Universität Trier, Wintersemester 24/25
Software Architectures for Enterprises
Michael Goichmann, 1417617

SA4E Übung 2

Den Quellcode und Videos für diese Abgabe finden Sie unter:

https://github.com/Michael-Goichmann/sa4e_ws2425/tree/main/U2

Aufgabe 1 Modellierung

Denken Sie über eine mögliche Implementierung von XmasWishes nach. Berücksichtigen Sie in Ihrem Architekturentwurf das tatsächlich zu erwartende, geographisch verteilte Datenaufkommen über die Zeit.

In dieser Aufgabe wird zunächst eine grobe Skizze für eine mögliche Implementierung gefordert. Im ersten Schritt sollte man im Rahmen von Softwareprojekten die gegebenen Anforderungen feststellen und darauf basierend erste Schlüsse für eventuelle konkretere Architekturen oder Technologien schließen.

1. „Wie sollte das zukünftige Softwaresystem aussehen, um den unerwartet und überraschend großen Ansturm, [..] im letzten Quartal des Jahres [..] abzufedern?“
    a. Hieraus lässt sich erkennen, dass saisonal bedingt große Lasten im System auftreten. Also sollte vermutlich irgendeine Form von lastenbedingter Form von Skalierung eingebaut werden.
2. „Es geht [..] um das einfache Wünscheverwaltungssystem, das die Weihnachtswünsche [..] aufnimmt und für den Nordpol speichert.“
    a. Auf den gesamten Globus verteilte Systeme, welche individuell Wünsche sammelt. Hier wären vermutlich individuelle Systeminstanzen pro z.B. Kontinent mit eigenen Datenbanken geeignet
3. „Dort, am Nordpol, wird [..] auf die Daten [..] zugegriffen“
    a. Zentrale Instanz, in der aggregierte Daten aus den externen Quellen einfließen und ausgelesen werden.
4. „Der Geschenkwunsch wird als einfache Zeichenkette gespeichert, [..]. Auch für die Speicherung des Personennamens reicht [..] eine einfache Zeichenkette. Das System speichert außerdem, ob der Wunsch (1) [..] (4) (Statusangabe).“
    a. Die Daten sind in einem simplen semi-strukturierten Format gegeben. Also könnte man die Daten als JSON Format abbilden.

Universität Trier, Wintersemester 24/25
Software Architectures for Enterprises
Michael Goichmann, 1417617

In der folgenden Grafik wird zusammenfassend abgebildet, wie die Architektur aussehen könnte. Auf den einzelnen Kontinenten kommen jeweils Anfragen über die Clients an die jeweilige Regionale Microservice-Instanz, welche beliebig mit weiteren Microservice Containern erweitert werden kann, sollte der Bedarf bestehen. Jede Instanz hat dabei eine eigene Datenbank, welche entweder batchweise oder per Streaming die Daten an die zentrale Aggregationsinstanz am Nordpol schickt. Am Nordpol werden alle einkommenden Daten gesammelt und ausgelesen für die weitere Bearbeitung.

