Aufgabe 1 & 2: Modell und Konkretisierung

Aus dem beschriebenen Szenario ergeben sich folgende Anforderungen an das System bzw. die Softwarearchitektur.
Da der Fokus auf der Softwarearchitektur liegen soll ist die folgende Liste unvollständig und soll eher die berücksichtigten Schwerpunkte verdeutlichen.

Funktionale Anforderungen

- **Wunschverwaltung**
  - Aufnahme von Wünschen und Namen.
  - Speicherung der Wünsche als Zeichenkette zusammen mit dem Namen der Person und dem Status des Wunschs.
  - Anzeigen der gespeicherten Daten (Wunsch, Name und Status).
  - Statusverwaltung der Wünsche (Formuliert/In Bearbeitung/In Auslieferung/Unter dem Weihnachtsbaum).

Nicht-funktionale Anforderungen

- **Skalierbarkeit/Elastizität**
  - Kaum bzw. kein Traffic über das Jahr hinweg.
  - Horizontale Skalierbarkeit, um den hohen Andrang im letzten Quartal bzw. Dezember zu bewältigen:
    - Rund **2,4 Milliarden Menschen weltweit feiern Weihnachten**[1].
    - Rund 1/3 der Weltbevölkerung sind Kinder (unter 18 Jahren). Es feiern demnach rund **800 Millionen Kinder und 1,6 Milliarden Erwachsene Weihnachten**.

    Gehen wir nun vereinfacht davon aus, dass jedes Kind und 12,5 % der Erwachsenen, die Weihnachten feiern, einen Wunsch übermitteln. Dann ergeben sich grob **1 Milliarde Wünsche weltweit**.
    Durch die Anzahl der Wünsche ergeben sich eine Menge API-Anfragen:
    - Zum Updaten des Status (*formuliert, in Bearbeitung, in Auslieferung, unter Weihnachtsbaum*) sind pro Wunsch mindestens 4 API-Anfragen notwendig. Also rund **4 Milliarden API-Anfragen**.

    Angenommen alle Wünsche werden im Dezember vor Weihnachten, also in einem Zeitraum von 3 Wochen gesendet, dann ergibt sich eine durchschnittliche Last wie folgt:
    - 4 Milliarden / (3 Wochen · 7 Tage · 24 Stunden) ≈ **8 Millionen API-Aufrufe/Tag**

[1] Quelle: https://dossiers.kleinezeitung.at/welt-in-zahlen-christen/

    * 4 Milliarden / (3 Wochen · 7 Tage · 24 Stunden · 3600 Sekunden) ≈ **2200 API-Aufrufe/Sekunde**

  Berücksichtigt man zusätzlich, dass Wünsche vorrangig am Abend und am Wochenende abgegeben werden, wird klar, dass die Last enorme Spitzen erreichen kann.

- **Verfügbarkeit/Zuverlässigkeit und Leistung**
  - Hohe Verfügbarkeit durch redundante Systeme um Ausfälle zu minimieren.
  - Weltweite Verfügbarkeit - global geringe Latenzen.
- **Sicherheit**
  - DSGVO-konforme Datenverarbeitung (Verschlüsselung der Daten und lokale Speicherung am Erhebungsort).
- **Benutzerfreundlichkeit**
  - Webservice für einfachen Zugriff.

Um die hohe Elastizität, große Maximallast und globalen Zugriff abbilden zu können, sollte das System horizontal skalierbar sein. Hierzu sollte eine Cloud- und Containerbasierte Microservice-Architektur eingesetzt werden. Zur Datenspeicherung sollten aus demselben Grund NoSQL-Datenbanken verwendet werden. Im Folgenden wird das Modell für die Softwarearchitektur und mögliche Technologien vorgestellt.

