firestore_client: Annotated[AsyncClient, Depends(get_firestore_client)],
) -> Response:
    pub_sub_publisher = PubSubPublisher(publisher_client)
    firestore_connector = FirestoreConnector(firestore_client)

    for msg in command.data.messages:
        await firestore_connector.save_request(...)

    prediction_command = PredictionNeededCommand(...)
    for component in command.data.components:
        pub_command = PublishableMessage(
            message=prediction_command,
            attributes={"component_name": component},
        )

        await pub_sub_publisher.publish(pub_command)
        await emit_prediction_requested_events(...)

    return Response(content="OK", status_code=200)


# .....
```

Die einzige Ausnahme geteilter Bibliotheken bilden die KI-Komponenten. Sie sind bis auf Modellaufrufe und Postprocessing gleich, weswegen für sie das `component_wrapper` Package entwickelt wurde. Dieses Package ermöglicht über Vererbung der enthaltenen Klassen die Injektion komponentenspezifischen Codes in ein ansonsten vollständiges Service-Chassis, siehe Listing A.2. Es stellt die Nutzung einheitlicher Datenstrukturen sicher und übernimmt Aufgaben wie die Persistierung von Daten und Versendung von Messages. Dadurch soll die Implementierung neuer Komponenten möglichst vereinfacht werden und auch ohne Wissen über die anderen nötigen Bestandteile eines Services des Nachrichtenmanagements möglich sein. Wie eingangs erwähnt, ist es allerdings nicht generisch

genug, um als Chassis für sämtliche NMG-Services zu dienen.

Für die KI-Modelle, die in der Vertex AI verwaltet und ausgeführt werden, wurde eine Bibliothek, bestehend aus einer Modell-Pipeline und konfigurierbaren Pre- und Postprocessingschritten, entwickelt. Diese Bibliothek wird zusammen mit einer FastAPI-Applikation in ein Container-Image verpackt, welches alle NMG-Modelle ausführen kann. Die Deployments der Modelle unterscheiden sich lediglich durch Umgebungsvariablen, Hardwarespezifikationen und einen Link zum jeweiligen Cloud Storage Bucket der Modelldateien.

7.3.3. Build- und Deployment-Prozess

CI/CD-Pipelines mit Azure Pipelines

Im Rahmen der Implementierung des Microservices-Projekts spielt die Automatisierung von Build- und Deployment-Prozessen eine entscheidende Rolle. *Azure Pipelines*, ein Service von Microsofts *Azure DevOps*, wurde für die Umsetzung von CI/CD genutzt. Dieser Abschnitt beschreibt die Konfiguration und Funktionsweise einer Pipeline, die durch eine YAML-Datei definiert wird.

Die Pipeline-Spezifikation in Listing A.5 definiert die verschiedenen Phasen und Aufgaben, die während des Build- und Deployment-Prozesses ausgeführt werden. Der Code ist in zwei Hauptabschnitte unterteilt: den Abschnitt für die Entwicklungsumgebung und den Abschnitt für die Produktionsumgebung. Die Pipeline-Definition enthält folgende Elemente:

- **Trigger**: Die Pipeline wird durch jeden *Git Tag* ausgelöst, der im Repository erstellt wird.
- **Variablen**: Verschiedene Variablen, wie die Python-Version der späteren Laufzeitumgebung, werden definiert.
- **Aufgaben**: Jede Aufgabe führt spezifische Aktionen wie das Einloggen in die Google Artifact Registry, das Erstellen eines Docker-Images und das Hochladen des Images in die Registry aus.

Die Verwendung von Azure Pipelines ermöglicht eine effiziente und zuverlässige Automatisierung des gesamten Software-Lebenszyklus. Durch die Integration in das NMG-Projekt wird sichergestellt, dass jeder Service ohne großen Aufwand gebaut und bereitgestellt werden kann.

Infrastrukturmanagement mit Terraform

Neben der Automatisierung der Build- und Deployment-Prozesse ist das Management der zugrundeliegenden Infrastruktur ein weiterer kritischer Aspekt der Implementierung von Microservices. Für diese Aufgabe wurde Terraform verwendet. Terraform ermöglicht die Automatisierung der Infrastrukturverwaltung durch Code (Infrastructure-as-Code, IaC). Im Kontext dieses Microservices-Projekts wird Terraform verwendet, um die Cloud-Infrastruktur in einer konsistenten und reproduzierbaren Weise aufzubauen.

Da Ressourcendefinitionen als Code geschrieben sind, können sie mit Git versionsverwaltet werden.

Terraform ist ein umfangreiches Thema mit einer Vielzahl von Funktionen und Möglichkeiten. Eine detaillierte Behandlung würde den Rahmen dieser Arbeit übersteigen. Daher wird kein Terraform-Code aus der Anwendung gezeigt oder im Detail erläutert. Obwohl es in dieser Arbeit nicht im Detail behandelt wird, spielt es eine entscheidende Rolle für die Automatisierung, Skalierbarkeit und Wartbarkeit der Infrastruktur des Nachrichtenmanagements.

8. Tests und Evaluation

8.1. Bewertung der Skalierbarkeit und Performance

Zur Bewertung der Skalierbarkeit und Performance der Anwendung wurden mehrere Testreihen auf der in Abschnitt 7.3 implementierten Anwendung durchgeführt. Es wurden jeweils zwei der momentan verfügbaren KI-Komponenten angefragt (*Topics* und *Entities*). Zwischen den Testreihen wurden die Skalierungsparameter der Services und KI-Modelle verändert und die Antwortzeit jeder Anfrage gemessen. Testläufe innerhalb der Testreihen gliederten sich in Kurzzeittests, Langzeittests und Lastspitzentests. Die Parameter für Anfragen pro Minute und gleichzeitige Anfragen bei Lastspitzen wurden den Anforderungen entnommen (vgl. Abschnitte 4.4.2 und 4.4.3) und folgende Erfolgskriterien festgelegt:

1. Die maximale Antwortzeit pro Nachricht soll zehn Minuten betragen.
2. Die Anwendung muss den prognostizierten Nutzerverkehr des ersten Betriebsjahres bewältigen können (tausende Anfragen am Tag).
3. Lastspitzen von bis zu 1000 simultanen Anfragen müssen toleriert werden.

Zum besseren Verständnis der gezeigten Graphen sei angemerkt, dass die Topic-Komponente dreimal so viele Modelle ansprechen muss, wie die Entity-Komponente, wodurch sie bei gleicher Ressourcenallokation schneller Leistungsgrenzen stößt.

8.1.1. Kurzzeittests

Bei den Kurzzeittests wurden je eine Minute lang Vorhersageanfragen mit gleichbleibender Frequenz an die Anwendung gesendet. Die Frequenz wurde zwischen Testläufen stufenweise von 20 auf 100 Anfragen pro Minute erhöht. Zusätzlich wurden die maximal zulässigen Instanzen aller Services und KI-Modelle variiert:

1. Services und KI-Modelle wurden auf jeweils eine Instanz beschränkt (keine Skalierung), dargestellt in Abb. A.4.
2. Nur Services konnten auf jeweils zehn Instanzen skalieren, dargestellt in Abb. A.5.
3. Services und KI-Modelle konnten auf jeweils zehn Instanzen skalieren, dargestellt in Abb. A.6.

Aus den Diagrammen geht hervor, dass bei erlaubter Skalierung der Services und KI-Modelle auf jeweils bis zu zehn Instanzen eine Abarbeitung von 100 Anfragen pro Minute in durchschnittlich unter zwei Sekunden erfolgen kann. Das würde 144000 Anfragen pro Tag entsprechen und genügt zur Erfüllung der Anforderungen.

8.1.2. Langzeittests

Die Ergebnisse der Kurzzeittests wurden in Langzeittests validiert, indem über einen Zeitraum von 15 Minuten 100 Anfragen pro Minute an die Anwendung geschickt wurden. Dabei wurden die maximal zulässigen Instanzen aller Services und KI-Modelle erneut variiert:

1. Modelle und Services wurden auf jeweils eine Instanz beschränkt (keine Skalierung), dargestellt in Abb. A.7.
2. Services und KI-Modelle konnten auf jeweils zehn Instanzen skalieren, dargestellt in Abb. A.8.

Die Ergebnisse der Dauerlasttests zeigen deutlich, dass das System bei zulässiger Skalierung der Services und KI-Modelle auf jeweils bis zu zehn Instanzen die Anforderungen bezüglich der Antwortzeit von maximal zehn Minuten einhalten kann. Ohne Skalierung wäre dies nicht möglich gewesen.

8.1.3. Lastspitzentests

Zur Evaluation der Elastizität des Systems wurden innerhalb kürzester Zeit Anfragen an die Anwendung geschickt. Die Frequenz der Anfragen war durch das verwendete Testskript auf ungefähr 30 Anfragen pro Sekunde limitiert. Alle Services und KI-Modelle durften auf bis zu zehn Instanzen skalieren. Es wurden zwei Testläufe durchgeführt:

1. 100 Anfragen wurden in ungefähr drei Sekunden gesendet, dargestellt in Abb. A.9.
2. 1000 Anfragen wurden in ungefähr 30 Sekunden gesendet, dargestellt in Abb. A.10.

Den Diagrammen ist zu entnehmen, dass Lastspitzen von bis zu 1000 Nachrichten leicht innerhalb der geforderten zehn Minuten abgearbeitet werden können. Dieses Maß an Elastizität wird für die ersten Betriebsjahre als vollkommen ausreichend eingeschätzt. Es sei zudem angemerkt, dass die Obergrenze von zehn Instanzen arbiträr gewählt ist. Bei noch größerem Anfragevolumen oder extremeren Lastspitzen wären die verwendeten Cloud-Dienste praktisch unendlich skalierbar.

