DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR

Grosse Studienarbeit

des Studiengangs Informationstechnik

der Dualen Hochschule Baden-Württemberg Mannheim

Performance Analysis of serverless cloud functions

Sebastian Wallat

3. Mai 2021

Bearbeitungszeitraum:	07.12.2020-05.05.2021
Matrikelnummer, Kurs:	1708267, TINF18-IT1
Betreuer:	Dr. Frank Schulz

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit mit dem Titel "Performance Analysis of serverless cloud functions" selbständig verfasst habe, dass ich sie zuvor an keiner anderen Hochschule und in keinem anderen Studiengang als Prüfungsleistung eingereicht habe und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht.

Datum, Ort

Unterschrift

Zusammenfassung

Performance Analysis of serverless cloud functions
Serverlose Architekturen gewinnen im Bereich des Cloud-Computing vermehrt an Bedeutung. Ein Vertreter hierfür sind serverlose Cloudfunktionen die mittlerweile von allen großen Cloud Service Providern (CSP) angeboten werden. Ziel dieser Arbeit soll es sein serverlose Cloudfunktionen auf ihre Performanz zu untersuchen.
Dafür soll zuerst geklärt werden worum es sich bei serverlosen Funktionen handelt, sowie ihre Vor-und Nachteile kurz erläutert werden. Die Performanz-Analyse soll am Beispiel von AWS Lambda erfolgen.
Zur Analyse sollen vier Test-Funktionen implementiert werden und auf verschiedene Eigenschaften, wie z.B. Latenz untersucht werden. Besonders die Coldstart-Problematik von serverlosen Systemen soll hierbei näher erläutert werden.
Zum testen der Funktionen wird eine selbst einwickelte NodeJS Anwendung verwendet, welche die Funktionen ansteuert und die Ergebnisse Protokolliert. Zur Auswertung der so gesammelten Daten wird eine Jupyter-Notebook entwickelt, welches die gängigen Data-Science Bibliotheken wie Pandas und Matplotlib verwendet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

SaaS **S**oftware-**a**s-**a**-**S**ervice

CAD **C**omputer-**A**ided **D**esign

CSP **C**loud **S**ervice **P**rovider

AWS **A**mazon **W**eb **S**ervices

SLA **S**ervice **L**evel **A**greements

CSV **C**omma-**s**eparated **v**alues

1 Einführung

Durch die stetig voranschreitende Digitalisierung von Unternehmen gewinnen im Besonderen Cloud-Basierte Softwarelösungen an Bedeutung. Hierzu zählen unter anderem Software-as-a-Service (SaaS) Angebote wie Office356 von Microsoft (Textverarbeitung) oder Fusion360 von Autodesk (CAD-Programm). Des weiteren sind viele der heutigen Web-Anwendungen in der Cloud gehostet. Alle diese Anwendungen haben gemein, dass die wesentliche Datenverarbeitung nicht auf dem Gerät des Nutzers erfolgt, sondern auf den Servern des Cloud Service Provider (CSP). Hierfür bieten die meisten CSP Baukastensysteme an um schnell komplexe Backend-Strukturen innerhalb ihrer Umgebung implementieren zu können. Einige Bespiele hierfür sind Datenbanksysteme, API-Gateways, Serverlose-Funktionen, (Container)-Hosting und weitere.
Das Ziel dieser Arbeit ist es sich mit einem dieser Systeme näher zu beschäftigen, den Serverlosen Cloudfunktionen. Diese bieten die meisten CSP in ihrer Infrastruktur an, z.B AWS-Lambda (Amazon Web Services) und Google Cloud Run.
Cloudfunktionen sind eine kostengünstige Lösung um bestimmte Backend Funktionalitäten zur Verfügung zu stellen. Sie sind zustandslose (stateless) Programmstücke, die über eine definierte Aktion aktiviert werden, z.B einen API Aufruf. Zur Ausführung erstellt der CSP einen neue Instanz der Funktion (falls nicht schon vorhanden) und führt diese aus. Die Besonderheit hierbei ist die zustandslösigkeit, also die Eigenschaft der Funktion keine persistenten Daten zu speichern. Alle für die Ausführung benötigten Daten, müssen vom Aufrufer mitgegeben werden, oder aus einer persistenten Datenquelle (z.b Datenbank) geladen werden. Da die Funktion aus diesem Grund nicht auf einem normalen Webserver läuft, sondern nur führ einen Aufruf existiert, werden sie auch als serverlos bezeichnet. [1]
Der größte Vorteil von Cloudfunktionen gegenüber einem herkömmlichen Webserver, liegt darin, dass nur pro Aufruf kosten einstehen. Da ein normaler Webserver ja durchgängig Ressourcen des CSP belegt, fallen für ihn auch kosten an, wenn er nicht benutzt wird.
Aus diesem Verhalten ergibt sich direkt auch ein wesentlicher Nachteil, der allerdings die meisten Cloud-Services betrifft, dieser liegt in der Varianz der Kosten. Erfreut sich ein Service unerwarteter Beliebtheit, so können unerwartet hohe Kosten durch den vermehrten Aufruf der Funktion entstehen.
Dies wird besonders durch die gute horizontale Skalierbarkeit der Cloudfunktionen verstärkt. Würde ein traditioneller Webserver durch zu viele Anfragen überlastet, kann man von einer Funktion beliebig viele Instanzen erzeugen, welche die Anfragen parallel verarbeiten können.