Modell

Abbildung 2 zeigt eine Skizze der Softwarearchitektur.
Die dargestellte Architektur besteht im Kern aus zwei Bestandteilen: Dem Verwaltungs- bzw. Administrations-System für den Weihnachtsmann und dessen Mitarbeiter am Nordpol und dem Wünsche-Service für Kinder und junggebliebene Erwachsene, um Wünsche an den Weihnachtsmann zu senden. Der Wünsche-Service ist nach geographischen Regionen (bspw. Kontinenten) in eigenständige Instanzen aufgeteilt. Die global verteilte Architektur mit regionalen Instanzen bietet mehrere Vorteile: Aufteilung der Last, Erhöhung der Ausfallsicherheit, DSGVO konforme regionale Speicherung der Daten und globaler Zugriff mit geringen Latenzen.
Da der Weihnachtsmann ein großes Geheimnis um die Anzahl seiner Mitarbeitenden - den Elfen und Rentieren - macht, ist nicht ganz klar wie das Administrations-System aufgebaut und wie es skaliert werden muss. Deshalb wird auch für dieses System eine horizontal skalierbare Architektur vorgeschlagen.

1. **Regionale Instanzen des Wünsche-Service**
   - **API-Gateway** (bspw. nginx): Zentraler Einstiegspunkt für alle Nutzeranfragen. Nimmt Nutzeranfragen entgegen und leitet sie weiter. Da sämtliche Anfragen nur an den Load-Balancer weitergeleitet werden, ist das API-Gateway

theoretisch nicht notwendig. Praktisch dient es jedoch zum Schutz, bspw. vor DDoS-Angriffen, und stellt die DSGVO-Konformität durch Protokollierung sicher.

- **Load-Balancer** (bspw. nginx): Verteilt eingehende Anfragen gleichmäßig auf die verfügbaren Instanzen der Wish-Services (per Round-Robin oder least-connection).
- **Wish-Service** (bspw. Node.js): Mehrere Instanzen leichtgewichtiger Webserver, die als Frontend eine benutzerfreundliche Weboberfläche bereitstellen. Sie empfangen HTTP POST Anfragen mit Namen und Wünschen der Nutzer, validieren die Daten und speichern sie direkt in einer NoSQL-Datenbank bzw. einem Datenbank-Cluster.
- **NoSQL-DB-Cluster** (bspw. MongoDB-Sharding, MongoDB Atlas): Jede Region verfügt über ein separates MongoDB-Cluster, eine Synchronisation erfolgt nicht. Die regionale Speicherung der Daten ist für die DSGVO-Konformität notwendig oder zumindest empfohlen. MongoDB bietet beispielsweise Sharding an, womit ein Cluster über mehrere Maschinen hinweg erstellt werden kann. Dabei kommunizieren die Applikationen mit einem Router, der wiederum mithilfe eines Config-Servers die Anfragen an den entsprechenden Shard weiterleiten. Die Shards sind einzelne Knoten des Clusters, wobei jede aus einem Replica Set besteht, sodass Daten redundant gespeichert werden (siehe Abbildung 1). Da sowohl die Anzahl der Router, der Config-Server und der Shards variiert werden kann, bietet sich die Architektur sehr gut zur horizontalen Skalierung an.

2. **Administrative Systeme am Nordpol**
   Um die Sicherheit zu erhöhen könnte die Verfügbarkeit der gesamten administrativen Systeme auf das interne Netzwerk am Nordpol (das Intranet des Weihnachtsmanns) beschränkt werden.
   - **API-Gateway** (bspw. nginx): Nimmt Anfragen entgegen und leitet sie weiter. Dient zusätzlich zur DSGVO-konformen Protokollierung. Kann auch einen Authentifikations-Service zur Authentifizierung mit JSON Web Tokens (JWTs) integriert haben.
   - **Authentification-Service** (JWT): Optional ein eigenständiger Service zur Verwaltung von JWTs. JWTs bieten sich zur Authentifizierung in Architekturen mit Microservices an, da sie self-contained sind, d.h. sie enthalten alle notwendigen Informationen zur Authentifizierung. Außerdem kann jeder Microservice den JWT validieren, ohne auf eine zentrale Datenbank oder einen Auth-Server angewiesen zu sein. Die JWTs können im Header von HTTP Anfragen mitgesendet werden.
   - **Load-Balancer** (bspw. nginx): Verteilt eingehende Anfragen gleichmäßig auf die Admin-Services.

