Erhöhte Skalierbarkeit durch eine Serverlessimplementierung von Downloadvorgängen im GeRDI Projekt unter dem Einsatz von Kubernetes

Bachelorarbeit

Matthias Oppermann

29. September 2019

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
M.Sc. Nelson Tavares de Sousa

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

_____

Zusammenfassung

In dieser Arbeit wird ein Einblick in die Skalierbarkeit von Donwloadvorgängen mittels einer Serverlessimplementierung gegeben. Serverless beinhaltet eine zustandslose Funktion, bei der eine Funktion auf einer von einem Cloud Anbieter gestellten Infrastruktur ausgeführt wird. Diese Ausarbeitung beschränkt sich auf Function-as-a-Service (FaaS) als Serverlessfunktion, bei der eine in einem Container bereitgestellte ereignisgesteuerte Funktion erstellt wird. In dem GeRDI Projekt befinden sich 2 Microservices, die in einem Kubernetes Cluster arbeiten und einen Downloaddienst anbieten. Diese Komponente wird in der neuen Implementierung in eine Serverlessfunktion exportiert. Dazu wird in dieser Ausarbeitung untersucht, wie der Stand der Technik, die grundlegenden Technologien und wie die Downloadfunktion in eine Serverlessfunktion exportiert werden kann. Die neugestaltete Implementierung der Komponente wird in einem Testcluster auf ihre Performance bei unterschiedlichen Skalierungsgraden der Serverlessfunktion mit der ursprünglichen Implementierung verglichen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Beitrag dieser Arbeit zum GeRDI Projekt besteht darin die Skalierbarkeit des Storings zu verbessern, indem die Downloadfunktion gekapselt und dann beliebig oft mittels Kubernetes repliziert werden kann. Die Vorteile einer Serverlessimplementierung motivieren wir in dem folgenden Kapitel. Zusätzlich wird der Aufbau dieser Arbeit detailliert erläutert.

1.1. Motivation

Durch die Entwicklung zur immer feineren Modularisierung von Diensten und Funktionen zu einzelnen Microservice hin zu Serverlessfunktionen können moderne Systeme grössere Nuteraufkommen bewältigen als vergangene Monolithische. In dieser Architektur werden alle Komponenten zu einer logischen Einheit, auch als Monolithen bezeichnet, gebündelt. Monolithen sind vertikal skalierbar (Scale-up), durch das Hinzufügen von Speicher oder Prozessorkerne. Da die Funktionen sehr eng gekoppelt sind und es nur eine große gemeinsame Datenbank zum Persistieren von Daten gibt, ist eine horizontale Skalierung (Scale-out) mit erheblichen Codeanpassungen verbunden oder aufgrund von relationalen Datenbanken unmöglich.
Da die Microservices der Storage-Komponente nicht zustandslos sind, können diese nicht horizontal skaliert werden und laufen daher nur auf einem Server. Aufgrund dieser Eigenschaft kann das bei vermehrter Nutzung zu Engpässen führen. Um das Gesamtsystem performant zu halten, ist es notwendig, diese Store-Komponente des Systems skalierbar zu halten. Hierzu wird die Downloadfunktion aus den Storeservices extrahiert und als eigenständige Serverless Anwendung entwickelt. Dabei bedeutet Serverless nicht, dass kein Server benötigt wird, sondern der Entwickler muss sich bereits bei der Anwendung von Containern nicht um Infrastrukturen kümmern. Hier ist der Vorteil, dass eine Funktion bereitgestellt wird, ohne dabei abhängig von anderen Systemteilen zu sein. [Fox+17][Hec][Luk18, Seite 5]

1.2. Aufbau

Meine Bachelorarbeit ist wie folgt aufgebaut:

Kapitel 2 präsentiert die Ziele die umgesetzt werden sollen.

Kapitel 3 stellt die grundlegenden Technologien da, die für die Umsetzung benötigt werden.

Kapitel 4 beschreibt die für diese Arbeit betroffenen Komponenten des Storeservice anhand von Klassendiagrammen und einzelnen Codeausschnitten.

Kapitel 5 illustriert die Umsetzung der in Kapitel 2 formulierten Ziele.

Kapitel 6 beschäftigt sich mit der Analyse und Auswertung der neuen Implementierung.

Kapitel 7 stellt einen Bezug zu verwandten Arbeiten her.

Kapitel 8 in dem das Fazit und ein Ausblick für zukünftige Erweiterungen des Storings gegeben wird.

Kapitel 2

Ziele

2.1. Ziel 1: Extrahieren der Downloadfunktion

Das erste Ziel dieser Arbeit ist es, die Downloadfunktion, die im Storeservice enthalten ist, zu extrahieren und als eigenständige Serverlessfunktion zu implementieren. Zur Zeit finden alle Downloadvorgänge eines Microservices auf dem gleichen Server statt. Das hat zur Folge, dass bei einer Vielzahl von Downloadvorgängen der Server an seine Grenzen stößt und es so zu Verzögerungen kommt. Durch die Serverlessimplementierung dieser Komponente wird die Last effektiver auf mehrere Server im Cluster verteilt und Engpässe können umgangen werden.

2.2. Ziel 2: Deployment der Downloadfunktion in Kubernetes

Das Deployen einer einzelnen Funktion als Function as a Service (FaaS) verursacht zusätzlichen Verwaltungsaufwand. Hierzu wird Kubernetes zur Orchestrierung verwendet. Der Master ermittelt freie Ressourcen und verteilt dann die Funktion auf die Nodes . Über die API können wir im laufenden Betrieb weitere Instanzen der Funktion erzeugen, das Prinzip der Skalierung wird in Kapitel 3 behandelt. [AUTa]

2.3. Ziel 3: Kommunikation

Der Storeservice sendet an die einzelnen Serverlessfunktionen eine Liste von Downloadaufgaben. Weiter kann ein Status abgerufen werden, ob die zugeteilten Downloads fertigstellt sind oder nicht. Dadurch hat der Storeservice einen Überblick, wann der gesamte Downloadvorgang abgeschlossen ist. Damit das Frontend einen detaillierten Zustand vom Prozessfortschritt darstellen kann, ruft der Storeservice detaillierte Listen von den Serverlessfunktionen ab, die alle Informationen von den Downloads haben. Diese Teillisten werden dann zusammengesetzt und an das Frontend weitergeleitet.

Kapitel 3

Grundlagen und Technologien

Die eingesetzten Technologien beschränken sich auf den Einsatz von Microservices, Function as a Service, Docker-Container und deren Orchestrierung durch Kubernetes. Beides wird in den folgenden Unterkapiteln eingehendst erläutert.

3.1. GeRDI

GeRDI (Generic Research Data Infrastructure) ist ein Infrastrukturprojekt für Forschungsdaten. Ziel ist, es eine fachübergreifende Weitergabe von Forschungsdaten zu ermöglichen. Dabei soll Wissenschaftlern der Zugang von Forschungsdaten erleichtert werden.

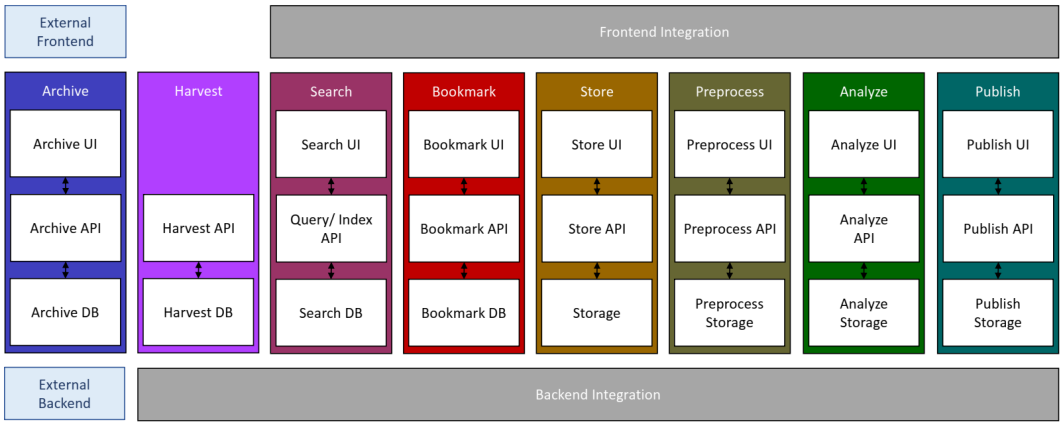

Abbildung 3.1. GeRDI System Architektur [Sou+]

Die in dieser Arbeit zu betrachtenden Komponenten sind die in Abbildung 3.1 dargestellten Bookmark und Store. Wobei im Bookmark eine Liste von Forschungsdaten erstellt wird, welche dann dem Store aufzeigt, welche Daten gedownloadet werden sollen. [Gru+] [pro]

3.2. Microservices

Microservices sind eigenständige Dienste, die genau eine Funktion implementieren. Dabei sind Microservices unabhängig voneinander und haben eigene für den Anwender sichtbare Nutzeroberflächen. Das hat zum Vorteil, dass sie sich leicht ersetzen lassen und von einem Entwicklerteam innerhalb kürzester Zeit neu erstellt werden können. Weiter sind Microservices unabhängig in der Auswahl der Programmiersprache oder Datenbank. Doch eine Microservice-Architektur bedarf einer umfangreichen Infrastruktur. Dabei geschieht die Auslieferung meist in Form von Containern und die Lastenverteilung externer HTTP-Anfragen wird durch Loadbalancer, wie zum Beispiel Kubernetes Services, realisiert. Vgl. [Luk18, Seite 3-10] [IEE+18]

3.3. Function as a Service (FaaS)

Function as a Service beschreibt eigens entwickelte zustandslose Funktionen, die in Containern bereitgestellt werden. Weiter können mehrere Serverless-Funktionen zu einem eigenen Microservice zusammengefasst werden. In dieser Bachelorarbeit wird Function as a Service mit dem Begriff Serverless gleichgesetzt. Grundlegende Eigenschaften von FaaS beinhalten, dass sie ereignisgetrieben und zustandslos sind, sowie eine Cloud-Plattform benötigen, um effizient auf und ab zu skalieren. Die Cloud-Plattformen stellen dabei die Infrastruktur. Diese ermöglicht dem Entwickler, das Infrastrukturdesign, Servicequalität, Skalierung und Fehlertoleranz komplett an den Plattformanbieter abgeben zu können. Ein weiterer großer Vorteil von FaaS ist, dass nur die verbrauchte Rechenzeit berechnet wird und dadurch keine extra Gebühren anfallen, wenn der Code sich beispielsweise im Leerlauf befindet. FaaS ist zwar vielseitig jedoch nicht überall von Vorteil. Aufgrund ihrer Zustandslosigkeit sind sie bspw. für Video-Streaming oder Datenbanken nicht geeignet. [Fox+17] [19]

3.4. Docker-Container

Eine der bedeutendsten Trends der Software Entwicklung ist es, die Software in Containern auszuliefern.
Ein Docker-Container Image umfasst eine einzelne Anwendung samt aller Abhängigkeiten wie Bibliotheken und Hilfsprogrammen. Dadurch wird das Deployen einer Anwendung vereinfacht, da nur das Image für die Container verteilt werden muss. Container sind dabei unabhängig vom ausführenden Betriebssystem , da sie in ihrer eigenen Laufzeitumgebung ausgeführt werden. Veranschaulicht wird dies in der Abbildung 3.2.

