Hochschule Flensburg

Entwicklung eines PDF-Designers in Go

Cloud Engineering

Lehrende: Florian Hansen

Hauke Ingwersen (670177)

Jannes Nebendahl (750594)

Robert Pfeiffer (740003)

Inhaltsverzeichnis

Inhaltsverzeichnis **2**

Abbildungsverzeichnis **3**

Tabellenverzeichnis **4**

1 Einleitung **5**

2 Allgemeine Grundlagen **6**

2.1 Microservice Architektur **6**

2.2 Test-Driven-Development **6**

2.3 Docker **6**

2.4 CI/CD **6**

2.5 Kubernetes **6**

2.6 Reverse Proxy und Load Balancer **7**

2.7 Thundering Herd Problem **7**

2.8 Request Coalescing **7**

2.9 gRPC **7**

3 Aktuelle Architektur **9**

4 Testaufbau und Testergebnisse **11**

4.1 Arbeitshypothese **11**

4.2 Coalescing Implementierung **11**

4.3 Testaufbau **12**

4.4 Testergebniss **13**

5 Fazit und Ausblick **20**

Literaturverzeichnis **21**

Abbildungsverzeichnis

Abbildung 1: Erste Microservice-Architektur mit Reverse Proxy 9
Abbildung 2: Architektur mit Load Balancer 10
Abbildung 3: Architektur mit Kubernetes Cluster 10
Abbildung 4: Schematische Darstellung der drei Varianten 12
Abbildung 5: Erwarteter Verlauf der Antwortzeiten 14
Abbildung 6: Erster Testlauf 17
Abbildung 7: Zweiter Testlauf 18
Abbildung 8: Dritter Testlauf 19

Tabellenverzeichnis

Tabelle 1: Erster Testlauf 15
Tabelle 2: Zweiter Testlauf 16
Tabelle 3: Dritter Testlauf 16

1 Einleitung

In einer Welt, die zunehmend von digitaler Kommunikation und Datenverwaltung geprägt ist, ist das Ziel dieser Hochschul-Projektarbeit eine Cloud-Anwendung für das Generieren von PDFs zu erstellen. Das Projekt trägt dabei den Namen *PDF-Designer*.

Der PDF-Designer soll es Nutzern ermöglichen, individuelle Vorlagen für ein PDF zu erstellen, welche anschließend dynamisch mit Daten befüllt werden können und PDFs generiert werden können. So kann beispielsweise ein Online-Shop eine Vorlage für seine Rechnungen erstellen. Auf Basis der Bestelldaten kann der Betreiber anschließend automatisch die Rechnungen als PDF generieren lassen und muss die Rechnungen nicht selbst manuell schreiben.

Das Projektteam besteht aus drei Entwicklern, die über unterschiedlich viel Erfahrung im Bereich Cloud Engineering verfügen. Neben der Entwicklung des PDF-Designers steht daher ebenfalls das Aneignen von neuen Fertigkeiten im Fokus des Projekts. Unser Team entschied sich, dass das Backend in der Programmiersprache *Go* entwickelt wird, da diese eine sehr verbreitete Sprache im Bereich des Cloud-Engineerings ist. Mit dieser Programmiersprache hat bislang nur einer der Entwickler Erfahrungen sammeln können. Es wird daher davon ausgegangen, dass das Erlernen von Go eine weitere Herausforderung bei der Umsetzung des Projekts darstellt.

Diese Ausarbeitung beschreibt den Entwicklungsprozess des PDF-Designers. In Abschnitt 2 werden Begriffe erklärt, die notwendig für das Verständnis der entwickelten Cloud-Komponenten sind. In Abschnitt 3 wird die Architektur des PDF-Designers vorgestellt und wie sich diese im Verlauf des Projekts entwickelt hat. Abschnitt 4 befasst sich mit der Arbeitshypothese, dass durch die Implementierung von Request Coalescing am Einstiegspunkt der Cloud-Anwendung geringere Antwortzeiten zu erwarten sind als in einer tiefer liegenden Komponente. In Abschnitt 5 wird abschließend ein Fazit gezogen und die Ergebnisse zusammengefasst.

2 Allgemeine Grundlagen

2.1 Microservice Architektur

Für die Umsetzung dieses Softwareprojektes wird die Microservice-Architektur verwendet. Die Microservice-Architektur wird als eine Alternative zu den monolithischen Architekturen verwendet. In der Microservice-Architektur werden die Aufgaben von einzelnen Einheiten durchgeführt [1].

2.2 Test-Driven-Development

Bei der Anwendung PDF-Designer wird nach dem Test Driven Development (TDD) vorgegangen. Bei TDD werden zuerst Unit-Tests für eine neue Funktionalität geschrieben, obwohl diese Funktionalität noch nicht existiert. Nach dem Schreiben von den Tests wird die Funktionalität geschrieben, damit diese den Test erfolgreich durchläuft [2].

2.3 Docker

Damit das Projekt nicht ständig manuell gestartet wird, werden Docker Container verwendet. Ein Docker Container ist eine isolierte Umgebung, dieser Docker Container hat keine Kenntnis von dem Betriebssystem oder unseren Dateien [3]. Für die Umsetzung von Docker wurde für jeden Microservice eine Dockerfile erstellt. Mit dem Befehl `docker compose up` wurden die einzelenen Dockerfiles gestartet.

2.4 CI/CD

Continous Integration (CI) und Continous Delivery (CD) wird verwendet um automatisiert zu Testen und die Anwendung bereitzustellen. Für die Umsetzung von CI/CD wird Github Actions verwendet. Für CI/CD wird jeweils ein YAML-Skript geschrieben [4]. CI wird verwendet, um die Tests automatisiert laufen zu lassen. CD wird verwendet, um automatisiert das aktuelle Docker-Image auf Docker-Hub zu veröffentlichen.

