Vergleich der Startzeit von Funktionen bei marktführenden FaaS-Anbietern (Januar 2019)

Marco Mehrhoff – mmehrhoff@stud.hs-bremen.de

Abstract—In diesem Paper wird die Startzeit von Funktionsaufrufen bei drei marktführenden Function-as-a-Service-Anbietern (Amazon AWS Lambda, Google Cloud Functions & Microsoft Azure Functions) durch einen Test ermittelt. Durch verschiedene Zeitintervalle bei den Funktionsaufrufen wird erkannt, ab welchem Intervall ein Kalt-Start wahrscheinlicher wird. Des Weiteren wird verglichen ob Unterschiede zwischen den genutzten Programmiersprachen (C#, Go, Java, JavaScript, Python & Ruby) existieren. Durch eine Auswertung der erhobenen Daten wird die wahrscheinliche Kalt-Startzeit für Funktionen der genutzten Programmiersprachen für jeden Anbieter berechnet und im Ergebnis präsentiert. Zudem wird festgehalten, dass Amazon vor Google und Microsoft als FaaS-Anbieter zu präferieren ist. Im Anschluss wird ein Ausblick auf zukünftige Erweiterungen des Papers gegeben.

Index Terms—Serverless, FaaS, Functions-as-a-Service, Performance, Startzeit, Amazon, AWS, Lambda, Google, Cloud, Functions, Microsoft, Azure

1 Einleitung

Serverless Computing ist ein aufstrebender Bereich in der Bereitstellung von Cloud-Applikationen. Der Begriff "Serverless" (übersetzt "ohne Server") ist etwas irreführend, da immer noch Server zur Bereitstellung genutzt werden. Es bezieht sich darauf, dass der Entwickler oder Betreiber nur seinen Dienst bereitstellt und das Ressourcen-Management komplett vom Cloud-Anbieter übernommen wird. Dieser entscheidet auf welchem Server mit welchen Ressourcen der Dienst ausgeführt wird. [1]

Functions-as-a-Service (FaaS) ist ein Teilbereich des Serverless Computing. Zustandslose Funktionen werden bei einem Cloud-Anbieter definiert und können über einen Trigger (z. B. HTTP, internes Event) aufgerufen werden. Findet der Aufruf zum ersten Mal seit längerer Zeit statt spricht man von einem sogenannten Kalt-Start [2, 3, 4, 5]. Bei diesem muss ein entsprechender Server mit den von der Funktionen geforderten Ressourcen ausgewählt, die Laufzeitumgebung und Funktion dort geladen und dann ausgeführt werden [5]. Bei paralleler Ausführung einer Funktion kann es somit auch zu Kalt-Starts kommen. Der Cloud-Anbieter behält es sich gegebenenfalls vor eine aufgerufene Funktion "warm" zu halten ("Idle"). Das bedeutet, dass die Ressourcen für diese Funktion für eine bestimmte Zeit auf dem Server reserviert werden und bei einer erneuten Ausführung die Ressourcen nicht erneut angefordert werden müssen. Dies entspricht einem Warm-Start.

Nach Baldini et al. liegt der Unterschied von FaaS zu Platform-as-a-Service (PaaS) und Software-as-a-Service (SaaS) in der Kontrolle des Entwicklers [3]. Bei PaaS wird die Infrastruktur sowie der Code nach den Wünschen des Entwicklers bereitgestellt. Bei SaaS wird die Infrastruktur sowie der Code vom Anbieter gestellt. Bei FaaS gibt der Entwickler die Kontrolle über die Infrastruktur ab, behält jedoch die Kontrolle über den ausgeführten Code. Des Weiteren hat FaaS nach Lloyd et al. weitere Restriktionen in der Ausführungszeit sowie der Größe des Codes, welche bei PaaS keine übergeordnete Rolle spielen [2]. Zur Übersicht ist in Tabelle 1 die Einordnung von FaaS im Vergleich zu IaaS, PaaS und SaaS dargestellt. Weißer Hintergrund bedeutet, dass der Kunde die Verwaltung in der Hand hat währenddessen grauer Hintergrund bedeutet, dass der Cloud-Anbieter diese verwaltet.

In diesem Paper soll die Forschungsfrage "*Wie unterscheiden sich die Startzeiten einer in einer FaaS-Umgebung bereitgestellten Funktion wenn diese in unterschiedlichen Zeitabständen aufgerufen wird?*" für marktführende Cloud-Anbieter und unterschiedliche Programmiersprachen beantwortet werden. Des Weiteren sollen die folgenden zwei Unterfragen pro Anbieter und genutzter Programmiersprache beantwortet werden:

- *Ab welchem Zeitabstand wird ein Kalt-Start wahrscheinlicher?*
- *Wie hoch ist die Startzeit bei einem Kalt-Start?*

Als marktführende Cloud-Anbieter wurden *Amazon AWS Lambda*[1], *Google Cloud Functions*[2] & *Microsoft Azure Functions*[3] ausgewählt. Dabei wird auf JavaScript und Python als Programmiersprache eingegangen. Diese wird von allen drei Anbietern unterstützt. Des Weiteren werden weitere unterstützte Programmiersprachen (C#, Go, Java & Ruby) in ihrem Verhalten untersucht.

Im nächsten Abschnitt 2 wird auf die Literatur im Bereich Performance von FaaS eingegangen. Im darauffolgenden Abschnitt 3 wird der Aufbau des Tests erläutert. Daraufhin findet im Abschnitt 4 eine Auswertung und Evaluation statt. Im letzten Abschnitt 5 werden die Ergebnisse zusammengefasst und ein Ausblick gegeben.