Im weiteren Verlauf der Arbeit kann aus mehreren Gründen nur auf einen CSP eingegangen werden, hierfür wurde AWS ausgewählt. AWS stellt Cloudfunktionen unter seinem AWS-Lambda Service bereit. Daher sind die folgenden Ausführungen nur in der AWS Umgebung gültig und die präsentierten Resultate gelten nur für AWS-Lambda Funktionen.

2 Technischer Hintergrund (AWS Lambda)

AWS beschreibt Lambda folgendermaßen: "AWS Lambda ist ein serverloser Datenverarbeitungsservice, der Ihren Code beim Eintreten bestimmter Ereignisse ausführt und für Sie automatisch die zugrunde liegenden Datenverarbeitungsressourcen verwaltet" [1] Dies lässt sich an einem kleinen Beispiel verdeutlichen:

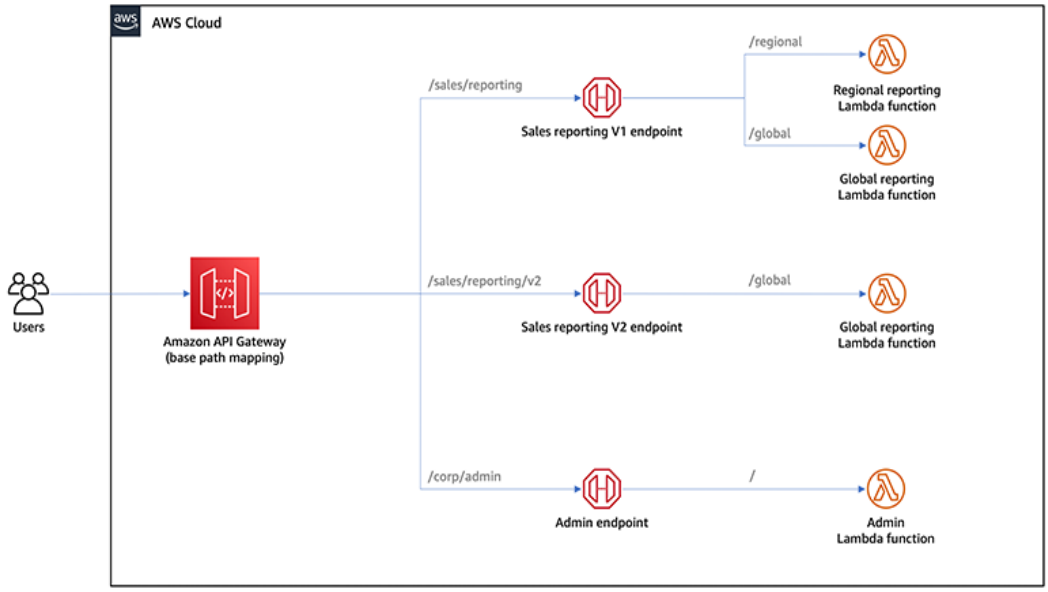

Abbildung 2.1: Beispiel für den Einsatz von Lambda Funktionen mit einem API-Gateway als Auslöser
Quelle: https://aws.amazon.com/de/blogs/compute/icymi-serverless-q1-2021/

Grafik 2.1 zeigt die Verwendung von Lambda Funktionen in Kombination mit einem AWS API Gateway. Der Gateway besitzt mehrere vordefinierte Endpunkte, die jeweils mit einer Lambda-Funktion verknüpft sind. Schickt ein Nutzer eine HTTP Anfrage mit einem bestimmten Pfad (z.B "/corp/admin") an den die URL des API Gateway, so extrahiert dieser den konkreten Pfad der Anfrage und startet die verknüpfte Lambda-Funktion (in diesem Fall die "Admin" Funktion) mit den mitgelieferten Parametern.

3 Performance Tests

Ziel soll es sein AWS Lambda Funktionen auf ihre Performanz zu untersuchen. Als Kriterien hierfür bieten sich Latenz und Verfügbarkeit an.
In den Service Level Agreements (SLA) von AWS Lambda garantiert AWS eine Verfügbarkeit von mindestens 99.95% pro Monat. Sollte dies nicht erfüllt werden, so bekommt der Kunde eine Gutschrift für zukünftige Lambda Kosten, in Form sogenannter "Service Credits". Ab einer Verfügbarkeit von unter 95% werden die gesamten Kosten als Service Credits gutgeschrieben. [2]
Weitere Informationen hierzu findet man in der offiziellen SLA von AWS Lambda (`https://aws.amazon.com/de/lambda/sla/`)
Da die Angaben der SLA dafür sprechen dass AWS von einer hohen Verfügbarkeit ausgeht und dies bei einem weltweit führenden CSP durchaus erwartet werden kann, soll die Verfügbarkeit nicht Fokus der weiteren Untersuchungen sein.
Da die Lambda SLA keine Angaben zur Latenz machen, soll diese näher untersucht werden. Besonders die Problematik des sogenannten "Cold start" (zu deutsch Kaltstart) soll untersucht werden.