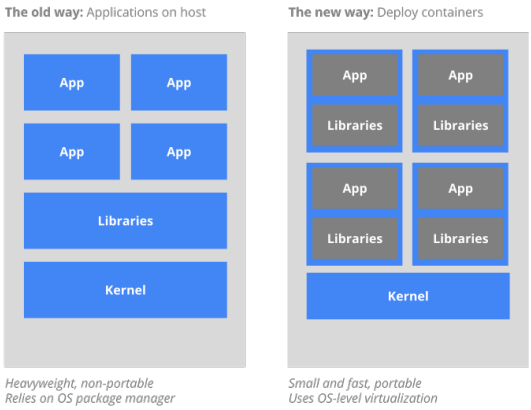

Abbildung 3.2. Container Vergleich [AUTi]

Weiter benötigen sie im Gegensatz zu virtuellen Maschinen deutlich weniger Ressourcen, da sie im Kontext des Host-Betriebsystems laufen. Aus diesem Grund werden Container auch gerne als leichtgewichtige Virtualisierung bezeichnet und es können bei gleicher Hardware deutlich mehr Container als virtuelle Maschinen betrieben werden. Ein zusätzlicher Vorteil besteht in der einfachen Skalierbarkeit dieser Software Architektur. Wenn weitere Instanzen eines Systems benötigt werden muss lediglich ein neuer Container gestartet werden. Dabei ist jeder Container über das Netzwerk erreichbar und besitzt eine eigene IP-Adresse. Wird jedoch eine größere Anzahl von Containern benötigt, empfiehlt sich der Einsatz von Orchestrierungstools wie Google Kubernetes. [Aug17] [Ism+15]

3.5. Kubernetes

Kubernetes nutzt Docker zum Verteilen von containerisierten Anwendungen. Es verwaltet die Container, die von einer Container-Engine wie Docker bereitgestellt werden. Kubernetes wird, wie in 3.3 dargestellt, nach der Master-Slave Architekur aufgebaut, wobei der Master die einzelnen Nodes überwacht und Pods erstellen kann. [Inc19]

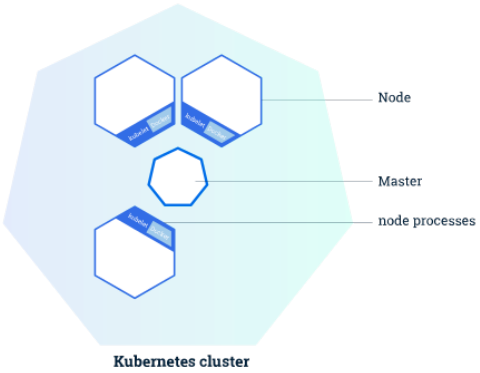

Abbildung 3.3. Kubernetes Komponenten [AUTg]

3.5.1. Master

Der Kubernetes-Master steuert und verwaltet den Cluster. Eine wichtige Komponente des Masters ist der API-Server. Dieser stellt einen Teil der Steuerebene dar und ist für die Kommunikation zwischen Master und Node verantwortlich. Eine bedeutende Funktion des API-Server ist der Replicationscontroller, mit dem die Anzahl von Instanzen einer Anwendung im Cluster bestimmt werden kann. Weitere Elemente der Steuerebene sind der Scheduler, der für die Planung und Überwachung von Pods zuständig ist, der Controller-Manager, der die Funktionen zur Zustandsänderung ausführt und die etcd Datenbank, welche die Konfiguration des Clusters speichert. [Luk18, Seite20-21, 50] [AUTb]

3.5.2. Nodes

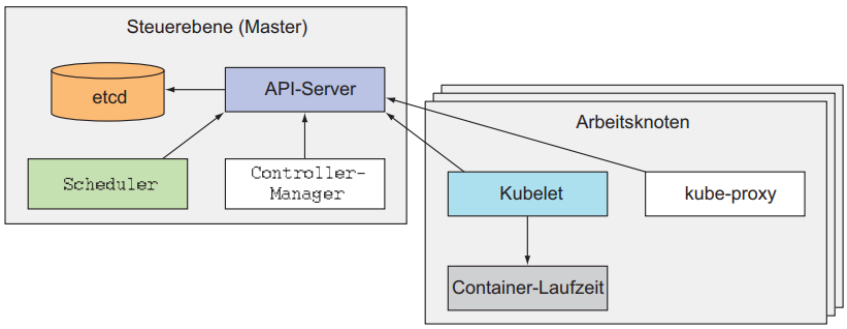

Abbildung 3.4. Kubernetes Aufbau [Luk18, Seite 21]

Eine Node oder auch Minion stellt einen physischen oder virtuellen Server dar. Sie beinhaltet wie in Abbildung 3.4 dargestellt eine Dockerlaufzeitumgebung, in der die

Container ausgeführt werden. Ein weiterer Bestandteil einer Node ist das Kubelet, welches die Schnittstelle für den Master bildet. Wie in Abbildung 3.4 dargestellt, kann der Master mittels API-Server über das Kubelet die Container steuern. Zusätzlich hat eine Node einen Kube-Proxy , der die Proxy- und Lastenausgleichsfunktion zwischen den Komponenten übernimmt. [Luk18, Seite20-22] [AUTb]

3.5.3. Pods

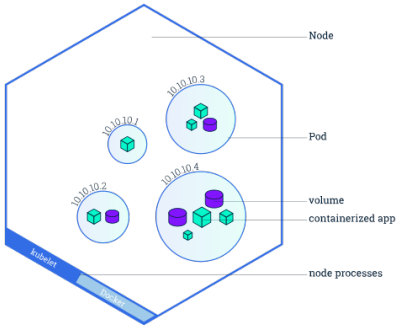

Abbildung 3.5. Kubernetes Node [AUTh]

Pods sind die kleinste Einheit im Kubernetescluster und können beliebig skaliert werden. Wie in 3.5 dargestellt, beinhaltet ein Pod einen oder mehrere Container, in denen die Services bzw. Microservices ihren Dienst anbieten. Mehrere Container innerhalb eines Pods teilen sich ein Dateisystem und einen Netzwerk-Namespace/Adresse. [Bry17] [AUTd][AUTc]

3.5.4. Volumes

Ein Kubernetes-Volume ist eine Komponente eines Pods und kann nur innerhalb dieser erstellt oder gelöscht werden. Dabei kann jeder Container innerhalb eines Pods auf das Volumen zugreifen, nachdem er dieses gemountet hat. Es gibt verschiedene Typen von Volumes innerhalb eines Kubernets-Clusters, wobei wir uns in dieser Arbeit auf die „persistente Volume" und „Claims" beschränken. Ein „PersistentVolumeClaim" (PVC) bietet uns die Möglichkeit, auf einen vorab definierten oder dynamisch bereitgestellten Speicher zuzugreifen. Damit in einem Kubernetes-Cluster Speicher angefordert werden kann, müssen zwei neue Ressourcen eingeführt werden, nämlich „persistente Volumes" und „Claims".

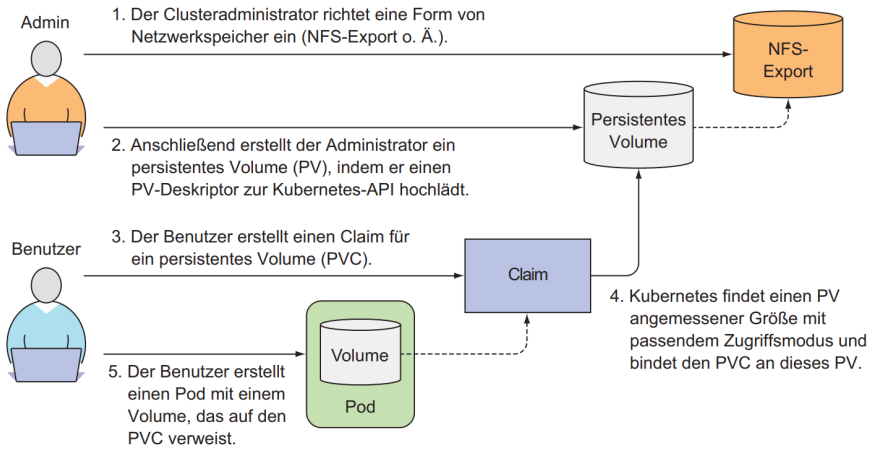

Abbildung 3.6. Persistente Volumes über Claim bereitgestellt [Luk18, Seite 196]

Dabei wird, wie in Abbildung 3.6 dargestellt, über einen Administrator ein Speichermedium zur Verfügung gestellt. Der Speicher wird dann als „PersistentVolume" über den API-Server als neue Ressource registriert. Damit ein Pod auf das persistente Volume zugreifen kann, wird zunächst eine Manifestdatei vom Typ „PersistentVolumeClaim" erzeugt, in der alle nötigen Angaben zur Größe und Zugriffsmodi enthalten sind. Das Manifest wird dann an den API-Server übergeben, der dann das Claim an das persistente Volume bindet. Nun kann ein oder mehrere Pods ein Volume erstellen, das auf den PVC verweist. [Luk18, Seite 178, 195–210]

3.5.5. Kubernetes-Deploymensts

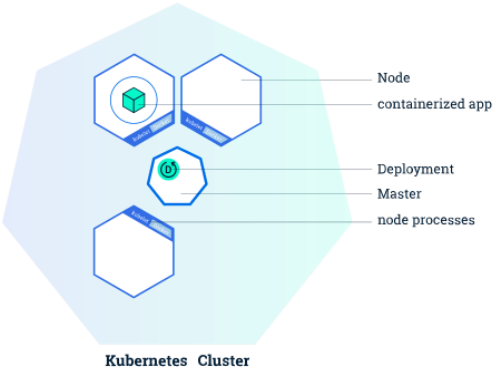

Abbildung 3.7. Kubernetes Deployment [AUTf]

Die in einem Container verpackte Anwendung wird im Deployment, wie in Abbildung 3.7 dargestellt, über den Master auf die einzelnen Nodes und Pods im Kubernetes-Cluster verteilt. Wie die einzelnen Instanzen erstellt, aktualisiert oder welchem Label zugeordnet werden, wird deklarativ in der Kubernetes Deployment-Konfiguration definiert. Die Überwachung einer jeden Instanz übernimmt der Deployment-Controller. Antwortet ein Pod nicht mehr, startet Kubernetes sie automatisch neu. Auch wenn eine Node ausfällt, wählt der Master eine neue aus und liefert dort alle Container neu aus und startet sie dort erneut. [AUTf] [Luk18, Seite 25-26 , 275–280]

3.5.6. Kubernetes-Services

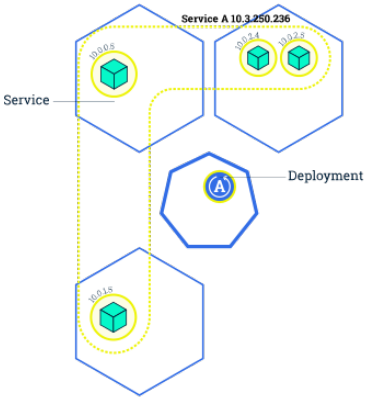

Abbildung 3.8. Kubernetes Service [AUTe]

Wie in der Abbildung 3.8 zu entnehmen, sind Kubernetes-Services eine Abstraktion einer Menge von Pods, wobei jeder Pod eine einzigartige ClusterIP-Adresse enthält. Der Service stellt eine eigene Ressource dar und besitzt ebenfalls eine IP-Adresse, die als Einstiegspunkt für die Pods dient. Der Client spricht die in einem Pod befindliche, containerisierte Anwendung nur über den Kubernetes-Service an. Dabei ist es irrelevant, wie viele Replikationen es davon gibt.[AUTd] [Luk18, Seite 136]

3.6. WebDAV

Ist eine Erweiterung des Hypertext Transfer Protocol (HTTP). Es ermöglicht den Transfer von Dateien oder ganzer Ordnerstrukturen von einer Quelle zu einem Ziel. Sowohl bei COPY als auch bei MOVE wird der Inhalt an die Quellressource gesendet. Dabei wird das Ziel durch den Zielheader angegeben. Der Zugriff auf ein WebDAV kann durch das einbinden als Netzlaufwerk in einem Betriebssystem oder über eine Nutzeroberfläche

im Web-Browser erfolgen. Diese Eigenschaft ermöglicht einen einfachen und direkten Datenzugriff.[E J97]