- **Admin-Services** (bspw. Node.js): Analog zu den Wish-Services. Ebenfalls leichtgewichtige Webserver, die eine Weboberfläche zur Anzeige der Daten und Aktualisierung des Status von Wünschen zur Verfügung stellen. Arbeiten mit HTTP GET Anfragen, um Daten aus der Datenbank abzurufen und HTTP PUT Anfragen, um den Status von Wünschen zu aktualisieren. Analog zum zuvor beschriebenen Sharding können die Admin-Services über Router die Daten anfragen. Die Router verweisen allerdings nun nicht direkt auf Shards, sondern wiederum auf die Router der einzelnen regionalen DB-Cluster. Aus der Perspektive der Admin-Services werden Daten also aus einem Cluster von Datenbank-Clustern abgerufen.

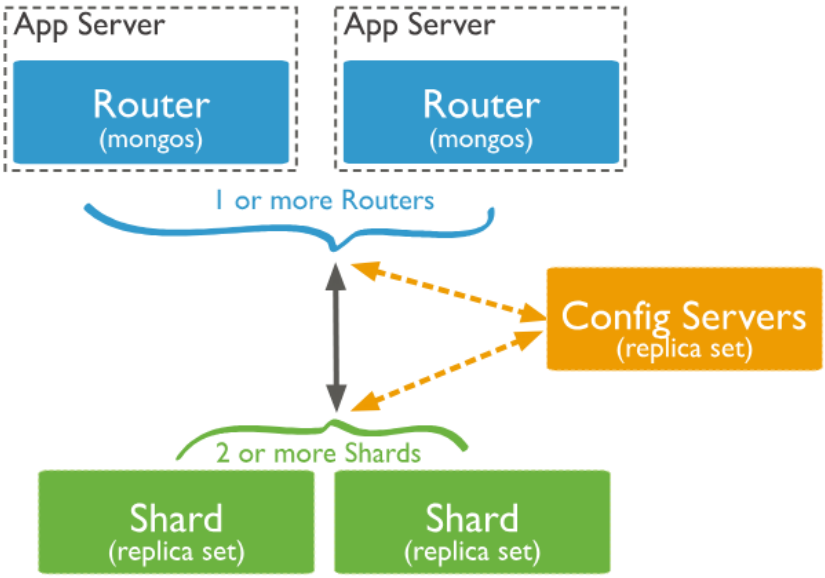


Abbildung 1: Interaktion der Komponenten eines Sharded-Clusters.

Skalierbarkeit

Ein wesentlicher Punkt der Übung ist die Skalierbarkeit der Softwarearchitektur. Das zuvor vorgestellte Modell weißt eine hohe horizontale Skalierbarkeit auf, sodass nahezu beliebige Lasten bewältigt werden können. Zum einen kann hierzu jede regionale Instanz horizontal skaliert werden, indem die Anzahl der verfügbaren Microservices sowie die Anzahl der Knoten des DB-Clusters erhöht werden. Zum anderen können regionale Instanzen weiter unterteilt werden, beispielsweise könnte die Instanz für Europa in EU-West und EU-East aufgeteilt werden.