3.1 Cold start (Kaltstart) von serverlosen Architekturen

Aufgrund der serverlosen Architektur von AWS Lambda liegt die Vermutung nah, dass er Kaltstart wesentlichen Anteil an der Latenz einer Lambda Funktion haben könnte. Daher soll zuerst geklärt werden worin genau die Kaltstart-Problematik besteht.
Da Lambda Funktionen keinen permanenten Server Prozess besitzen, sondern Instanz basiert sind, existieren zu einer bestimmten Zeit t eine bestimmte Anzahl von Instanzen n der Funktion. Wie groß n zu einem konkreten Zeitpunkt ist, wird von Skalierungsalgorithmen bestimmt und hängt in erster Linie von der Anzahl der Aufrufe in einem bestimmten Zeitfenster Δt ab. (vgl. Autoscaling von AWS Lambda [1]).
Um Ressourcen zu sparen wird die Anzahl der Funktions-Instanzen auf null reduziert, falls in einem festgelegten Intervall keine Aufrufe der Funktion stattfinden.
Trifft nun ein neuer Aufruf für die Funktion ein, so muss der CSP zuerst eine neue Instanz erstellen, welche dann die Anfrage bearbeiten kann. Dies wird als Kaltstart bezeichnet (Skalierung von $n = 0$ auf $n \geq 1$)
Die Gesamtzeit des (oder der) ersten Funktionsaufrufe(s) erhöht sich somit um die Zeit, die zur Instanziierung der Funktion benötigt wird. Je nach verwendeter Programmiersprache kann es zusätzlich passieren dass das Programm deutlich langsamer Ausgeführt

wird als nach einiger Laufzeit (z.B JIT Laufzeitoptimierung von Java [3]). In diesem Fall vergrößert sich die Laufzeit nach Kaltstart zusätzlich.

3.2 Testaufbau

Um verschiedene Performance-Aspekte von AWS Lambda Funktionen zu untersuchen wurde der im folgenden näher beschriebene Testaufbau verwendet.

3.2.1 AWS

Der Testaufbau besteht aus 4 in AWS implementierten Lambda-Funktionen. Diese sind über einen API Gateway erreichbar.

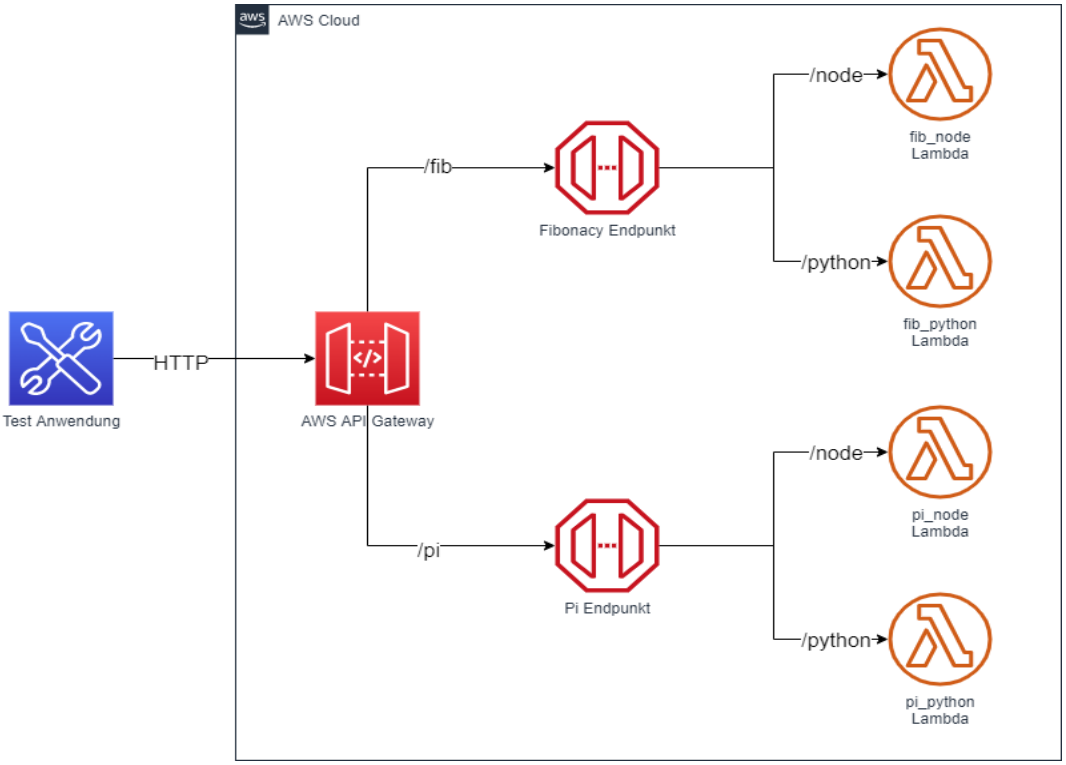

Abbildung 3.1: Überblick über die in AWS implementierten Funktionen
Quelle: Eigene Darstellung mit DrawIO