2.5 Kubernetes

Kubernetes ist ein von Google veröffentlichtes Open-Source Projekt. Google veröffentliche Kubernetes im Jahr 2014. Kubernetes wird verwendet um die Verwaltung von containerisierten Services [5]. Für das Projekt wurde ein Kubernetes-Cluster erstellt. Dazu wurden drei virtuelle Maschinen (VM) mit dem Ubuntu 20.04 Betriebssystem erstellt. Zwei VMs wurden als Worker-Nodes eingerichtet. Eine VM wurde als Master-Node eingerichtet. Am Anfang wurde den Worker-Nodes zu wenig Festplattenspeicher zugewiesen. Dies führte dazu, dass die Worker-

Nodes nicht korrekt funktionierten. Anschließend wurden zwei neue Worker-Nodes mit mehr Festplattenspeicher eingerichtet. Ein kurioses Problem tritt auf, als wir auf den Worker-Nodes das Projekt gestartet haben, dass der Festplattenspeicher immer voller wurde. Einige Services konnten nicht ordnungsgemäß heruntergefahren werden, da die Datenbank nicht korrekt konfiguriert wurde. Für das Problem wurden Umgebungsvariabeln erstellt, um die Datenbank zu verwenden.

2.6 Reverse Proxy und Load Balancer

Reverse Proxy ist eine Art Server der Anfragen an einen anderen Server weiterleitet [6].Das Load-Balancing ist eine Technik zur Verteilung des eingehenden Datenverkehr auf die Server, damit die Anfragen bearbeitet werden können. Um ein Verständnis für Load Balancer herzustellen, wurden drei Load Balancer Algorithmen entwickelt. Unser Team entschied sich für die Load-Balancer Algorithmen Round-Robin, Weighted Round-Robin und IP-Hash. Round-Robin erhält Anfragen, diese Anfragen werden unabhängig von der Serverlast zirkulär verteilt. Weighted Round-Robin, die Server werden nach ihrer Leistungsfähigkeit gewichtet und die Anfragen werden in der Reihenfolge der höheren bis zu niedrigeren Gewichtung der Server entgegengenommen [7]. Für unser Projekt wurden die Gewichte zufällig berechnet und zugewiesen. Beim IP-Hash wird eine Hash-Funktion verwendet, die dann bestimmt welcher Server die Anfrage bearbeiten wird [8] .

2.7 Thundering Herd Problem

Das Thundering Herd Problem ist ein bekanntes Problem. Diese Situation kann unter Unix auftreten, wenn mehrere Prozesse auf ein einzelnes Ereignis warten. Tritt dieses Ereignis auf, wenn zum Beispiel eine Verbindung zum Webserver hergestellt wird, dann werden alle Prozesse, die das Ereignis bearbeiten könnten, gleichzeitig aktiviert. Letztendlich wird nur einer dieser Prozesse in der Lage sein, die Arbeit zu verrichten [9].

2.8 Request Coalescing

Um einer großen Anzahl von Nutzern gerecht zu werden, wird Request Coalesing verwendet. Wenn mehrere Nutzer dasselbe Ziel Anfragen, dann werden diese Anfragen zusammengefasst und die Datenbank wird nur einmal abgefragt [10].

2.9 gRPC

gRPC (Remote Procedure Call) ist ein von Google entwickeltes Open-Source-Framework für die Kommunikation zwischen Anwendungen [11]. RPC ermöglicht eine Client-Server-

Kommunikation, bei der die Anforderungen des Client-APIs so erscheinen, als würden sie lokal aufgerufen oder seien Teil des internen Servercodes. gRPC verwendet HTTP 2. REST verwendet HTTP 1.1 mit den Methoden POST, GET, PUT und DELETE [12].

3 Aktuelle Architektur

Unsere Anwendung besteht aus vier Microservices, dem Web-Service, Creation-Service, Templateing-Service und User-Service. Der Web-Service stellt das User-Interface in Form einer Website bereit. Für die Verwaltung der Nutzer, inklusive Login und Registrierung, ist der User Service zuständig. Der Template-Service wiederum erlaubt dem Nutzer, Vorlagen eines PDFs zu erstellen, in welche anschließend vom Creation-Service verschiedene Daten eingetragen werden können, um daraus eine PDF-Datei zu generieren. Diese Microservice-Architektur bietet den Vorteil der Skalierbarkeit [13].

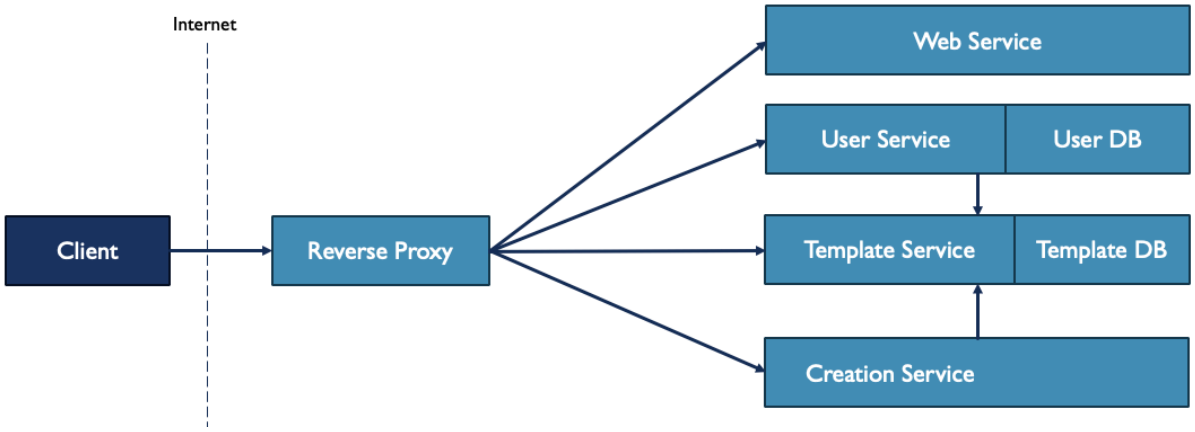

Abbildung 1: Erste Microservice-Architektur mit Reverse Proxy

Die erste funktionale Architektur der Anwendung für ein Bereitstellung mittels Docker ist in Abbildung 1 zu sehen. Die einzelnen Services sind durch einen Reverse Proxy als API-Gateway vor der Außenwelt geschützt. Die Anfrage eines Clients wird über den Reverse Proxy an den jeweiligen Service weitergeleitet. Zur Bearbeitung dieser Anfragen können die Services untereinander kommunizieren und stellen anschließend wieder über den Reverse Proxy eine entsprechende Antwort bereit.