2 Verwandte Arbeiten

Im Bereich Serverless bzw. FaaS ist wenig Literatur in Bezug zur Performance vorhanden. Insbesondere der Aspekt der

1. https://aws.amazon.com/lambda/
2. https://cloud.google.com/functions/
3. https://azure.microsoft.com/en-us/services/functions/

TABLE 1: Einordnung *-as-a-Service nach Jacobs [6]

Infrastructure (IaaS)	Platform (PaaS)	Function (FaaS)	Software (SaaS)
Geschäftslogik	Geschäftslogik	Geschäftslogik	Geschäftslogik
Applikation	Applikation	Applikation	Applikation
Daten	Daten	Daten	Daten
Laufzeitumgebung	Laufzeitumgebung	Laufzeitumgebung	Laufzeitumgebung
Betriebssystem	Betriebssystem	Betriebssystem	Betriebssystem
Virtualisierung	Virtualisierung	Virtualisierung	Virtualisierung
Server, Speicher, Netzwerk	Server, Speicher, Netzwerk	Server, Speicher, Netzwerk	Server, Speicher, Netzwerk

Kalt-/Warm-Starts von Funktionen wird nicht intensiv behandelt. Baldini et al. sehen die Kalt-Starts als Herausforderung für das ganze Serverless- bzw. FaaS-System [3]. Es stellt eine hohe Priorität dar, die Kalt-Startzeit von Funktionen durch den Einsatz von Techniken so gering wie möglich zu halten. Dieser Punkt wird von Bardsley et al., Lynn et al. aufgegriffen aber nicht weiter vertieft bzw. so weit eliminiert, dass der Kalt-Start-Effekt nicht auftritt [5, 7]. Lloyd et al. erstellen einen Vergleich zwischen einem gehosteten Docker-Machine-Server, *Amazon AWS Lambda* und *Microsoft Azure Functions* [2]. Der Fokus lag darauf zu erkennen welcher der drei kalten Zustände der Funktion vorliegt. Dazu wurde bei *Amazon AWS Lambda* erkannt, dass nach 10 Minuten die ersten Container-Kalt-Starts, nach 30 Minuten die ersten VM-Kalt-Starts eintreten und somit zu einem Kalt-Start der Funktion beitragen. Für *Microsoft Azure Functions* sind die Zeiten mit 10 und 40 Minuten ähnlich. Jedoch wurden bei der Betrachtung unterschiedliche Programmiersprachen genutzt.

McGrath and R. Brenner führen einen "Backoff"-Test mit *Amazon AWS Lambda*, *Apache OpenWhisk*, *Google Cloud Functions*, *Microsoft Azure Functions* sowie einem eigenen Prototypen durch [4]. Durch diesen Test soll herausgefunden werden, ab welchem Zeitabstand Funktionen bei erneutem Aufruf einen Kalt-Start durchführen. Gegenüber Lloyd et al. [2] wurde für alle Funktionen die Programmiersprache JavaScript verwendet. Der Test wurde mit Zeitabständen von 1 bis 30 Minute(n) durchgeführt und ergab, dass Amazon und Google quasi keinerlei Kalt-Start-Verhalten bis 30 Minuten Zeitabstand aufweisen. Bei Microsoft wurde gezeigt, dass es dort bereits ab einem Zeitabstand von 5 Minuten zu einem Kalt-Start kommt. Es ist aus dem Paper nicht zu erkennen wie oft oder zu welcher Tageszeit der Test durchgeführt wurde. Im Abschnitt 4.1 werden die Daten aus dem Test mit den ausgewerteten Daten verglichen.

Des Weiteren führen Ishakian et al. einen Test zur Ermittlung der Kalt-Startzeit auf der *Amazon AWS Lambda*-Platform durch [8]. Dafür werden fünf Aufrufe in einem Abstand von 10 Minuten durchgeführt. Der Test wird für drei Funktionen, welche ein neuronales Netz ausführen, durchgeführt. Die Ergebnisse sind durch die schwergewichtige Implementierung wahrscheinlich nicht mit den Ergebnissen dieses Papers vergleichbar.

Abseits der wissenschaftlichen Literatur gibt es mehrere Artikel die über das Verhalten von Funktionen in einer FaaS-Umgebung berichten. Dort wird insbesondere auf das Phänomenen des Kalt-Starts bei *Amazon AWS Lambda* eingegangen [9, 10, 11] sowie auf dessen Vermeidung [12, 13]. Dabei wird herausgearbeitet wie hoch die Kalt-Startzeit für verschiedene Programmiersprachen ist [10], sowie wie groß der Zeitabstand bis zu einem Kalt-Start ist [11]. Die Ergebnisse der letzten beiden Artikel werden im Abschnitt 4.3 noch einmal aufgegriffen und verglichen.

Nach Untersuchung der verwandten Arbeiten lässt sich festellen, dass keines der genannten Werke explizit das Gebiet der Kalt-Startzeit bei verschiedenen Anbietern und Programmiersprachen in Bezug zum Zeitabstand der Funktionsaufrufe behandelt.

3 Aufbau des Tests

Der Test bezieht sich auf drei marktführende Anbieter und deren unterstützte Programmiersprachen, welche im Folgenden aufgeschlüsselt sind:

- Amazon AWS Lambda
  - C#
  - Go
  - Java
  - JavaScript
  - Python
  - Ruby
- Google Cloud Functions
  - JavaScript
  - Python (Beta)
- Microsoft Azure Functions
  - C#
  - Java (Beta)
  - JavaScript
  - Python (Beta)

Um herauszufinden ab wann eine Funktion bei einem Aufruf kalt gestartet wird (ergo eine höhere Startzeit), wurden neun Zeitintervalle definiert (30 Sekunden, 5 & 15 Minuten, 1, 2, 12, 24, 36 & 48 Stunde(n)). Für jedes Intervall wird bei jedem Anbieter in jeder Programmiersprache eine Funktion angelegt. Der Standort wird auf West-Europa und der Speicher auf 128 MB festgelegt. Die Funktionen geben eine Antwort in JSON zurück, welche das aktuelle Datum sowie die Zeit in UTC auf dem Server beinhaltet. Dadurch wird vermieden, dass die Antwort evtl. zwischengespeichert wird. Des Weiteren wird die Laufzeit der Funktion minimal gehalten. Als Trigger wird für jede Funktion ein HTTPS-Aufruf eingestellt.