Darstellung 3.1 zeigt eine Überblickskizze über die in der Cloud implementierten Funktionalitäten. Insgesamt wurden vier Lambda-Funktionen implementiert, hierbei wurden zwei verschiedene Programmiersprachen und zwei Arten von Workloads berücksichtigt.

Als verwendete Sprachen werden Python und Java Script (NodeJS-Framework) verwendet. Bei den implementierten Workloads handelt es sich einmal um die iterative Berechnung von π auf eine festgelegte Anzahl stellen, sowie um die rekursive Berechnung einer Zahl der Fibonacci folge. Der Sourcecode kann unter folgendem Link eingesehen werden: `https://github.com/SebastianWallat/StudienarbeitAWS`
Des weiteren dokumentiert jede Lambda Funktion ihre Laufzeit und sendet diese, sowie das Ergebnis der Berechnung an den Aufrufer zurück.

3.2.2 Test-Anwendung

Um die zuvor implementierten Funktionen zu testen wurde eine einfache NodeJS Anwendung entwickelt. Diese aktiviert die Lambda-Funktionen über ihre korrespondierenden API-Pfade und speichert die Ergebnisse als CSV. Hierbei misst die Applikation die Ausführungszeit der gesamten Http Get Anfrage (Roundtrip). Des weiteren wird die in der API Antwort enthaltene serverseitige Laufzeit, sowie der Http Statuscode gespeichert.
Über eine CSV Datei (Timings.csv) kann eine Sequenz für das Durchführen der Abfragen angegeben werden, in dieser wird die Anzahl an derselben, sowie die Wartezeit zwischen den Anfragen festgelegt.
Die nachfolgende Tabelle 3.1 zeigt eine mögliche Request-Sequenz:

Wartezeit (in ms)	Anzahl Aufrufe
1000	5
5000	10

Tabelle 3.1: Beispiel einer Abfragesequenz

Diese Tabelle kann folgendermaßen interpretiert werden:
Führe zuerst 5 Aufrufe mit jeweils 1 Sekunde Pause aus, danach führe 10 Aufrufe mit jeweils 5 Sekunden Pause aus.

3.3 Auswertung

Zur Auswertung der gesammelten Daten wird ein Jupyter-Notebook (Python) verwendet, dieses visualisiert die Latenzzeiten unter Verwendung der üblichen Data-Science Frameworks (unter anderem Pandas, Matplotlib).
Um eine relativ gute Datenbasis zu erhalten wurden alle versuche mindestens drei Mal durchgeführt und ihre Ergebnisse gemittelt.

3.3.1 Client-Daten

	timestamp	status code	request time	server time	request type	function type
0	1619780358277	200	1111	63.000000	get	/default/fib_node
1	1619780358279	200	1163	162.000000	get	/default/fib_node
2	1619780358278	200	1171	164.000000	get	/default/fib_node
3	1619780358281	200	1519	571.211459	get	/default/fib_python
4	1619780358280	200	1525	566.245233	get	/default/fib_python

Abbildung 3.2: Ausschnitt aus den von der lokalen Test Anwendung gesammelten Daten
Quelle: Eigene Darstellung (Pandas)

Grafik 3.2 zeigt einen Ausschnitt der gesammelten Daten. Diese werden innerhalb des Jupyter-Notebook zuerst für die Darstellung vorbereitet. Hierzu zählt unter anderem die Umwandlung des Zeitstempels in die vergangene Zeit seit Start der Anwendung. Dafür wird für jeden Zeitstempel die Differenz zwischen ihm und dem allerersten berechnet und das Zeitintervall in Millisekunden angegeben. Danach können die Daten in einigen Grafiken dargestellt werden.

