Entwicklung einer serverlosen Anwendung für das Testen von Java Programmen in einer isolierter Umgebung

Bachelorarbeit

von

Sirat Ahmadi

aus

Essen

vorgelegt am

Lehrstuhl für Rechnernetze
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

August 2020

Betreuer:
Dr. Christian Meter

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer serverlosen Anwendung für das Ausführen und Testen einfacher Java Konsolenanwendungen in einer sicheren Umgebung. Um fremden Quellcode schnell und einfach zu testen, wird er häufig ungeachtet jeglicher Sicherheitsrisiken auf dem eigenen System ausgeführt. Der ausgeführte Code kann jedoch Sicherheitslücken ausnutzen und somit dem eigenen System schaden. Um den Quellcode von Java Konsolenanwendungen abgesichert zu kompilieren und zu starten, wurde eine Applikation entwickelt, die den Quellcode in einer sicheren Umgebung ausführen kann. Das Resultat wird dem Benutzer zurückgesendet. Die Applikation baut auf den Serverless Prinzipien auf. Dabei handelt es sich um ein neues Cloud-Computing Konzept, mit dem skalierbare Anwendungen ohne komplexe Serververwaltung bereitgestellt werden können. Bevor die eigentliche Entwicklung beginnt, wird eine Einführung in das Thema Serverless-Computing gegeben. Es werden die wichtigsten Eigenschaften und unterschiedliche serverlose Dienstarten, wie FaaS, vorgestellt.

Für die Entwicklung der Anwendung wurde Cloud Run benutzt, ein serverloser Dienst von Google. Die Anwendung stellt eine REST-API bereit. Über unterschiedliche Endpunkte können Benutzer ihren Java Quellcode oder ihr Gradle-Projekt versenden. Der Java Quellcode wird durch Pythons Subprocess-Modul und dem Java Development Kit erst kompiliert und danach ausgeführt. Bei den Gradle-Projekten wird durch den Subprocess-Modul und den passenden Gradle-Befehlen das Projekt gebaut und ausgeführt. Das Resultat wird dem Benutzer zurückgesendet. Werden zusätzlich JUnit Tests mitgesendet, können diese über weitere Endpunkte gestartet werden. Zusätzlich zur serverlosen Anwendung wurde ein Frontend entwickelt. Das Frontend stellt eine kleine Entwicklungsumgebung zur Verfügung, über die der Java Quellcode geschrieben und an das Backend versendet werden kann.

Während der Evaluation traten einige Probleme auf. Das Ausführen von Gradle-Projekten funktioniert nicht immer. Bei großer Nachfrage werden ungefähr 85% der Anfragen nicht erfolgreich beantwortet. Es wird angenommen, dass dies Aufgrund eines Fehlers bei dem temporären Speichern des gesendeten Zip-Archivs ist. Außerdem braucht die serverlose Anwendung im Durchschnitt zu lange für das Ausführen und Testen. Die Anwendung muss für eine produktionsreife Bereitstellung verbessert und erweitert werden. In Anbetracht der vielen Probleme kommt man zu dem Schluss, dass die Serverless Prinzipien und serverlose Dienste für die entwickelte Anwendung nicht geeignet sind.

Danksagung

Einen ganz herzlichen Dank an Dr. Christian Meter für die tatkräftige Unterstützung und wöchentliche Betreuung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird die Problemstellung und das Ziel der Arbeit vorgestellt. Außerdem wird ein Überblick über die einzelnen Kapitel gegeben.