TABLE 2: Minimale & maximale Startzeit von Funktionen

Anbieter			Startzeit [ms]	
		Anzahl	Minimum	Maximum
Amazon	C#	91448	11	4618
	Go	91043	11	2035
	Java	90955	12	2637
	JavaScript	91269	11	7194
	Python	91261	13	1745
	Ruby	91058	13	1462
Google	JavaScript	91206	15	6522
	Python	91457	17	4043
Microsoft	C#	90568	15	22992
	Java	76231	18	44120
	JavaScript	90028	15	38349
	Python	90558	21	20626

Der Test erfolgt über vier Wochen. Dafür wurde eine virtuelle Maschine in der Cloud gemietet welche für jede Funktion und Intervall einen Docker-Container ausführt. Dieser Container führt ein Bash-Skript zur Messung aus und speichert das Ergebnis in einer Datensenke. Das Skript führt die Messung über das Werkzeug cURL[4] aus. Für den Aufruf der Webseite bietet cURL die Möglichkeit Metriken zu sammeln [14]. Dadurch kann der DNS-Lookup sowie der SSL-Handshake aus der Zeit herausgerechnet werden und die reine Laufzeit (inkl. Startzeit) der Funktion ermittelt werden. Da die Laufzeit minimal gehalten wurde, kann diese vernachlässigt und die Zeit als Startzeit betrachtet werden. Diese Startzeit beinhaltet auch die Infrastruktur von jedem Anbieter (z. B. das API Gateway von Amazon).

4 Ergebnis

In diesem Abschnitt wird auf die einzelnen Ergebnisse der Abbildungen 1 bis 5 eingegangen. Dabei wird ein Vergleich der Startzeit der Sprachen JavaScript und Python zwischen den unterschiedlichen Anbietern gezogen (s. Abschnitte 4.1 & 4.2). Des Weiteren wird die Startzeit bei jedem Anbieter zwischen dessen unterstützter Sprachen untersucht (s. Abschnitte 4.3, 4.4 & 4.5).

Die Anzahl der Ergebnisse pro Intervall und Programmiersprache sind für jeden Anbieter in der Tabelle 2 aufgeschlüsselt. Ergebnisse für die Intervalle >= 12 Stunden liegen nur in einer geringen Anzahl vor, so dass diese mit einer gewissen Sorgfalt zu betrachten sind.

4.1 JavaScript

In der Abbildung 1 sind die Startzeiten von JavaScript-Funktionen im Vergleich zwischen den Anbietern dargestellt. In der Tabelle 2 ist zu erkennen, dass bei einem Warm-Start einer Funktion eine Zeit von ca. 15 ms eingehalten werden kann. Bei den maximalen Werten (s. Tabelle 2) ist zu erkennen, dass Microsoft mit bis zu 38 Sekunden Startzeit gegenüber Amazon und Google mit 7 bzw. 6,5 Sekunden um den Faktor 5-6 langsamer sein kann.

4. https://curl.haxx.se

Die Abbildung 1a zeigt, dass 50 % der Anfragen bei Google mit einem Zeitintervall $>$ 12 Stunden eine Startzeit von mindestens 500 ms aufweisen. Hingegen steigt die Startzeit bei Amazon bereits ab einem Zeitintervall $>$ 1 Stunde, jedoch nur auf 250 ms. Bei Microsoft dagegen ist keine nennenswerte Änderung zu erkennen. 50 % der Funktionen werden dort unter 100 ms gestartet.

In Abbildung 1b verschiebt sich das Zeitintervall bei Google von $>$ 12 auf $>$ 2 Stunden. 5 % der Anfragen haben nun eine Startzeit von $>$ 1,5 s. Bei Amazon gibt es analog zu Google einen Anstieg der Startzeit für 5 % der Anfragen. Ab einem Intervall $>$ 1 Stunde liegt die Startzeit bei $>$ 1,25 s. Nach dem Intervall von 12 h gibt es jedoch einen Abfall der Startzeit. Dies könnte darauf zurückzuführen sein, dass nicht genügend Ergebnisse vorliegen. Es liegt nicht nahe, dass Amazon Funktionen wieder Ressourcen zuteilt, sobald das Intervall $>$ 12 Stunden ist. Für Microsoft wurde für 95 % der Funktionsaufrufe eine Startzeit von unter 750 ms ermittelt - unabhängig vom Zeitintervall.

Das 99. Perzentil, dargestellt in Abbildung 1c, zeigt für Google und Amazon ein ähnliches Verhalten wie beim 95. Perzentil. Amazon bleibt konstant bei einer Erhöhung ab einem Zeitintervall $>$ 1 Stunde. Zusätzlich erhöht sich die Startzeit auf bis zu 5 s. Bei Google verschiebt sich das Intervall auf $>$ 15 Minuten. Die Startzeit bleibt bei etwa 2 s. 1 % der Funktionsaufrufe bei Microsoft benötigen jedoch $>$ 9 s. Besonders auffällig ist, dass dieses bei einem Zeitintervall von 30 s der Fall ist.

Zusammenfassend lässt sich sagen, dass Microsoft sehr lange Startzeiten von bis zu 9 s von Funktionen, trotz häufigem Aufruf, nicht vermeiden kann. Im äußersten Fall kann der Start der Funktion bis zu 38 s dauern (s. Tabelle 2). Amazon und Google können in 99 % der Aufrufe eine Startzeit von unter 2,5 s aufweisen, welche im äußersten Fall auf bis zu 5 s steigen kann. Dieses ist im Vergleich zu Microsoft jedoch gering.

4.1.1 Vergleich mit Ergebnissen von McGrath and R. Brenner [4]

Das Paper von McGrath and R. Brenner hat festgestellt, dass für Amazon und Google keine nennenswerte Änderung der Antwortzeit für einen Zeitabstand von bis zu 30 Minuten zu bestimmen ist [4]. Zwar ist kein direkter Vergleich der Zeiten möglich, da die Werte die Antwortzeit (inkl. DNS-Lookup, SSL-Handshake, etc.) darstellen und die Resultate aus Abbildung 1 die Startzeit der Funktion, jedoch lässt sich die Aussage bis zu einem Zeitabstand von 15 Minuten validieren. Für die Zeit zwischen 15 und 30 Minuten lässt sich in diesem Paper keine Aussage treffen, die Tendenz geht jedoch in die gleiche Richtung. Für Microsoft ist das Ergebnis dieses Papers gespiegelt. In Abbildung 1c ist zu erkennen, dass die Latenz im ersten Intervall bis zu 9 s betragen kann und danach bei 1 s liegt. McGrath and R. Brenner hingegen messen für Microsoft eine erhöhte Antwortzeit erst ab 5 Minuten [4]. Dieses Ergebnis kann dahingehend nicht validiert werden.