3.7. REST und HTTP

REpresentational State Transfer (REST) bezeichnet einen ressourcenorientierten Architektur-Stil für die Client/Server Kommunikation. Ein weiteres Merkmal von REST ist die Zustandslosigkeit. Das heißt, ein Zustand wird vom Client gehalten oder vom Server in einen Ressourcenstatus umgewandelt. Ein bekannter Vertreter von REST ist HTTP (Hypertext Transfer Protokoll) mit den Methoden GET, POST, PUT und DELETE. Dabei stellt der Client einen Request an den Server und der antwortet mit einem Response. In dieser Arbeit werden lediglich GET und POST Befehle genutzt. Mit Anfragen der Form GET <URL> wird direkt eine Ressource angesprochen. Beim POST können wir zusätzliche Inhalte über den „ObjectBody" versenden. Dieser weitere Austausch von Daten-Formaten im Bereich Webservice wird mittels JavaScript Oject Notation (JSON) oder Extensible Markup Language (XML) bewerkstelligt. In dieser Arbeit nutzen wir JSON zum Übermitteln von Inhalten, da in JAVA eine Vielzahl von Parsern zu Verfügung steht, die Java Objekte in JSON Strings umwandeln kann. Dieser Vorgang wird auch als Serialisierung und die Rückrichtung als Deserialsierung bezeichnet.[Zin] [Daz] [Ber][Nur+]

3.8. JSON

Ein JSON-Objekt bildet eine Baumstruktur, wie an einem einfachen Beispiel in Listing 3.1 illustriert. Die äußeren geschweiften Klammern bilden die Wurzel und stellen ein Objekt dar. Die darin enthaltenen Knoten werden durch ihre Schlüßelwerte repräsentiert. Der Inhalt der Knoten folgt nach dem Doppelpunkt. Ein Objekt wird mit geschweiften und eine Liste mit eckigen Klammern dargestellt. Die Übergabewerte werden in Hochkommata eingefasst. Die Elemente eines Objektes oder Liste werden durch einfache Kommata getrennt.

```
{
 "name":"Max Mustermann",
 "alter": 5,
 "adressen":["Wohnsitz1" ,"Wohnsitz2"]
}
```

Listing 3.1. Beispiel JSON

Kapitel 4

Stand der Technik

4.1. Architektur

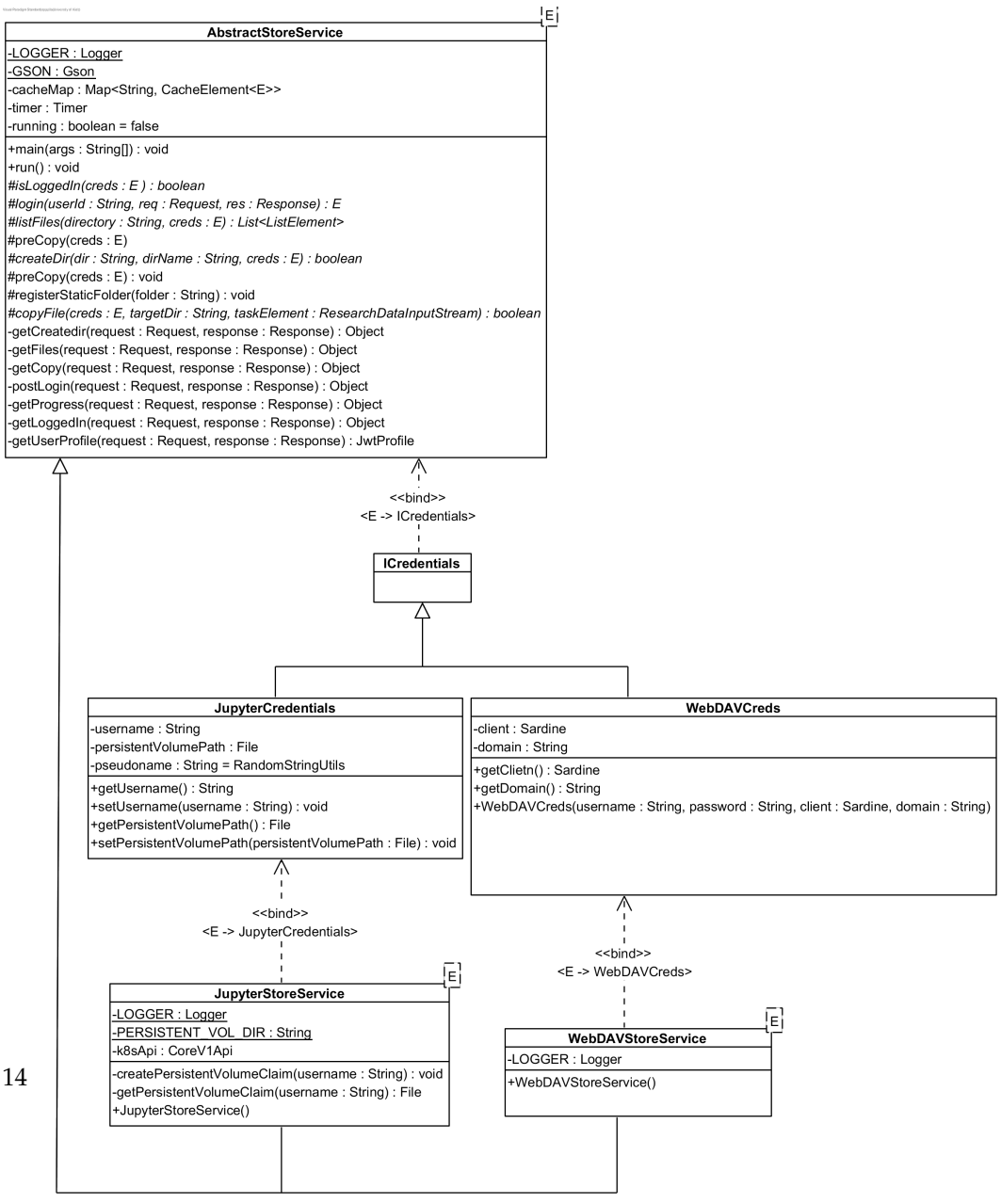

Abbildung 4.1. Klassendiagramm des AbstractStoreService

Die in Abbildung 4.1 dargestellte ursprüngliche Architektur des Storeservice bildet sich aus einer abstrakten Klasse, dem AbstractStoreService, und den abgeleiteten Klassen JupyterStoreService und WebDAVStoreService. Die Hauptklasse hat einen generischen Typen, der von dem Interface ICredentials erbt. Die eigentlichen Storeservices belegen dieses Feld mit dem JuptyterCredentials und dem WebDAVCreds die das Interface ICredentials implementiert haben.

4.2. Implementierung von Downloads

Nach dem sich ein Nutzer in dem Storeservice eingeloggt und die Downloaddaten via JSON an den Storeservice übermittelt hat, wird eine Session-ID generiert und an den Nutzer zurückgegeben. Anhand der generierten Session werden die Downloaddaten in einer Cache-Map gespeichert, wodurch der Storeservice nicht mehr zustandslos ist.// Vor Beginn der Umstrukturierung war die **copyFile** Methode ,wie in Abbildung 4.1 dargestellt, als abstract gekennzeichnet und musste in den jeweiligen Storeservices explizit implementiert werden. Wenn ein User mittels GET-Request die **getCopy** Methode des Storeservice aufruft, wird zuerst der „request" Parameter mit der „Session_ID" ausgelesen. Mit dieser ID werden dann die Downloadelemente vom Typ „ResearchDataInputStream" als Liste aus der „cacheMap" ausgegeben und anschließend für jedes Element die **copyFile** Methode aufgerufen, wie in Listing 4.1 dargestellt..

```
private Object getCopy(Request request, Response response) {
        final String session = request.params(StoreConstants.SESSION_ID);
        final CacheElement<E> cacheElement = cacheMap.get(session);
                ...
        for (final ResearchDataInputStream entry : cacheElement.getTask().getElements()) {
            if (!this.copyFile(creds, targetDir, entry)) {
                acknowledgedAll = false;
            }
        }
        return "";
    }
```

Listing 4.1. getCopy

Der Download findet dann in der implementierten **copyFile** Methode innerhalb des jeweiligen Storeservices in einem seperaten Thread statt.
In dem JupyterStoreService werden die Downloads in einem PVC in den Kubernetes-Cluster speichert. In diesem PVC wurde zuvor beim Loginvorgang des Storeservice eine Ordnerstruktur für den User erstellt, in der die Daten abgelegt werden. Beim WebDAV

handelt es sich um eine entfernte Ressource, auf die mit einem Client vom Typ Sardine zugegriffen wird.

Beide Storeservices arbeiten jeweils im realen Betrieb als eigenständiger Kubernetes-Service in einem Pod. Aufgrund der Zustandsgebundenheit des Storeservice , ist dieser nicht skalierbar, da sonst der Login und der anschließende Downloadauftrag an unterschiedliche Pods gesendet werden könnte. Das hat zur Folge, dass die **getCopy** Methode eine Session-ID übergeben wird für die es keinen Eintrag in der Cache-Map gibt. Das gleiche Problem entsteht beim Aufruf der **getPorgress** Methode, wo anhand der übergebenen Session-ID eine serialisierte Liste aus dem Cache-Map Eintrag erzeugt und an das Frontend zurückgegeben wird. Weil die Ressourcen eines Pods begrenzt sind, ist dies der limitierende Faktor, wenn eine hohe Last durch die Downloadfunktion erzeugt wird.

Kapitel 5

Implementierung

5.1. Architektur nach Umbau

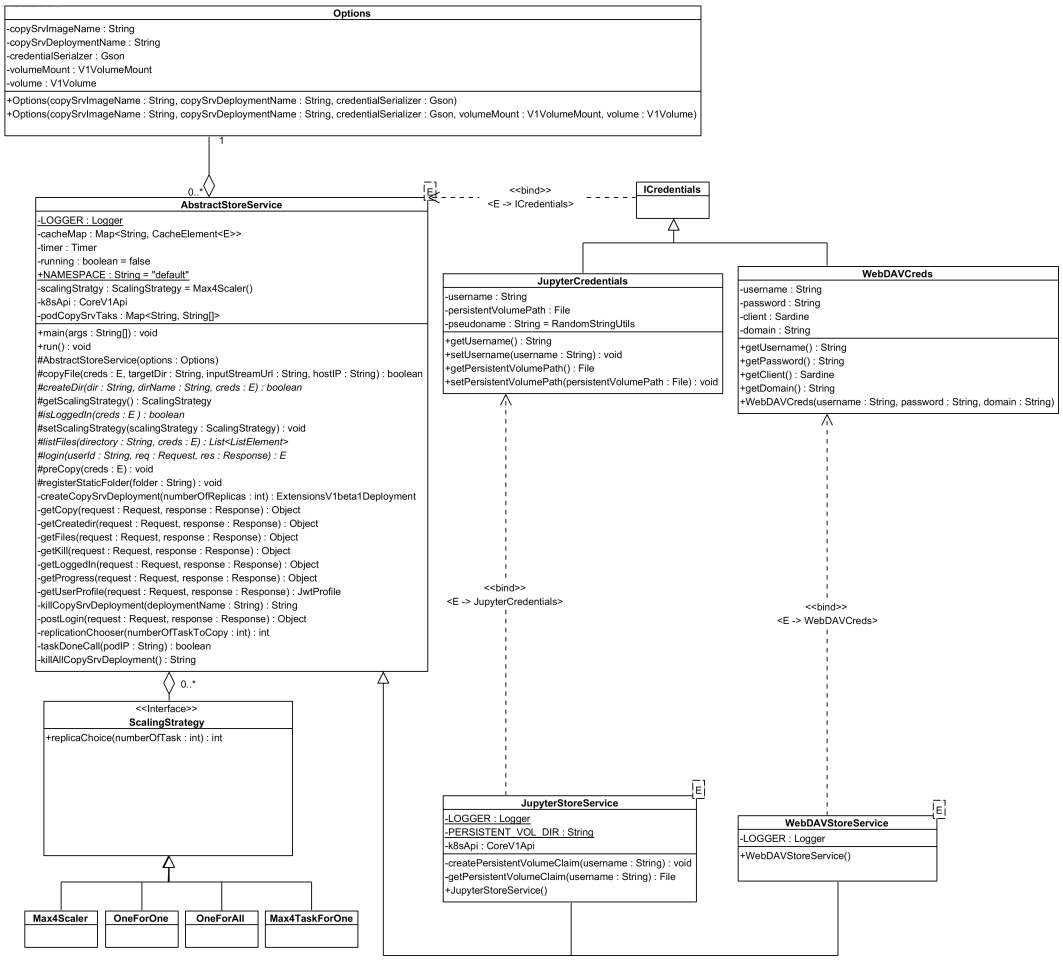

Abbildung 5.1. Klassendiagramm des AbstractStoreService nach Umbau

Die in Abbildung 5.1 dargestellte Architektur des neugestalteten „AbstractStoreService" enthält neue Klassen vom Typ „ScalingStrategy" und „Options". Von diesen Klassen erzeugte Instanzen werden über den Konstruktor an den „AbstractStoreService" übergeben. In der Options-Klasse werden die nötigen Daten für das interaktive Deployment der für den Download zuständigen Faas und ein Serialisierungs-Objekt vom Typ Gson gespeichert. Die „ScalingStrategy" gibt vor wie viele Instanzen der Serverlessfunktion erzeugt werden sollen. Der Deployprozess wird im Unterkapitel 5.3.2 detailliert erläutert. Die Bedeutung des Attributes „credentailSerializer" aus dem Options-Objekt wird im Abschnitt 5.4 verdeutlicht und die Neugestaltung der **copyFile** Methode wird direkt im Anschluss beschrieben.
In dem Attribut „podCopySrvTaks" vom Typ Map, werden die IP-Adressen der Pods gespeichert, die für eine Session-ID den Downloaddienst als Serverlessfunktion ausführen.