Bevor im Verlauf des Projekts eine Skalierbarkeit der Anwendung durch den Einsatz von Kubernetes erzielt wird, wird zunächst ein eigener Load Balancer für den Templating-Service entwickelt. Dieser Service erhält ein Load Balancing, da sowohl der Client als auch die anderen Services auf diesen Service zugreifen und somit bei diesem Service eine hohe Last zu erwarten ist. Wie in Abbildung 2 zu erkennen ist, wird dafür der Load Balancer zwischen Reverse-Proxy und Templating-Service geschaltet. Auf diese Weise wird die Last durch Anfragen des Clients auf Instanzen des Templating-Services verteilt. Alle Instanzen greifen wieder auf nur eine

Instanz der Template Datenbank zu, um eine Konsistenz dieser gewährleisten zu können und sich nicht mit der Verteilung der Daten auf mehrere Datenbank-Instanzen beschäftigen zu müssen.

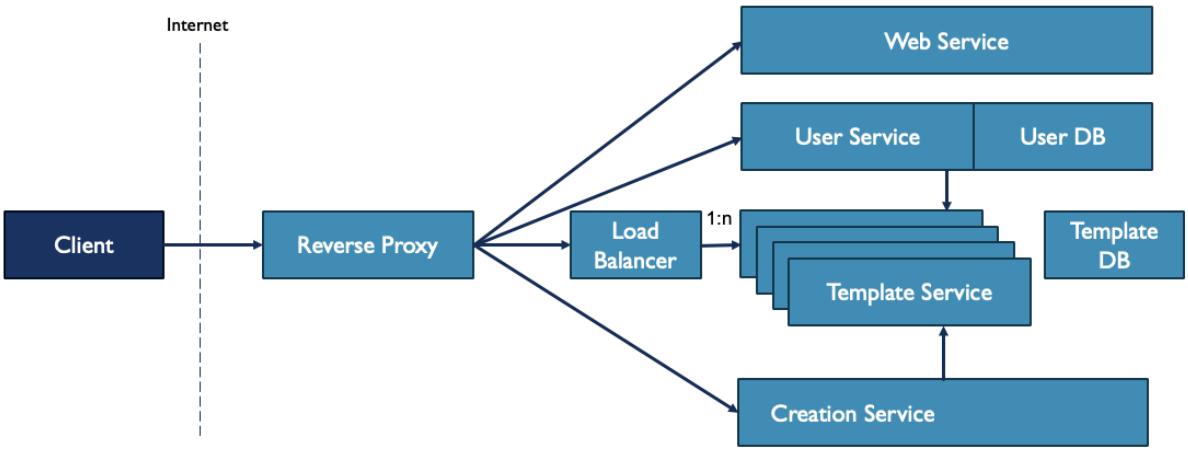

Abbildung 2: Architektur mit Load Balancer

Request Coalescing wurde für das Templating-Service implementiert. Load Balancer wurde nun deaktiviert, stattdessen wird Kubernetes Load Balancer verwendet.

Im Fortgang des Projekts wird der Load Balancer wieder entfernt und für eine Bereitstellung der Anwendung mit Docker wird wieder auf die Architektur in Abbildung 1 zurückgegriffen. Bei der Bereitstellung der Anwendung mit einem Kubernetes Cluster kommt die Architektur in Abbildung 3 zum Einsatz. Diese Architektur unterscheidet sich von der in Abbildung 1 lediglich darin, dass Kubernetes die Last auf mehrere Instanzen der Service verteilt, je nachdem wie viele Instanzen des Services in dem entsprechenden Manifest hinterlegt sind.

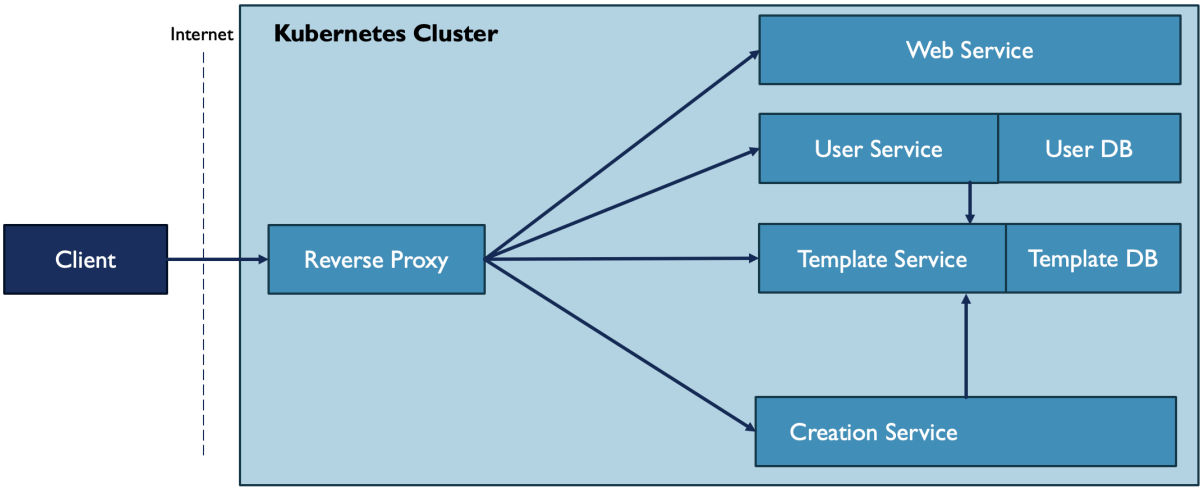

Abbildung 3: Architektur mit Kubernetes Cluster

4 Testaufbau und Testergebnisse

4.1 Arbeitshypothese

Bei der Implementierung von Request Coalescing direkt am Einstiegspunkt der Cloudanwendung, am API-Gateway, sind die kürzere Antwortzeiten zu erwarten, als bei einer Coalescing Implementierung in den Controllern der einzelnen Services oder keinem Coalescing.