Abbildung 3.3: Serverseitige Laufzeiten der Lambda Funktionen
Quelle: Eigene Darstellung (Matplotlib)

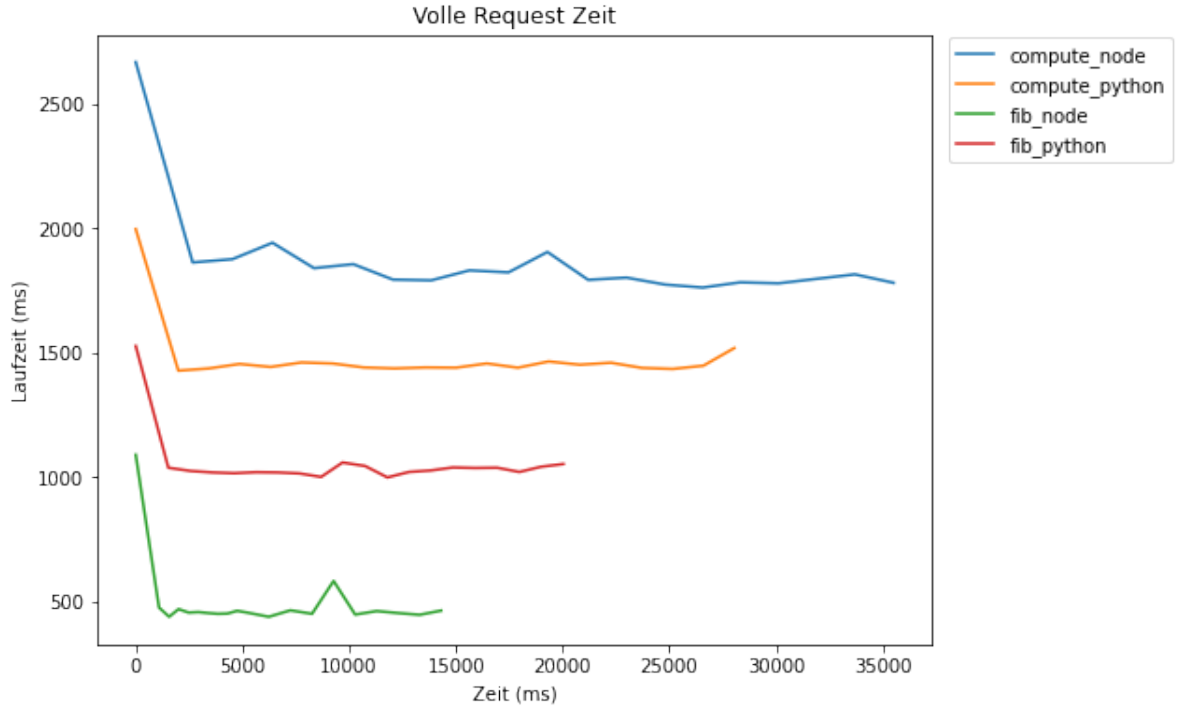

Abbildung 3.4: Laufzeiten des gesamten Funktionsaufrufs (Roundtrip)
Quelle: Eigene Darstellung (Matplotlib)

Die Abbildung 3.3 zeigt die serverseitigen Laufzeiten aller vier Lambdafunktionen. Zu erkennen ist eine relativ konstante Ausführungszeit für jede Funktion, außer der Node-JS Fibonacci Implementierung, welche eine stark erhöhte bei ihrer ersten Ausführung aufweist.

Abbildung 3.4 zeigt die Clientseitige Latenz der gesamten HTTP Abfrage (Roundtrip). Hier ist eine deutliche Erhöhung der Benötigten Zeit für die erste Antwort erkennbar. Dies lässt sie mithilfe des zuvor erläuterten Kaltstart-Problems erklären. Zur Verdeutlichung zeigen die nächsten zwei Grafiken nur die Anfragen die in den ersten 5 Sekunden des Tests erfolgt sind.