Den in Abb. 8.1 dargestellten Metriken der Vertex AI ist zudem entnehmbar, dass die KI-Modelle eine maximale Latenz von ungefähr 17 Sekunden melden, die höchsten Antwortzeiten der Anfragen aber bei über 400 Sekunden liegen. Die Ursache für diesen Umstand konnte zum Abgabezeitpunkt dieser Arbeit nicht weiter untersucht werden. Es wird allerdings vermutet, dass die größten Verzögerungen durch die Wiederholungsrichtlinien der Pub/Sub-Subscriptions verursacht werden und dass eine Optimierung der entsprechenden Parameter zu einer starken Verbesserung führen könnte, siehe Abschnitt 8.2.1.

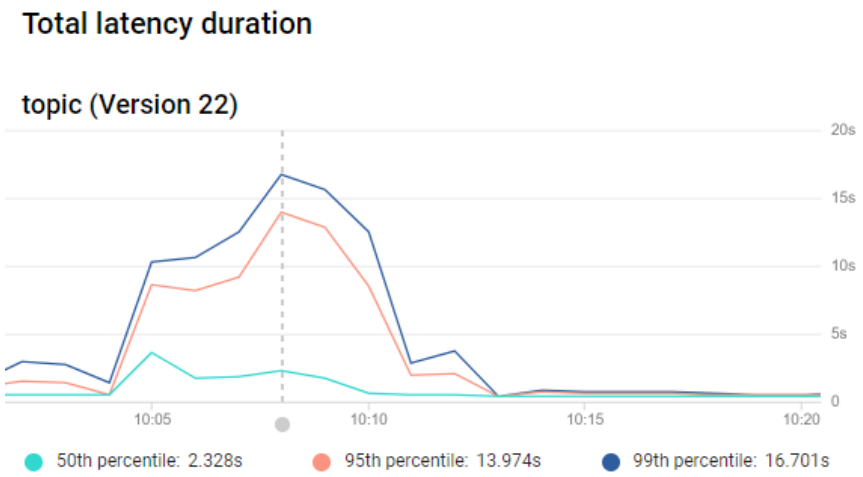


Abbildung 8.1.: Antwortzeiten der KI-Modelle in der Vertex AI. Quelle: Eigene Darstellung.

8.1.4. Ergebnisse

Die durchgeführten Lasttests haben gezeigt, dass die Anwendung die Anforderungen bezüglich Performance, Skalierbarkeit und Elastizität durch die automatischen Skalierungsmechanismen der verwendeten Serverless-Technologien problemlos erfüllen kann. Es sei außerdem angemerkt, dass die Anwendung sehr zuverlässig ist – während der Tests wurde keine einzige Anfrage fallengelassen, was noch im PoC-System ein großes Problem darstellte.

Dank der praktisch unendlichen Skalierbarkeit von Serverless-Technologien ist die Anwendung auch weit über die heutigen Anforderungen hinaus skalierungsfähig. Die Kosten werden zudem durch das *Pay-per-Use*-Modell dieser Technologien niedrig gehalten – gerade in den ersten Betriebsmonaten ist durch kostenlose Kontingente mit fast keinen Kosten zu rechnen.

8.1.5. Limitationen

Die hier durchgeführten Tests konnten nicht alle denkbaren Belastungsszenarien des Produktivbetriebs abdecken. Es wurde nicht untersucht, wie sich die Anwendung unter extremer Last über einen längeren Zeitraum verhält, wie sie zum Beispiel durch einen böswilligen Angreifer auftreten könnte. Die Services hatten während der Tests immer mindestens eine Instanz hochgefahren, wodurch keine Kaltstarts auftraten. Es wurden zudem keine Anfragen an alle Komponenten des Nachrichtenmanagements gleichzeitig getestet.

8.2. Identifikation von Optimierungspotenzialen

8.2.1. Wiederholungsrichtlinien

Wie bereits in Abschnitt 8.1.3 angesprochen, könnte durch die Optimierung der Wiederholungsrichtlinien auf Pub/Sub-Subscriptions die Bearbeitungszeit von Anfragen bei hoher Systemlast verringert werden. Der *exponentielle Back-Off* bei Zustellversuchen steigt momentan auf bis zu 60 Sekunden. Pub/Sub bringt keine zufällige Streuung in diese Verzögerung ein. In Kombination mit dem *Concurrent Request Limit* von Cloud Runs (momentan auf 80 gleichzeitige Anfragen pro Instanz gesetzt) konnte das Phänomen extrem hoher Long-Tail Latency beobachtet werden:

1. Pub/Sub sendet erfolgreich die ersten 80 Nachrichten an einen Service.
2. Bis der Service einige Anfragen abgearbeitet hat, schlagen die nächsten Zustellversuche fehl.
3. Die Nachrichten werden wieder auf die Subscription gelegt und mit einer Verzögerung versehen. Es bilden sich Cluster von Nachrichten, deren geplanter Zustellzeitpunkt immer fast gleich sein wird.
4. Pub/Sub schickt weiter Nachrichten an den Service. Ist gerade zufällig ein Slot frei, ist eine Zustellung erfolgreich. Diese freien Slots werden fast immer von Nachrichten besetzt, die sich im ersten Zustellversuch befinden.
5. In immer größeren Zeitabständen (bis zur Obergrenze) werden die Cluster verzögerter Nachrichten an den Service geschickt.
6. Sind die Cluster größer als das Concurrent Request Limit des Services, werden einige ihrer Anfragen selbst dann noch einmal abgewiesen, wenn der Service sich zwischen den Zustellversuchen komplett im Leerlauf befand.
7. Ist die maximale Verzögerung zu hoch eingestellt, können Serviceinstanzen zwischen Zustellversuchen sogar wieder abgeschaltet werden. Der dadurch erforderliche Kaltstart erhöht die Latenz weiter und führt im schlimmsten Fall zu erneutem Fehlschlagen der Zustellung.

Das hier beschriebene Szenario tritt im normalen Betrieb in der Regel nicht auf, ist aber ein *Edge Case*, den es zu beachten gilt. Zur Vermeidung wäre es zum Beispiel denkbar, dass Pub/Sub die Wartezeit in gewissem Maße zufällig wählt, ähnlich dem CSMA/CD-Zugriffsverfahren von Ethernet. Bis eine solche Funktionalität unterstützt wird, muss durch Optimierung von Wiederholungsrichtlinien und der Parameter der Cloud Runs eine eigene Lösung entwickelt werden.

8.2.2. Kaltstartlatenzen

Kaltstarts bringen verhältnismäßig viel Latenz in ein verteiltes Serverless-System. Während Kommunikation über das Netzwerk in der Regel hunderte Millisekunden dauert, braucht ein Cloud Run des Nachrichtenmanagements zum Starten bis zu 15 Sekunden. Durch „Warmhalten“ einer Instanz für jeden Service kann die Performance des Systems

weiter verbessert werden. Dazu muss in den Skalierungsoptionen der Cloud Runs eine Anzahl minimaler Instanzen größer null eingestellt werden. Solche Instanzen kosten dauerhaft Geld, auch wenn gerade keine Anfragen bearbeitet werden. Anders als bei dauerhaft laufenden Produkten wie Google Compute Engine VMs gibt es bei Cloud Runs allerdings einen vergünstigten Tarif für Leerlaufzeiten [21]. Der monatliche Warmhaltepreis bei mindestens einer Instanz für jeden der sieben Services des NMG MVPs würde ungefähr 90 € betragen[1].

[1] cloud.google.com/products/calculator/#id=a519c4e6-a68c-477f-9342-a80854bca9f3

9. Zusammenfassung und Fazit

Diese Bachelorarbeit beschrieb die Transformation eines bestehenden Softwareprototyps für KI-gestützte Nachrichtenverarbeitungsdienstleistungen namens *Nachrichtenmanagement* in eine Microservices-Architektur. Es wurden Anforderungen an die Applikation und Schwachstellen des Prototyps erfasst, ein *MVP* entwickelt und dieses in einer produktionsreifen Cloud-Umgebung auf der Google Cloud Platform ausführlich getestet. Die Arbeit erfolgte vor dem Hintergrund einer sich rasant entwickelnden Cloud-Technologielandschaft, die durch den Übergang von monolithischen Architekturen zu Microservices und Serverless Computing gekennzeichnet ist (siehe Kapitel 3). Im Kontext des aktuellen Stands der Technik wurde insbesondere auf die Herausforderungen und Vorteile von Serverless-Technologien eingegangen.

Vier Hauptforschungsfragen leiteten die Arbeit: die konzeptionelle Aufteilung der Software in Microservices, die Bewältigung der Herausforderungen eines verteilten Systems, die Umsetzung einer skalierbaren und erweiterbaren Architektur auf der GCP und die Auswirkungen der Verwendung von Serverless-Technologien auf das System. Die erarbeiteten Erkenntnisse zu diesen Forschungsfragen werden in Abschnitt 9.2 noch einmal diskutiert.

9.1. Bewertung der Ergebnisse

Im Rahmen dieser Arbeit wurde eine Microservices-Architektur für die Anwendung *Nachrichtenmanagement* konzipiert (siehe Kapitel 5) und als *Minimum Viable Product* (MVP) umgesetzt (siehe Abschnitt 7.3). Besondere Aufmerksamkeit wurde der Modularität, Skalierbarkeit und dem effizienten Entwicklungsprozess gewidmet.

Das MVP konzentriert sich auf Kernfunktionen der geplanten Anwendung, um eine zügige Markteinführung zu erreichen. Es verzichtet bewusst auf erweiterte Funktionalitäten wie Feedback-Mechanismen und komplexe Monetarisierung, stellt aber eine solide technische Grundlage für zukünftige Entwicklungszyklen dar.

Die Auswahl von Entwicklungsstrategien und -tools trug signifikant zur Effizienz des Projekts bei. Die *Multirepo*-Struktur fördert die Unabhängigkeit der Microservices und erleichtert das parallele Arbeiten im Team. *DevContainer* und *Poetry* unterstützen einen konsistenten und reibungslosen Entwicklungsprozess.

Mit Azure Pipelines und Terraform wurde eine robuste Infrastruktur für die Automatisierung des gesamten Software-Lebenszyklus geschaffen. Diese Auswahl hat es ermöglicht, die Cloud-Ressourcen effizient zu verwalten und eine stetige Integration und Bereitstellung der Anwendungskomponenten zu gewährleisten.

Die mit dem MVP durchgeführten Tests (siehe Kapitel 8) haben gezeigt, dass die Anwendung nicht nur alle gestellten Anforderungen bezüglich der Skalierbarkeit und Per-

formance erfüllt, sondern auch noch Potenzial für zukünftige Erweiterungen bietet. Insbesondere konnten durch automatische Skalierungsmechanismen Lastspitzen von bis zu 1000 gleichzeitigen Anfragen bearbeitet werden. Auch bei einer dauerhaften Last von 100 Anfragen pro Minute über einen Zeitraum von 15 Minuten hat das System alle Anforderungen erfüllt. Diese Leistungsmerkmale lassen darauf schließen, dass die Anwendung den prognostizierten Nutzerverkehr des ersten Betriebsjahres leicht bewältigen wird.