5.2. Extrahierung der Downloadfunktion

Um die Downloadfunktion als Severlessfunktion zu exportieren und mit dieser kommunizieren zu können , bedarf es eines Webservers, der dann mittels Http-Request über die IP-Adresse, dem Port und dem Pfad erreicht werden kann. Dazu werden die statischen „get", „post" und „port" Methoden von Apache Spark importiert, siehe Abbildung 5.2.

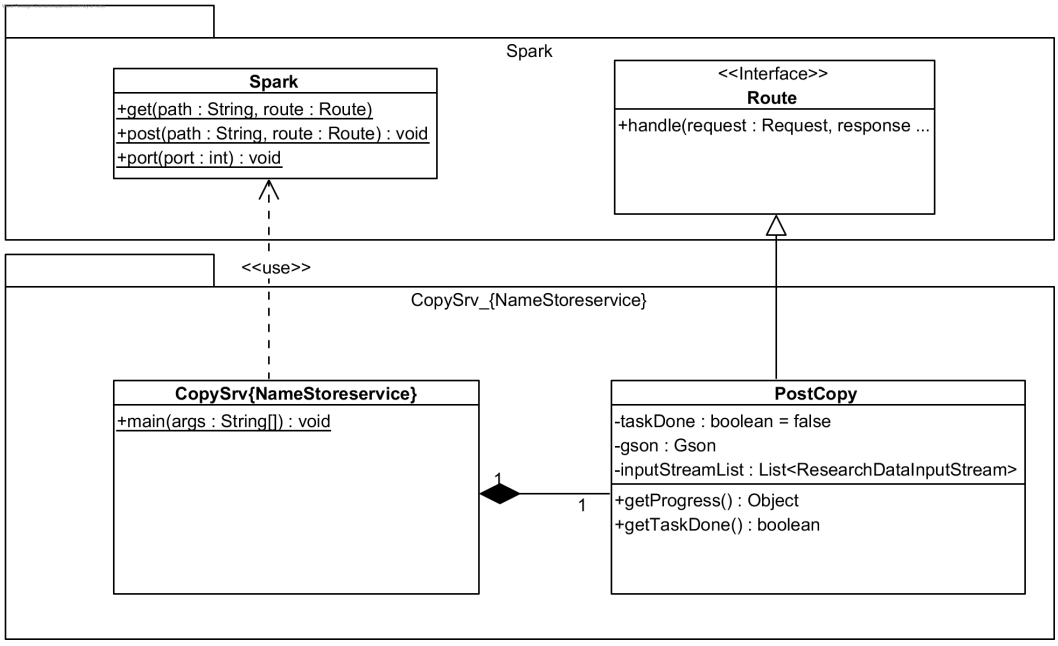

Abbildung 5.2. Klassendiagramm der Serverlessfunktionen für den WebDAV und JupyterHub Storeservice

Diese werden in der **main** Methode aufgerufen und legen den Port für die Kommunikation und den Pfad für die GET und POST Requests an die Serverlessfunktion fest. Dabei bilden die folgenden Pfade und der Port die Schnittstellen für die Interaktion:

PORT: 5679

POST: /copy

GET: /progress

GET: /taskDone

Um einen Bezug auf die **copyFile** Funktion zu haben, in dieser der Download vom Storeservice versendet wird, beginnen die Klassen für die Serverlessfunktionen mit „CopySrv". Im weiteren Verlauf dieser Arbeit wird der Begriff „CopySrv" für eine Serverlessfunktion stehen, die den Downloadservice anbietet.

Da der Downloadvorgang aus **copyFile** Methode für jeden StoreService extrahiert und in einer separaten Serverlessfunktion exportiert wird, kann die Methode vereinheitlicht werden, so dass sie in der AbstractStoreService Klasse ausformuliert werden kann. Dazu werden,wie in Abbildung 5.1, die Übergabeparameter der Methode angepasst, so dass eine Liste aus Strings, in der die Downloadadressen enthalten sind, aus der die Serverlessfunktion eine Liste vom Typ ResearchDataInputStream erzeugt. Die IP, mit der dieser Service in Kombination mit dem Port 5679 erreicht werden kann, wurde mit dem Feld „hostIP" realisiert. Dabei werden alle für den Downlaod nötigen Objekte in eine Map verpackt, die dann an den Service mittels POST:/copy Request versendet werden. Dieser Prozess wird im nächsten Unterkapitel 5.4 erläutert.
In der der CopySrv wird dann die Methode **handle** aufgerufen, in der die Logik für den Downloadvorgang ausgelagert wird. Aus dem „request" Parameter der **handle** Methode werden dann die nötigen Daten für den Downloadvorgang extrahiert und deserialisiert. Wie dieser Vorgang im einzelnen geschieht, wird in der Sektion 5.4 und 5.4.2 beschrieben. Nach der Deserialisierung gibt es eine Liste vom Typ „ResearchDataInputStream", die die Downloadaufgaben für den jeweiligen CopySrv beinhalten. Über eine for-Schleife werden dann die einzelnen ResearchDataInputStream-Objekte aus der Liste ausgelesen. Diese enthalten die Downloadadresse und den Prozessstatus. Der Status wird bei der Instanzierung des Objektes auf „PENDING" oder „UNKNOWN_SIZE" (bei unbekannter Dateigröße) gesetzt. Nach dem Starten des Downloadvorgangs wird der Wert auf „RUNNING", bei Fehlschlägen auf „ERROR" und nach Fertigstellung auf „FINISHED" geändert.

Da der Download in den Serverlessfunktionen stattfinde,t hat der Storeservice keinen Überblick mehr über den Fortschritt des Downloadsprozesses. Aus diesen Grund wurde die **getProgress** Methode angepasst, so dass der aktuelle Status von den Serverlessfunktionen über den GET:/progress Request abgerufen werden kann. In den CopySrvs

wird dann die **getProgress** Methode aufgerufen, die von den Downloadaufgaben vom Typ „ReserachDataInpuStream" in eine serialisierte JSON-Liste erzeugen. Diese Listen werden anschließend vom Storeservice wieder zusammen gesetzt, sodass dieser bei einer „GET:/progress/SESSION_ID" Anfrage den Downloadstatus für eine Session wiedergeben kann. Dieser wird bei einem Downloadvorgang vom Frontend kontinuierlich abgefragt.

Um zu ermitteln, wann der komplette Download abgeschlossen ist und der Storservice die erzeugten CopySrvs wieder entfernen kann, gibt es den „GET:/taskDone" Request. Durch diese Anfrage wird die **getTaskDone** Methode des PostCopy Objektes aufgerufen, die den Wert des „taskDone" Attributes wiedergibt. Dieser ist zur Initialisierung des „PostCopy" Objektes auf den Wert **false** gesetzt und wird nach Fertigstellung der Downloads auf **true** geändert. Beim aktiven Download sendet die CopySrv als Response den HTML-Code:200 und bei einem abgeschlossenen den HTML-Code:201. Anhand dieser Codes entscheidet der Storeservice, ob eine CopySrv beendet werden kann.

5.3. Deployment der Downloadfunktion

Bevor ein Storeservice eine CopySrv deployen kann, muss zuerst mit dem Build-Tool Maven der Java Code kompiliert und dann in eine ausführbare .JAR Datei inklusive aller Abhängigkeiten verpackt werden. Aus der JAR und dem Docker File wird dann mit Docker ein Docker Image erstellt und in eine Registry hochgeladen. Aus dieser Registry wird dann das Image im Deployprozess heruntergeladen und auf die Pods im Kubernetes-Cluster verteilt. Der in Abbildung 5.3 dargestellte Vorgang wird in den nächsten Unterkapiteln detailliert erläutert.

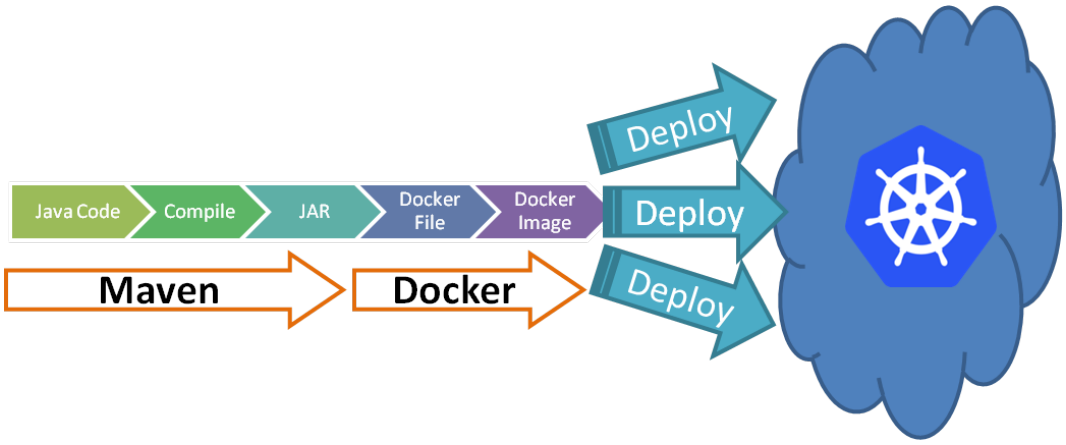

Abbildung 5.3. Deployprozess

5.3.1. Dockerfile

Zuerst wird das Dockerfile, wie in Listing 5.1 zu sehen, editiert. Dabei handelt es sich um eine Reihenfolge von Anweisungen, die Docker beim Erstellen des Images ausführt. In der **FROM** Zeile legen wir „openjdk Tag 8" als Basisimage fest, auf den wir das Containerimage aufbauen wollen. In der **COPY** Anweisung legen wir fest, wo die ausführbare „.jar" inklusive aller Abhängigkeiten kopiert werden soll. Über **WORKDIR** legen wir den Pfad fest, in dem man sich beim Ausführen des Images in einem Container befindet und **ENTRYPOINT** legt den Einstiegspunkt für das Programm fest.

```
FROM openjdk:8
COPY ./target/...jar−with−dependencies.jar /usr/src/store−prototype/app.jar
WORKDIR /usr/src/store−prototype
ENTRYPOINT ["java", "−jar" , "app.jar"]
```

Listing 5.1. Struktur Dockefile

Mit dem Befehl „docker build -t" wird aus den Definitionen aus dem Dockerfile ein Dockerimage erstellt. Um das Image zur Verfügung zustellen, wird es mit der Anweisung „docker push" in eine Registry hochgeladen.

5.3.2. Deployprozess durch den Storeservice

Das Deployment der Serverlessfunktion wird, wie in Abbildung 5.4 dargestellt, in der **getCopy** Methode über den Aufruf **createCopySrvDeployment** des jeweiligen Storeservice erzeugt. Der Name des Deployments bildet sich aus dem in dem Options Objekt enthaltenen Feld „DeploymentName" und der „SESSION_ID" der beim Login erzeugt und als Parameter im Request übergeben wird. Die Funktion **replicationChooser** entscheidet anhand der Menge der herunterzuladenden Dateien, wie viele Replika erzeugt werden sollen. Im Anschluss wird über eine Schleife der Status des Deployments so lange abgefragt, bis die zu erstellenden CopySrvs mit den fertig deployten Instanzen , hier als „readyReplica" gekennzeichnet,übereinstimmen. Über den Api-Client wird dann eine Liste von Pods ausgegeben, die zu dem Deployment gehören. Aus dieser Liste werden dann die IP-Adressen der Pods extrahiert und in der Hash-Map „podCopySrvTaks",siehe Abbildung 5.1 Klassenattribut AbstractStoreservice, als Key-Value Paar bestehend aus „SESSION_ID" und PodIP-Liste hinzugefügt. So hat der Storeservice einen Überblick, welche Pods für eine „SESSION_ID" den CopySrv ausführen.