Im Rahmen der Vorlesung haben sich die meisten anderen Gruppen für eine Coalescing Implementierung in den Controllern ihrer Services entschieden. Wir hingegen hatten einen Ansatz mit einem eigenem Request Service gewählt, welcher zwischen unser API-Gateway und Templating Service geschalten werden sollte. Eine solche Implementierung würde die Microservice Architektur befolgen. Wir befürchten allerdings, dass der Zeitgewinn des Coalescings durch das Verarbeiten der Kommunikation zwischen den Servicen kompensiert werden könnte und so nicht die schnellstmöglichen Antwortzeiten erreicht werden können. Aus diesem Grund verschieben wir die Coalescing Implementierung aus dem eigenständigen Request Service mit in das API-Gateway.

4.2 Coalescing Implementierung

Das Coalescing wird ausschließlich zum Testen der Arbeitshypothese implementiert und nicht für einen performanteren Betrieb der Cloud-Anwendung. Für das Test-Scenario kommt unserer Templating-Service zum Einsatz. Für diesen sind 3 Varianten implementiert, um das PDF-Template mit einer bestimmten ID bereitzustellen. Diese sind schematisch in der folgenden Abbildung dargestellt.

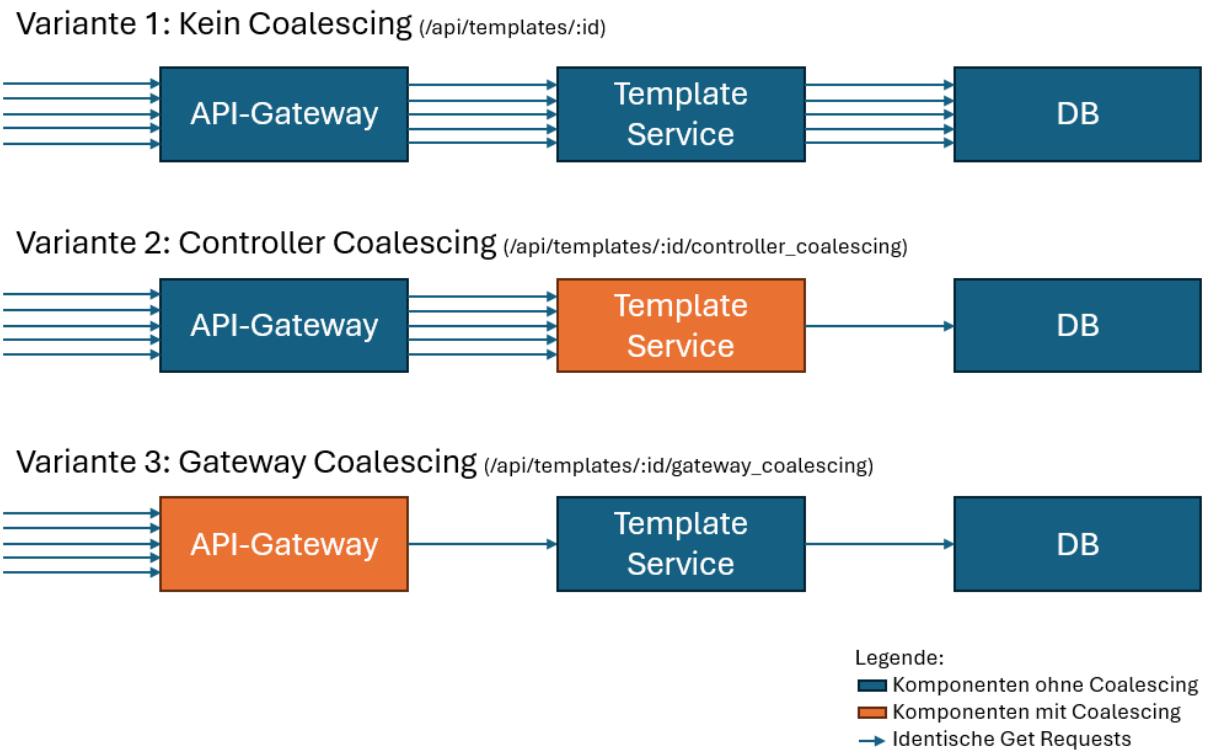

Abbildung 4: Schematische Darstellung der drei Varianten

Bei Variante 1 kommt kein Coalescing zum Einsatz und jede GET-Request wird durch die gesamte Anwendung hindurch geleitet. Dabei bewirken x Anfragen für das gleiche PDF-Template ebenfalls, x Abfragen der Datenbank nach diesem PDF-Template. Bei Variante 2 kommt ein Coalescing dem Controller des Template Service zum Einsatz. Dadurch werden zwar x Anfragen durch das API-Gateway an den Template Service geleitet, von dort an werden die Anfragen für das gleiche PDF-Template jedoch durch das Coalescing zusammengefasst. Bei Variante 3 findet das Coalescing bereits im API-Gateway statt. Hierdurch wird die Last an anfragen direkt zusammengefasst und in der Datenbank muss das Objekt nur ein einziges Mal abgefragt werden.

4.3 Testaufbau

Um die Arbeitshypothese zu überprüfen, entsteht das Modul *coalescingtest*. Zum Ausführen des Tests muss die *main.go* Datei dieses Moduls ausgeführt werden.

Der Test ist so aufgebaut, dass sich die Anzahl an gesendeten Request an jede der 3 Varianten in x Schritte um eine Schrittgröße y erhöht. Dabei wird für jede Request die Zeit zwischen dem Absenden der Request und dem Empfangen der Response gemessen. Nachdem alle Zeiten gemessen sind, wird die mittlere Antwortzeit ermittelt. Dabei werden die kürzesten und längsten

10 % der Antwortzeiten ignoriert, um Ausreißer auszufiltern. Anschließend wird die mittlere Antwortzeit aller 3 Varianten auf der Konsole ausgegeben und in einer csv Datei abgespeichert.

Konfiguriert werden kann der Test durch die *config.yaml* Datei. In dieser *config.yaml* müssen die URLs für das Senden der GET-Request an die 3 Varianten hinterlegt werden. Zudem sind die die Anzahl an Schritten und Schrittgröße festzulegen, welche zusammen die maximale Anzahl an Request definieren. Es ist eine maximal Anzahl von 200 Request empfehlenswert. Der Test muss ggf. diese Anzahl an http-Verbindungen gleichzeitig aufrechterhalten. Dies kann bei vielen Request leicht fehlschlagen, wodurch der Test scheitert. Der entwickelte Lasttest arbeitet nach der *Fire-And-Forget*-Methode und hält keine http-Verbindung bis zur Response aufrecht. Auf diese Weise kann er die Anwendung mit deutlich mehr Request fordern. Er ist allerdings für das Messen der Antwortzeiten ungeeignet und kommt daher nicht zum Testen der Arbeitshypothese zum Einsatz.