Dennoch gibt es Raum für Optimierungen, insbesondere im Bereich der Wiederholungsrichtlinien des Messaging-Services und der Kaltstartlatenzen. In Abschnitt 8.2 wurde erläutert, dass die aktuellen Wiederholungsrichtlinien unter bestimmten Bedingungen zu unerwartet hohen Latenzen führen können. Eine Überarbeitung dieser Mechanismen könnte die Systemleistung unter hohen Lastbedingungen weiter verbessern.

Die Anwendung ist dank des Pay-per-Use-Modells der verwendeten Serverless-Technologien sehr kosteneffizient. In den ersten Betriebsmonaten ist, dank der kostenlosen Kontingente der Cloud-Dienste, mit fast keinen Kosten zu rechnen. Langfristige Kosten, wie sie etwa durch das Warmhalten von Instanzen entstehen könnten, sind überschaubar und im Kontext der erreichten Leistungssteigerung gerechtfertigt.

9.2. Beantwortung der Forschungsfragen

In der Einleitung wurden aus den Zielsetzungen dieser Arbeit insgesamt vier Forschungsfragen abgeleitet, siehe Abschnitt 1.3.1. Es lässt sich sagen, dass alle gestellten Forschungsfragen beantwortet werden konnten und dass die dabei entstandenen Ergebnisse die Zielsetzungen dieser Arbeit erfüllen.

Die konzeptionelle Aufspaltung der Anwendung wurde in Kapitel 5 durch die Anwendung eines systematischen Dekompositionsverfahrens unter Zuhilfenahme von *Domain-Driven Design* durchgeführt. Als Grundlage dienten ein bereits existierender Prototyp und eine Projektvision, sowie die Ergebnisse der vorgenommenen Anforderungsanalyse.

Zur Bewältigung der speziellen Herausforderungen und Komplexität einer verteilten Anwendung wurden in Abschnitt 2.3 verschiedene Konzepte wie das *Saga-Pattern* und *Idempotenz* vorgestellt. In den Abschnitten 7.3.2 und 7.3.3 wurde beschrieben, wie die Nutzung moderner Entwicklungswerkzeuge den Aufwand der Entwicklung und Bereitstellung mehrerer Softwareartefakte reduziert und somit einige der größten Kritikpunkte an Microservices-Architekturen leichter zu bewältigen macht.

Eine besondere Herausforderung dieses Projektes war, dass der Auftraggeber großen Wert auf minimalen operationalen Aufwand und hohe Kosteneffizienz legt. Dadurch wurden schon zu Projektbeginn gängige Technologien wie Kubernetes und das dazugehörige Microservices-Ökosystem ausgeschlossen und es mussten andere Lösungsansätze untersucht werden. Um eine Microservices-Architektur ohne hohes Maß an operationalen Tätigkeiten umzusetzen, wurden in Abschnitt 2.4 und Kapitel 6 Serverless-Technologien in den Fokus dieser Arbeit gerückt. Einige verfügbare Dienste der Google Cloud Platform wurden im Hinblick auf ihre Eignung für das vorliegende Projekt diskutiert. Es wurden projektspezifische, aber auch allgemeingültige Einschätzungen bezüglich dieser Technologien gegeben, die als Entscheidungshilfen für zukünftige Arbeiten auf dem Gebiet der Serverless-Microservices-Architekturen dienen können.

Die Skalierbarkeit der Anwendung wurde in Kapitel 8 durch die Durchführung von Lasttests demonstriert und es wurden Optimierungspotenziale identifiziert. Lediglich die Gewährleistung der Erweiterbarkeit der Anwendung wurde nicht explizit nachgewiesen. Sie leitet sich aus den zugrunde liegenden Designprinzipien und Dekompositionsverfahren ab, die im Verlauf der Arbeit erläutert wurden, und wird sich erst durch die kontinuierliche Weiterentwicklung der Anwendung in kommenden Iterationszyklen bestätigen.

In Abschnitt 7.1 wurden die Herausforderungen im Hinblick auf die Anwendbarkeit des Saga-Patterns behandelt, die durch die Verwendung von Serverless-Technologien entstehen. Lösungsansätze aus dem Google Cloud Architecture Center wurden kritisch untersucht und für nicht praktikabel befunden. Lösungen durch die Verwendung von Cloud-Diensten, die im Nachrichtenmanagement nicht zum Einsatz gekommen sind, sowie neueste Entwicklungen aus der Forschung wurden aufgezeigt, damit sie als Grundlage für weitere Arbeiten dienen können.

9.3. Ausblick und zukünftige Entwicklungen

Das Nachrichtenmanagement wird auch nach Abschluss dieser Arbeit bei der Mittelstand.ai iterativ weiterentwickelt werden. Zukünftige Arbeiten werden das MVP durch die Implementierung fehlender Komponenten, wie Monitoring und Analytics, sowie die Einführung von Feedback-Mechanismen erweitern. Die bestehende Architektur und die verwendeten Entwicklungs- und Automatisierungstools bilden eine solide Grundlage für die kontinuierliche Verbesserung und Erweiterung des Systems. Für das vierte Quartal 2023 ist eine geschlossene Testphase geplant und im ersten Quartal 2024 soll die Anwendung mit den in Abschnitt 4.4 beschriebenen Funktionalitäten produktiv geschaltet werden.

In einer fortgeschrittenen Implementationsphase der hier behandelten Iteration des Nachrichtenmanagements wurde eine weitere Möglichkeit zur Umsetzung von *Transactional Messaging* mit Google Cloud Functions erkannt (siehe Abschnitt 7.1.2). Diese Möglichkeit wird das Projektteam zukünftig auf ihren praktischen Nutzen untersuchen.

Literatur

[1] „CCW - Der Kundenservice & Call Center Kongress in Berlin: Trendreport AI: KI-Anwendungen im Kundenservice.“ (3. März 2022), Adresse: https://www.ccw.eu/blog/trendreport-ai-ki-anwendungen-im-kundenservice.html (besucht am 20. 08. 2023).

[2] Alex. „New Spiral Cycle: Why Microservices Are Overrated,“ Medium. (26. Juni 2023), Adresse: https://medium.com/@CalculusBear/new-spiral-cycle-why-microservices-are-overrated-fae0b39aaa3b (besucht am 13. 08. 2023).

[3] J. Soldani, D. Tamburri und W.-J. Heuvel, „The Pains and Gains of Microservices: A Systematic Grey Literature Review,“ *Journal of Systems and Software*, Jg. 146, 1. Sep. 2018. DOI: 10.1016/j.jss.2018.09.082.

[4] „Google Trends,“ Google Trends. (), Adresse: https://trends.google.com/trends/explore?date=all&q=microservices (besucht am 22. 07. 2023).

[5] S. Newman, „Chapter 1: What Are Microservices?“ In *Building Microservices: Designing Fine-Grained Systems*, Second Edition, Sebastopol, CA: O'Reilly Media, 2021, S. 3–34, ISBN: 978-1-4920-3402-5.

[6] C. Richardson, „Chapter 1: Escaping Monolithic Hell,“ in *Microservices Patterns: With Examples in Java*, Shelter Island, New York: Manning Publications, 2019, S. 1–32, ISBN: 978-1-61729-454-9.

[7] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, New York: Manning Publications, 2019, 490 S., ISBN: 978-1-61729-454-9.

[8] M. Richards und N. Ford, „Chapter 9: Foundations,“ in *Fundamentals of Software Architecture: An Engineering Approach*, First edition, Sebastopol, CA: O'Reilly Media, Inc, 2020, S. 119–132, ISBN: 978-1-4920-4345-4.

[9] C. Richardson, „Chapter 2.1: What Is the Microservice Architecture Exactly?“ In *Microservices Patterns: With Examples in Java*, Shelter Island, New York: Manning Publications, 2019, S. 34, ISBN: 978-1-61729-454-9.

[10] „Calculating the True Cost of Moving to the Cloud - Part 2.“ (25. Feb. 2020), Adresse: https://www.weave.works/blog/calculating-the-true-cost-of-moving-to-the-cloud-part-2 (besucht am 04. 10. 2023).

[11] M. Fowler und J. Lewis. „Microservices - a Definition of This New Architectural Term,“ martinfowler.com. (25. März 2014), Adresse: https://martinfowler.com/articles/microservices.html#footnote-etymology (besucht am 22. 08. 2023).

[12] M. Fowler. „MicroservicePremium,“ martinfowler.com. (13. Mai 2015), Adresse: https://martinfowler.com/bliki/MicroservicePremium.html (besucht am 26. 08. 2023).

[13] S. Newman, „Chapter 4: Microservice Communication Styles,“ in *Building Microservices: Designing Fine-Grained Systems*, Second Edition, Sebastopol, CA: O'Reilly Media, 2021, S. 89–117, ISBN: 978-1-4920-3402-5.

[14] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, Second Edition. Sebastopol, CA: O'Reilly Media, 2021, 586 S., ISBN: 978-1-4920-3402-5.

[15] C. Richardson, „Chapter 3: Interprocess Communication in a Microservice Architecture," in *Microservices Patterns: With Examples in Java*, Shelter Island, New York: Manning Publications, 2019, S. 65–109, ISBN: 978-1-61729-454-9.

[16] M. Richards und N. Ford, „Chapter 14: Event-Driven Architecture Style," in *Fundamentals of Software Architecture: An Engineering Approach*, First edition, Sebastopol, CA: O'Reilly Media, Inc, 2020, S. 179–210, ISBN: 978-1-4920-4345-4.

[17] S. Newman, „Chapter 6: Workkflow," in *Building Microservices: Designing Fine-Grained Systems*, Second Edition, Sebastopol, CA: O'Reilly Media, 2021, S. 175–196, ISBN: 978-1-4920-3402-5.

[18] C. Richardson, „Chapter 4: Managing Transactions with Sagas," in *Microservices Patterns: With Examples in Java*, Shelter Island, New York: Manning Publications, 2019, S. 110–145, ISBN: 978-1-61729-454-9.

[19] Apache Software Foundation. „1.3. Eventual Consistency — Apache CouchDB® 3.3 Documentation." (2023), Adresse: https://docs.couchdb.org/en/stable/intro/consistency.html (besucht am 15. 10. 2023).