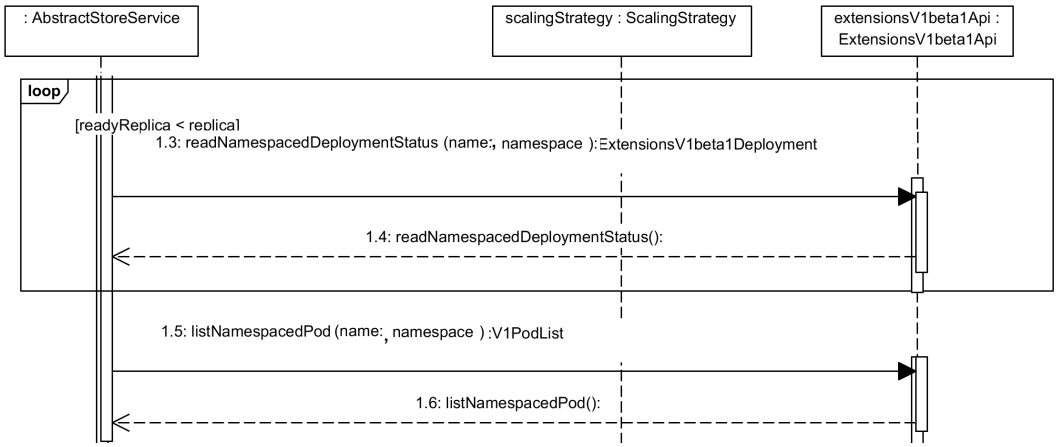

Abbildung 5.4. Ausschnitt der getCopy Methode für den Deployprozess

Wie die Methode **replicationChooser** entscheidet, wird über das Attribut **scalingStrategy** vom Typ **IStartegy**, siehe Abbildung 5.1, festgelegt. Dabei wird die Methode **replicaChioce** aufgerufen, die für eine Anzahl von Downloadaufgaben eine Menge von CopySrv Instanzen berechnet. Wird bei der Instanziierung eines Storeservice keine Strategie ausgewählt, so ist diese eine **Max4Scaler** Instanz, um ein ausgewogenes Verhältnis an Deployten CopySrvs und Downloads für jede Instanz zu erhalten. Im Paket „de.gerdiproject.store.util.ScalingStrategy"befindet sich das Interface samt aller Scaling-Strategien.
Die Konzepte der einzelnen Strategien sind:

Max4Scaler teilt die Anzahl der Downloadaufgaben durch 5 und addiert das Ergebnis um 1. Ist das Ergebnis der Berechnung größer als 4, wird der Rückgabewert auf 4 gesetzt, um die Anzahl der Replika zu begrenzen.

Max4TaskForOne hier wird die Anzahl der Downloads durch 4 geteilt und dieser Wert plus eins bildet die Anzahl der Deployten Instanzen.

OneForOne gibt die Quantität der Downloadaufgaben zurück, so dass jede CopySrv einen Download tätigt.

OneForAll gibt immer den Wert 1 zurück, so dass nur eine CopySrv deployt wird, die dann alle Downloads ausführt.

Dabei bieten„OneForAll"die geringste und „OneForOne"die maximalste Skalierung. „Max4TaskForOne" und „Max4Scaler"reihen sich dazwischen ein und bilden ein ausgewogenes Verhältnis an Downloadaufgaben und CopySrv-Instanzen. Die „scalingStartegyëines Storeservice kann jederzeit durch die Methode **setStrategy** gewechselt werden.

In der **createCopySrvDeployment** Funktion wird, wie oben erwähnt, das Deployment definiert und über den Kuberbetes Api-Client vom Typ „ExtensionV1Api" ein Deployment erzeugt. Dabei werden alle nötigen Informationen in einem „ExtensionsV1beta1Deployment" Objekt gespeichert. In diesem Objekt ist der Name des Deployments, Name des Docker Image und Anzahl der Replika enthalten. Um die CopySrv im Kubernetes-Cluster zu finden, besitzt das Deployment-Objekt 2 Labels. Ein Label enthält den Namen der CopySrv, der als Parameter in der **reateCopySrvDeployment** Methode übergeben wird. Das zweite Label dient der Gruppierung und erhält den Namen der Serverlessfunktion ohne die Session-ID. Zusätzlich wird erkannt, ob es sich um die Serverlessfunktion für den „WebDAVStoreService" oder den „JupyterStoreService"handelt. Denn der CopySrv für den „JupyterHubStoreService"greift auf ein PVC zu, welches vom instanziierten Pod gemountet werden muss.

Nach dem der Download fertiggestellt ist, müssen alle der ID zugeordneten Pods und das Deployment entfernt werden. Dazu werden in einer Schleife die zuständigen CopySrvs mittels des in Kapitel 5.4 vorgestellten „GET:/taskDone" Request abgefragt, ob der Download beendet ist. Im Anschluss wird das Deployment inklusive aller CopySrv Instanzen mit der **killCopySrvDeployment(deploymentName)** Methode und der Eintrag für die Session in der der Hash-Map „podCopySrvTaks" entfernt . Kommt es nach dem Erstellen des Deployments zu einem Fehler, wird ebenfalls die **killCopySrvDeployment(deploymentName)** Methode aufgerufen und der Eintrag in der Has-Map entfernt. Sollte durch einen Absturz eines Storeservice herrenlose CopySrvs existieren, werden diese bei der Initialisierung des Storeservice bereits im Konstruktur durch die **killAllCopySrv** Methode entfernt. Falls solche Instanzen existieren, werden diese über das gruppierende Label ermittelt und entfernt. Durch diese Mechanismen wird sichergestellt, dass im System keine untätigen CopySrv Instanzen existieren.

5.4. Kommunikation

Wie in Abbildung 5.5 illustriert, kommuniziert der Storeservice direkt mit dem Pod, auf dem der Container mit dem CopySrv gestartet ist. Dabei entscheidet der Hauptservice selbst, mit welcher CopySrv er spricht, wie in Unterkapitel 5.4.1 beschrieben.

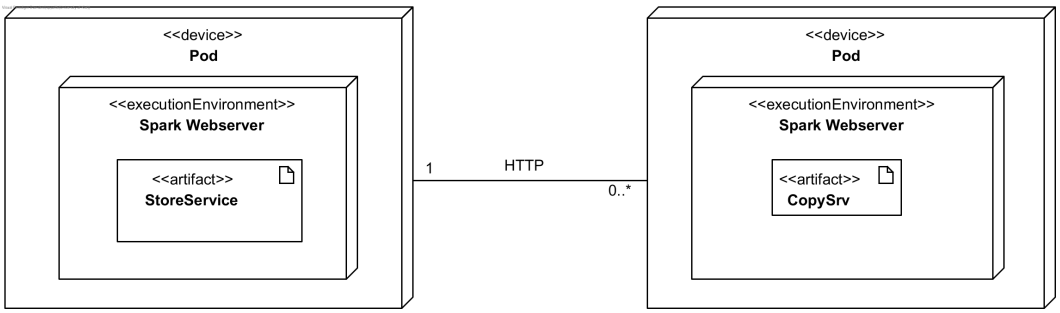

Abbildung 5.5. Kommunikation Storeservice und Serverlessfunktion

In den Folgenenden Unterkapiteln wird die Verteilung der Downloads in der **getCopy** und der anschließenden Übermitllung an die CopySrvs in der **copyFile** Methode, sowie der in dem Unterkapitel 5.2 erwähnten De/Serialisierung der JSON Objekte erläutert.

5.4.1. Verteilung der Downloads

Da kein Kubernetes-Service erstellt wird, der alle Instanzen des jeweiligen CopySrv als Ganzes zur Verfügung stellt, übernimmt der zuständige Storeservice die Lastenverteilung, indem er direkt auf die Pods zugreift. Dabei wird ein neuer Thread in der **getCopy** Methode des Storeservice nach dem Erstellen des Deployment erzeugt. In dem Prozess wird dann der Deploymentstatus abgefragt, die Downloadaufgaben in Sublisten aufgeteilt und an die **copyFile** Methode mit der IP-Adresse des zuständigen CopySrv asynchron in einer zweifach geschachtelten Schleife aufgerufen. Wobei, wie in Listing 5.2 dargestellt, der Index „i" der äußeren Schleife die IP des Pods aus dem Array PodIP ausgibt und in dem inneren Loop der Index „j" mit dem momentanen Wert von „i" initialisiert wird. Der Zähler „j" wird bei jedem Durchgang um die Menge der Pods inkrementiert. Dadurch wird eine gleichmäßige Aufteilung der Downloadelemente in den Sublisten erzeugt, wie in Listing 5.2 und Abbildung 5.6 illustriert. Die **copyFile** Methode sendet eine „POST" Anfrage an den CopySrv und wartet nicht auf eine Antwort, da der Downloadstatus in einer separaten „GET" Anfrage behandelt wird, siehe Sektion 5.3.2.

```
for (int i = 0; i < podSize; i++) {
        List<String> sublist = new ArrayList<String>();
        for (int j = i; j < elementsize; j += podSize) {
                        sublist.add(inputStreamUrl.get(j));       }
        boolean result = copyFile(creds, targetDir, sublist, podIP[i]);
  }
```

Listing 5.2. Asynchroner copyFile Aufruf

Zum besseren Verständnis der Lastenaufteilung in Sublisten ist in Abbildung 5.6 ein Beispiel dieser Prozedur mit 4 Pods dargestellt.

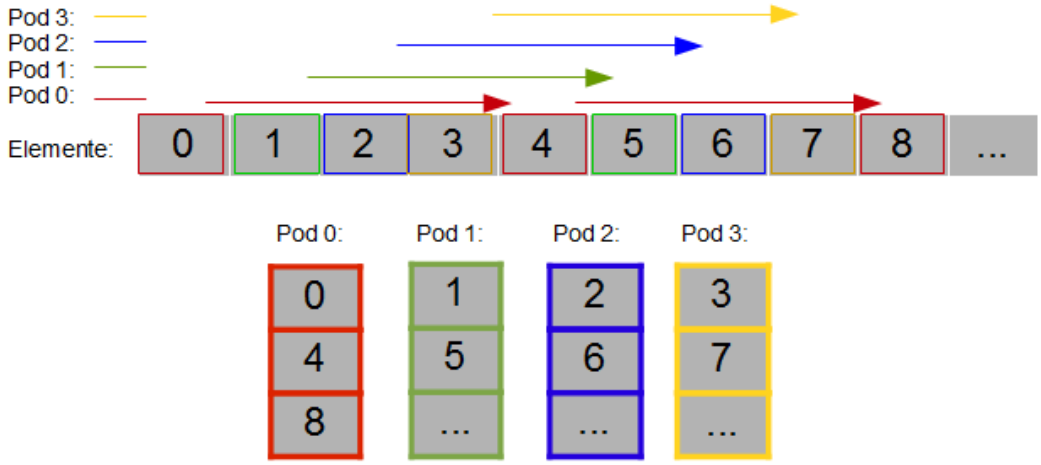

Abbildung 5.6. Konzept Lastenverteilung

5.4.2. Serialisierung und Deserialisierung mit Gson

Die Kommunikation zwischen dem Storeservice und der CopySrv findet über HTTP statt. In der **copyFile** Methode wird ein Webclient erzeugt, der dann eine POST Anfrage an die CopySrv sendet. Mit dem POST Request senden wir ein JSON-Objekt, das eine Map mit den Objekten creds, targetDir und inputStreamUrl enthält. Die beiden letzten Parameter sind vom Typ String und List<String>. Diese können ohne weiteren Aufwand mit einem Objekt vom Typ GSON serialisiert werden. Da der erste Eintrag creds vom Typ JupyterCredentials oder WebDAVCreds sein kann, wurde das Gson Objekt erweitert, so dass dieser die Credentialtypen De-/Serialisieren kann. Als Beispiel dient hierzu ein JSON, das vom WebDAVStoreService erzeugt wird.

```
{
 "cred":
        {"username":"user","password":"password","domain":"Domain"},
 "targetDir":"dir",
 "inputStreamUrl":["testURL1" ,"testURL"]
}
```