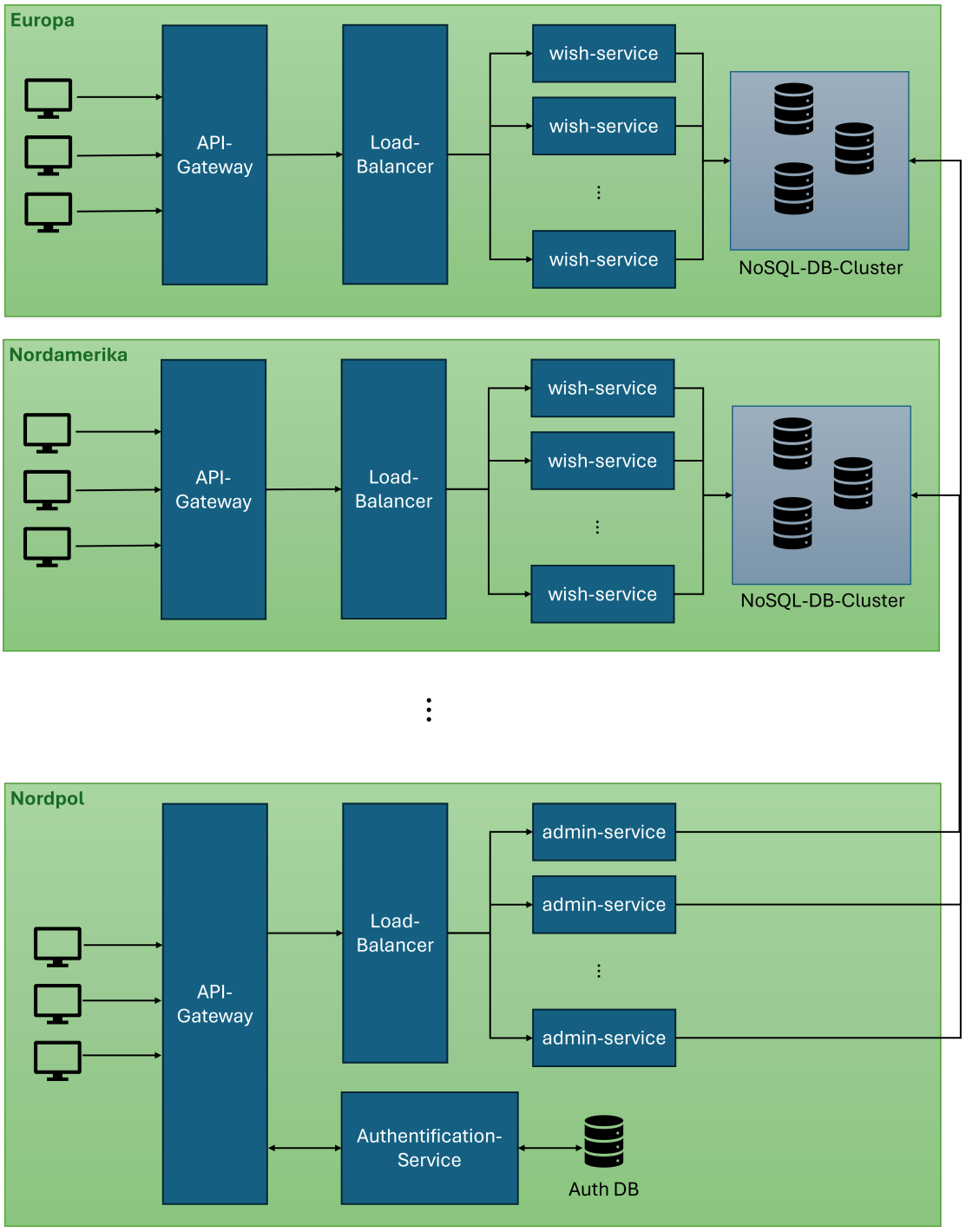


Abbildung 2: Modell der Softwarearchitektur.

Aufgabe 3 & 4: Prototyp inklusive Apache Camel Integration

Der Quellcode des Prototypen inklusive der `README`-Datei mit Erklärung zum Setup steht in folgendem Repository zur Verfügung: `https://github.com/s4paweil/SA4E_Ueb02.git`.
Im Folgenden wird kurz der Prototyp vorgestellt und anschließend der Lasttest und dessen Ergebnisse präsentiert.

Prototyp

Abbildung 3 zeigt die Architektur des implementierten Prototypen. Ergänzend zeigt Abbildung 4 Screenshots der Frontends des Wish-, sowie des Admin-Services. Im Admin-Service kann der Status von Wünschen über eine Dropdown Menü geändert werden. Damit der Prototyp einfach ausgerollt werden kann, wurde er als Docker-Container entwickelt und mit Docker Compose orchestriert.
Da die Architektur zuvor bereits im Detail beschrieben wurde, wird der Prototyp nicht nochmal genauer beschrieben. Im Vergleich zu der vorgestellten Softwarearchitektur wurde der Prototyp wie folgt vereinfacht:

- **Eine regionale Instanz:** Der Prototyp simuliert eine einzelne regionale Instanz des Wünsche-Services, sowie den Admin-Service des Nordpols. Da die regionalen Instanzen der Wünsche-Services unabhängig voneinander laufen, bietet die Integration weiterer regionaler Instanzen keinen Mehrwert.
- **Keine Authentifizierung:** Auf die Integration einer Authentifizierung für den Admin-Service wurde verzichtet, da es den Prototypen nur unnötig verkompliziert hätte.
- **Verzicht auf API-Gateways:** Da in dem Prototypen keine Protokollierung notwendig ist und auf die Authentifizierung verzichtet wurde, haben API-Gateways keinen Nutzen für den Prototypen und wurden deswegen vernachlässigt.
- **Einzelner Admin-Service:** Für die Wish-Services enthält der Prototyp mehrere Instanzen (standardmäßig 2) und einen Load-Balancer. Da die Struktur für mehrere Instanzen des Admin-Services identisch wäre wurde der Prototyp auf eine Admin-Service Instanz beschränkt und der Load-Balancer für die Admin-Services als Konsequenz weggelassen.
- **Kein DB-Cluster:** Da für den kleinen Prototypen eine einfache MongoDB Instanz ausreicht wurde auf die Integration eines komplexen DB-Clusters verzichtet.

Die Apache Camel Integration läuft in einem eigenen Service und sorgt dafür, dass eingescannte Wünsche automatisch in das System integriert werden. Der Weihnachtsmann verfügt über wahnsinnig fortgeschrittene Scanner mit KI Funktionen, die Wunschzettel voll automatisch auslesen können und die Namen und Wünsche in einer Textdatei abspeichern. Der Apache Camel Service überwacht dauerhaft ein definiertes Verzeichnis

und schreibt die Daten neuer Textdokumente automatisch als elektronische Wünsche in die Datenbank. Dies ermöglicht eine nahtlose Integration von analogen Wünschen in das digitale System.

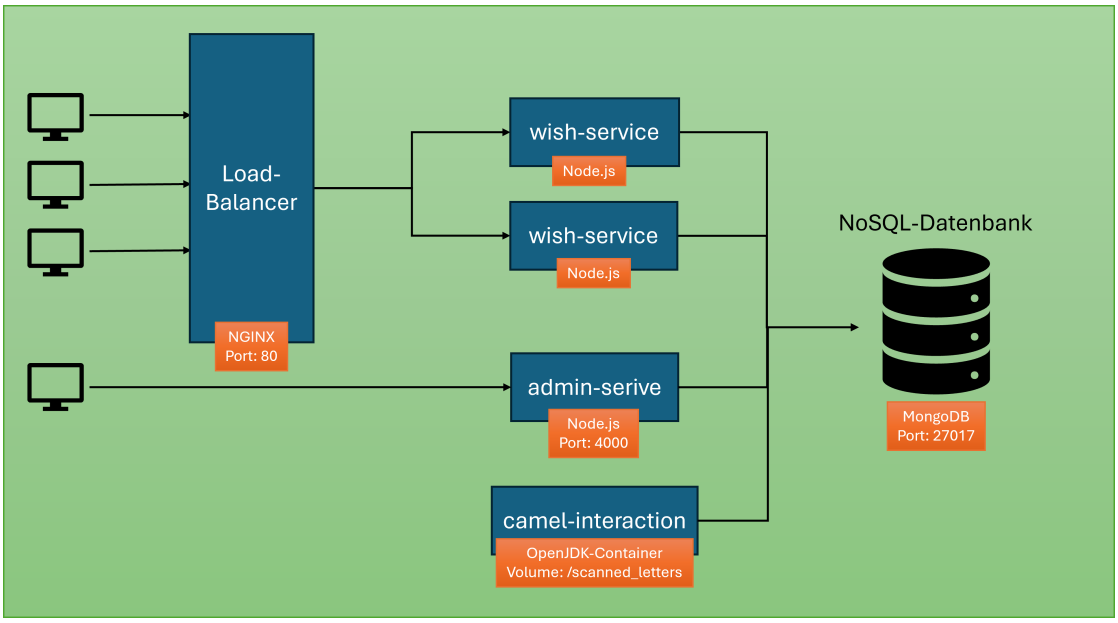


Abbildung 3: Architektur des implementierten Prototypen.