[20] W. Venema, „Chapter 1: Introduction," in *Building Serverless Applications with Google Cloud Run: A Real-World Guide to Building Production-Ready Services*, First edition, Beijing ; Boston: O'Reilly, 2020, S. 1–12, ISBN: 978-1-4920-5709-3.

[21] Google. „Pricing | Cloud Run," Google Cloud. (), Adresse: https://cloud.google.com/run/pricing (besucht am 12. 10. 2023).

[22] S. Newman, „Chapter 8: Deployment," in *Building Microservices: Designing Fine-Grained Systems*, Second Edition, Sebastopol, CA: O'Reilly Media, 2021, S. 219–274, ISBN: 978-1-4920-3402-5.

[23] Google. „Cloud Run Quotas and Limits," Cloud Run Documentation. (9. Okt. 2023), Adresse: https://cloud.google.com/run/quotas (besucht am 11. 10. 2023).

[24] W. Venema, *Building Serverless Applications with Google Cloud Run: A Real-World Guide to Building Production-Ready Services*, First edition. Beijing ; Boston: O'Reilly, 2020, 175 S., ISBN: 978-1-4920-5709-3.

[25] M. Kolny. „Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%," Prime Video Tech. (22. März 2023), Adresse: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90 (besucht am 22. 07. 2023).

[26] A. Wu, „Taking the Cloud-Native Approach with Microservices," Google Cloud Platform, White Paper, 2017. Adresse: https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf (besucht am 13. 10. 2023).

[27] Tom Grey. „5 principles for cloud-native architecture—what it is and how to master it," Google Cloud Blog. (20. Juni 2019), Adresse: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it (besucht am 13. 10. 2023).

[28] D. M. Naranjo Delgado, „Serverless Computing Strategies on Cloud Platforms,“ Universitat Politècnica de València, Valencia (Spain), Dez. 2020. DOI: 10.4995/Thesis/10251/160916. Adresse: https://riunet.upv.es/handle/10251/160916 (besucht am 13. 10. 2023).

[29] S. Wang, „Thin Serverless Functions with GraalVM Native Image,“ unter Mitarb. von Fraga Barcelos Paulus Bruno, Rodrigo, Müller, Ingo und Alonso, Gustavo, 67 p. 22. Apr. 2021. DOI: 10.3929/ETHZ-B-000480335. Adresse: http://hdl.handle.net/20.500.11850/480335 (besucht am 13. 10. 2023).

[30] G. C. Fox, V. Ishakian, V. Muthusamy und A. Slominski, „Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research,“ White Paper, 2017. arXiv: 1708.08028 [cs]. Adresse: http://arxiv.org/abs/1708.08028 (besucht am 13. 10. 2023).

[31] Cloud Native Computing Foundation. „Cloud Native Computing Foundation Annual Survey 2022,“ Cloud Native Computing Foundation. (31. Jan. 2023), Adresse: https://www.cncf.io/reports/cncf-annual-survey-2022/ (besucht am 13. 10. 2023).

[32] Datadog. „The State of Serverless,“ The State of Serverless. (Aug. 2023), Adresse: https://www.datadoghq.com/state-of-serverless/ (besucht am 13. 10. 2023).

[33] A. Thurai. „Serverless in the Enterprise: Building Stateful Applications,“ The New Stack. (2. Juni 2020), Adresse: https://thenewstack.io/serverless/serverless-in-the-enterprise-building-stateful-applications/ (besucht am 13. 10. 2023).

[34] J. Wachtel. „Serverless vs. Kubernetes: The People's Vote,“ The New Stack. (9. Dez. 2022), Adresse: https://thenewstack.io/serverless-vs-kubernetes-the-peoples-vote/ (besucht am 13. 10. 2023).

[35] S. Eismann, J. Scheuner, E. V. Eyk u. a., „The State of Serverless Applications: Collection, Characterization, and Community Consensus,“ *IEEE Transactions on Software Engineering*, Jg. 48, Nr. 10, S. 4152–4166, 1. Okt. 2022, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2021.3113940. Adresse: https://ieeexplore.ieee.org/document/9543531/ (besucht am 11. 10. 2023).

[36] H. Balzert und P. Liggesmeyer, „Kapitel 9: Nichtfunktionale Anforderungen,“ in *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*, Ser. Lehrbücher der Informatik, 3. Aufl, Heidelberg: Spektrum Akademischer Verl, 2011, S. 109–133, ISBN: 978-3-8274-2246-0.

[37] P. D. W. Schramm, „Anforderungsanalyse Und -Spezifikation.“ Adresse: https://services.informatik.hs-mannheim.de/~schramm/see/files/Kapitel03.pdf (besucht am 20. 09. 2023).

[38] Google. „About instance autoscaling,“ Cloud Run Documentation. (15. Sep. 2023), Adresse: https://cloud.google.com/run/docs/about-instance-autoscaling (besucht am 21. 09. 2023).

[39] M. Fowler. „BoundedContext,“ martinfowler.com. (15. Jan. 2014), Adresse: https://martinfowler.com/bliki/BoundedContext.html (besucht am 04. 09. 2023).

[40] V. Khononov und J. Lerman, *Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy*, First Edition. Sebastopol, CA: O'Reilly Media, Inc, 2021, 312 S., ISBN: 978-1-09-810013-1.

[41] C. Richardson, „Chapter 2.2: Defining an Application's Microservice Architecture," in *Microservices Patterns: With Examples in Java*, Shelter Island, New York: Manning Publications, 2019, S. 44–64, ISBN: 978-1-61729-454-9.

[42] Martin Fowler. „CQRS," martinfowler.com. (14. Juli 2011), Adresse: https://martinfowler.com/bliki/CQRS.html (besucht am 12. 10. 2023).

[43] F. Milovanović. „Answer to "What Is the Difference between Business Capability and Subdomain Decomposition in Microservices?"" Software Engineering Stack Exchange. (4. Nov. 2021), Adresse: https://softwareengineering.stackexchange.com/a/433291 (besucht am 12. 10. 2023).

[44] Google. „App Engine locations," Google Cloud. (9. Okt. 2023), Adresse: https://cloud.google.com/appengine/docs/flexible/locations (besucht am 12. 10. 2023).

[45] Google. „Choose an App Engine environment," Google Cloud. (10. Okt. 2023), Adresse: https://cloud.google.com/appengine/docs/the-appengine-environments (besucht am 12. 10. 2023).

[46] „Serverless Archives," The New Stack. (), Adresse: https://thenewstack.io/serverless/ (besucht am 12. 10. 2023).

[47] Google. „Understand Cloud Tasks," Google Cloud. (9. Okt. 2023), Adresse: https://cloud.google.com/tasks/docs/dual-overview (besucht am 12. 10. 2023).

[48] U. Bharti, A. Goel und S. C. Gupta, „ReactiveFnJ: A choreographed model for Fork-Join Workflow in Serverless Computing," *Journal of Cloud Computing*, Jg. 12, Nr. 1, S. 63, 24. Apr. 2023, ISSN: 2192-113X. DOI: 10.1186/s13677-023-00429-3. Adresse: https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-023-00429-3 (besucht am 05. 10. 2023).

[49] A. Arjona, P. G. López, J. Sampé, A. Slominski und L. Villard, „Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, Jg. 124, S. 215–229, Nov. 2021, ISSN: 0167739X. DOI: 10.1016/j.future.2021.06.004. Adresse: https://linkinghub.elsevier.com/retrieve/pii/S0167739X21001989 (besucht am 05. 10. 2023).

[50] D. H. Liu, A. Levy, S. Noghabi und S. Burckhardt, „Doing More with Less: Orchestrating Serverless Applications without an Orchestrator," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, S. 1505–1519, ISBN: 978-1-939133-33-5. Adresse: https://www.usenix.org/conference/nsdi23/presentation/liu-david.

[51] M. Kleppmann, „Chapter 9: Consistency and Consensus," in *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*, Sebastopol, CA: O'Reilly Media, 2017, S. 321–383, ISBN: 978-1-4919-0311-7.

[52] Google. „Transactions," Google Cloud. (9. Okt. 2023), Adresse: https://cloud.google.com/appengine/docs/legacy/standard/java/datastore/transactions (besucht am 12. 10. 2023).

[53] Prectron. „Does Google Pub/Sub Support Transactional Messaging?" Stack Overflow. (21. Mai 2023), Adresse: https://stackoverflow.com/q/74310559 (besucht am 12. 10. 2023).

[54] V. Tokarev. „Is There a Way to Implement Transactional Outbox Pattern with GCP Firestore, PubSub and Message Ordering,“ Stack Overflow. (14. Okt. 2021), Adresse: https://stackoverflow.com/q/69572549 (besucht am 12. 10. 2023).

[55] Google. „Transactional workflows in a microservices architecture on Google Cloud | Cloud Architecture Center,“ Google Cloud. (4. Apr. 2022), Adresse: https://cloud.google.com/architecture/transactional-workflows-microservices-architecture-google-cloud (besucht am 22. 07. 2023).

[56] Mete Atamel. „Long-running containers with Workflows and Compute Engine,“ Google Cloud Blog. (9. Feb. 2022), Adresse: https://cloud.google.com/blog/topics/developers-practitioners/long-running-containers-workflows-and-compute-engine (besucht am 14. 10. 2023).

[57] S. Newman, „Chapter 7: Build,“ in *Building Microservices: Designing Fine-Grained Systems*, Second Edition, Sebastopol, CA: O'Reilly Media, 2021, S. 197–217, ISBN: 978-1-4920-3402-5.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

`pydantic_settings` Pydantic Settings bietet optionale Pydantic-Funktionen zum Laden von Einstellungen oder Konfigurationsklassen aus Umgebungsvariablen oder Secrets-Dateien., siehe docs.pydantic.dev/latest/concepts/pydantic_settings/. 58