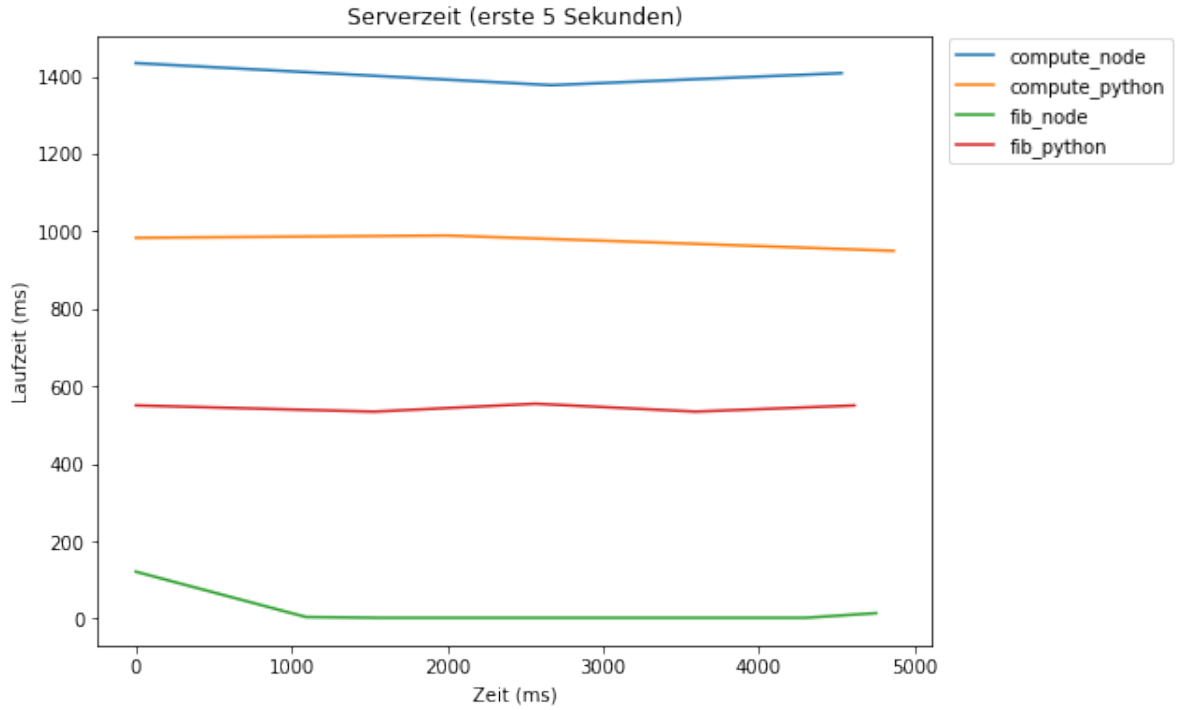

Abbildung 3.5: Serverseitige Laufzeiten der Lambda Funktionen (Ausschnitt)
Quelle: Eigene Darstellung (Matplotlib)

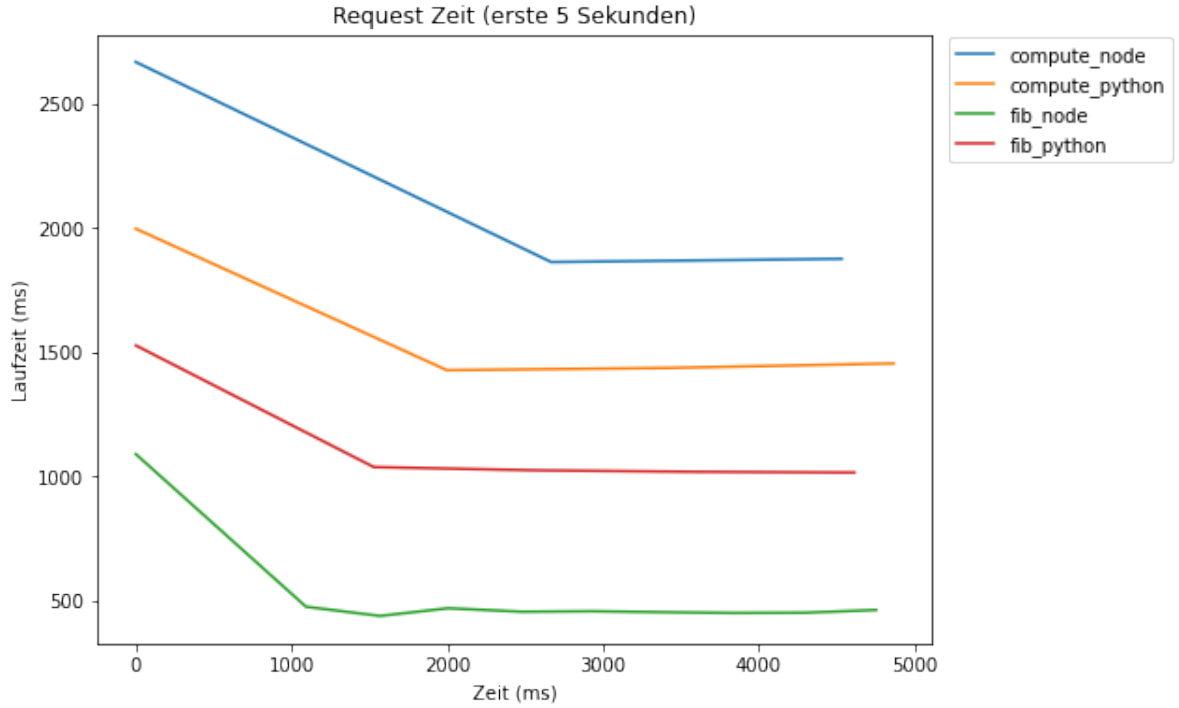

Abbildung 3.6: Laufzeiten des gesamten Funktionsaufrufs (Ausschnitt)
Quelle: Eigene Darstellung (Matplotlib)

In dem in Grafik 3.6 gezeigten Ausschnitt wird die Start erhöhte Latenz der ersten Abfrage besonders deutlich. Zudem fällt auf dass die NodeJS Fibonacci Implementierung die größte Verzögerung aufweist. Dies lässt sich mit einem Blick in Grafik 3.5 erklären, da die Funktion bei ihrer ersten Ausführung bereits eine erhöhte Laufzeit aufweist, summiert sich diese mit der Kaltstartverzögerung.

3.3.2 AWS Cloudwatch

Neben den von der Testanwendung gesammelten Daten, können ebenfalls die von AWS gesammelten Log Daten der Lambda Funktionen analysiert werden. Diese können mithilfe des AWS CloudWatch Service eingesehen und als CSV exportiert werden.