1.1 Motivation

Während meiner Korrektor-Tätigkeit neigte ich schnell dazu den Quellcode der Studenten sofort zu kompilieren und das resultierende Programm auszuführen, weil dadurch bereits ein sehr aussagekräftiger Eindruck über die Qualität der Abgabe ersichtlich wird. Jedoch bringt das Ausführen von fremden und ungeprüften Programmen einige Gefahren mit sich. Es können Sicherheitslücken der *Java Runtime Environment* (JRE) [Ora20] oder des eigenen Systems ausgenutzt werden, wodurch Schadprogramme eingeschleust werden könnte [MIT20]. Um dies zu verhindern, kann man das Programm in einer isolierten Umgebung laufen lassen. Das manuelle Aufsetzen einer *Virtual-Machine* (VM) [Red20d] auf dem eigenen Computer benötigt relativ viele Ressourcen und ist zugleich auch aufwendiger nur um ein kleines Programm auszuführen. Das Ausweichen auf traditionelle externe Dienste von *Cloud-Providern* hat den Nachteil, dass man einen stündlichen bzw. monatlichen festen Betrag zahlen muss, auch wenn der Dienst nicht genutzt wird. Man muss im Vorhinein grob die Auslastung abschätzen, um den Server passend einzustellen. Außerdem muss der Server regelmäßig auf Sicherheitslücken geprüft und dementsprechend aktualisiert werden. Effizienter wäre eine serverlose (engl. *serverless*) Anwendung, dass den Quellcode in einer sicheren Umgebung baut und ausführt.

Das Konzept des *Serverless-Computing* beinhaltet das Entwickeln und Bereitstellen von skalierbaren Anwendungen ohne komplexe Serververwaltung, bei der nur der eigentliche Verbrauch in Zahlung gestellt wird [Com19]. Dafür bieten Cloud-Provider Dienste an, welche sie selbst verwalten. Entwickler brauchen sich nur um ihre Programme und neue Erweiterungen zu kümmern, da die mühselige Arbeit der Skalierung, Provisionierung und Instandhaltung bereits von den Cloud-Providern übernommen werden.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es eine serverlose Anwendung für das sichere Ausführen und Testen von einfachen *Java*[1] und *Gradle*[2] Konsolenanwendungen zu entwickeln. Der Quellcode soll über eine REST-API an den Dienst gesendet werden, wo er anschließend gebaut und ausgeführt oder getestet wird.

1.3 Aufbau der Arbeit

Kapitel 2 - Grundlagen Zu Beginn werden die nötigen Grundlagen und Eigenschaften von Serverless erläutert. Ein wesentlicher Bestandteil dieser Arbeit ist, dass die Programme in einer sicheren Umgebung, getrennt voneinander ausgeführt werden. Dafür werden *Container* und Virtual-Machines näher erkundet. Diese zwei Konzepte sind das Fundament der Umgebungen, in denen Cloud-Provider die Anwendungen ihrer Kunden sicher und getrennt voneinander ausführen. Es werden zwei unterschiedliche serverlose Dienstarten vorgestellt, die den Entwicklern zur Verfügung stehen. Außerdem werden aktuell verfügbare serverlose Open Source Lösungen erwähnt. Abschließend wird der Aufbau und die Architektur einer serverlosen Plattform erläutert.

Kapitel 3 - Entwicklung In diesem Kapitel wird die eigentliche Architektur und Implementierung der entwickelten Anwendung präsentiert. Davor werden kurz die benötigten Werkzeuge vorgestellt.

[1] `https://www.oracle.com/de/java/`
[2] `https://gradle.org/`

Kapitel 4 - Probleme und Entscheidungen Es werden kritische Entscheidungen und Probleme erläutert, welche die Architektur und die Implementierung der Anwendung beeinflusst haben.

Kapitel 5 - Evaluation Nach dem Aufbau folgt das Testen. Es werden Performance-Probleme aufgelistet und mögliche Verbesserungen vorgeschlagen.

Kapitel 6 - Ähnliche Anwendungen Danach folgt ein Vergleich mit ähnlichen Lösungen. Dadurch werden Anwendungsgrenzen und Erweiterungen besser hervorgehoben.

Kapitel 7 - Fazit Abschließend wird ein Fazit der entwickelten Anwendung und seiner Serverless Eigenschaften gezogen.