4.2 Python

Die Abbildung 2 zeigt die Startzeiten von Funktionen, welche in Python geschrieben wurden, im Vergleich zwischen den Anbietern. Es sei hier nochmal darauf

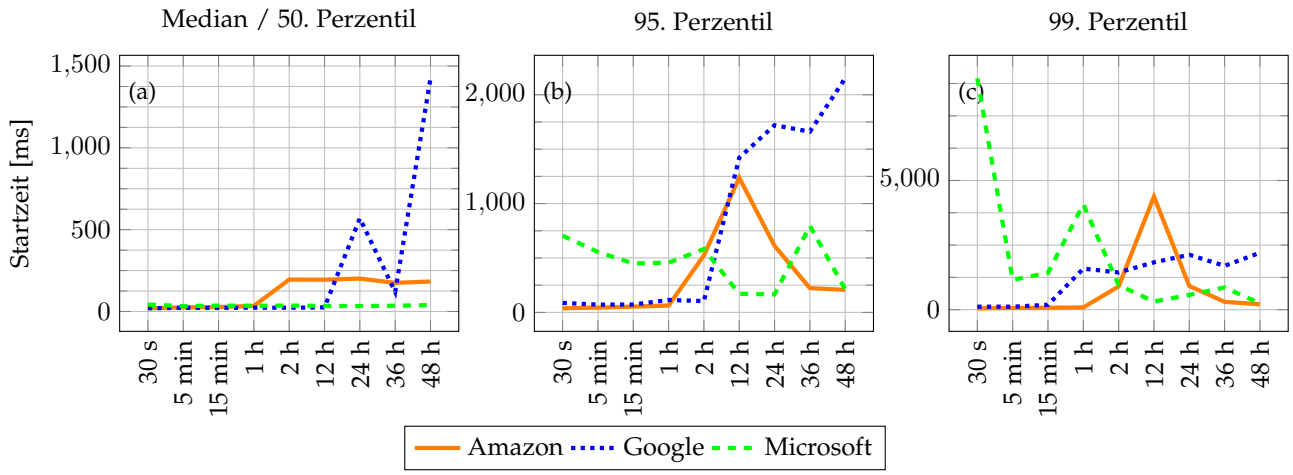

Fig. 1: (a) Median, (b) 95. und (c) 99. Perzentil der Startzeit für jedes Intervall (JavaScript)

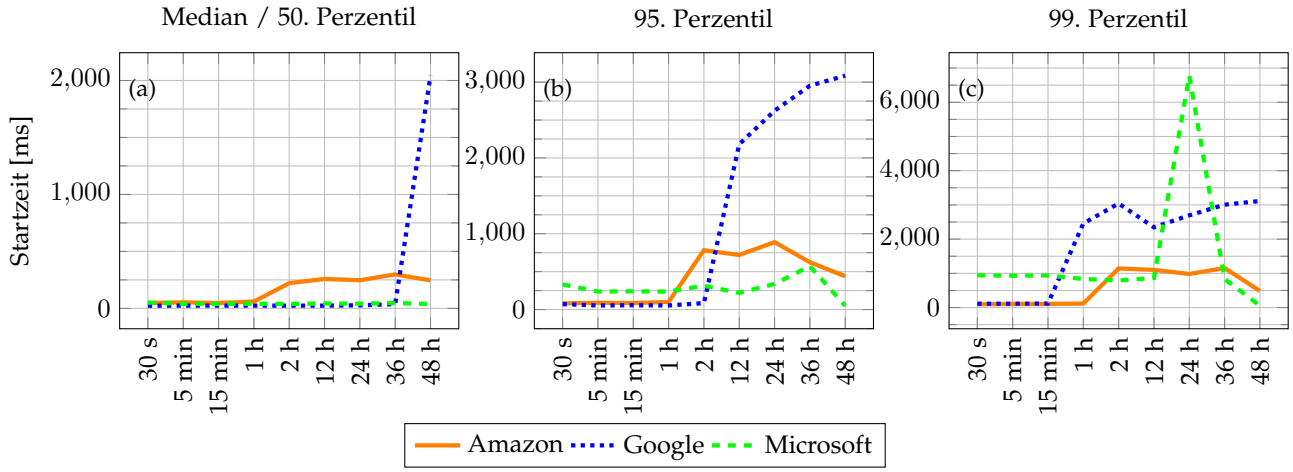

Fig. 2: (a) Median, (b) 95. und (c) 99. Perzentil der Startzeit für jedes Intervall (Python)

hingewiesen, dass sich die Skriptsprache bei Google und Microsoft in einer Beta befindet und die Auswertung nur einen Blick auf den aktuellen Stand der beiden Anbieter zulässt. Amazon bietet Python als funktionsfähige Laufzeitumgebung an. In Tabelle 2 ist eine minimale Startzeit von < 25 ms für alle Anbieter und eine maximale Startzeit von 1,75 s für Amazon, 4 s für Google & 20 s für Microsoft gelistet. Die maximale Startzeit bei Google ist um den Faktor 2, bei Microsoft um den Faktor 11 größer als bei Amazon.

Die Abbildung 2a zeigt ein ähnliches Bild wie Abbildung 1a. 50 % der Anfragen bei Google mit einem Zeitintervall von > 24 Stunden weisen eine Startzeit von 2 s auf. Bei Amazon hingegen steigt die Startzeit bei einem Zeitintervall > 1 Stunde auf 250 ms. Bei Microsoft werden 50 % der Funktionen unter 100 ms gestartet.

Das 95. Perzentil in Abbildung 2b zeigt Ähnlichkeiten zur Abbildung 1b. Google verhält sich analog zur Sprache JavaScript. Das Zeitintervall verschiebt sich von > 24 auf > 2 Stunden mit einer Startzeit > 2 s. Amazon verhält sich analog zum 50. Perzentil in Abbildung 2a, jedoch mit einer erhöhten Startzeit von 900 ms. Bei Microsoft lässt sich eine Startzeit von < 500 ms über alle Intervalle erkennen.