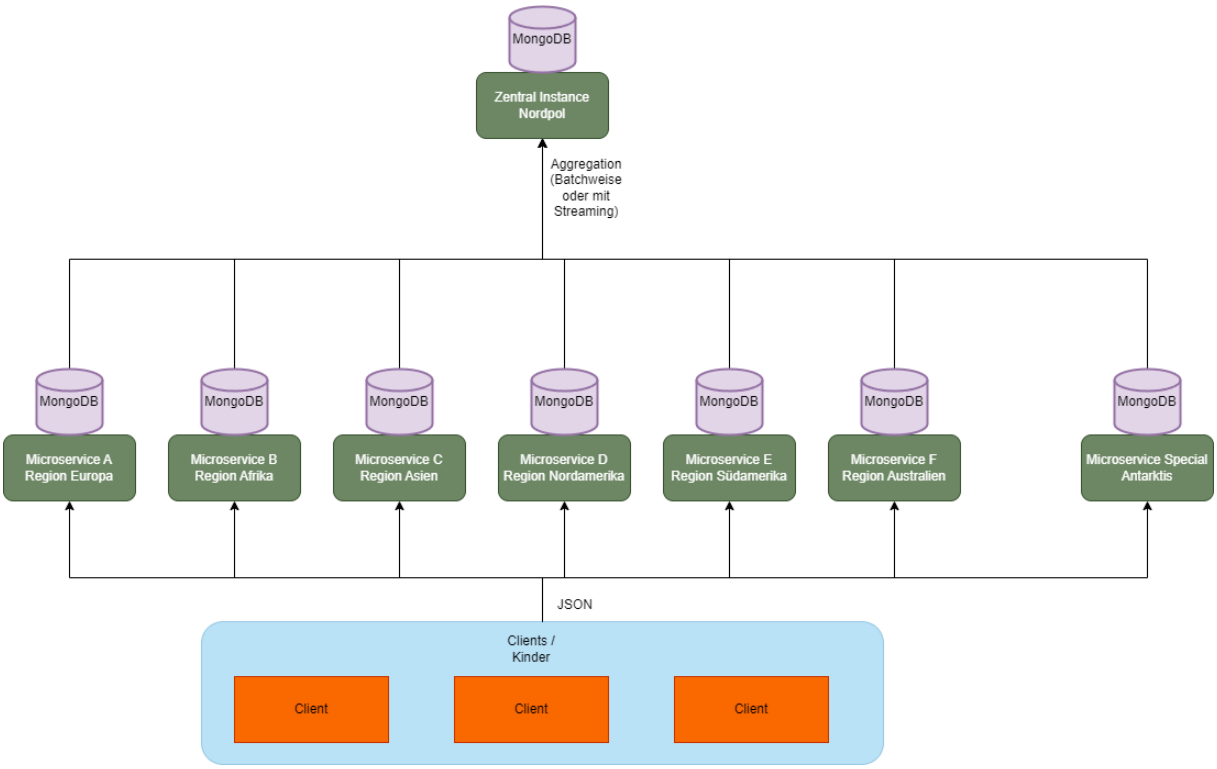


Abbildung 1 Skizze Architektur Lösung

Aufgabe 2 Konkretisierung Technologien

Da Skalierung ein wichtiges Thema für diese Übung ist und in Aufgabe 1 bereits eine Microservice Architektur angelehnt wurde, würde **Docker** hervorragen passen, um die individuellen Microservices einfach verwaltbar und beliebig erweiterbar zu gestalten. Für das Backend der Services könnte man ein beliebiges Stack verwenden. Im Rahmen dieser Aufgabe habe ich Python als Programmiersprache gewählt. Weit verbreitet sind für Python Flask oder FAST-API. Hier entscheide ich mich für **Flask**, da dieses Framework für prototypische Arbeit optimal ist. Mit jeder Docker Instanz wird eine **MongoDB** Datenbank verbunden, da die Daten gut als JSON abbildbar sein. Dabei haben die einzelnen Regionen lokale Datenbanken und am Nordpol ist eine zentrale MongoDB, welche alle Wünsche zusammenfasst. Um die verteilte Architektur abzubilden wird pro Region / Kontinent ein eigener Container in einer Virtuellen Maschine gestartet, welcher jeweils das Flask Backend und eine MongoDB Instanz besitzt. Diese Konfiguration wurde mit Vagrant und Virtualbox umgesetzt. Dies reicht für dieses kleine Projekt. Für größere Projekte wären eher Tools wie Kubernetes oder Kafka notwendig.

Zu demonstrationszwecken wird das Framework Locust verwendet, welche eine kleine und simpel zu bedienende Last-Test Plattform beinhaltet.