Kapitel 2

Grundlagen

Für das Verständnis der Ausarbeitung werden einige Grundlagen benötigt. In diesem Kapitel werden diese Grundlagen übermittelt. Es werden die wichtigsten Eigenschaften des Serverless-Computing erläutert.

2.1 Serverless-Computing

Serverless-Computing (Serverless) ist ein Konzept, mit dem man skalierbare Anwendungen entwickeln und ohne komplexe Serververwaltung bereitstellen kann [Com19]. Das Konzept ist bei den öffentlichen Cloud-Providern beliebt, die viele serverlose Dienste für Entwickler und Unternehmen bereitstellen. Trotz des Namens werden selbstverständlich immer noch *Server* verwendet. Der Name symbolisiert die geringe und einfache Serververwaltung im Vergleich zu üblichen selbstverwalteten Diensten [Fou18]. Es wird meist zwischen zwei Arten von serverlosen Diensten unterschieden. Cloud-Provider bieten mehrere Laufzeitumgebungen und Programmiersprachen zur Auswahl an, sodass Entwickler durch isolierte, kompakte und zustandslose Funktionen ihre Anwendungen entwickeln können. Diese Dienstart wird auch *Function-as-a-Service* (FaaS) genannt, wie beispielsweise AWS Lambda[1]. Serverless ist ein relativ neues Thema, sodass der Begriff immer weiter ausgebaut wird [Bal+17]. So haben Cloud-Provider mit *Serverless Containers* eine weitere serverlose Dienstarten eingeführt [Rai19]. Googles serverloser Container Dienst, Cloud Run [Goo20a], gibt Entwicklern die

[1] `https://aws.amazon.com/de/lambda/`

Freiheit die Laufzeitumgebung ihrer Anwendung selbst zu definieren, denn für die Bereitstellung erwartet es ein Dockerimage. Ein Dockerimage ist eine mehrschichtige Schablone, welche alle nötigen Abhängigkeiten und Instruktionen für das Erzeugen eines Containers enthält [Doc20] [Red20b]. Entwickler können somit die Laufzeitumgebung ihrer Anwendungen genaustens definieren. Sie sind nicht gezwungen ihre Anwendungen aus einzelnen Funktionen zu entwickeln, sondern können auch eine übliche komplexere Applikation bereitstellen [Bal+17].

2.1.1 Function as a Service

Häufig wird der Begriff Serverless mit einer ganz bestimmten serverlosen Dienstart assoziiert, den Function-as-a-Service (FaaS) [Bal+17]. Die Idee ist, dass Entwickler ihre Anwendungslogik aus isolierten und zustandslosen Funktionen aufbauen. Es ist vorgesehen, dass jede Funktion nur für eine ganz spezifische Aufgabe zuständig ist. Die Funktionen können durch Ereignisse (englisch *Events*) ausgelöst werden. Durch Events können einzelnen Funktionen miteinander oder mit weiteren Diensten der Architektur verbunden werden. Treffen solche Ereignisse ein, provisioniert der Cloudanbieter die passenden Ressourcen und führt die jeweilige Funktion aus. Aufgrund des geringen Aufgabenbereichs und der Zustandslosigkeit einer Funktion, sind Cloudanbieter in der Lage sehr schnell mehrere Instanzen einer Funktion zu starten, um eine erhöhte Nachfrage zu bewältigen. Verringert sich die Nachfrage wieder, werden die Instanzen abgebaut. Anders ist es bei traditionellen selbstverwalteten Diensten, denn dort müssen Entwickler die Nachfrage grob abschätzen und bei zu hoher Nachfrage die Kapazitäten rechtzeitig erhöhen. Verringert sich die Nachfrage wieder muss daran gedacht werden die Kapazitäten zu verkleinern, da sonst Ressourcen in Zahlung gestellt werden die nicht gebraucht wurden [MR17] [Bal+17].