Die Abbildung 2c zeigt das 99. Perzentil. Amazon verhält sich analog zur Abbildung 2b mit einer weiteren leichten Erhöhung der Startzeit auf 1 s. Die Intervallgrenze bleibt bei einer 1 Stunde. Google verschiebt das Zeitintervall zu > 15 Minuten und erhöht die Startzeit auf bis zu 3 s. Bei Microsoft erhöht sich die Startzeit auf < 1 s über alle Intervalle hinweg. Die Spitze bei 24 h kann vernachlässigt werden, welche auf eine geringe Menge an Messwerten zurückzuführen ist.

Als Ergebnis lässt sich festhalten, dass Amazon und Google sich analog zur Programmiersprache JavaScript (vgl. Abschnitt 4.1) verhalten. Je nach Perzentil verkleinert sich bei Google das Zeitintervall, bei Amazon ist das Verhalten über die Perzentile konstant. Microsoft zeigt, abgesehen von der Spitze, keine Tendenz ab einem bestimmten Intervall einen Kalt-Start zu nutzen. Dort wird gleichmäßig über die Intervalle ein Kalt-Start durchgeführt.

4.3 Amazon

In der Abbildung 3 sind die Startzeiten der von Amazon unterstützten Programmiersprachen dargestellt. Dies lässt einen Vergleich zwischen den Sprachen zu um eventuell vorhandene Unterschiede zu ermitteln. Die minimale Startzeit bewegt sich für alle Sprachen auf einem

TABLE 3: Kalt-Start Intervalle (Amazon)

C#	Java	Go	JavaScript	Python	Ruby
$>$ 30 s	$>$ 5 min	$>$ 1 h			

TABLE 4: Kalt-Start Intervalle (Google)

JavaScript	Python
$>$ 15 min	

Niveau ($<$ 15 ms). Die maximale Startzeit liegt zwischen 1,5 s für Ruby bei bis zu 7 s für JavaScript (vgl. Tabelle 2).

Beim 50. Perzentil ist zu erkennen, dass die Sprachen Go, JavaScript, Python & Ruby sich gleich verhalten. Ab einem Zeitintervall von $>$ 1 Stunde steigt die Startzeit für $>$ 50 % von $>$ 100 ms auf $>$ 450 ms. Bei C# und Java steigt bei einem Zeitintervall von $>$ 1 Stunde die Startzeit von $>$ 100 ms über 1,5 s auf $>$ 2,5 s für C# und 1 s für Java. Dies könnte unter anderem daran liegen, dass C# und Java Sprachen mit einer Laufzeitumgebung sind, welche vor der Ausführung erst geladen werden muss. Damit wird die Startzeit um einen nahezu konstanten Wert erhöht.

In der Abbildung 3b setzt sich C# weiter von Java ab. Das Zeitintervall verschiebt sich auf $>$ 15 Minuten und die Startzeit für 5 % der Aufrufe steigt von 100 ms über 3 s auf 4 s. Java hingegen rückt näher mit den Skriptsprachen und Go zusammen. Das Intervall verschiebt sich auf $>$ 15 Minuten und die Startzeit steigt auf 1,75 s. Bei den Skriptsprachen steigt die Startzeit für 95 % der Aufrufe auf $<$ 1 s und das Intervall bleibt bei 1 Stunde. Die Funktionen in der Sprache Go verhalten sich ähnlich zu denen in Skriptsprachen geschriebenen Funktionen.

Für das 99. Perzentil (s. Abbildung 3c) gelten die selben Merkmale wie für das 95. Perzentil außer für C# und Java. Ersteres beginnt für 1 % der Aufrufe bereits bei einem Intervall $>$ 30 s Kalt-Starts durchzuführen. Java neigt bereits bei Intervallen $>$ 5 Minuten dazu Kalt-Starts für 1 % der Aufrufe durchzuführen. Von der Startzeit her ist Java den Skriptsprachen sowie Go (1,75 s zu 1 s) unterlegen.

Zusammenfassend lässt sich sagen, dass die Skriptsprachen bei Amazon ein besseres Kalt-Start-Verhalten gegenüber den Sprachen mit Laufzeitumgebung zeigen. Die Skriptsprachen zeigen erst bei größeren Intervallen ein Kalt-Start-Verhalten, hingegen die Sprachen mit Laufzeitumgebung dieses früher zeigen. Die Sprache Go als kompilierte ausführbare Datei zeigt ein ähnliches Verhalten wie die Skriptsprachen. Auf Basis der Abbildung 3c wurde die Tabelle 3 erstellt. In dieser ist aufgeschlüsselt, bei welcher Sprache ab welchem Intervall mit einem Kalt-Start zu rechnen ist.

4.3.1 Vergleich mit den Ergebnissen von Cui [10]

Cui vergleicht in seinem Artikel die Kalt-Antwortzeit verschiedener Programmiersprachen (C#, Java, JavaScript & Python) miteinander [10]. Dabei wird auch zwischen unterschiedlichen Speichergrößen unterschieden. Mit diesem Paper werden nur die Werte mit einer Speicherzuweisung von 128 MB verglichen. Die Antwortzeit ist zwar nicht direkt mit der berechneten Startzeit der Funktion vergleichbar, lässt aber trotzdem einen Vergleich zu. Daher lässt sich die Aussage soweit validieren, dass C# und Java langsamer als JavaScript und Python sind. In diesem Paper wurden jedoch niedrigere Werte gemessen (C# 1 s, Java 4 s schneller), was auf eine Optimierung seitens Amazon hindeuten könnte. Für JavaScript und Python wurden deutlich höhere Werte ermittelt als in den Ergebnissen von Cui festgehalten [10]. Die in diesem Paper gemessenen Kalt-Startzeiten sind um den Faktor 10-20 langsamer. Dies könnte dadurch zustande kommen, dass Amazon direkt erstellte Funktionen in Skriptsprachen bereits lädt und somit kein Kalt-Start provoziert werden kann wie Cui annimmt [10].