4.4 Testergebniss

Um die Arbeitshypothese zu bestätigen, wäre folgender Verlauf der Antwortzeit bei steigender Anzahl Anfragen zu erwarten.

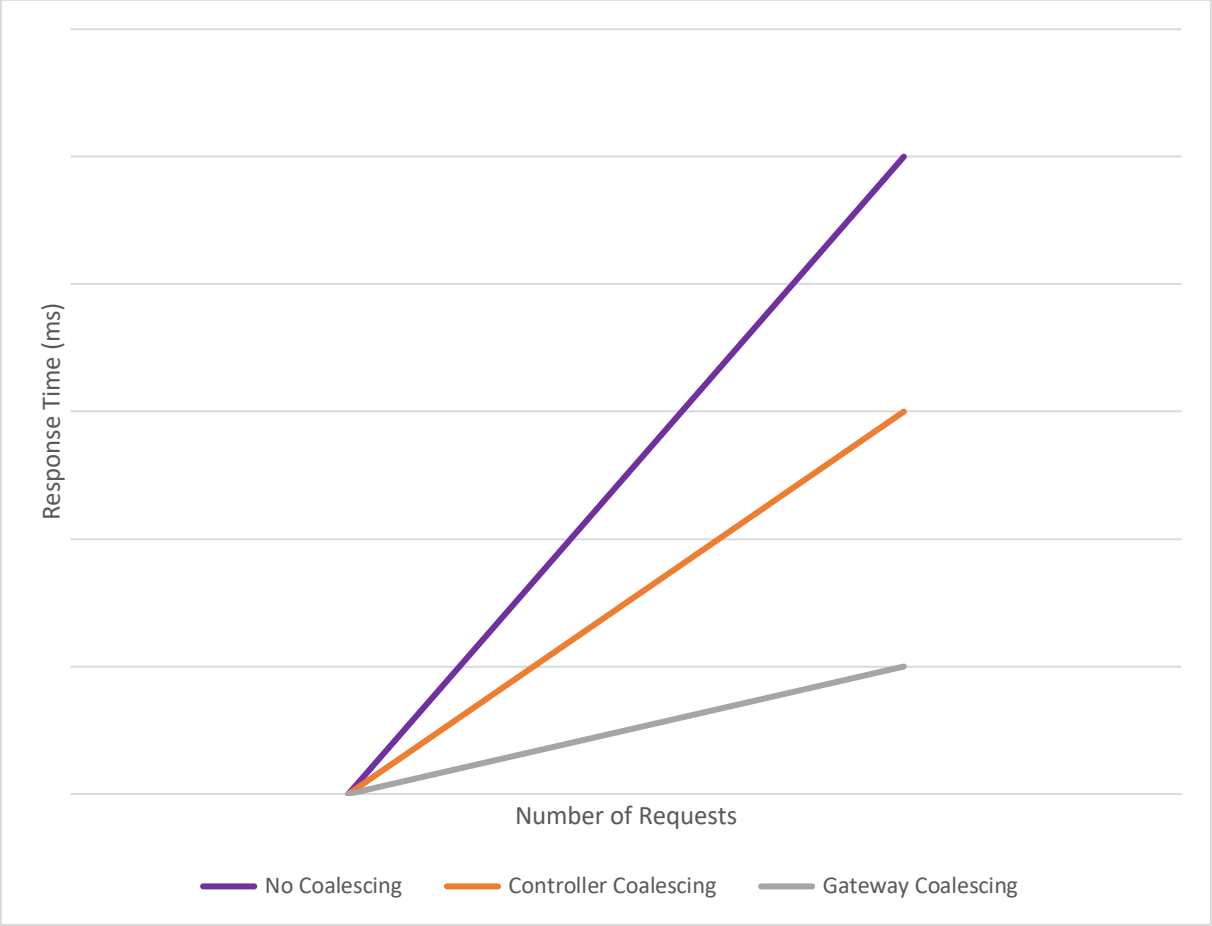

Abbildung 5: Erwarteter Verlauf der Antwortzeiten

Die Variante 1, ohne Coalescing, hat die längsten Antwortzeiten, da bei dieser jede Anfrage bis zur Datenbank durchgereicht werden muss.Beim Coalescing steigt die Antwortzeit weniger stark, da die höhere Anzahl an Anfragen nur bis zu den Controllern durchgereicht werden müssen und zur Datenbank nur einmalig die erste Anfrage durchgereicht wird. Beim Coalescing in dem API-Gateway wiederum muss auch bei steigender Anzahl an Anfragen nur einmalig die erste Anfrage bis zur Datenbank durchgereicht werden. Alle folgenden können direkt vom API-Gateway beantwortet werden und die Antwortzeit steigt am geringsten bei einer steigenden Anzahl von Anfragen.

Durchgeführt wird der Test zur Überprüfung der Arbeitshypothese mit einer Schrittanzahl von 20 und einer Schrittweite von 10, sodass im letzten Schritt 200 Requests gesendet werden. Gehostet wird die Cloud-Anwendung in einer Docker Umgebung und auf keinem Server-Cluster.

Beim mehrmaligen Durchführen der Tests fällt auf, dass keine konstanter Anstieg zwischen der Anzahl an Request und den Antwortzeiten beobachtet werden kann. Im Rahmen dieser Arbeit

werden die Testergebnisse von drei Testläufen dargestellt. Entnommen werden können die Ergebnisse den folgenden Tabellen.