2.1.2 Serverless Containers

Eine weitere serverlose Dienstart sind die Serverless Containers [Rai19]. Ein großer Nachteil von FaaS Diensten der Cloud-Provider ist, dass man die Laufzeitumgebung nicht erweitern kann. Bei der Entwicklung seiner Funktionen muss man aus einer geringen Auswahl an Programmiersprachen und Laufzeiten auswählen. Serverlose Container bieten hier einen großen Vorteil, denn sie erwarten ein Dockerimage [Rai19]. Dadurch haben Entwickler mehr Kon-

trolle und Freiheit bei der Anwendungsentwicklung. Zusätzlich bleiben die Vorteile der FaaS Dienste erhalten.

2.1.3 Eigenschaften von Serverless

Damit ein Dienst dem Serverless Prinzip folgt, sollte es gewisse Charakteristiken aufweisen [SKM19]. Diese Eigenschaften gelten für FaaS sowie für Serverless Containers können jedoch von Cloud-Provider zu Cloud-Provider leicht abweichen.

Einfache Bereitstellung Eine Anwendung sollte so einfach wie möglich bereitgestellt werden können. FaaS und Serverless Container Dienste erwarten meist nur eine Anwendungslogik und eine Dockerfile. Mit nur wenigen Kommandos und Interaktionen in der Kommandozeile oder der grafischen Benutzeroberfläche sollte die Anwendung bereitgestellt werden können [Bal+17].

Einfache Verwaltung und Wartung Nach dem Aufsetzen sollte die weitere Verwaltung der Anwendung, wie das Aktualisieren ohne Komplexität möglich sein. Google Cloud Run gibt Entwicklern die Möglichkeit Containerressourcen, wie beispielsweise die Speicherkapazität [Goo20c] und CPU-Anzahl [Goo20d], anzupassen. Die Dienste und darunterliegende Infrastruktur wird von den Cloud-Providern selbst auf Sicherheitslücken überwacht [Bal+17].

Automatisches Skalieren Bei erhöhter Nachfrage sollten weitere Instanzen der Anwendung schnell genug gestartet werden, sodass die Nachfrage gedeckt wird. Verringern sich die Anfragen, sollten die zusätzlich gestarteten Instanzen wieder abgebaut werden. Durch das automatische Skalieren passt sich der Dienst an die aktuelle Nachfrage der Anwendung an. Werden gar keine Anfragen an die Anwendung getätigt, sollte keine Instanz laufen. Dies wird auch *scale-to-zero* genannt [Bal+17].

Bezahlung des eigentlichen Verbrauchs Kosten fallen nur dann an, wenn mindestens eine Instanz der Anwendung arbeitet. Oft wird die eigentliche Rechendauer, Anzahl der Aufrufe pro Monat sowie die Konfiguration der Instanzen in Zahlung gestellt [Bal+17]. Verarbeitet

die Anwendung keine Anfragen, fallen keine Kosten an. Selbstverwaltete Dienste dagegen werden oft stündlich oder monatlich in Zahlung gestellt. Dies ist die *pay-for-use* Eigenschaft von Serverless [Ser18]. Zusätzlich bieten Google Cloud Run und AWS Lambda auch ein kostenloses Kontingent an [Goo20b] [Ama20]. Erst nachdem die jeweiligen Ressourcen ausgelastet sind, fallen Kosten an.

Somit lässt sich das Serverless-Computing Konzept gut auf Applikationen anwenden, die über den Tag hinaus eine starke Schwankung in der Nachfrage haben. Der Cloudanbieter wird die angefragten Funktionen passend hoch und runter skalieren. Dies kann unter Umständen zu geringeren Kosten führen. Auch wenn bereits stark auf andere Services eines Cloud-Providers zurückgegriffen wird, sind die serverlosen Dienste gut für das Verknüpfen der Services geeignet. Da die einzelnen Funktionen durch Events aktiviert werden, können Daten in eine Funktion eingeführt werden, um dann an die benötigten Komponenten in passendem Format wieder ausgeführt zu werden [Bal+17] [SMM18].