Listing 5.3. JSON Map das an den CopySrvWebDAV gesendet wird

Bei der Deserialisierung in der **PostCopy.handle** Methode wird das JSON aus dem Requestbody extrahiert. Wie in Listing 5.4 aufgezeigt, travesiert eine JsonParser-Instanz durch das Objekt. Dabei wird von der Wurzel zu den einzelnen Konten gesprungen, wo dann die Deserialisierung stattfindet.

```
public Object handle(Request request, Response response) throws Exception {
        JsonParser parser= new JsonParser();
        JsonElement rootNode= parser.parse(request.body());

        WebDAVCreds creds = gson.fromJson(rootNode.getAsJsonObject().get("cred"),
                                                                WebDAVCreds.class);
        String targetDir   = rootNode.getAsJsonObject().get("targetDir").getAsString();
        String[] inputStreamUrl = rootNode.getAsJsonObject().
                                                        get("inputStreamUrl").getAsString();
        for (String url : inputStreamUrl) {
                        inputStreamList.add(new ResearchDataInputStream(new URL(url)));
                ...
```

Listing 5.4. Deserialisierung CopySrvWebDAV

Kapitel 6

Evaluierung

In diesem Kapitel wird anhand von Testdaten die exportierte Downloadkomponente als Severlessfunktion mit der ursprünglichen Variante verglichen.

6.1. Ziele

Microservices und Serverlessfunktionen benötigen aufgrund ihrer verteilten Natur andere Teststrategien als monolitsche Systeme, da es viele Schnittstellen, wie das Erstellen und Löschen von Deployments oder der HTTP-Kommunikation zwischen den Services gibt. Aufgrund dieser Gegebenheiten werden folgende Ziele für die Evaluation definiert:

- Ziel 1: Machbarkeit der Exportierung der Downloadfunktion und der Interaktion zwischen Microservice und Serverlessfunktion.

- Ziel 2: Performancevorteil durch Skalierung der CopySrv für den jeweiligen Storeservice

6.2. Versuchsaufbau

Um die neue Implementierung der Storekomponente im GeRDI-Projekt zu testen, gibt es einen Test Cluster, dieser befindet sich im Leibniz-Rechenzentrum der Bayrischen Akademie der Wissenschaft (LRZ). Der Kubernetes Cluster besteht aus einem Master und 3 Minions. Zur weiteren Infrastruktur zählt ein NFS (Network File Storage), das als PVC für den Storeservice JupyterHub und die dazugehörge CopySrv dient. Der WebDAV Microservice greift über den CopySrv auf einen WebDAV-Speicher zu. Beide Speicherkomponenten befinden sich ebenfalls im LRZ. Da bei beiden CopySrv das Container-Image eine Größe von ungefähr 45 Megabyte hat, entsteht beim Deployen ein gewisser Overhead. Um zu ermitteln, wie sich der Initialisierungsaufwand des Deployments auf die Performance auswirkt, werden 12 Testdaten mit Dateigrößen zwischen 1 - 466 MB und einem Gesamtvolumen von 767 MB gewählt . Zustätzlich werden auch die Geschwindigkeitsunterschiede in den verschiedenen Skalierungsstrategien betrachtet.

6.3. Durchführung

Um den Deployprozess zu analysieren, werden die Logs des Kubernetesclusters ausgewertet und überprüft, ob die Soll-Instanzen des jeweiligen CopySrv mit den erstellten und gelöschten Instanzen übereinstimmen. Für die Ermittlung der Vorgangszeiten für das Deployen und des Downloads wird beim Aufruf der **getCopy** Methode und nach dem fertigen Deployment die Systemzeit in Millisekunden gespeichert. Erst nach dem die Verteilung der Downloads an die jeweiligen CopySrv vollzogen ist, werden die gespeicherten Zeiten in das Logging geschrieben. Die einzelnen CopySrv schreiben die aktuelle Systemzeit in den Log sobald der Download beendet ist. Aus den gesammelten Daten werden dann die Zeiten errechnet.

6.4. Ergebnisse und Diskussion

6.4.1. Deployment der CopySrv

Die Analyse der Protokolle ergibt, das das Deployen der CopySrv funktioniert und auch nach der Fertigstellung der Downloads oder bei Fehlern alle Instanzen und das Deployment wieder entfernt werden. Dazu werden die Logs nach der Nachricht: „Ende Kopiervorgang: (Zeitstempel in Millisekunden)" gefiltert. Diese Information wird von einer CopySrv erstellt, sobald der Downloadvorgang abgeschlossen ist. Dabei werden bei 12 Downloaddateien abhängig von der „ScalingStrategy" folgende Mengen von Pods erstellt:

- **Max4Scaler :** 3
- **Max4TaskForOne:** 4
- **OneForAll :** 1
- **OneForOne :** 12

Nach Durchlaufen sämtlicher Test, wurde über den Api-Client nach existierenden CopySrv gesucht und keine gefunden.

6.4.2. WebDAV Performance

In dieser Sektion werden die Ergebnisse der Tests grafisch dargestellt. In der Grafik 6.1 werden die errechneten Durchschnittswerte der Downloads in einem Balkendiagramm illustriert. Dabei unterteilt sich der Balken in die Zeit für das Deployment und den Downloadvorgang. In dem Diagramm 6.2 wird die Diskrepanz zwischen der schnellsten und der langsamsten Zeit für einen Downloadvorgang inklusive Deployment verglichen.

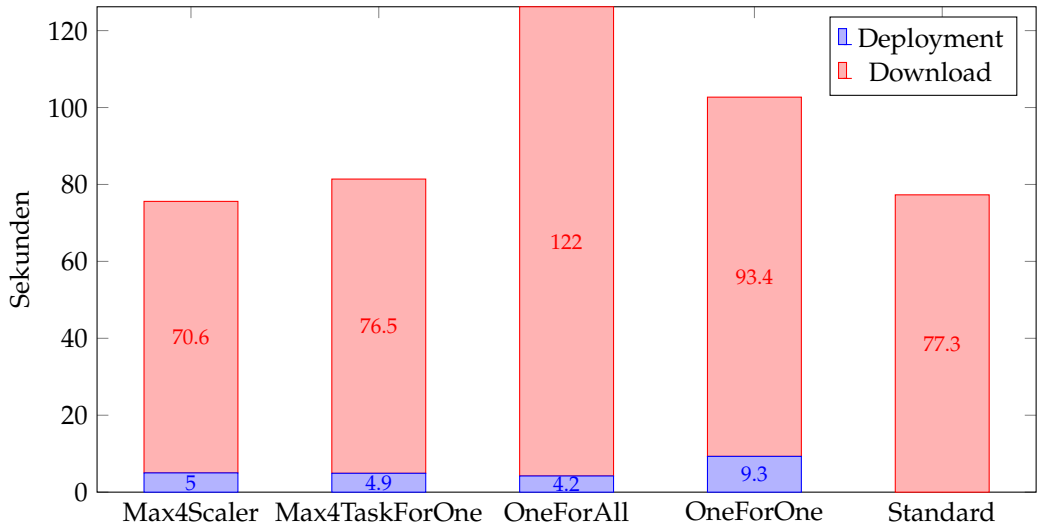

Abbildung 6.1. Download von 12 Dateien von 1 - 466 MB im Durchschnitt

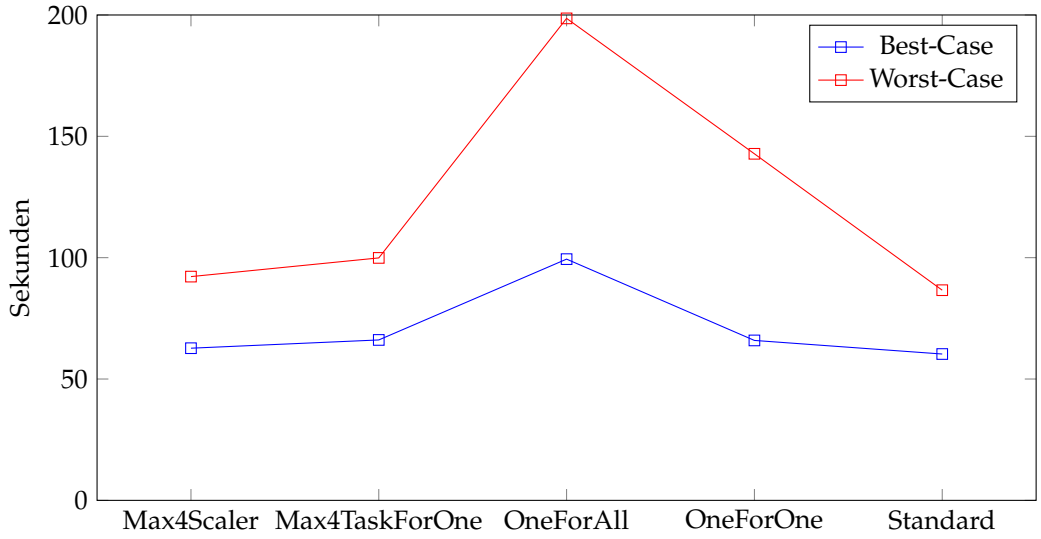

Abbildung 6.2. Best/Worst-Case Vergleich 12 Dateien von 1 - 466 MB

6.4.3. JupyterHub Performancve

Wie in dem Unterkapitel 6.4.2 werden auch die Durchschnittswerte für die JupyterHub CopySrv grafisch aufbereitet und die Stabilität der Downloadgeschwindigkeit in einem

Worst/Bes-Case vergleich aufbereitet.