Number of Request	No Coalescing (ms)	Controller Coalescing (ms)	Gateway Coalescing (ms)
10	27	14	14
20	32	28	18
30	35	50	24
40	66	108	21
50	138	77	48
60	74	59	50
70	76	307	60
80	126	187	51
90	256	142	57
100	169	366	86
110	230	408	109
120	159	329	107
130	210	468	132
140	296	265	134
150	277	224	139
160	275	296	196
170	317	336	215
180	383	413	255
190	325	398	237
200	348	435	305

Tabelle 1: Erster Testlauf

Number of Request	No Coalescing (ms)	Controller Coalescing (ms)	Gateway Coalescing (ms)
10	56	14	5
20	19	20	16
30	27	25	15
40	25	31	19
50	34	65	24
60	60	44	31
70	52	84	42
80	85	101	47
90	183	145	99
100	305	141	77
110	210	172	58
120	224	261	57
130	163	136	74
140	166	147	80
150	158	163	88
160	253	287	117
170	302	268	119
180	236	243	123

190	356	252	156
200	246	250	171

Tabelle 2: Zweiter Testlauf

Number of Request	No Coalescing (ms)	Controller Coalescing (ms)	Gateway Coalescing (ms)
10	35	14	13
20	20	25	13
30	27	32	24
40	156	41	40
50	47	115	40
60	76	78	53
70	87	79	62
80	93	131	71
90	132	128	58
100	133	148	94
110	199	159	116
120	226	454	129
130	389	302	119
140	472	211	327
150	316	502	157
160	773	676	410
170	453	458	325
180	783	669	191
190	436	485	351
200	777	910	285

Tabelle 3: Dritter Testlauf

Eine leichte Auswertungsmöglichkeit bietet das Öffnen der erstellten csv Dateien mit Excel und der Darstellung der Daten in folgenden Graphen.