(a) Frontend des Wish-Service

(b) Frontend des Admin-Service

Abbildung 4: Frontends der Services

Leistungstests

Zur Bestimmung der maximalen Anzahl an API-Calls pro Sekunde, die das System stabil verarbeiten kann, bevor die Fehlerrate signifikant ansteigt, wurde das Tool *Grafana k6*[2] verwendet. Auf Basis eines einfachen, in JavaScript geschriebenen, Tests, können mit Hilfe des Tools Virtual Users (VUs) erzeugt werden, die ein realistisches Nutzerverhalten simulieren.

Testaufbau:

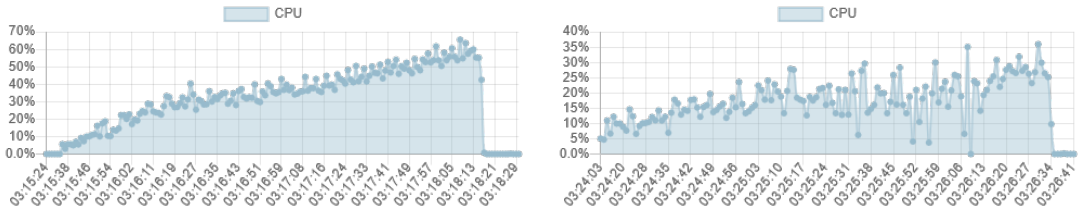
- Maximal 500 virtuelle Benutzer, wobei die Nutzerzahl um 100 Benutzer pro 30 Sekunden ansteigt.
- Dauer eines Testdurchlaufs sind somit 3 Minuten.
- HTTP Post Anfragen mit Namen und Wünschen werden über den Load-Balancer an die Wish-Services gesendet und auf eine Antwort des Services gewartet.

Die folgende Tabelle zeigt die Resultate zweier Tests, einmal mit 2 und einmal mit 5 Wish-Service Instanzen.

Metrik	**2 Wish-Services**	**5 Wish-Services**
Maximale Virtuelle Nutzer	500	500
Anfragen gesamt	37.211	37.321
Erfolgreiche Anfragen	37.149	37.201
Fehlgeschlagene Anfragen	62	120
Fehlerrate	0,16%	0.32%
Maximale Anfragerate	245 API-Calls/Sekunde	227 API-Calls/Sekunde
Durchschnittliche Antwortzeit (avg)	10,72 ms	10,78 ms
Durchschnittliche Antwortzeit (p90)	16,69 ms	13,07 ms
Daten empfangen	26 MB	26 MB
Daten gesendet	7,2 MB	7,2 MB

In beiden Tests war zu erkennen, dass sämtliche fehlgeschlagenen Anfragen gegen Ende des Tests, also zur maximalen Last, aufgetreten sind. In den beiden Testdurchläufen hat das System mit 2 Wish-Service Instanzen etwas besser abgeschnitten als das System mit 5 Instanzen. Vermutlich liegt dies allerdings an der Gesamtbelastung des Systems, da drei Container zusätzlich betrieben werden müssen. Abbildung 5 zeigt beispielhaft die Gegenüberstellung der Auslastung jeweils eines Wish-Services in beiden Tests. Es zeigt sich, dass die Auslastung bei nur 2 Wish-Service Instanzen deutlich höher ist als bei 5 Instanzen. Die Auslastung des Load-Balancers sowie des MongoDB-Containers sind auf maximal 15 % gestiegen. Die horizontale Skalierung durch hinzufügen weiterer Container scheint also zu funktionieren, solange das Gesamtsystem die Last verarbeiten kann. Alternativ können Container natürlich auf mehrere Maschinen aufgeteilt werden.

[2] Siehe `https://k6.io/`



(a) Auslastung bei 2 Wish-Services

(b) Auslastung bei 5 Wish-Services

Abbildung 5: Container CPU-Auslastung von jeweils einem Wish-Service während den beiden Tests.