2.1.4 Nachteile von Serverless

Neben den zahlreichen Vorteilen bergen serverlose Dienste auch einige Nachteile. Diese gelten, mit einer Ausnahme, für FaaS sowie für Serverless Containers Dienste.

Abhängigkeit der Cloud-Provider Das Verwenden von vielen serverlosen Diensten kann zur Abhängigkeit der Cloud-Provider (englisch *vendor lock-in*) führen. Die einzelnen Services lassen sich leicht miteinander verbinden, was weiter die Komplexität der gesamten Anwendungsarchitektur verringert. Jedoch erwarten FaaS Dienste unterschiedliche Funktionsschreibweisen, sodass der Wechsel zu einem anderen Anbieter nicht mühelos wäre [AC17]. Dieses Problem haben Serverless Container nicht. Deren Laufzeitumgebung wird durch ein Dockerimage festgelegt. Somit ist der Wechsel zu anderen Diensten ohne großen Aufwand möglich.

Flüchtig und zustandslos Die gestarteten Instanzen sind flüchtig (englisch *ephemeral*). Dadurch eignet sich Serverless nicht für alle Anwendungen. Einige Dienste, wie Google Cloud Run, haben auch eine maximale Zeitspanne (englisch *timeout*), in der eine Anfrage abgearbeitet werden muss. Dauert es länger, wird die Bearbeitung abgebrochen [Goo20g].

Auch dürfen Anfragen nicht davon ausgehen, dass sie von der gleichen Instanz bearbeitet werden.

Cold-Starts Ein weiteres Problem von Serverless sind die *Cold-Starts*. Bei einer eintretenden Anfrage, braucht das Hochfahren einer neuen Anwendungsinstanz gewisse Zeit, denn Anwendungslogik und Laufzeitumgebung müssen erst initialisiert werden. Cloud-Provider versuchen diese Zeit zu minimieren [Ser18]. Wird eine Instanz mit der Bearbeitung einer Anfrage fertig so wird sie nicht sofort herunterfahren, sondern erst nach einer gewissen Zeit. Dadurch kann sie bei erneuter Anfrage während diese Zeit wiederverwendet werden, denn Laufzeitumgebung und Anwendungslogik sind bereits geladen [Mik20].

2.1.5 Open Source serverless Platform

Neben den serverlosen Diensten der Cloud-Provider stehen auch Open Source Lösungen zur Verfügung. Platformen wie Apache OpenWhisk [Apa19] geben Entwicklern die Möglichkeit ihre Applikationen nach dem Serverless Konzept bereitzustellen und von den Vorteilen zu profitieren. Wie bei Serverless Container Diensten, haben sie den Vorteil, dass die Laufzeitumgebung der Instanzen durch eine Dockerfile konfiguriert werden kann. Jedoch werden einzelne ereignisgesteuerte Funktionen benutzt, um Anwendungen zu entwickeln ähnlich den FaaS Diensten der Cloud-Provider [Apa19]. Einen großen Nachteil haben sie dennoch. Für das Ausführen in einer Produktionsumgebung setzen einige der Platformen Werkzeuge für Container-Orchestrierung voraus. Diese Werkzeuge müssen erst aufgesetzt werden oder Entwickler greifen auf einen verwalteten Dienst der Cloud-Provider zurück. Jedoch fallen dann stündliche bzw. monatliche Fixkosten an und die Verwaltung der Anwendung steigt, sodass die Vorteile von Serverless nicht ganz ausgeschöpft werden.