4.3.2 Vergleich mit den Ergebnissen von Cui [11]

Cui ermittelt in dem Artikel das Zeitintervall wann eine Funktion beim nächsten Aufruf einen Kalt-Start vollzieht [11]. Es wird offen gelassen in welcher Programmiersprache die Funktion geschrieben wurde. Des Weiteren wird nur auf die Ergebnisse für einen Speicherverbrauch von 128 MB eingegangen. Cui hat eine Grenze von 1 Stunde für einen Speicherverbrauch von 128 MB ermittelt [11]. Dies deckt sich mit den Ergebnissen aus diesem Paper für die Skriptsprachen sowie Go. Für die Sprachen mit Laufzeitumgebung sind die Ergebnisse aus diesem Paper wesentlich geringer, diese werden bereits ab $>$ 30 s bzw. 5 Minuten kalt gestartet.

4.4 Google

Abbildung 4 stellt die unterstützten Programmiersprachen von Google und deren Startzeit in unterschiedlichen Perzentilen dar. Es sei anzumerken, dass sich Python noch im Beta Status befindet. Minima und Maxima der Startzeit liegen für JavaScript bei 15 ms und 6,5 s; für Python bei 17 ms und 4 s (s. Tabelle 2).

Die Abbildung 4a zeigt, dass 50 % der Funktionsaufrufe erst nach 36 Stunden (JavaScript & Python) eine Startzeit $>$ 1,5 s haben. Die Spitze bei 24 Stunden für JavaScript liegt bei 500 ms und kann vernachlässigt werden.

Das 95. Perzentil in Abbildung 4b lässt erkennen, dass das Intervall für beide Sprachen auf $>$ 2 Stunden verschoben wurde. 95 % der Funktionsaufrufe liegen bei einem höheren Intervall unter 2 bzw. 3 s (JavaScript bzw. Python) Startzeit.

Abbildung 4c mit dem 99. Perzentil zeigt das selbe Bild wie Abbildung 4b. Ein Unterschied ist, dass das Intervall sich auf $>$ 15 Minuten verschiebt und dadurch 1 % der Anfragen mit einem höheren Intervall eine Startzeit $>$ 2 bzw.$>$ 3 s (JavaScript bzw. Python) haben.

Zusammenfassend besteht bei Google kein merklicher Unterschied zwischen dem Zeitintervall ab wann eine Funktion in einer bestimmten Sprache kalt gestartet wird. JavaScript und Python verhalten sich dort ähnlich. Einen merklichen Unterschied gibt es in der Startzeit der Funktionen im 95. und 99. Perzentil. Dort ist Python bis zu 1 s langsamer als JavaScript. Auf Basis der Abbildung 4c wurde die Tabelle 4 erstellt. Diese beinhaltet die Information ab wann ein Kalt-Start wahrscheinlich wird.

4.5 Microsoft

Abbildung 5 zeigt die von Microsoft unterstützten Programmiersprachen (s. Tabelle 2) im Vergleich zu deren Startzeit

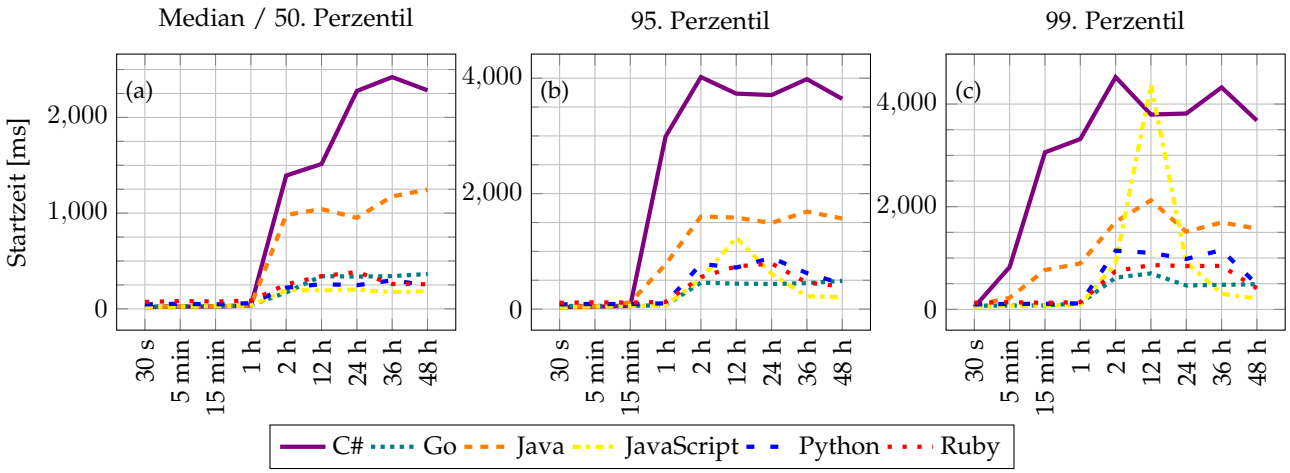

Fig. 3: (a) Median, (b) 95. und (c) 99. Perzentil der Startzeit für jedes Intervall (Amazon)

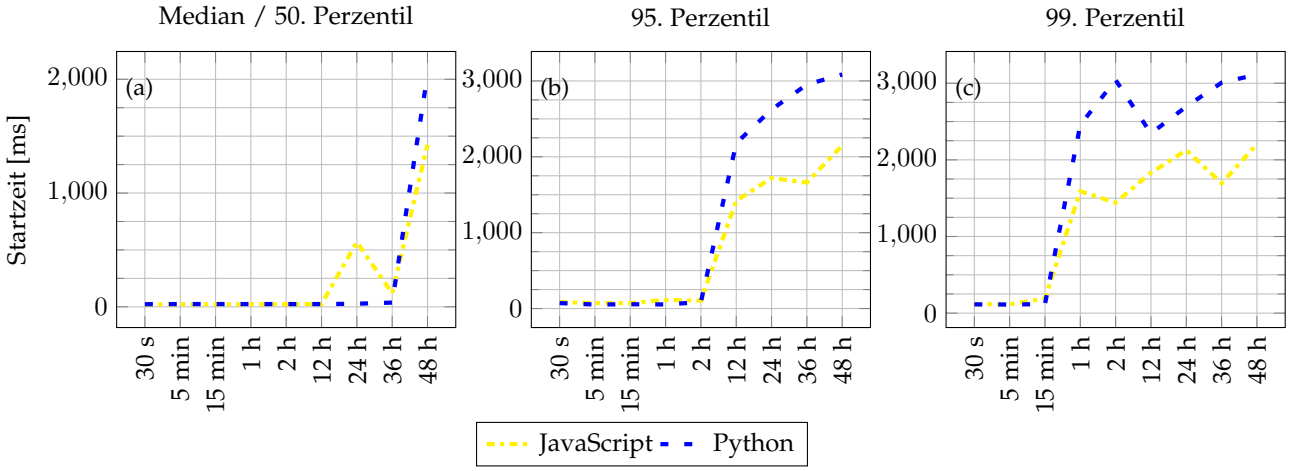

Fig. 4: (a) Median, (b) 95. und (c) 99. Perzentil der Startzeit für jedes Intervall (Google)