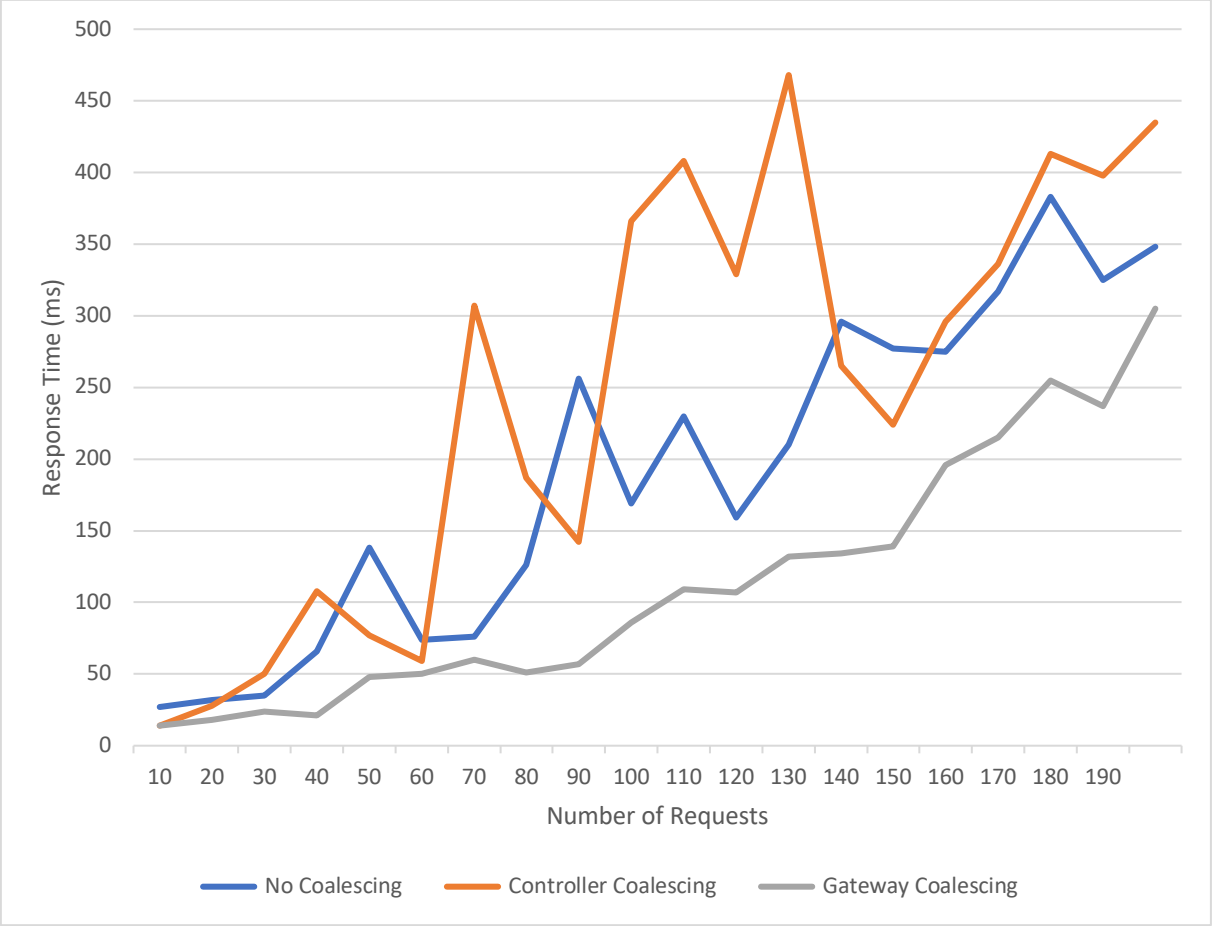

Abbildung 6: Erster Testlauf

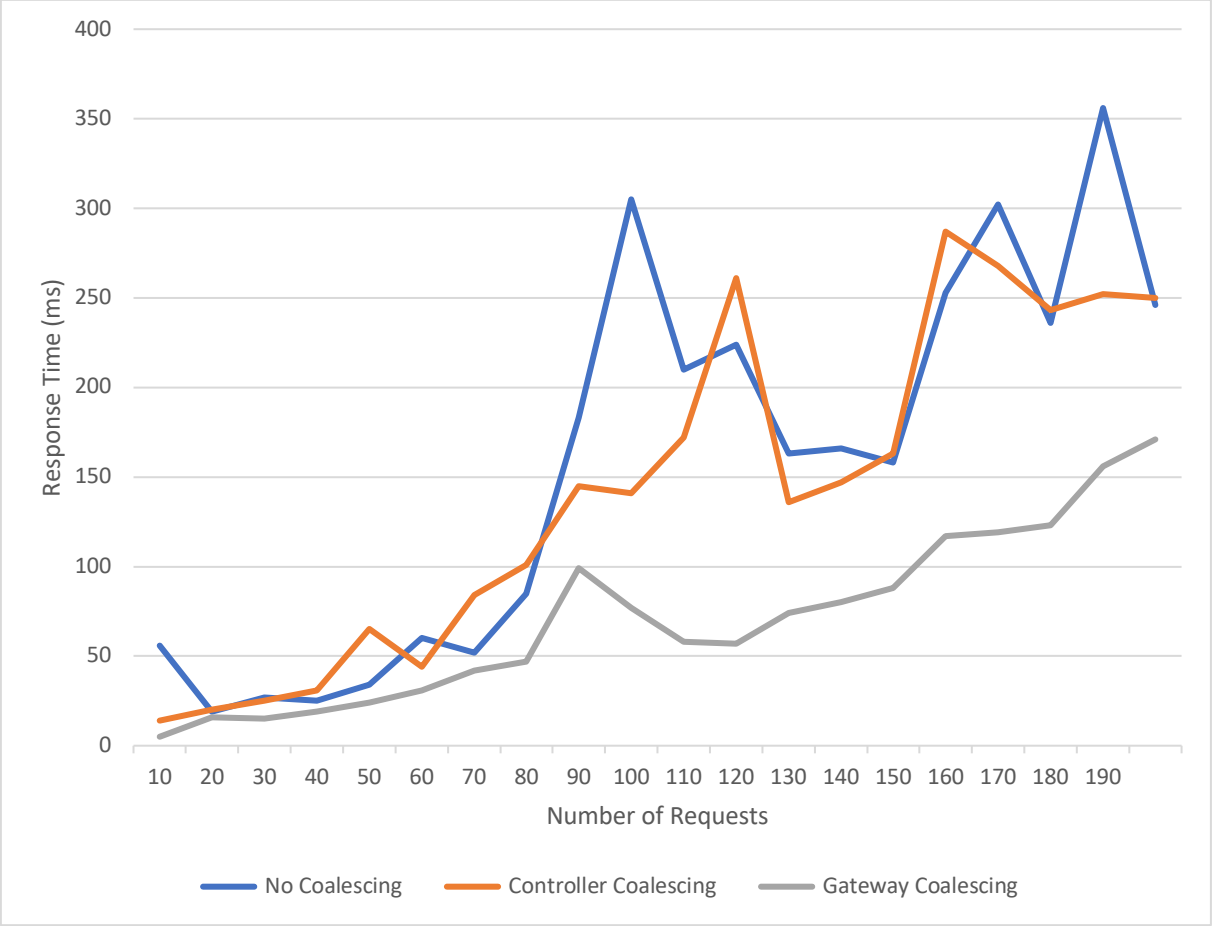

Abbildung 7: Zweiter Testlauf

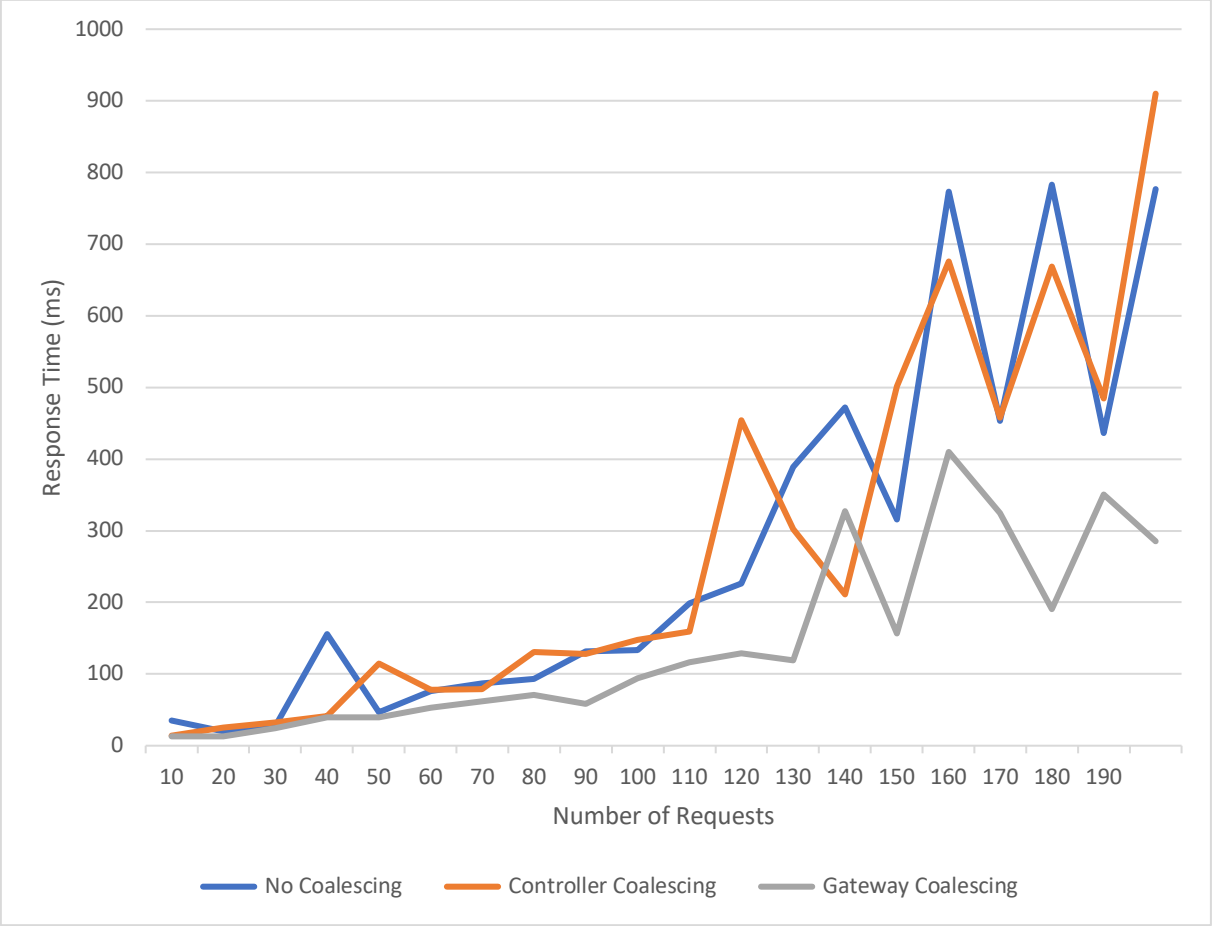

Abbildung 8: Dritter Testlauf

Zunächst fällt auf, dass keine der Varianten einen konstanten Anstieg der Antwortzeiten aufweist. Stattdessen streuen die Messergebnisse in jedem Schritt. Es ist allerdings der Trend zu erkennen, dass Variante 3, Coalescing im Gateway, die geringsten Antwortzeiten besitzt. Wohingegen Variante 1 & 2 einen recht ähnliches Erscheinungsbild aufweisen und nicht wie erwartet, dass Variante 2 geringere Antwortzeiten besitzt.