2.1.6 Aufbau einer serverless Platform

Obwohl die ausgearbeitete Anwendung im folgenden Kapitel mithilfe von Google Cloud Run entwickelt wird, wird hier der Aufbau einer serverlosen Platform anhand von Apache OpenWhisk vorgestellt. Aus Googles Dokumentationen wird nicht eindeutig ersichtlich, wie

deren Service aufgebaut ist. Dagegen ist OpenWhisk eine gut dokumentierte Open Source Lösung.

In OpenWhisk werden serverlose Funktionen *Actions* genannt. Dafür wird in einer Datei eine Funktion deklarieren. Die Funktion muss gewisse Voraussetzungen erfüllen, damit OpenWhisk sie verwenden kann. Unter anderem muss der Rückgabewert als JSON Objekt interpretiert werden können [Apa20].

```
1  function main(params) {
2      console.log('Hello World');
3      return { msg: 'Hello World' };
4  }
```

Listing 2.1: Hello World JS Funktion. In Anlehnung an [Apa19] [Apa20]

Die Funktion in Auflistung 2.1, die nach Ausführung den Text *Hello World* in der Standardausgabe ausgibt und selbigen Text auch als JSON-Objekt zurücksendet, erfüllt alle Voraussetzungen für die Umwandlung zu einer Action.

```
1  wsk action create myAction action.js
```

Listing 2.2: Erstellen einer Action [Apa19]

Die Action wird mithilfe der OpenWhisk CLI (`wsk`) und dem Zusatz `action create` erzeugt. Mit `myAction` und `action.js` in Auflistung 2.2 sind der Name der Action und der Name der Datei angegeben [Apa19].

```
1  /api/v1/namespaces/$userNamespace/actions/myAction
```

Listing 2.3: Action Endpunkt [Apa19]

Erstellen Entwickler ihre Actions werden sie durch einen HTTP-Endpunkt der Form in Auflistung 2.3, zugänglich. Die Variable `$userNamespace` ist hier ein Platzhalter für einen Namensraum. In OpenWhisk können Actions für eine bessere Verwaltung und Autorisierung in Namensräumen aufgeteilt werden [Apa19].

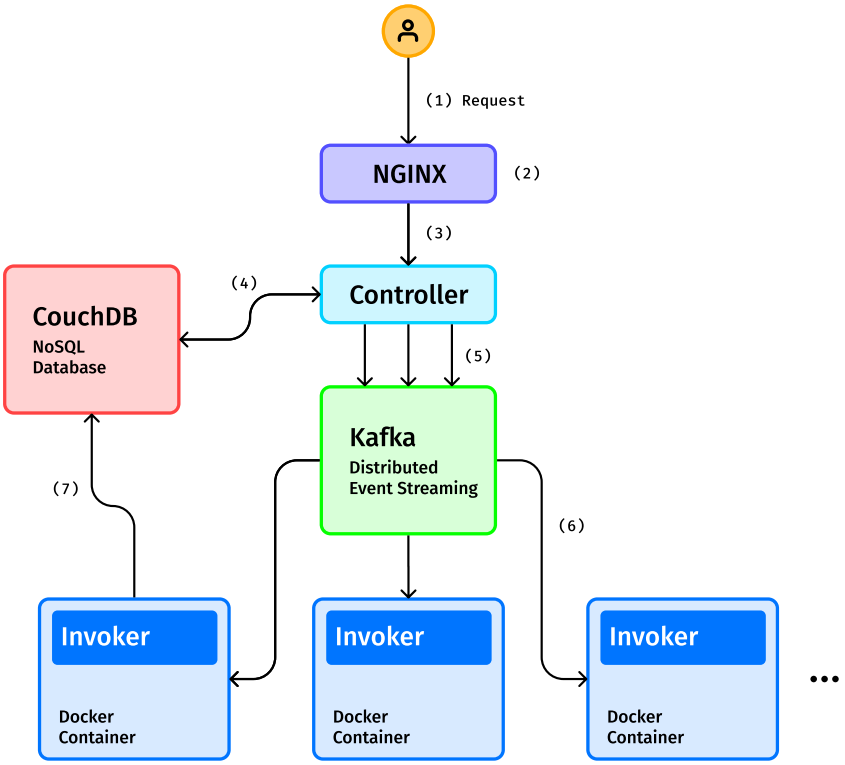

Abbildung 2.1: Apache OpenWhisk Architektur. In Anlehnung an [Apa19]