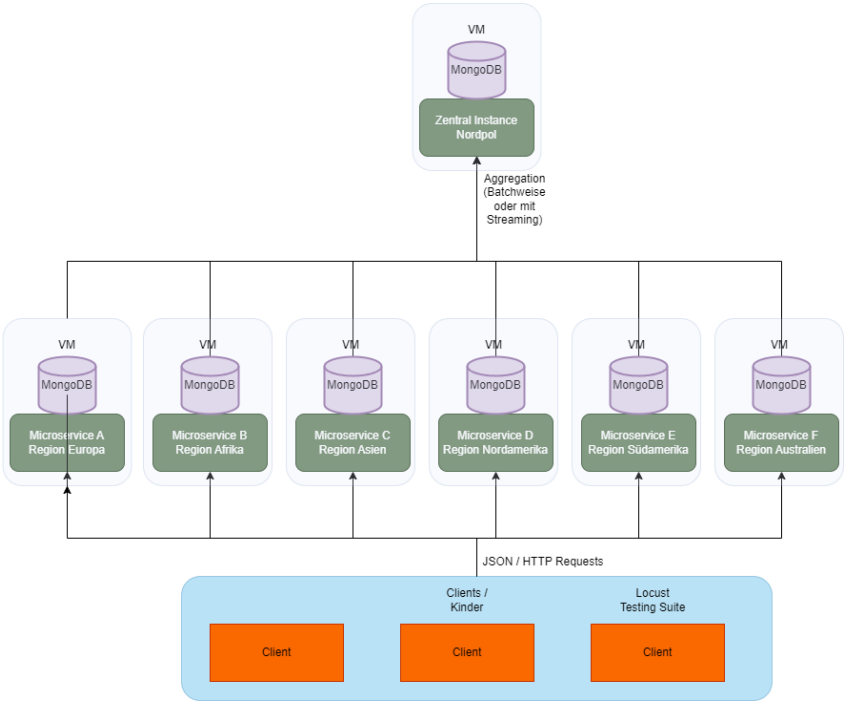


Abbildung 2 Konkretere Technologien

Aufgabe 3

In dieser Aufgabe wurde die Implementierung getestet. Dafür wurde für jede Region eine eigene VM erstellt und ein Docker Container mit einer ansprechbaren Flask API und einer eigenen Datenbank erstellt. Aus Hardware- und Praktikabilitätsgründen beschränke ich die folgenden Tests auf eine einzelne VM und prüfe die Skalierungsoptionen.

Test 1: 1 Container

- Maximal 5000 Clients (3050 erreicht)
- Wachstum von 50 Clients pro Sekunde
- jeder Client stellt eine Anfrage zwischen 1 und 2 Sekunden
- Lesen und Schreiben
- Laufzeit 1 Minute