	Timestamp	log	DurationInMS	BilledDurationInMS	MemorySetInMB	MemoryUsedInMB
0	2021-04-30 11:03:09.243	257307065737:/aws/lambda/compute_node	1521.15	1522	128	88
1	2021-04-30 11:02:58.890	257307065737:/aws/lambda/compute_python	1017.28	1018	128	51
2	2021-04-30 11:02:57.283	257307065737:/aws/lambda/compute_node	1491.58	1492	128	88
3	2021-04-30 11:02:49.676	257307065737:/aws/lambda/fib_python	576.06	577	128	50
4	2021-04-30 11:02:47.330	257307065737:/aws/lambda/compute_python	1035.14	1036	128	51

Abbildung 3.7: Ausschnitt aus den von AWS CloudWatch gesammelten Daten
Quelle: Eigene Darstellung (Pandas), Daten von AWS CloudWatch bereitgestellt

Stellt man diese Daten ebenfalls graphisch dar, so erhält man folgende Übersicht:

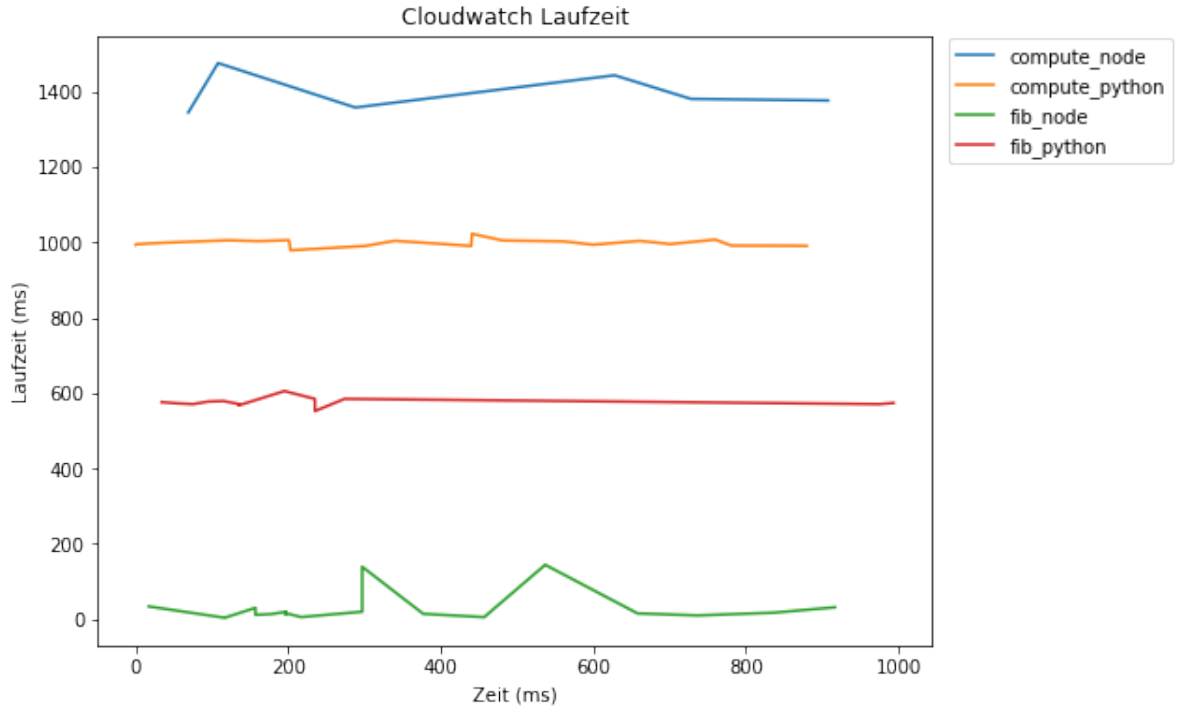

Abbildung 3.8: AWS CloudWatch Laufzeiten
Quelle: Eigene Darstellung (Matplotlib), Daten von AWS CloudWatch bereitgestellt

Die von AWS bereitgestellten Daten bestätigen die von den Funktionen selbst gemessenen Laufzeiten. Neben den Laufzeiten misst Cloudwatch ebenfalls den Speicherverbrauch der Funktionen.