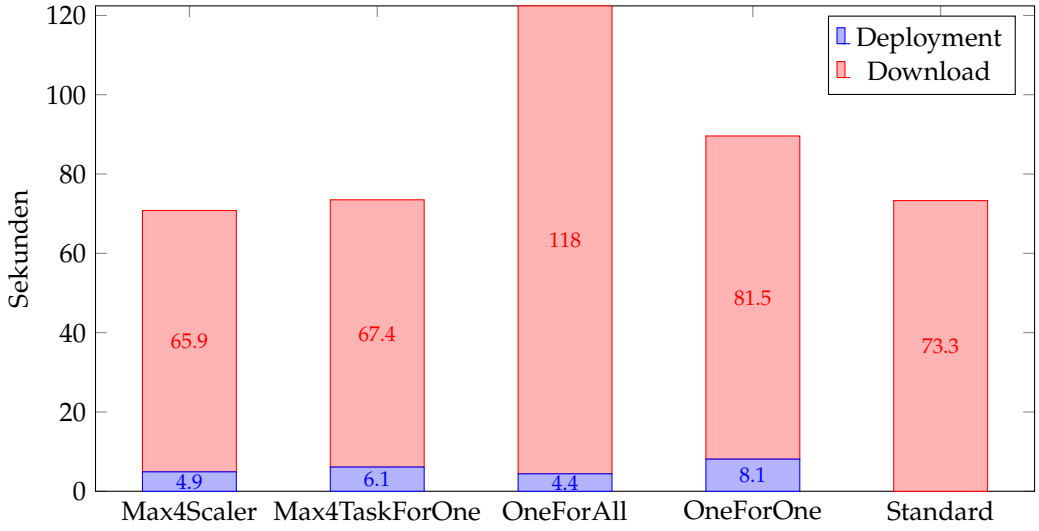

Abbildung 6.3. Download von 12 Dateien von 1 - 466 MB im Durchschnitt

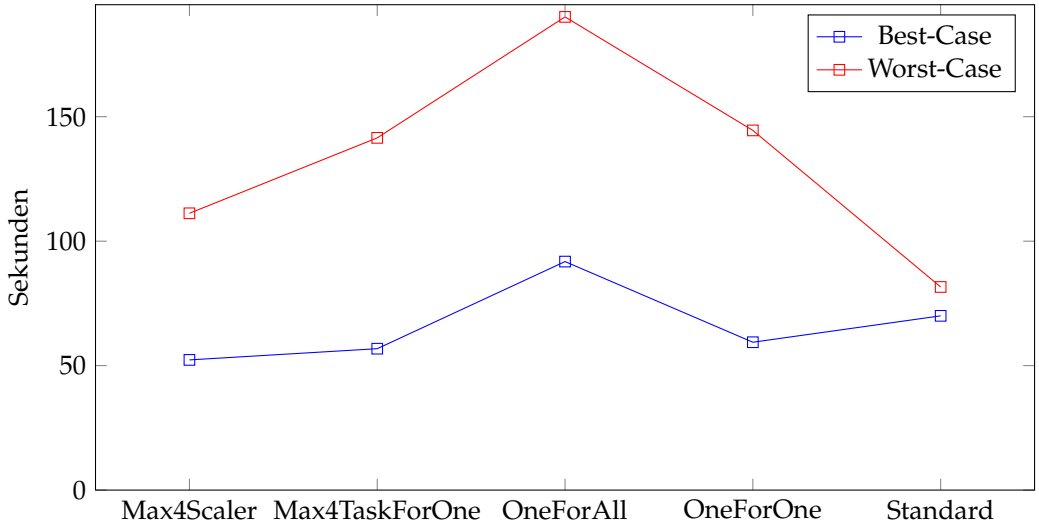

Abbildung 6.4. Best/Worst-Case Vergleich 12 Dateien von 1 - 466 MB

6.4.4. Diskussion

Die Serverlessimplementierung der Downloadfunktion entlastet den jeweiligen Storeservice, da in der ursprünglichen Implementierung der Downloadprozess in einem separaten Thread innerhalb des Storeservice arbeitet, der permanent Ressourcen in Anspruch nimmt. In der neugestalteten Version wird nach der Instanziierung des CopySrv ein neuer Thread gestartet, eine http-Anfrage mit den Downloadaufgaben versendet und dann vom System schlafen gelegt, so dass er keine Rechenlast in Anspruch nimmt. Der Prozess wird lediglich zum Abfragen des Gesamtfortschritts geweckt, der dann die CopySrv löscht, um die Ressource wieder frei zu geben. Da das Instanziieren und Beenden einer CopySrv durch Anfragen an die Kubernetes-Api geschieht, wird der Storeservice ebenfalls nur in einem geringen Maß beansprucht. Dadurch kann ein Storeservice ein höheres Aufkommen von Downloadanfragen bewältigen.
Nachteilig wirkt sich der Overhead des Deployen der CopySrvs und der zusätzliche Kommunikationsaufwand, wie das De/Serialisieren der JSON-Objekte und das Versenden an die Downloadservices, auf die Performance aus.

Weiter muss die „ScalingStrategy" des Storeservice auf die Gegebenheiten des Kubernetes-Cluster angepasst werden. Es sollte vermieden werden, dass ein einzelner Worker mit zu vielen CopySrv Instanzen belastet wird und dadurch mehr mit dem Verwalten der Ressourcen als mit dem eigentlichen Downloadprozess belastet wird. Wie in den Grafiken 6.1 und 6.3 zu erkennen, wird bei maximaler Skalierung kein weiterer Speedup im Vergleich zu ausgewogenen Strategien wie „Max4Scaler" oder „Max4TaskForOne" erzeugt. Deutlicher wird der Scheduling-Aufwand bei „OneForOne" in den Best/Worst-Case Diagrammen 6.2 und 6.4, wo die Differenz bei ca. 59% liegt. Aus diesem Grund sollte die „OneForOne" Strategie mit Bedacht eingesetzt werden.

Eine geringe Skalierung wie bei „OneForAll" hat jedoch den Nachteil, dass ein langsamer Download alle weiteren blockiert und damit, wie in der JupyterHub Performance-Anlayse Abbildung 6.3 dargestellt, nachteilig auf den gesamten Downloadvorgang auswirkt.
Den maximalen Benefit liefern die ausgewogenen Skalierungen „Max4Scaler" und „Max4TaskForOne" im Bereich Geschwindigkeit und Konsistenz. Die Anzahl der erzeugten Instanzen liegt beim erst genannten bei 3 und bei der 2ten Stratgie bei 4. Da das Test-System aus 3 Workern besteht, kann in diesem Test bei „Max4Scaler" jede CopySrv einem zugewiesen werden. Dadurch werden die negativen Effekte durch vereinzelte langsame Downloads und das Ressourcen-Mangement möglichst gering gehalten.

6.4.5. Bedrohungen der Validität

Das Ergebnis der Performance hängt stark von den Gegebenheiten ab. Beim WebDAV Storeservice könnte als Ziel ein WebDAV eingebunden sein, hinter dem ein Speicher mit nur sehr geringer Bandbreite arbeitet. In diesem Fall würden sich die Skalierungseffekte nicht bemerkbar machen. Eine weitere Einschränkung für beide Storeservices bilden die Downloadgeschwindigkeiten der externen Quellen. Diese können aufgrund von Technik oder durch die Frequentierung anderer Teilnehmer stark variieren.

Eine weitere Einschränkung bilden externe Anbieter eines WebDAV. Denn diese können Uploads in den Speicher mit unbekannter Dateigröße verbieten. Da der Download als Stream von einer Quelle zu einem Ziel umgesetzt ist und die Speichergröße von diesem oft nicht mitgeliefert wird, kann es zu Fehlern kommen. Bei den Anbietern „NextCloud" und „OpernDrive" werden bei diesen Downloads nur 0 Byte große Dateien erzeugt.

Darüber hinaus bildet die Struktur des Kubernetes-Cluster nicht unerhebliche Limitierungen. Angenommen, es steht nur ein einziger Worker zur Verfügung, dann werden sämtliche durch das Deployment erzeugten Pods auf dieser einen Maschine ausgeführt. In diesem Fall erledigt ein Server alle Downloadaufgaben und es gäbe keine wirksame Skalierung.

Kapitel 7

Verwandte Arbeiten

In diesem Kapitel werden ähnliche Arbeiten im Bereich Skalierung mittels Serverlessfunktionen präsentiert.

7.1. Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions

In dieser Arbeit werden ebenfalls Serverlessfunktionen und Faas auf ihre Anwendbarkeit auf datenintensivere Operation betrachtet. Zusätzlich werden dabei verschiedenste serverbasierte Architekturen für die Ausführung wissenschaftlicher Workflows verglichen, siehe Abbildung 7.1.

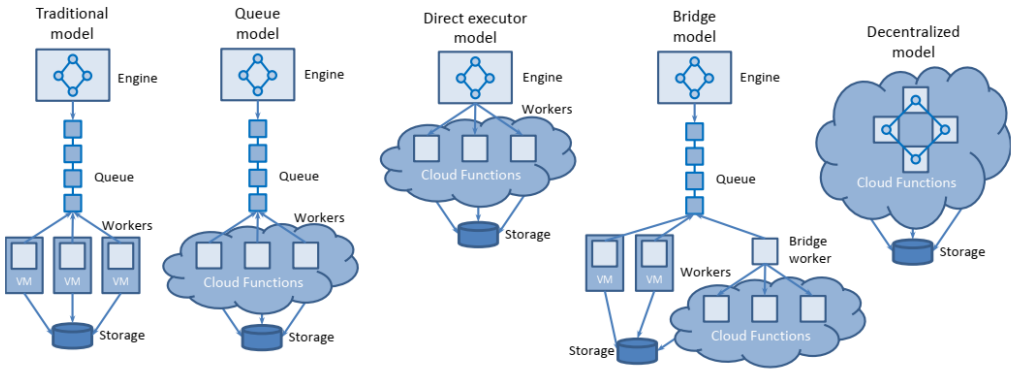

Abbildung 7.1. serverless architectures for execution of scientific workflows [Mal+17, Seite 5]

Dabei werden die zu speichernden Daten je nach Architektur über eine Engine unterschiedlich verteilt. Zum Vergleich, geschieht die Verteilung der Downloads in der Neuimplemtierung des Storeservice im GERDI-Projekt in der getCopy-Methode, wo über das „ScalingStrategy" Objetk festgelegt wird, wie viele Instanzen der CopySrv existieren, die dann eine zugeteilte Liste von Downloads ausführen.

7.2. FaaSter, Better, Cheaper: The Prospect ofServerless Scientific Computing and HPC

In Josef Spillner, Cristian Mateos, and David A. Monge Veröffentlichung wird das FaaS-Modell mit herkömmlichen monolithischen Algorithmusausführungen verglichen. Dabei werden rechenintensive Prozesse, wie die Gesichterkennung oder das Entschlüsseln von Passwörtern als Performancetest genutzt. Dabei wurden Passwörter der Länge 3 in verschiedenen Parallelisierungsgraden bis zu 10 Workern und Mappern bearbeitet. Ähnlich wie in dieser Arbeit werden hier die Ausführungszeiten von lokaler und verteilter Parallelisierung verglichen und wie sich der Grad der Skalierung auf die Performance auswirkt, siehe Grafik 7.2.

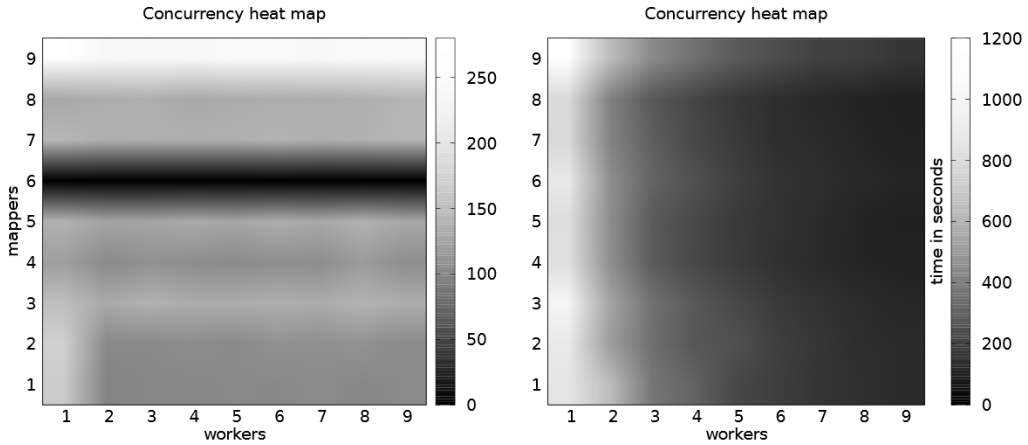

Abbildung 7.2. Entschlüsseln von 100 Passwörtern durch einen Double-Map-Reduktionsprozess mit Kombinationen von Arbeitern und Mappern; links: lokale parallele Berechnung; rechts: Parallele verteilte Verarbeitung. [SMM17]

Anstelle von Downloads werden hier hier Rechenintensieve Operationen mit parallelen und verteilten Ausführungen verglichen.

Kapitel 8

Fazit und Ausblick

8.1. Fazit

Bei der Serverlessimplementierung, mit der „Max4Scaler" Strategie bei der 3 Instanzen einer CopySrv erstellt werden, kann ein Speedup im Vergleich mit der Standardversion erzielt werden. Andere Strategien mit höherer Skalierung hingegen erzeugen keinen Vorteil in der Downloadgeschwindigkeit, weil zu viele Pods, die eine CopySrv ausführen, auf eine zu geringe Anzahl von Nodes verteilt werden. Aufgrund der in dem Unterkapitel 6.4.4 gewonnen Erkenntnisse, bietet sich die ausgewogene und auch begrenzte Skalierung „Max4Scaler" an, weil sie die Testdaten konstant am schnellsten herunterladen kann. Weiter würde sich auch die Strategie „OneForAll" anbieten. Wenn mehrere Clients zeitgleich einen Download ausführen, hat diese Strategie den geringsten Ressourcenverbrauch im Kubernetes Cluster und entlastet ebenfalls den Storeservice. Jedoch könnten dann vereinzelte Downloads den gesamten Vorgang ausbremsen und zu unangenehmen Wartezeiten führen.

8.2. Ausblick

Für zukünftige Arbeiten können folgende Aspekte weiterbehandelt werden.