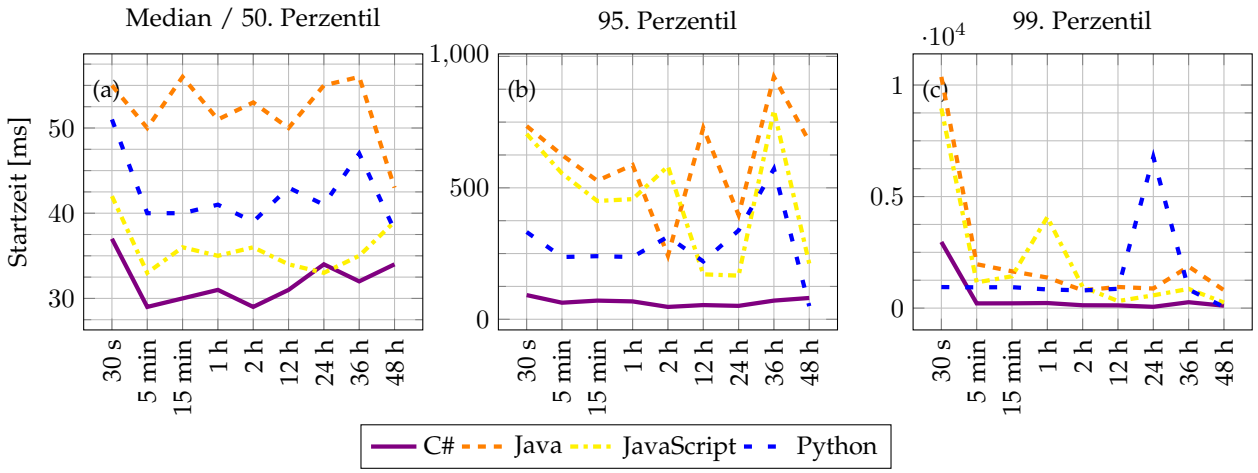

Fig. 5: (a) Median, (b) 95. und (c) 99. Perzentil der Startzeit für jedes Intervall (Microsoft)

TABLE 5: Kalt-Start Intervalle (Microsoft)

C#	Java	JavaScript	Python
> 30 s			

für jedes definierte Zeitintervall. Es ist zu beachten, dass sich die Programmiersprache Java und Python im Beta-Status befinden. Die maximalen Startzeiten (s. Tabelle 2) sind vergleichsweise hoch. Von 20 s (Python) bis zu 44 s (Java) sind diese sehr hoch. Die minimalen Startzeiten sind mit < 25 ms schnell.

Der Median in Abbildung 5a zeigt keine besondere Auffälligkeit ab einem bestimmten Zeitintervall. Für jedes Intervall wird für 50 % der Funktionsaufrufe eine Startzeit von < 70 ms erreicht. Dabei ist C# schneller als die anderen Sprachen. Im Vergleich mit Java liegt die Differenz aber nur bei ca. 20 ms.

In Abbildung 5b ist das 95. Perzentil dargestellt. Alle Sprachen liegen für jedes Intervall bei einer Startzeit < 1 s für 95 % der Funktionsaufrufe. Python liegt mit der Startzeit niedriger als Java und JavaScript. C# ist auch im 95. Perzentil die schnellste startende Sprache für Funktionen. Die Startzeit für 95 % der Funktionsaufrufe steigt nicht über 100 ms.

Das 99. Perzentil in Abbildung 5c zeigt für die Funktionen in Python ein ähnliches Bild wie in Abbildung 5b. 1 % der Funktionsaufrufe zeigen eine Startzeit > 1 s. Beim Intervall von > 12 Stunden steigt die Startzeit auf 7 s. Java zeigt für 1 % der Aufrufe ab einem Zeitintervall von 5 Minuten eine Startzeit von über 1 s. JavaScript lässt eine ähnliche Startzeit erkennen. Auffällig ist, dass Java und JavaScript für 1 % der Aufrufe beim kleinsten Zeitintervall von 30 Sekunden eine Startzeit von 9 bis 10 s aufzeigen. C# ist auch beim 99. Perzentil eine der schnellen Sprachen für Funktionen. Für das Intervall 30 Sekunden ist zu erkennen, dass die Startzeit für 1 % der Aufrufe bei 2-3 s liegt. Für die restlichen Intervalle liegt die Startzeit bei unter 225 ms.

Zusammenfassend ist zu erkennen, dass Microsoft der hauseigenen Programmiersprache C# anscheinend eine wichtigere Rolle als andere Programmiersprachen zukommen lässt. In allen Perzentilen zählt C# zu den schnelleren Sprachen für die Startzeit von Funktionen. Hervorzuheben ist auch, dass alle Programmiersprachen für 95 % der Funktionsaufrufe nicht länger als 1 s benötigen. Nachteilig zu bewerten ist die unter Umständen sehr hohe, maximale Startzeit der Funktionen beim Kalt-Start (vgl. Tabelle 2). Diese zeigen sich besonders im 99. Perzentil in den Sprachen Java und JavaScript, sowie abgeschwächt bei C#. In der Tabelle 5 ist für jede Sprache das Zeitintervall angegeben, ab welchem ein Kalt-Start wahrscheinlich wird. Für alle Sprachen wurde das Zeitintervall 30 Sekunden ausgewählt, weil die Kalt-Starts in allen Intervallen vorkommen. Bei Python kommt hinzu, dass diese besonders ab einem Zeitintervall von 24 Stunden stattfinden.

4.6 Ergebnis

Das Ergebnis der Untersuchungen der vorherigen Abschnitte zeigt, dass es zwischen den Anbietern und genutzten Programmiersprachen deutliche Unterschiede in der Startzeit nach unterschiedlichen Zeitintervallen gibt.