In Abbildung 2.1 ist die Architektur von OpenWhisk abgebildet. Sie baut auf unterschiedliche Komponenten auf, um eine Anfrage zu bearbeiten. Bei einem eintretenden Event **(1)** ist *Nginx*[2] **(2)** die erste Station. Diese wird zur SSL-Terminierung und Weiterleitung verwendet [Apa19]. Nginx leitet den Aufruf weiter an den *Controller* **(3)**. Der Controller implementiert die Rest-API und filtert die eintretenden Anfragen durch, um die passende Action und den weiteren Verlauf zu ermitteln. Er autorisiert die Anfrage und schaut auf nötige Authentifizierungen. Dafür befragt er CouchDB[3] **(4)**. Nachdem Autorisierung und Authentifizierung erfolgreich abgeschlossen sind, wird die Actions-Logik aus der Datenbank entnommen **(4)** und an den Load-Balancer weitergeleitet. Der Load-Balancer ist Bestandteil des Controllers und enthält eine Sicht auf alle *Invoker*. Die Aufgabe der Invoker ist es, die Action am Ende auszuführen. In der Theorie würde der Load-Balancer sich einen freien Invoker suchen und ihm die Action-Logik übergeben. Jedoch kann es zu unerwarteten Störungen kommen oder es lässt sich kein freier Invoker finden. Deshalb sendet der Load-Balancer eine Nachricht an Apache Kafka[4] **(5)**. Kafka speichert seine einkommenden Nachrichten, sodass auch

[2] `https://www.nginx.com/`
[3] `https://couchdb.apache.org/`
[4] `https://kafka.apache.org/`

bei einem Systemabsturz alle Nachrichten vorhanden sind. Diese enthält welcher Invoker die Action ausführen soll, auch wenn dieser momentan beschäftigt ist. Kafka verarbeitet die Nachricht und sendet eine Empfangsbestätigung mit der `activationId` zurück. Mithilfe der `activationId` lässt sich später das Resultat der Action abfragen. Wenn der Invoker aus der Nachricht frei ist, wird die Action ausgeführt. Dafür wird ein Docker Container gestartet **(6)**, welcher eine passende Laufzeitumgebung für die Action zur Verfügung stellt. Die Action-Logik wird in den Container geladen und ausgeführt. Das Resultat wird entnommen und der Container wird heruntergefahren. Das Resultat sowie die `activationId` werden in die *activations-Datenbank* **(7)** gespeichert. Nun kann, wie bereits zu Beginn, über den Endpunkt und der neugewonnenen `activationId` das Resultat aus der `activations`-Datenbank abgerufen werden [Apa19].

2.2 Sichere Umgebungen für Serverless

Für Cloud-Provider ist es wichtig, dass die einzelnen Anwendungen ihrer zahlreichen Kunden sicher und abgekapselt voneinander laufen, sodass niemand unbefugten Zugriff auf andere Anwendungen hat [Ser18]. In diesem Abschnitt werden Virtual-Machines und Container näher erläutert, um ein Verständnis zu bekommen, wie die einzelnen Anwendungen untereinander abgesichert sind und ob sie für serverlose Anwendungen gut geeignet sind.