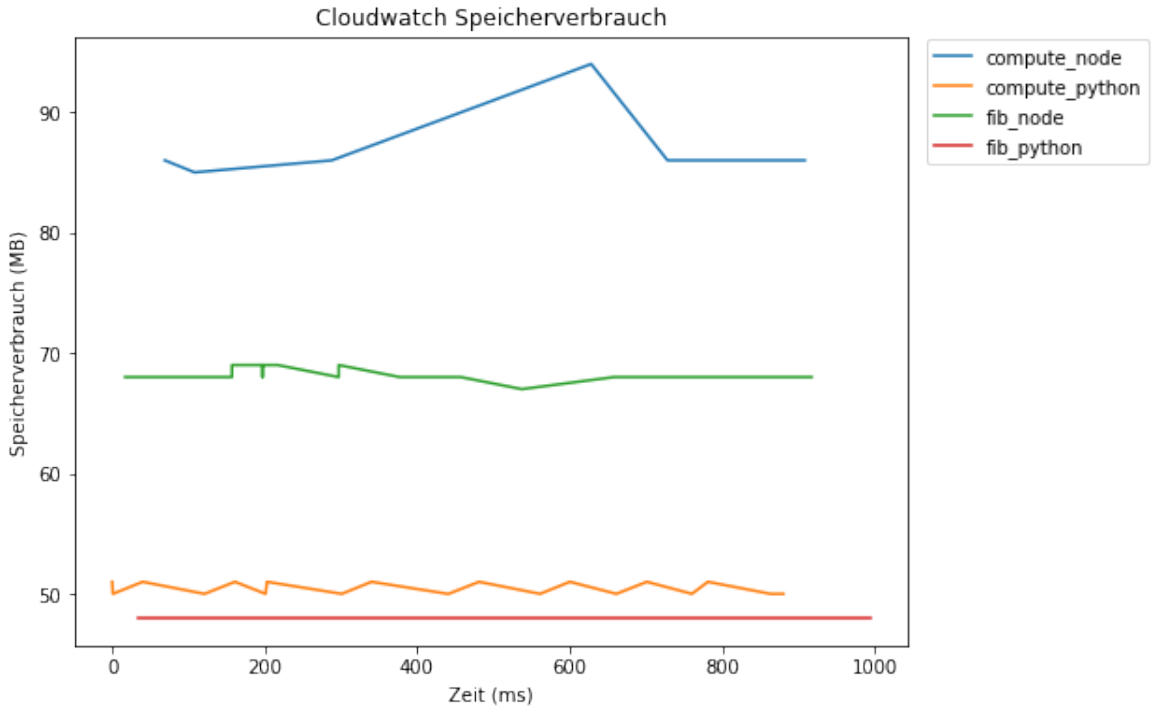

Abbildung 3.9: AWS CloudWatch Speicherverbrauch
Quelle: Eigene Darstellung (Matplotlib), Daten von AWS CloudWatch bereitgestellt

Abbildung 3.9 zeigt den Speicherverbrauch der Funktionen. Hier fällt der deutlich höhere Speicherverbrauch der NodeJS Funktionen auf, was auf die Unterschiede zwischen dem NodeJS Framework und Python zurückzuführen ist.

3.3.3 Coldstart Abweichungen

Berechnet man die Abweichung in der Laufzeit der ersten Abfrage verglichen mit dem Durchschnitt aller anderen Abfragen so erhält man folgende Werte:

Funktion	Absolute Abweichung
compute_node	798.0ms
compute_python	557.0ms
fib_node	632.0ms
fib_python	503.0ms

Funktion	Relative Abweichung
compute_node	70%
compute_python	72%
fib_node	42%
fib_python	67%

Tabelle 3.2: Auswirkungen des Coldstart

Der Tabelle lässt sich entnehmen, das die durchschnittliche Kaltstartverzögerung der untersuchten Funktionen zwischen 42% und 72% der normalen Laufzeit liegt.

3.4 Schlussfolgerungen

Aus den durchgeführten Versuchen kann man schließen, dass es bei AWS Lambda-Funktionen ein Coldstart verhalten gibt. Allerdings scheint Lambda so weitgehend Optimiert, dass man nur bei der allerersten Abfrage nach einer längeren Wartezeit eine deutliche Verzögerung feststellen kann. Daher betrifft die Kaltstart Problematik nur selten genutzte Funktionen die genügend Lehrlauf besitzen damit sie auf null Instanzen skaliert werden.
Bei den verwendeten Testfunktionen mit durchschnittlichen (serverseitigen) Laufzeiten zwischen wenigen Millisekunden und ca 1.4 Sekunden trat eine relative Verzögerung zwischen 40% und 70% auf. Diese ist allerdings nur Clientseitig messbar, da die tatsächliche Laufzeit der Funktion (nach Instanziierung) fast unverändert ist.
Weitere Untersuchungen für komplexeren Funktionen, die weitere AWS Services einbinden (z.B Datenbankabfragen) bieten sich für zukünftige Versuche an, da die Auswirkungen dieser Services auf das Kaltstartverhalten nicht untersucht wurden.

Literaturverzeichnis

[1] "Funktionen von aws lambda." https://aws.amazon.com/de/lambda/features/. Accessed: 2021-04-20.

[2] "Aws lambda sla." https://aws.amazon.com/de/lambda/sla/. Accessed: 2021-04-28.

[3] "Java jit optimierung." https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html. Accessed: 2021-04-20.

[4] Dr. Dillinger, "Verteilte systeme," Wintersemester 2020. Vorlesungs-Skript.