Die Variation der Testergebnisse lässt vermuten, dass es einen bislang unbeachteten Einfluss auf die Antwortzeit gibt. Dieser Einfluss scheint keinen konstanten Bias darzustellen, sondern je nach Moment der Durchführung zu variieren.

Die Arbeitshypothese kann allerdings insofern bestätigt werden, als dass der Trend zu erkennen ist, dass durch die Implementierung von Coalescing am Einstiegspunkt der Cloud-Anwendung, im API-Gateway, geringere Antwortzeiten erzielt werden können, als ohne Coalescing oder mit Coalescing in den Controllern der einzelnen Services.

5 Fazit und Ausblick

In dieser Ausarbeitung wurde die Cloud-Anwendung *PDF-Designer* vorgestellt und dessen Entwicklungsprozess beschrieben. Zum Abschluss des Projekts verfügt der PDF-Designer über ein rudimentäres Frontend, welches über alle grundlegenden Steuerungsmöglichkeiten der Anwendung verfügt. Ein Nutzer kann sich anmelden, eigene PDF-Templates erstellen sowie bearbeiten und daraus PDF-Dateien generieren lassen. Für eine tatsächliche Nutzung des Tools, wäre allerdings noch eine weitergehende Überarbeitung von dem Frontend nötig.

Das Projekt zeigt jedoch einen funktionalen Prototypen der Anwendung samt Hosting auf einem Kubernetes Cluster oder in einer Docker Umgebung. Zusätzlich verfügt die entwickelte Cloud-Anwendung auch über ein Monitoring des Kubernetes Clusters mittels eines Dashboards, sowie eine Skalierbarkeit der einzelnen Service.

Beim Überprüfen der Arbeitshypothese waren die erzielten Ergebnisse zunächst sehr unbefriedigend, durch die starke Streuung der Messergebnisse. Es gelang jedoch auch nach mehreren Versuchen nicht deutlichere Ergebnisse produzieren, welche eher dem erwarteten Verlauf folgen. Es wurden daher die aufgezeigten Ergebnisse genutzt, welche ebenfalls einen Trend abzeichnen und durch die die Arbeitshypothese unter Vorbehalt bestätigt werden kann.

Abschließend kann erwähnt werden, dass im Laufe dieses Projekts zahlreiches Wissen über das Themengebiet des Cloud-Engineerings erlangt werden konnte. Ebenfalls kann abschließend ein funktionaler Prototyp des PDF-Designers präsentiert werden.

Literaturverzeichnis

[1] Y. Abgaz *u. a.*, „Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review“, *IEEE Trans. Softw. Eng.*, Bd. 49, Nr. 8, S. 4213–4242, Aug. 2023, doi: 10.1109/TSE.2023.3287297.

[2] E. M. Maximilien und L. Williams, „Assessing test-driven development at IBM“, in *25th International Conference on Software Engineering, 2003. Proceedings.*, Portland, OR, USA: IEEE, 2003, S. 564–569. doi: 10.1109/ICSE.2003.1201238.

[3] „What is a container?“, Docker Documentation. Zugegriffen: 27. Dezember 2023. [Online]. Verfügbar unter: https://docs.docker.com/guides/walkthroughs/what-is-a-container/

[4] C. Singh, N. S. Gaba, M. Kaur, und B. Kaur, „Comparison of Different CI/CD Tools Integrated with Cloud Platform“, in *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, Noida, India: IEEE, Jan. 2019, S. 7–12. doi: 10.1109/CONFLUENCE.2019.8776985.

[5] „Was ist Kubernetes?“, Kubernetes. Zugegriffen: 27. Dezember 2023. [Online]. Verfügbar unter: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/

[6] „Nginx: the High-Performance Web Server and Reverse Proxy“. Zugegriffen: 3. Januar 2024. [Online]. Verfügbar unter: https://dl.acm.org/doi/full-Html/10.5555/1412202.1412204

[7] M. Rahman, S. Iqbal, und J. Gao, „Load Balancer as a Service in Cloud Computing“, in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, Oxford, United Kingdom: IEEE, Apr. 2014, S. 204–211. doi: 10.1109/SOSE.2014.31.

[8] M. R. Baihaqi, R. M. Negara, und R. Tulloh, „Analysis of Load Balancing Performance using Round Robin and IP Hash Algorithm on P4“, in *2022 5th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, Yogyakarta, Indonesia: IEEE, Dez. 2022, S. 93–98. doi: 10.1109/ISRITI56927.2022.10052975.

[9] „thundering herd problem“. Zugegriffen: 27. Dezember 2023. [Online]. Verfügbar unter: http://www.catb.org/jargon/html/T/thundering-herd-problem.html

[10] „How Discord Stores Trillions of Messages“. Zugegriffen: 29. Dezember 2023. [Online]. Verfügbar unter: https://discord.com/blog/how-discord-stores-trillions-of-messages

[11] „About gRPC“, gRPC. Zugegriffen: 29. Dezember 2023. [Online]. Verfügbar unter: https://grpc.io/about/

[12] „gRPC im Vergleich zu REST – Unterschiede zwischen den Anwendungsdesigns – AWS“, Amazon Web Services, Inc. Zugegriffen: 29. Dezember 2023. [Online]. Verfügbar unter: https://aws.amazon.com/de/compare/the-difference-between-grpc-and-rest/

[13] „Was sind Microservices? Architektur im Überblick“. Zugegriffen: 27. Dezember 2023. [Online]. Verfügbar unter: https://www.redhat.com/de/topics/microservices/what-are-microservices