Agile Softwareentwicklung Ein iterativer und inkrementeller Ansatz zur Softwareentwicklung, der auf Prinzipien wie Zusammenarbeit und Anpassung basiert, siehe agilemanifesto.org/principles.html. 1

BigQuery BigQuery ist ein vollständig verwalteter, serverloser Datenbankdienst der Google Cloud Platform, der für die Analyse großer Datenmengen entwickelt wurde. Es ermöglicht SQL-basierte Abfragen und ist besonders leistungsstark bei der Verarbeitung von Big Data und der Durchführung von Business-Intelligence-Analysen. 53

CI/CD CI/CD (Continuous Integration/Continuous Delivery) ist ein Praxisansatz in der Softwareentwicklung, der die kontinuierliche Integration von Codeänderungen und deren automatisierte Bereitstellung in produktionsnahe Umgebungen betont. Dies beschleunigt die Entwicklung, verbessert die Qualität der Software und ermöglicht eine schnellere Reaktion auf Anforderungsänderungen und auftretende Fehler. 60

CSMA/CD Carrier Sense Multiple Access with Collision Detection, ein Netzwerkzugriffsprotokoll, das Kollisionen in Ethernet-Netzwerken erkennt und behandelt, siehe www.elektronik-kompendium.de/sites/net/1406181.htm. 65

Dead Lettering Dead Lettering ist ein Konzept von Messaging-Diensten, bei dem Nachrichten in ein spezielles Ablageverzeichnis - oft als „Dead Letter Queue“ bezeichnet - verschoben werden. Dies geschieht, wenn die Nachricht aufgrund von Fehlern oder anderen Gründen nicht an ihren vorgesehenen Empfänger übertragen werden kann. Dead Lettering ermöglicht es, diese Nachrichten zu überwachen, zu analysieren und gegebenenfalls Maßnahmen zu ergreifen, um Probleme in der Kommunikation zu beheben. 40

Devcontainer Development Container ermöglicht es, einen Container als vollwertige Entwicklungsumgebung zu nutzen, siehe containers.dev/. 58

Distributed Big Ball of Mud Verteilte Form des Big Ball of Mud, siehe www.ben-morris.com/microservices-rest-and-the-distributed-big-ball-of-mud/. 6

FastAPI FastAPI ist ein hochleistungsfähiges Web-Framework zur Erstellung von APIs mit Python, siehe fastapi.tiangolo.com. 59

Function-as-a-Service Function-as-a-Service (FaaS), ein Serverless-Computing-Modell, bei dem Funktionen in Reaktion auf Trigger ausgeführt werden. 9, 17

GCP Google Cloud Platform, eine Cloud-Computing-Plattform von Google, siehe cloud.google.com/. 37

Google Artifact Registry Google Artifact Registry ist ein von Google Cloud bereitgestellter Dienst zur Verwaltung und Speicherung von Softwareartefakten und Container-Images. Er bietet eine Plattform zur Verwaltung von Build-Artefakten und ermöglicht die einfache Bereitstellung und Verteilung von Software in Cloud-nativen Umgebungen. 60

Google Cloud CLI Die Google Cloud CLI (Command-Line Interface) ist ein Kommandozeilenprogramm, das von Google Cloud bereitgestellt wird und es Entwicklern und Administratoren ermöglicht, mit den Diensten und Ressourcen der Google Cloud Platform zu interagieren. 58

Google Cloud Firestore Trigger Google Cloud Functions direkt durch Datenbankveränderungen anstoßen (Vorschau), siehe cloud.google.com/functions/docs/calling/cloud-firestore. 50

gRPC gRPC ist ein Open-Source RPC (Remote Procedure Call)-Framework, das von Google entwickelt wurde. Es verwendet HTTP/2 und ermöglicht effiziente, plattformübergreifende Kommunikation zwischen Anwendungen. 8, 53

Infrastructure-as-a-Service Infrastructure-as-a-Service (IaaS), eine Cloud-Computing-Servicekategorie, die Infrastrukturressourcen wie virtuelle Maschinen bereitstellt. 17, 37

Infrastructure-as-Code Infrastructure-as-Code (IaC), die Praxis, die Bereitstellung und Konfiguration von IT-Infrastruktur mithilfe von Code zu automatisieren. 60

Kubernetes Kubernetes, auch als K8s abgekürzt, ist ein Open-Source-Containerorchestrierungs-Framework, das von Google entwickelt wurde. Es dient dazu, die Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen in einem Cluster zu automatisieren. Siehe kubernetes.io. 3

Large Language Model Ein Large Language Model ist ein künstliches neuronales Netzwerk, das darauf spezialisiert ist, menschliche Sprache zu verstehen und zu generieren. Diese Modelle sind besonders leistungsfähig und werden in verschiedenen Anwendungen wie maschinellem Übersetzen, Textgenerierung und automatisierter Textanalyse eingesetzt. Sie bestehen aus Millionen bis Milliarden von Parametern und werden oft für Aufgaben des Natural Language Processing (NLP) verwendet. 21

Long-Tail Latency Ein Konzept in der Netzwerkleistung, das sich auf unerwartet lange Verzögerungen bei der Datenübertragung bezieht, siehe www.section.io/blog/preventing-long-tail-latency/. 65

Mittelstand.ai Gießener Softwaredienstleister für den Finanzsektor und Partnerunternehmen dieser Arbeit, siehe mittelstand.ai/. 1

Multistage-Build Durch ein Multistage-Dockerfile kann man Docker-Images schneller bauen und kleiner halten, siehe docs.docker.com/build/building/multi-stage/. 59

MVP Minimum Viable Product, ein Produkt mit ausreichenden Funktionen, um die Bedürfnisse der ersten Benutzergruppe zu erfüllen und Feedback zu sammeln, siehe wirtschaftslexikon.gabler.de/definition/minimum-viable-product-mvp-119157. 3

Named Entity Recognition Auch NER, Eigennamenerkennung; eine Technik des Natural Language Processing (NLP), um benannte Entitäten in Texten zu identifizieren, siehe www.ibm.com/topics/named-entity-recognition. 21

NoSQL NoSQL (Not Only SQL) ist eine Kategorie von Datenbankmanagementsystemen, die alternative Ansätze zur Speicherung und Verwaltung von Daten bieten, im Gegensatz zu traditionellen relationalen Datenbanken. NoSQL-Datenbanken sind flexibler und können unstrukturierte oder semi-strukturierte Daten effizient speichern, was sie ideal für Big Data und verteilte Anwendungen macht. 43

Observability Observability ist ein Konzept in der Systemanalyse und -überwachung, das die Fähigkeit beschreibt, den internen Zustand eines Systems anhand seiner externen Ausgaben zu verstehen. In der IT und Softwareentwicklung bezieht sich Observability oft auf die Möglichkeit, komplexe verteilte Systeme und Anwendungen in Echtzeit zu überwachen, um Probleme zu identifizieren, zu diagnostizieren und zu beheben. 41

Operations Suite Integrierte Monitoring-, Logging- und Trace-Dienste für Anwendungen und Systeme, die in Google Cloud ausgeführt werden, siehe cloud.google.com/products/operations. 14

Platform-as-a-Service Platform-as-a-Service (PaaS), eine Cloud-Computing-Servicekategorie, die fertige Deployment-Plattformen für Entwickler bereitstellt. 17, 37

Poetry Poetry ist ein Werkzeug zur Verwaltung von Abhängigkeiten und zur Paketerstellung in Python, siehe python-poetry.org/. 58

PostgreSQL Ein Open-Source relationales Datenbanksystem, siehe www.postgresql.org/. 20

Pydantic Pydantic ist die am meisten genutzte Validierungsbibliothek für Python, siehe docs.pydantic.dev/latest/. 58

ReactiveFnJ ReactiveFnJ ist ein vollständig serverloses und skalierbares Designmodell für Fork-Join-Workflows. GitHub: github.com/anitagoel/ReactiveFnJ. 48

REST REST (Representational State Transfer) ist ein bewährter Architekturstil für die Entwicklung von webbasierten Anwendungen. Er basiert auf dem Einsatz von HTTP-Protokollmethoden zur Datenübertragung und fördert Prinzipien wie Einfachheit, Skalierbarkeit und lose Kopplung, wodurch er zum bevorzugten Ansatz für die Erstellung von Webdiensten geworden ist. 1, 8

Serverless auf AWS Siehe aws.amazon.com/serverless/. 14

Serverless auf Azure Siehe azure.microsoft.com/en-us/solutions/serverless. 14

Serverless auf der GCP Siehe cloud.google.com/serverless. 14, 37

Terraform Ein Open-Source-Tool von HashiCorp zur Bereitstellung und Verwaltung von Infrastructure-as-Code, siehe www.terraform.io/. 60

Triggerflow Triggerflow ist eine skalierbare, erweiterbare und serverlose Plattform für die ereignisbasierte Orchestrierung von serverlosen Workflows. GitHub: github.com/triggerflow/triggerflow. 48

Union Durch Unions kann FastAPI einen Request Body automatisch in eines von mehreren Pydantic-Modellen umwandeln, siehe fastapi.tiangolo.com/tutorial/extra-models/#union-or-anyof. 59

Unum Unum ist ein System zum Aufbau und Betrieb großer FaaS-Anwendungen, die aus vielen FaaS-Funktionen bestehen. GitHub: github.com/LedgeDash/unum. 48

UUID Universally Unique Identifiers (UUIDs) sind 128 Bit lange Zahlen, die als eindeutige Bezeichner im Kontext aller jemals existierenden UUIDs verwendet werden können, siehe datatracker.ietf.org/doc/html/rfc4122. 48

Uvicorn Uvicorn ist eine ASGI-Webserver-Implementierung für Python, siehe www.uvicorn.org. 59

A. Anhang

A.1. Quellcode

Python

Listing A.1: Laufzeit-Konfiguration über `pydantic_settings`