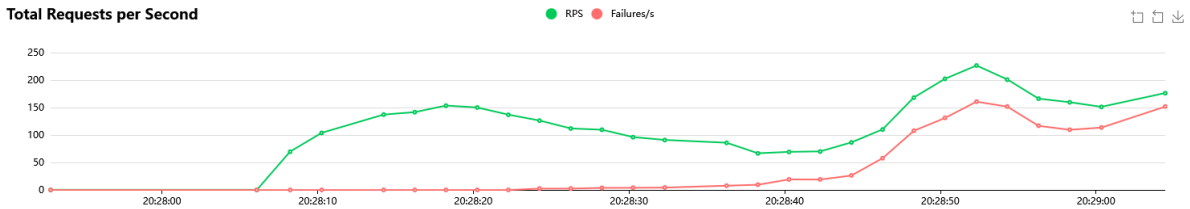


Abbildung 3 Requests und Failures pro Sekunde

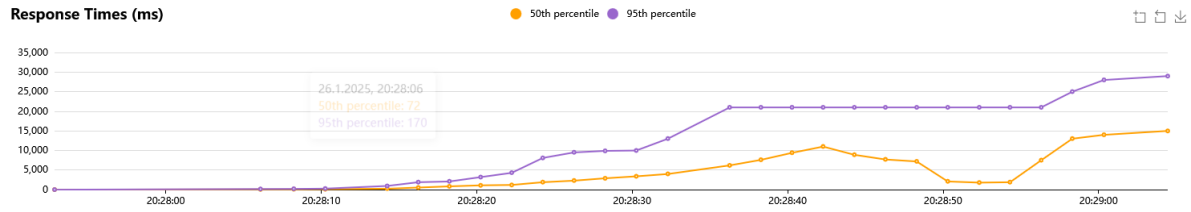
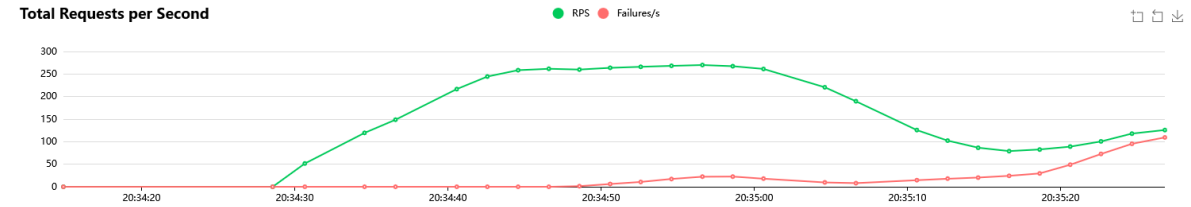


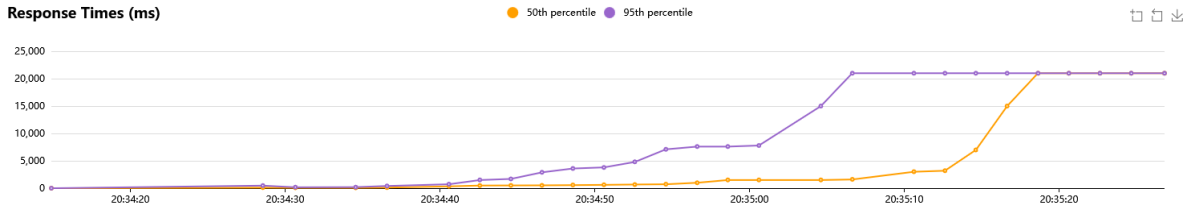
Abbildung 4 Response Times in ms

In diesem Test wurde gleichzeitiges Lesen und Schreiben auf den Service geprüft. Dabei erkennt man, dass zu in der ersten Hälfte der Zeit die Requests pro Sekunde erst bis knapp 150 steigen und dann langsam abfallen. In der zweiten Hälfte steigen die gesamten Requests stark an und gleichzeitig steigt die Fehlerquote enorm, so dass gegen Ende durchschnittlich etwa **68 Requests die Sekunde erfolgreich** durchgekommen sind und etwa **28 Requests die Sekunde fehlgeschlagen** sind.

Test 2: 1 Container

- Maximal 5000 Clients (3050 erreicht)
- Wachstum von 50 Clients pro Sekunde
- jeder Client stellt eine Anfrage zwischen 1 und 2 Sekunden
- Schreiben
- Laufzeit 1 Minute