TABLE 6: Kalt-Startzeiten

Anbieter		Anteil [‰]	Kalt-Startzeit-Perzentil [ms] 50.	95.	99.
Amazon	C#	9,66	1356	3795	4284
	Go	0,18	852	1876	2003
	Java	7,18	906	1574	1691
	JavaScript	0,36	783	2679	5997
	Python	0,59	783	1256	1533
	Ruby	0,65	626	978	1285
Google	JavaScript	1,47	1452	2219	2891
	Python	1,20	2573	3653	3988
Microsoft	C#	12,81	3171	4312	5584
	Java	67,45	900	14624	20606
	JavaScript	65,16	888	14657	19656
	Python	37,89	788	1698	3605

Auf Grundlage der gesammelten Daten wird nun für jede Programmiersprache von jedem Anbieter die ermittelte Kalt-Startzeit in der Tabelle 6 angegeben. Als Grenze zwischen Kalt- und Warm-Start wird eine Startzeit von 500 ms gesetzt. Alle Startzeiten größer diesem Wert wurden aggregiert und der Median (50. Perzentil) sowie das 95. und 99. Perzentil ermittelt (s. Tabelle 6).

Ergebnis ist, dass unabhängig von den Programmiersprachen die Anbieter in folgender Reihe zu bevorzugen sind:

1) Amazon AWS Lambda
2) Google Cloud Functions
3) Microsoft Azure Functions

Die Reihenfolge ergibt sich dadurch, dass Amazon über alle Programmiersprachen eine geringe Kalt-Startzeit im Median aller Kalt-Starts vorweisen kann. Des Weiteren ist der Anteil der Kalt-Starts bei Skriptsprachen sowie Go im Vergleich zu den anderen Anbietern sehr gering. C# und Java als Sprachen mit einer benötigten Laufzeitumgebung benötigen öfter Kalt-Starts, sind im Vergleich zu Microsoft aber geringer. Google wird hinter Amazon aufgelistet, da die Kalt-Startzeit im Vergleich zu Amazon höher ist. Die Startzeiten sind zwar auch höher als bei Microsoft im Median, jedoch ist der Anteil der Kalt-Starts im Vergleich gering. Microsoft sollte als Anbieter von Cloud-Funktionen in Anbetracht der Kalt-Startzeit vermieden werden. Für JavaScript und Python bieten sich Amazon sowie Google als Alternative an. Für C# und Java sollte auf Amazon zurückgegriffen werden.

5 Zusammenfassung & Ausblick

Dieses Paper untersuchte den aktuellen Stand dreier marktführender FaaS-Anbieter und ihrer unterstützten Programmiersprachen in Bezug zur Startzeit der Funktionen in unterschiedlichen Zeitabständen. Dazu wurde ein vierwöchiger Test bei den Anbietern durchgeführt. Daraus konnten Rückschlüsse gezogen werden, welcher Anbieter mit welcher Programmiersprache die beste Wahl in Bezug

zur Startzeit bietet. Basierend darauf wurde festgestellt, dass Amazon momentan die beste Variation aus Anzahl der Kalt-Starts und Dauer der Kalt-Starts über alle Programmiersprachen hinweg bietet. Danach folgen Google und Microsoft. Des Weiteren wurde herausgearbeitet ab welchem Zeitabstand der Funktionsaufrufe die Wahrscheinlichkeit auf einen Kalt-Start steigt. Dieses wurde für jeden Anbieter herausgearbeitet und graphisch sowie tabellarisch aufgearbeitet.

Als zukünftige Arbeit könnten weitere Anbieter wie *IBM Cloud Functions*[5] oder den bald verfügbaren *Oracle Functions*[6] in den Vergleich aufgenommen werden um deren Performance zu analysieren. Eine weitere Möglichkeit wäre die Artikel von Cui bezüglich des Speichers wissenschaftlich aufzuarbeiten [10, 11].

Die angefallenen Daten sind auf GitHub[7] verfügbar.

REFERENCES

[1] M. Roberts, "Serverless Architectures," 05 2018, accessed at 2019-01-05. [Online]. Available: https://www.martinfowler.com/articles/serverless.html

[2] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, Florida, USA, April 2018, pp. 159–169.

[3] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. Springer, Singapore, 2017, pp. 1–20.

[4] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Atlanta, GA, USA, July 2017, pp. 405–410.

[5] D. Bardsley, L. Ryan, and J. Howard, "Serverless Performance and Optimization Strategies," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, New York, New York, September 2018, pp. 19–26.

[6] S. Jacobs, "Function as a service," 02 2018, accessed at 2019-01-06. [Online]. Available: https://blog.oio.de/2018/02/14/function-as-a-service/

[7] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Hong Kong, China, December 2017, pp. 162–169.

[8] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving Deep Learning Models in a Serverless Platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, Florida, USA, April 2018, pp. 257–262.

[9] R. Vojta, "AWS journey — API Gateway & Lambda & VPC performance," 10 2016, accessed at 2019-01-06. [Online]. Available: https://www.robertvojta.com/aws-journey-api-gateway-lambda-vpc-performance/

[10] Y. Cui, "How does language, memory and package size affect cold starts of AWS Lambda?" 06 2017, accessed at 2019-01-06. [Online]. Available: https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76

[11] ——, "How long does AWS Lambda keep your idle functions around before a cold start?" 07 2017, accessed at 2019-01-06. [Online]. Available: https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810

[12] G. Neves, "Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues," 04 2017, accessed at 2019-01-06. [Online]. Available: https://www.serverless.com/blog/keep-your-lambdas-warm/

[13] S. Corcos, "How to Keep Your Lambda Functions Warm," 05 2017, accessed at 2019-01-06. [Online]. Available: https://read.acloud.guru/how-to-keep-your-lambda-functions-warm-9d7e1aa6e2f0

[14] D. Stenberg, "curl - how to use," accessed at 2019-01-06. [Online]. Available: https://curl.haxx.se/docs/manpage.html

5. https://console.bluemix.net/openwhisk/

6. https://blogs.oracle.com/cloud-infrastructure/announcing-oracle-functions

7. https://www.github.com/marcOcram/faas-start-time