```python
from pydantic import AliasChoices, Field
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    command_route: str = "/command"
    project_id: str = Field(
        default=...,
        validation_alias=AliasChoices(
            "google_cloud_project",
            "project_id",
        ),
    )
    output_topic: str = "component-service.output"
    service_name: str

    model_config = SettingsConfigDict(env_file=".env", extra="allow")
```

Listing A.2: Implementation einer KI-Komponente

```python
# .....
from component_wrapper.abstract_handler import (
    AbstractBaseHandler,
    AbstractHTTPHandler,
    AbstractPubSubHandler,
)
from component_wrapper.models import domain


class EntityComponentMixin(AbstractBaseHandler):
    """Handler Implementation der 'Entitätenkomponente'
    Stellt vor allem die ‘execute()‘ Methode zur Abfrage der Vertex AI Modelle der Entitätenkomponente.

    Logik für übergreifenden Workflow, Persistierung, Authentikation mit Vertex AI, etc. kommen vom AbstractBaseHandler.
    Diese Klasse kann nicht allein verwendet werden - sie muss mit einem
    Kommunikationsadapter kombiniert werden.
    Siehe ‘EntityComponentPubSubHandler‘ und ‘EntityComponentHTTPHandler‘.
    """

    def execute(
```

```
        self, messages: List[domain.CustomerMessage]
    ) -> List[domain.CustomerMessage]:
        """Komponentenspezifischer Modellaufruf.
        Args:
            messages (List[domain.CustomerMessage]): Liste mit Nachrichten, für die
    Predictions generiert werden sollen.
        """
        ...

    def postprocess(
        self, messages: List[domain.CustomerMessage]
    ) -> List[domain.CustomerMessage]:
        """Komponentenspezifisches Postprocessing."""
        ...


class EntityComponentPubSubHandler(EntityComponentMixin, AbstractPubSubHandler):
    """Verbindung der Handler Logik mit einem Pub/Sub Adapter.
    Für Production Usage.
    """

    def __init__(self, *args, **kwargs):
        EntityComponentMixin.__init__(self, *args, **kwargs)
        AbstractPubSubHandler.__init__(self, *args, **kwargs)


class EntityComponentHTTPHandler(EntityComponentMixin, AbstractHTTPHandler):
    """Verbindung der Handler Logik mit einem HTTP Adapter.
    Für Service Tests.
    """

    def __init__(self, *args, **kwargs):
        EntityComponentMixin.__init__(self, *args, **kwargs)
        AbstractHTTPHandler.__init__(self, *args, **kwargs)
```

Docker

Listing A.3: Dockerfile für Devcontainer mit Python und Poetry

```
FROM mcr.microsoft.com/devcontainers/base:bullseye
ARG DEBIAN_FRONTEND=noninteractive
ARG USER=vscode
# Dependency Installation
RUN DEBIAN_FRONTEND=noninteractive \
    && apt-get update \
    && apt-get install -y build-essential --no-install-recommends make \
        ...
# Python und Poetry Installation
USER $USER
ARG HOME="/home/$USER"
ARG PYTHON_VERSION=3.11
ENV PYENV_ROOT="${HOME}/.pyenv"
ENV PATH="${PYENV_ROOT}/shims:${PYENV_ROOT}/bin:${HOME}/.local/bin:$PATH"
RUN curl https://pyenv.run | bash \
    && pyenv install ${PYTHON_VERSION} \
    && pyenv global ${PYTHON_VERSION} \
```

```
    && curl -sSL https://install.python-poetry.org | python3 - \
    && poetry config virtualenvs.in-project true
```

Listing A.4: Dockerfile eines NMG Services

```
# syntax=docker/dockerfile:1

ARG PYTHON_VERSION=3.11
ARG POETRY_VERSION

################################
# PYTHON-BASE
################################
FROM python:${PYTHON_VERSION}-slim-bullseye as python-base
# Python setup
ENV PYTHONUNBUFFERED=1 \
    # prevents python creating .pyc files
    PYTHONDONTWRITEBYTECODE=1 \
    # pip
    PIP_DISABLE_PIP_VERSION_CHECK=on \
    PIP_DEFAULT_TIMEOUT=100 \
    # Mount source code here
    APP_PATH=/opt/app \
    # venv will be created here by poetry
    VENV_PATH=/opt/app/.venv

################################
# BUILDER-BASE
# Used to build deps
################################
FROM python-base as builder-base
ARG POETRY_VERSION
# Poetry setup
ENV POETRY_VERSION=$POETRY_VERSION \
    # make poetry install to this location
    POETRY_HOME="/opt/poetry" \
    # make poetry create the virtual environment in the project's root
    # it gets named `.venv`
    POETRY_VIRTUALENVS_IN_PROJECT=true \
    # do not ask any interactive question
    POETRY_NO_INTERACTION=1
# Prepend poetry and that venv to PATH so that its binaries are found and used first
# https://pythonspeed.com/articles/activate-virtualenv-dockerfile/
ENV PATH="$POETRY_HOME/bin:$PATH"

# Install needed deps for Python deps installation
RUN apt-get update \
    && apt-get install --no-install-recommends -y \
    # deps for installing poetry
    curl \
    # deps for building python deps
    build-essential

# Installing this way respects 'POETRY_VERSION' and 'POETRY_HOME' environment variables
# https://python-poetry.org/docs/master/#installation
RUN --mount=type=cache,target=/root/.cache \
    curl -sSL https://install.python-poetry.org | python3 -

```

```
# Installs the 'bundle' poetry plugin
# https://pypi.org/project/poetry-plugin-bundle/
# Without it, the project source code gets linked into the venv, not actually
# installed, leading to erros when copying the venv to another stage
RUN poetry self add poetry-plugin-bundle

WORKDIR $APP_PATH
# bring source code into the container
COPY component-service/src/ src/
# install code and runtime deps
# uses $POETRY_VIRTUALENVS_IN_PROJECT internally
RUN --mount=type=cache,target=/root/.cache \
    --mount=type=bind,source=./component-service/pyproject.toml,target=$APP_PATH/
    pyproject.toml \
    --mount=type=bind,source=./component-service/poetry.lock,target=$APP_PATH/poetry.
    lock \
    --mount=type=bind,source=./component-service/README.md,target=$APP_PATH/README.md \
    poetry bundle venv $VENV_PATH --without dev

################################
# PRODUCTION
# Final image used for runtime
################################
FROM python-base as production
ENV PATH="$VENV_PATH/bin:$PATH"
WORKDIR $APP_PATH
COPY --from=builder-base $VENV_PATH $VENV_PATH
CMD uvicorn component_service.app:app --host=0.0.0.0 --port=$PORT --workers=$NUM_WORKERS
```

Azure Pipelines

Listing A.5: Auszug der Build Pipeline eines NMG Services

```
# Build Component Service Image und Push to GCP Artifact Registry
trigger:
  tags:
    include:
      - "*"

variables:
  python_version: "3.11"

pool:
  vmImage: ubuntu-latest

steps:
  #-----  Generelle Schritte------  #

  - task: Docker@2
    displayName: "Login to Google Artifact Registry"
    inputs:
      command: login
      containerRegistry: <Bezeichner>

  - task: Docker@2
    displayName: "Build the Docker Image"
```

```
    inputs:
      repository: <Service Repo>
      command: "build"
      dockerfile: "**/Dockerfile"
      tags: latest, $(Build.SourceBranchName)

  #-----   Ausführung von Tests------  #
  - task: bash
  # ...

  #---- ------------------  --------  #

  - task: Docker@2
    displayName: "Push the Docker Image"
    inputs:
      repository: <Service Repo>
      command: "push"
      tags: latest, $(Build.SourceBranchName)