8.2.1. Entfernen der Zustände in den Storeservices

Um den Jupyterhub oder WebDAV Storeservice skalieren zu können, müssen diese zustandlos sein. Da beim Login in den jeweiligen Storservice eine Session generiert wird, in der die Information für den Downloadvorgang gespeichert wird, enthält der Pod, in dem ein Storeservice arbeitet, zustandsgebundene Informationen. Des weiteren wird beim Aufrufen der **getCopy** Methode ein weiterer Zustand erzeugt, in dem eine Liste von IP-Adressen für eine Session gespeichert wird, mit denen die CopySrvs erreicht werden. Ein Ansatz wäre, einen Storemaster zu entwickeln, der einen Überblick hat, welche Session zu welchem Pod gehört, auf dem eine Instanz eines Storeservices arbeitet und die Anfragen an diesen weiterleitet. Eine andere Alternative wäre, dass sich die Storeservice untereinander austauschen, und sobald ein neuer Zustand erzeugt wurde, er diesen allen anderen mitteilt.

8.2.2. Zukünftige Tests

Eine weitere Besonderheit ist bei weiteren Test entstanden, bei denen 12 Testdaten mit einem Gesamtvolumen von 25,4 GB eine höhrer Last erzeugen sollen . Dazu wurden verschiedene Linuxdistributionen von unterschiedlichen Servern ausgewählt und heruntergeladen. Ein wiederholtes Ausführen dieser Test führt zu einer Sperrung auf den Servern, so dass die Downloads nicht mehr ausgeführt werden können. Um für zukünftige Tests, die eine höhere Last als in dieser Evaluation erzeugen, sollten eigene Server gemietet werden, auf denen die Testdaten gelagert sind, um eine Sperrung zu umgehen.

8.2.3. Vom Faas zur reinen Serverlessanwendung

In diesem Projekt werden die Downloadfunktionen in einem Container exportiert, was dem Prinzip von Faas entspricht. Doch es gibt Open Source Frameworks für Kubernetes wie Kubeless, OpenWhisk und OpenFaaS, mit denen Funktionen ohne Container verteilt werden können. Das hat zum Vorteil, dass nur noch der reine Code in die Serverless-Plattform exportiert werden muss und somit der durch den Container entstandene Overhead wegfällt. Die Ausführung und Skalierung wird dabei komplett an das Serverlessframework abgegeben. Eine weitere Möglichkeit besteht in der Auslagerung der Infrastruktur an einen externen Anbieter, durch die Nutzung von „AWS Lambda". Diese von Amazon gestellte Plattform erlaubt das Erstellen und Ausführen von Serverlessfunktionen , dabei muss nur für die genutzte Datenverarbeitungszeit gezahlt werden. [MPD+18],[Vil+16]

8.2.4. Google Clouddrive

Ein weiterer Ansatz zur Persistierung wäre die Integration von Cloudspeichern wie GoogleDrive. Für den Zugriff auf den Speicher bietet Google über die „Cloud Storage Client Libraries" eine Java Implementierung. Auf dieser Basis kann ein weiterer Storeservice implementiert werden, der sich ebenfalls vom **AbstractStoreservice** ableitet und mit einer eigens entwickelten CopySrv die Downloads durchführt. Weiter unterstützt dieser Cloudspeicher das Ausführen von parallelen Uploads . Dadurch können parallele Speicherzugriffe von den Downloadservices ausgeführt werden, was weitere Performancevorteile bietet. Ein weiterer Ansatz wäre die Implementierung ohne CopySrv, indem das Skalieren des Speichervorgangs an den CloudStorage abgegeben wird. Dabei wird ein Objekt in mehrere Teile aufgeteilt und diese gleichzeitig in einen temporären Speicherort hochgeladen. Anschließend wird mit dem Befehl **compose** das ursprüngliche Objekt aus den Teilen,wie Listing 8.1 dargestellt , wieder zusammengesetzt.

```
POST /storage/v1/b/example-bucket/o/composite-object/compose
Host: www.googleapis.com
Content-Length: 216
Content-Type: application/json
Authorization: Bearer ya29.AHES6ZRVmB7fkLtd1XTmq6mo0S1wqZZi3-Lh_s-6Uw7p8vtgSwg

{
  "sourceObjects": [
    {
      "name": "component-obj-1"
    },
    { "name": "component-obj-2"
    },
    { "name": "component-obj-3"
    }
  ],
  "destination": {
   "contentType": "application/octet-stream"
 }
}
```

Listing 8.1. Objekte mit der JSON-API zusammensetzen [Goob]

Dieser Prozess birgt jedoch erhebliche Risiken in der Konsistenz der Daten, indem ein anderer Prozess versehentlich den Namen eines oder mehrerer der bereits hochgeladenen temporären Teile verwendet, wodurch in der **compose**-Phase eine inkorrekte Zusammensetzung entsteht. Um diese Art von Fehlern in dem späteren Gefüge zu vermeiden, können für die Objekte, die die Downloadaten enthalten, Meta-Generierungsnummern erzeugt werden, so dass keine Verwechslungen beim Erstellen mehr auftreten. Doch diese Generierung hat zum Nachteil, dass die Gesamtvorgangslatenz verdoppeln werden kann und mit 0,004 $ pro 10.000 Vorgänge kostenpflichtig berechnet wird. [Gooa],[Goob],[Gooc]

Literaturverzeichnis

[19] CNCF Serverless WG. Contribute to cncf/wg-serverless development by creating an account on GitHub. original-date: 2017-04-28T22:37:05Z. Mai 2019. URL: https://github.com/cncf/wg-serverless (besucht am 09. 05. 2019) (siehe Seite 6).

[Aug17] Stephan Augsten. Was sind Docker-Container? Apr. 2017. URL: https://www.dev-insider.de/was-sind-docker-container-a-597762/ (siehe Seite 7).

[AUTa] KUBERNETES AUTHORS. Horizontal Pod Autoscaler. en. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (besucht am 08. 05. 2019) (siehe Seite 3).

[AUTb] KUBERNETES AUTHORS. Kubernetes Components. en. URL: https://kubernetes.io/docs/concepts/overview/components/ (besucht am 08. 05. 2019) (siehe Seiten 8, 9).

[AUTc] KUBERNETES AUTHORS. Pods. en. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/ (besucht am 08. 05. 2019) (siehe Seite 9).

[AUTd] KUBERNETES AUTHORS. Services. en. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (besucht am 08. 05. 2019) (siehe Seiten 9, 11).

[AUTe] KUBERNETES AUTHORS. Using a Service to Expose Your App. en. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/ (besucht am 09. 05. 2019) (siehe Seite 11).

[AUTf] KUBERNETES AUTHORS. Using kubectl to Create a Deployment. en. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/ (besucht am 09. 05. 2019) (siehe Seiten 10, 11).

[AUTg] KUBERNETES AUTHORS. Using Minikube to Create a Cluster. en. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/ (besucht am 08. 05. 2019) (siehe Seite 8).

[AUTh] KUBERNETES AUTHORS. Viewing Pods and Nodes. en. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ (besucht am 08. 05. 2019) (siehe Seite 9).

[AUTi] KUBERNETES AUTHORS. What is Kubernetes. en. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (besucht am 08. 05. 2019) (siehe Seite 7).

[Ber] Tim Berners-Lee. HTTP: A protocol for networked information. Englisch. Internet Draft working document. URL: https://www.w3.org/Protocols/HTTP/HTTP2.html (besucht am 04. 08. 2019) (siehe Seite 12).

[Bry17] Daniel Bryant. Deploying Java Applications with Docker and Kubernetes. en. Okt. 2017. URL: https://www.oreilly.com/ideas/how-to-manage-docker-containers-in-kubernetes-with-java (besucht am 08.05.2019) (siehe Seite 9).

[Daz] Michael Dazer. „‚RESTful APIs - Eine Übersicht‘“. dejournal+issuetitle, Seite 8 (siehe Seite 12).

[E J97] Jr. E. James Whitehead. Lessons from WebDAV for the Next Generation Web Infrastructure. 1997. URL: https://www.ics.uci.edu/~ejw/http-future/whitehead/http_pos_paper.html (besucht am 30.07.2019) (siehe Seite 12).

[Fox+17] Geoffrey C. Fox u. a. „‚Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research‘“journal+issuetitle. URL: http://arxiv.org/abs/1708.08028 (siehe Seiten 1, 6).

[Gooa] Google. Cloud Storage Client Libraries | Cloud Storage. en. URL: https://cloud.google.com/storage/docs/reference/libraries (besucht am 07.08.2019) (siehe Seite 37).

[Goob] Google. Zusammengesetzte Objekte und parallele Uploads | Cloud Storage. de. URL: https://cloud.google.com/storage/docs/composite-objects?hl=de (besucht am 13.08.2019) (siehe Seite 37).

[Gooc] Goolge. Generierungsnummern und Vorbedingungen | Cloud Storage. de. URL: https://cloud.google.com/storage/docs/generations-preconditions?hl=de (besucht am 13.08.2019) (siehe Seite 37).

[Gru+] Richard Grunzke u. a. „‚Challenges in Creating a Sustainable Generic Research Data Infrastructure‘“. enjournal+issuetitle, Seite 4 (siehe Seite 5).

[Hec] Felix Heck. „‚Einsatz von Microservices in Multi-Cloud-Systemen‘“. dejournal+issuetitle, Seite 13 (siehe Seite 1).

[IEE+18] IEEE International Conference on Cloud Computing Technology and Science u. a. IEEE 10th International Conference on Cloud Computing Technology and Science: proceedings : 10–13 December 2018, Nicosia, Cyprus. English. OCLC: 1086472191. 2018. URL: https://ieeexplore.ieee.org/servlet/opac?punumber=8590843 (besucht am 05.05.2019) (siehe Seite 6).

[Inc19] Incloud. Kubernetes - Alles, was Sie darüber wissen müssen. Incloud, Feb. 2019 (siehe Seite 7).

[Ism+15] B. I. Ismail u. a. „‚Evaluation of Docker as Edge computing platform‘“. *2015 IEEE Conference on Open Systems (ICOS)*. Aug. 2015, Seiten 130–135 (siehe Seite 7).

[Luk18] Marko Luka. Kubernetes in Action. Carl Hanser Verlag GmbH & Co. KG, Juli 2018. URL: https://www.hanser-elibrary.com/doi/book/10.3139/9783446456020 (besucht am 08.05.2019) (siehe Seiten 1, 6, 8–11).

[Mal+17] Maciej Malawski u. a. „‚Serverless execution of scientific workflows: experiments with hyperflow, AWS lambda and Google cloud functions‘“journal+issuetitle (siehe Seite 33).

[MPD+18] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco u. a. „‚An Evaluation of Open Source Serverless Computing Frameworks.‘“ *CloudCom*. 2018, Seiten 115–120 (siehe Seite 36).

[Nur+] Nurzhan Nurseitov u. a. „‚Comparison of JSON and XML Data Interchange Formats: A Case Study‘“. enjournal+issuetitle, Seite 6 (siehe Seite 12).

[pro] GeRDI project. GeRDI – Generic Research Data Infrastructure – The GeRDI project is a three-year research project funded by the DFG to develop a prototype for generic research data infrastructure in Germany enabling multidisciplinary and FAIR research data management. en-US. URL: https://www.gerdi-project.eu/ (besucht am 16. 05. 2019) (siehe Seite 5).

[SMM17] Josef Spillner, Cristian Mateos und David A Monge. „‚Faaster, better, cheaper: The prospect of serverless scientific computing and hpc‘“. *Latin American High Performance Computing Conference*. Springer. 2017, Seiten 154–168 (siehe Seite 34).

[Sou+] Nelson Tavares de Sousa u. a. „‚Designing a Generic Research Data Infrastructure Architecture with Continuous Software Engineering‘“. enjournal+issuetitle, Seite 4 (siehe Seite 5).

[Vil+16] Mario Villamizar u. a. „‚Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures‘“. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, Seiten 179–182 (siehe Seite 36).

[Zin] Mathias Zinnen. „‚Design und Implementierung einer RESTful API für heterogene Daten‘“. enjournal+issuetitle, Seite 85 (siehe Seite 12).