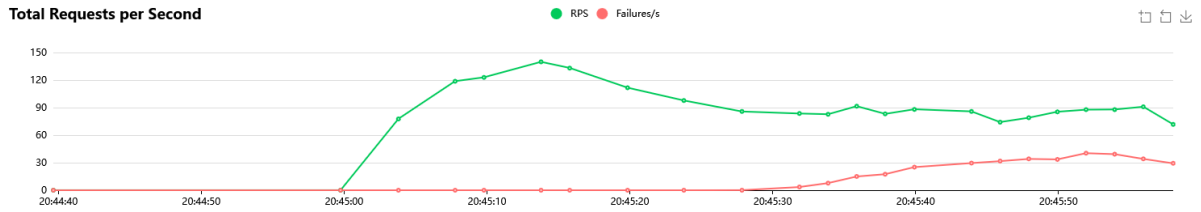


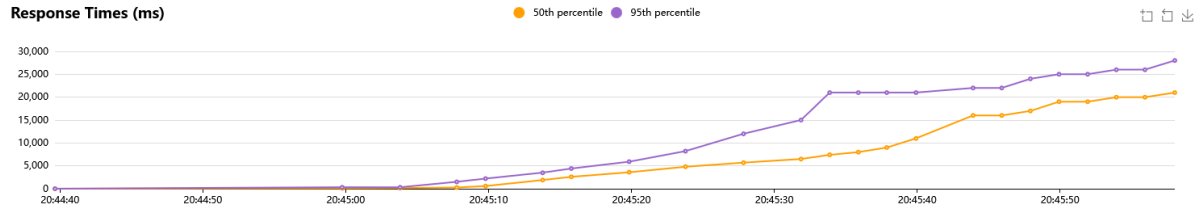


In diesem Test wurde nur das Schreiben auf den Service geprüft. Hier hat sich die Durchschnittliche Zahl an erfolgreichen Schreibzugriffen mehr als verdoppelt auf etwa 177 Posts pro Sekunde bei knapp 32 Post Failures die Sekunde.

Universität Trier, Wintersemester 24/25
Software Architectures for Enterprises
Michael Goichmann, 1417617
Test 3: 3 Container

- Maximal 5000 Clients (3050 erreicht)
- Wachstum von 50 Clients pro Sekunde
- jeder Client stellt eine Anfrage zwischen 1 und 2 Sekunden
- Lesen und Schreiben
- Laufzeit 1 Minute



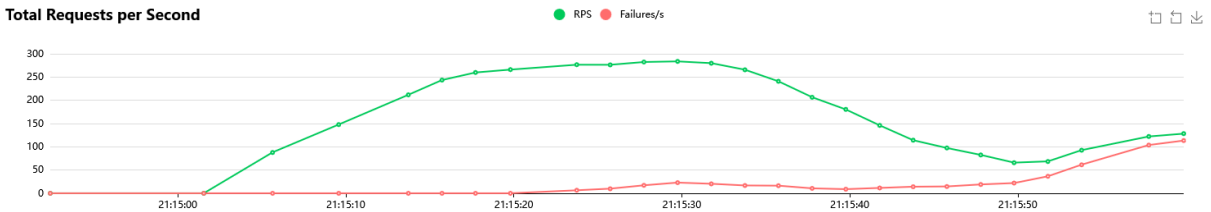


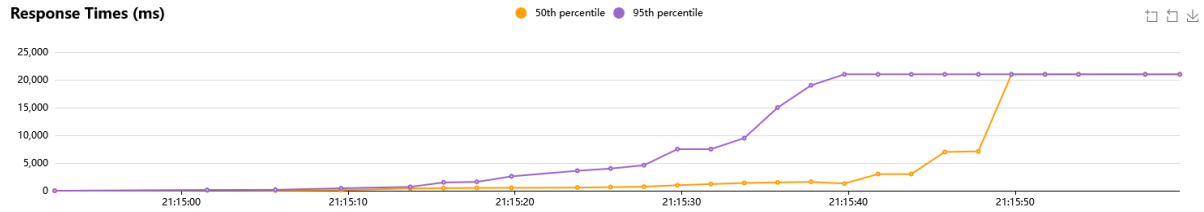
Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/wishes	2757	465(16.87%)	8750	4	38537	5400	45.47	7.67
POST	/wishes	2870	496(17.28%)	7896	14	34407	5200	47.34	8.18
	Aggregated	5627	961(17.08%)	8314	4	38537	5300	92.81	15.85