  # Abweichende Schritte in der Production Pipeline
  - task: Docker@2
    # Push in Production erfolgt nur, wenn kein 'rc' in der Package Version steht
    condition: not(contains(variables['Build.SourceBranchName'], 'rc'))
    displayName: "Additional Steps for Production"
    # ... (Gleiche Schritte wie oben, aber in Production Cloud Project)
```

A.2. Abbildungen und Diagramme

Schaubilder

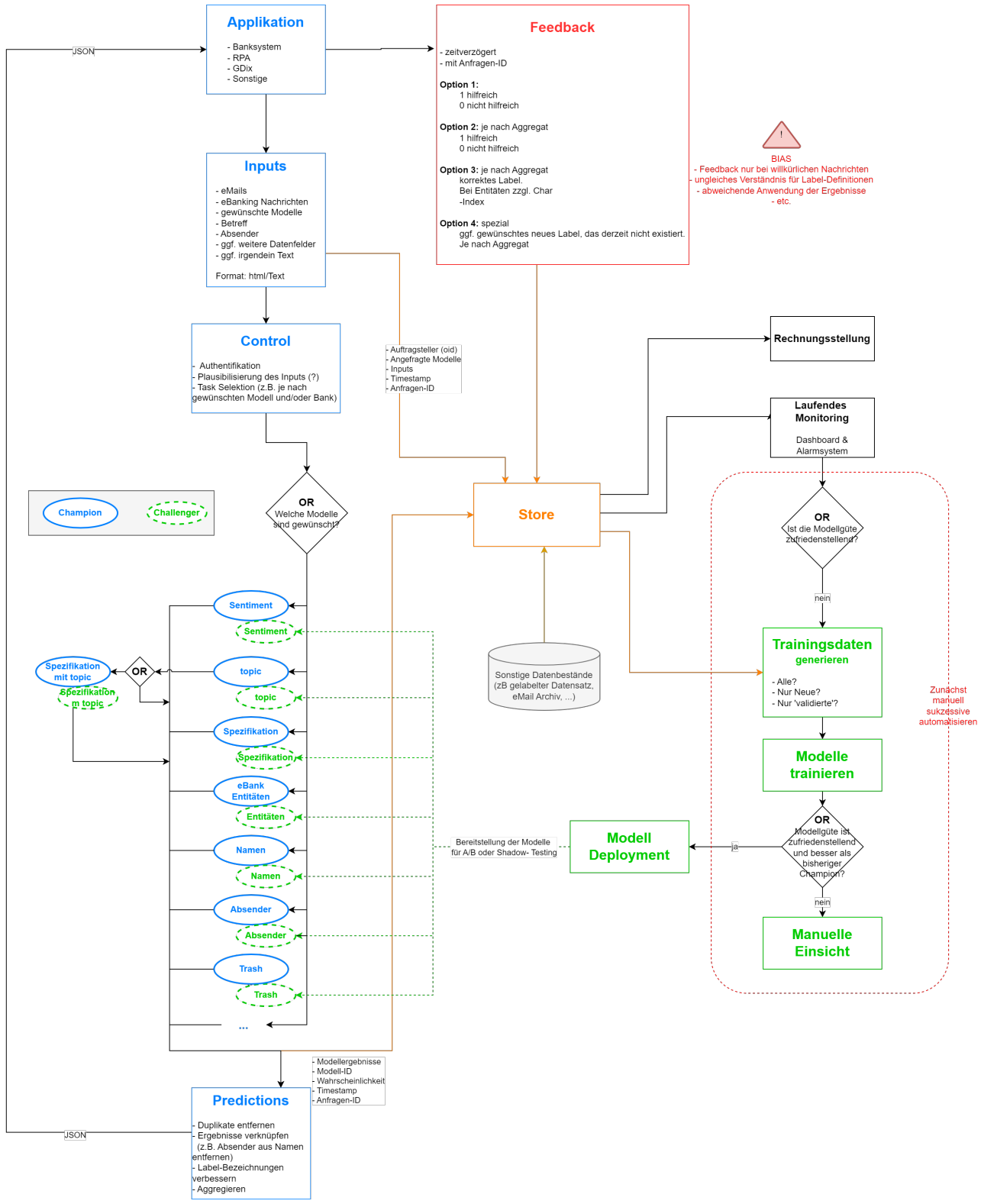


Abbildung A.1.: Geplanter Datenfluss des Produktivsystems. Quelle: Internes Wiki der Mittelstand.ai

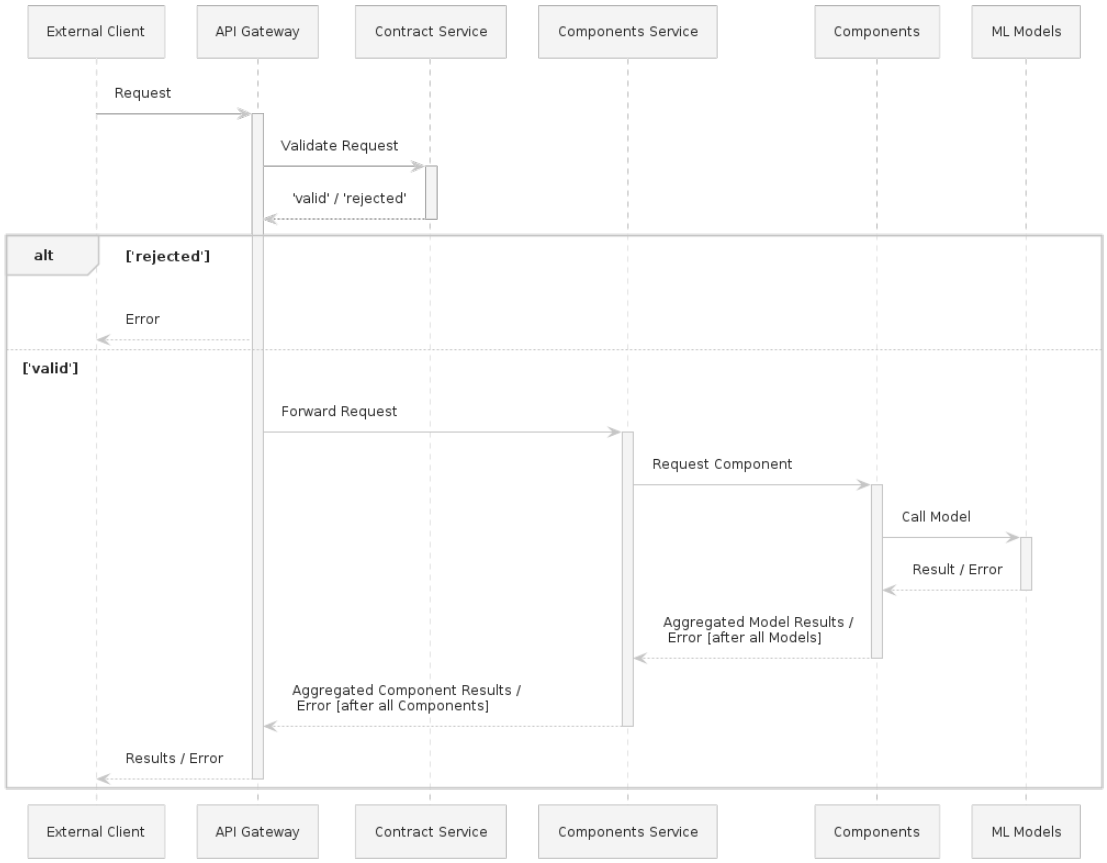


Abbildung A.2.: Sequenzdiagramm einer Vorhersageanfrage ohne CQRS. Quelle: Eigene Darstellung.

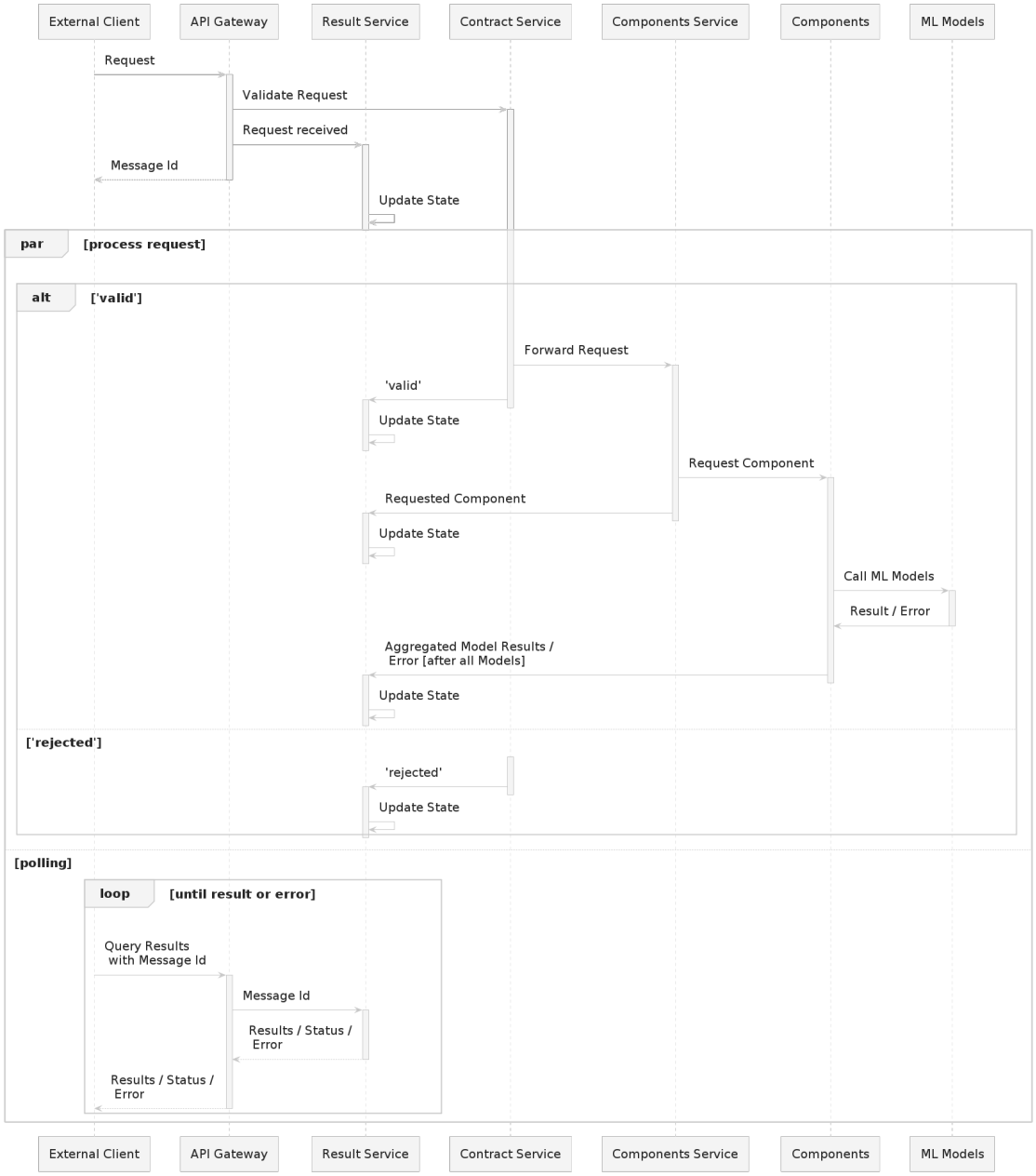
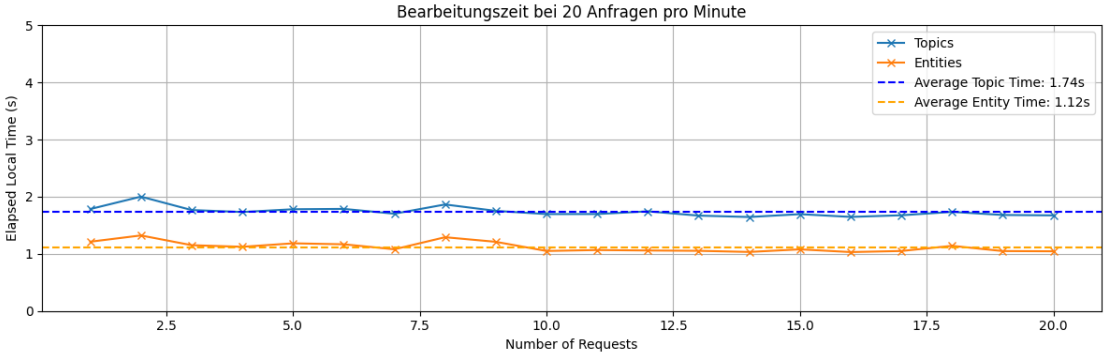


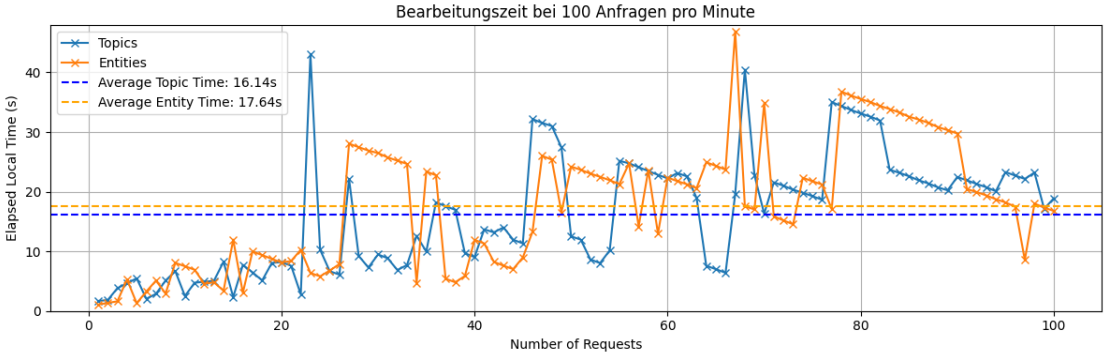
Abbildung A.3.: Sequenzdiagramm einer Vorhersageanfrage mit CQRS. Quelle: Eigene Darstellung.

Lasttests

Kurzzeittests

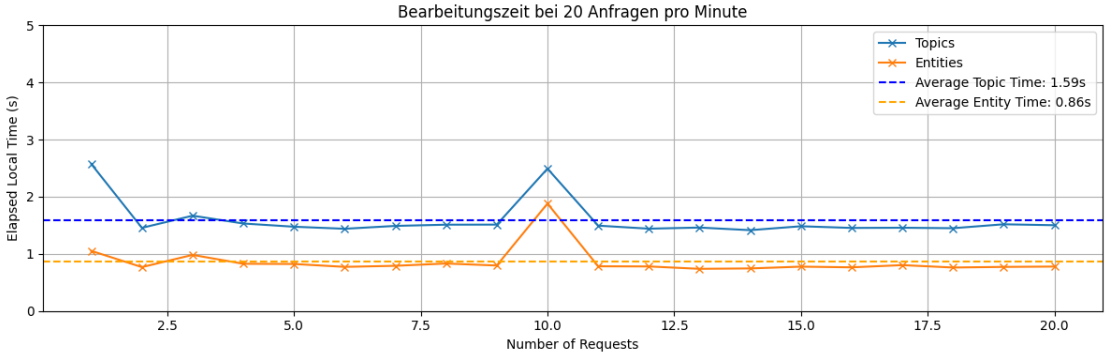


(a) 20 Anfragen/min

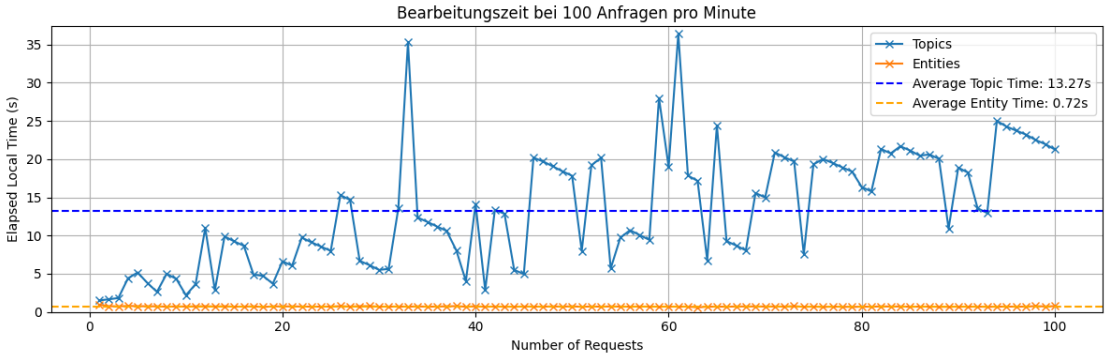


(b) 100 Anfragen/min

Abbildung A.4.: Kurzzeitlasttest ohne Skalierung. Quelle: Eigene Darstellung.