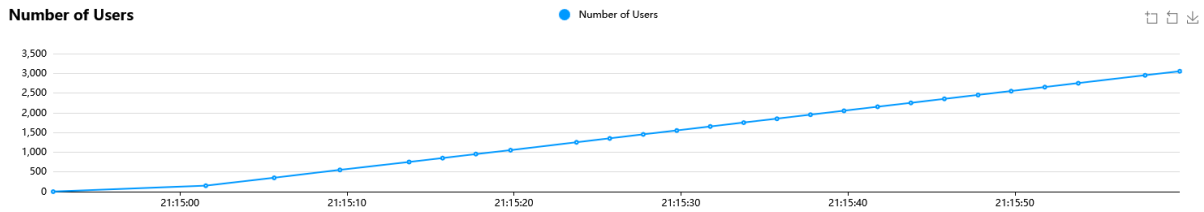
In diesem Test wurde die Anzahl der Container von 1 auf 3 erhöht, was den automatischen Loadbalancer von Docker Swarm auslöst. Den Effekt der Lastenverteilung kann man vor allem daran erkennen, dass am Ende nur noch sowohl beim Lesen als auch beim Schreiben nur knapp **8 Fehler die Sekunde** ausgelöst werden, bei jeweils **45 erfolgreichen Requests die Sekunde**. Dennoch scheint die allgemeine Menge an Zugriffen nicht sonderlich besser zu sein in diesem Szenario.

Test 4: 3 Container

- Maximal 5000 Clients (3050 erreicht)
- Wachstum von 50 Clients pro Sekunde
- jeder Client stellt eine Anfrage zwischen 1 und 2 Sekunden
- Schreiben
- Laufzeit 1 Minute







In diesem Beispiel wurde nur das Schreiben auf 3 verteilte Backendes geprüft. Hier wurde ein Optimum an Schreibzugriffen bei etwa 270 Zugriffen die Sekunde mit weniger als 20 Fehler die Sekunde erreicht. Gleichzeitig scheint wie bei allen anderen Versuchen die Leistung des Systems ab knapp 1600 gleichzeitigen Clients stark abzunehmen. Am Ende wurde Werte von Durchschnittlich 178 Schreibzugriffen die Sekunde und 33 Fehlern beim Schreiben erreicht.

Zusammenfassend für diese kleine Experimentreihe würde ich sagen, dass das Verteilen der Lasten hilfreich sein kann, aber einen starken Flaschenhals erfährt bei höheren verbundener Clientmenge. An dieser Stelle muss erwähnt sein, dass zwar jeder Region eine eigene VM Instanz hat und das Backend problemlos erweiterbar war, aber bei der Umsetzung aufgefallen ist, dass die Datenbanken sich nicht ohne weiteres vermehren und hier scheinbar mehr Arbeit nötig wäre, um die Logik so anzupassen, dass beim Skalieren der Container, über den Befehl: *docker service scale xmas_xmaswishes_app_eu=3*, auch jeweils die Datenbank erweitert wird. Aktuell wird nur das Backend, also die Business Logik Ebene erweitert, aber alle greifen auf dieselbe Datenbank Instanz zu. Weitere Arbeit an der Logik zum Skalieren wäre notwendig.

Aufgabe 4

Für diese Aufgabe wurde eine standalone Java Lösung für das separate Einlesen von Bilddateien in Apache Camel erstellt. Dabei wird zusätzlich zu den Kernfunktionen noch die Library Tesseract verwendet, um OCR Funktionalitäten umzusetzen, damit Bilder und Texte richtig erkannt werden.

Damit neu reingekommene Daten nicht mehrmals eingelesen werden, wird ein idempotent Repository – also eine extra Bibliothek – geführt, die sich merkt, welche Dateien bereits gelesen wurden und welche nicht.

Sobald die Bilder mit Tesseract gelesen und hoffentlich erkannt wurden, werden die einzelnen Texte mit einfachen Stringverarbeitungsfunktionen aufgeteilt und über einen Builder zusammengestellt, sodass diese im Nachgang einfach über ein Map Objekt in einem JSON nahen Format verarbeitet werden und zu guter letzt an den lokalhost Port mit einer POST Methode an den Microservice für den Nordpol gesendet werden.