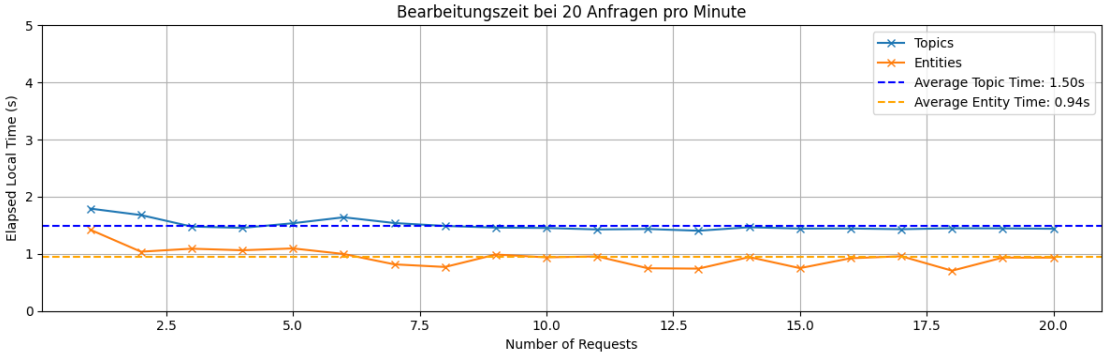


(a) 20 Anfragen/min

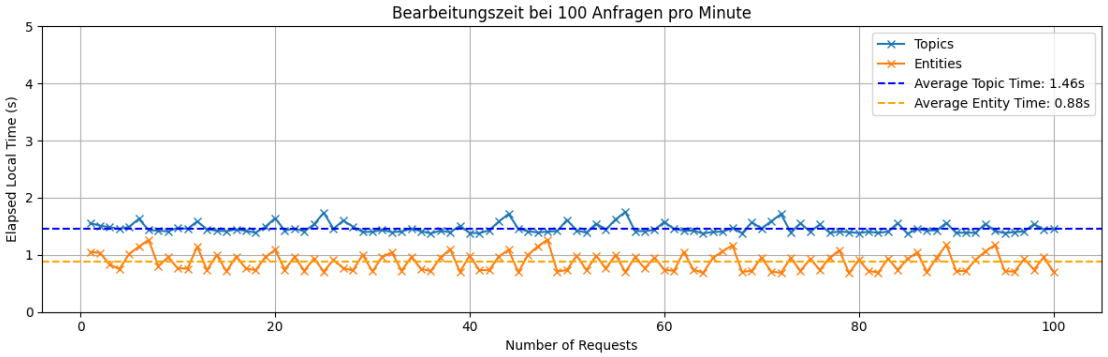


(b) 100 Anfragen/min

Abbildung A.5.: Kurzzeitlasttest mit skalierenden Cloud Runs. Quelle: Eigene Darstellung.



(a) 20 Anfragen/min



(b) 100 Anfragen/min

Abbildung A.6.: Kurzzeitlasttest mit skalierenden Cloud Runs und KI-Modellen. Quelle: Eigene Darstellung.

Dauerlasttests

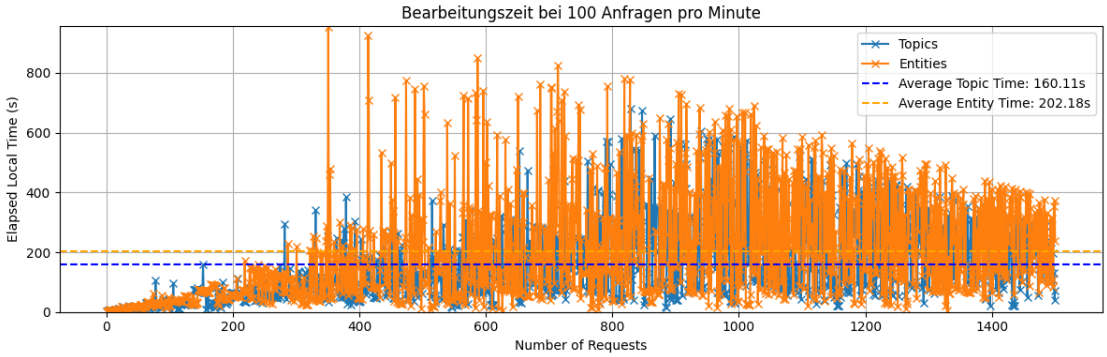


Abbildung A.7.: Dauerlasttest ohne Skalierung bei 100 Anfragen/min. Quelle: Eigene Darstellung.

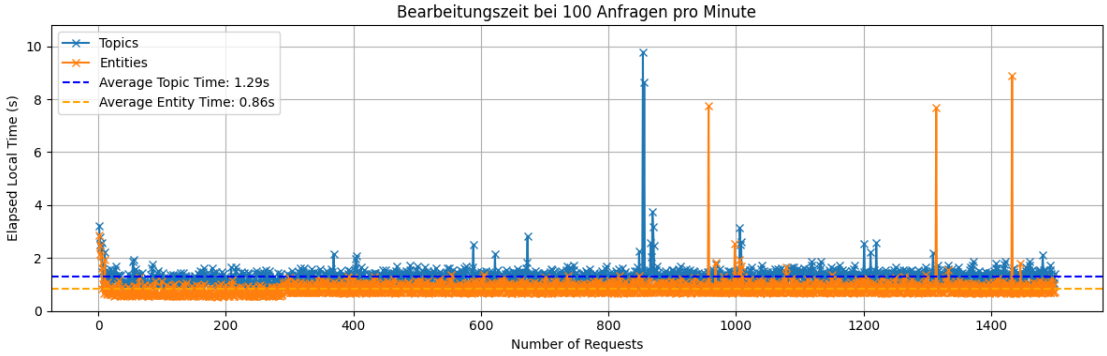


Abbildung A.8.: Dauerlasttest mit skalierenden Cloud Runs und skalierenden KI-Modellen bei 100 Anfragen/min. Quelle: Eigene Darstellung.

Lastspitzentests

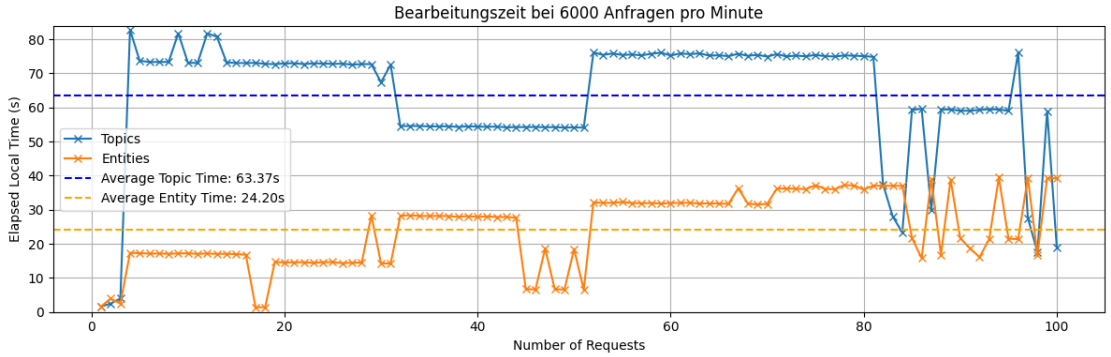


Abbildung A.9.: Lastspitze von 100 Anfragen mit skalierenden Cloud Runs und skalierenden KI-Modellen. Quelle: Eigene Darstellung.

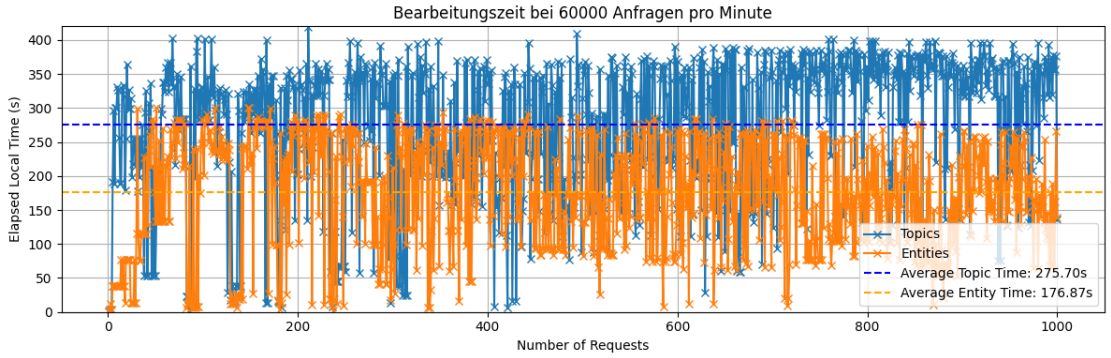


Abbildung A.10.: Lastspitze von 1000 Anfragen mit skalierenden Cloud Runs und skalierenden KI-Modellen. Quelle: Eigene Darstellung.