2.2.1 Virtual-Machine

Bei einer Virtual-Machine handelt es sich um eine virtuelle Umgebung, mit eigenen virtuellen Ressourcen, wie unter anderem CPU und Speicher. Ein *Virtual Machine Monitor* (VMM), auch *Hypervisor* genannt, ist ein Softwarestück, das dafür zuständig ist die virtuellen Umgebungen einschließlich seiner virtuellen Ressourcen zu verwalten und Instruktionen auf die physischen Ressourcen abzubilden. Der Hypervisor reserviert physische Ressourcen und plant die Zuweisung an die einzelnen VMs. Er ist der Kommunikationskanal zwischen den virtuellen Umgebungen und der Hardware. Ein großer Vorteil der Virtualisierung von Hardwareressourcen ist, dass jede Virtual-Machine ihr eigenes Betriebssystem benutzen kann. Somit können auf einer Maschine mehrere Betriebssysteme laufen, die vollständig voneinander abgekapselt sind [Red20d] [Red20c]. Es werden zwei Arten von Hypervisor

unterschieden.

Typ 1 Hypervisor Ein nativer Hypervisor, auch *Typ 1 Hypervisor* genannt, läuft direkt auf der physischen Hardware und verwaltet die virtuellen Umgebungen und ihre Betriebssysteme. Der Hypervisor benötigt selbst kein zugrundeliegendes Betriebssystem. Gängige Typ 1 Hypervisor sind unter anderem Microsoft Hyper-V[5] und die Kernel-basierte Virtual-Machine (*KVM*). Die KVM [Red20e] ist in den Linux-Kernel integriert und wandelt das Linux-Betriebssystem in einen Typ 1 Hypervisor. Sie unterstützt Erweiterungen für Hardware-Virtualisierung. Diese Erweiterungen wurden eingeführt, um die Virtualisierung und die Entwicklung von Hypervisor zu beschleunigen. KVM kann auch mit einem weiteren Typ 2 Hypervisor benutzt werden, um diesen zu unterstützen und zu beschleunigen [IBM20].

Typ 2 Hypervisor Ein hosted Hypervisor, auch *Typ 2 Hypervisor* genannt, läuft als übliche Applikation auf dem Host-Betriebssystem. Der hosted Hypervisor kommuniziert mit dem Host-Betriebssystem und hat keinen direkten Zugriff auf die physischen Ressourcen. Erst in Zusammenarbeit mit ihm werden die Instruktionen an die physischen Hardwareressourcen weitergeleitet. Somit laufen die virtuellen Umgebungen auf dem Host-Betriebssystem, haben aber jeweils ein eigenes Betriebssystem [Red20c]. Im Vergleich zu einem nativen Hypervisor ist das Erstellen einer VM mit einem hosted Hypervisor ohne großen Aufwand möglich. Ein Nachteil ist jedoch, dass sie langsamer und ineffizienter sind, da sie nicht direkt mit der Hardware kommunizieren [IBM20].

2.2.2 Container

Virtualisierung ist auch mithilfe von Linux Containern möglich, was auch Containerisierung (englisch *Containerization*) genannt wird. Diese enthalten isolierte Prozesse. Container teilen sich den Host-Kernel. Durch eine Konfigurationsdatei (*Image*) werden alle nötigen Ressourcen und Dateien definiert, die für das Ausführen der Prozesse benötigt werden. Dadurch spart man sich ein komplettes Betriebssystem zu laden, wodurch sie den Vorteil haben, dass sie kompakter und sparsamer sind. Entwicklern stehen eine eindeutig definierte Laufzeitumgebung zur Verfügung, worauf sie ihre Anwendungen testen können [Red20f]. Container

[5] `https://docs.microsoft.com/de-de/virtualization/hyper-v-on-windows/`

bauen auf drei Komponenten des Linux-Kernels auf, den Namensräumen (*namespaces*), den Kontrollgruppen (*cgroups*) und den Filtern (*seccomp-bpf*).

Mithilfe von namespaces stellt der Kernel eine Möglichkeit zur Isolation für Prozesse bereit. Mit ihnen können globale Ressourcen abstrahiert werden. Prozesse innerhalb des namespaces sehen nur ihre isolierte lokale Ausprägung der Ressource [Ubu20]. Dadurch können mehrere Container gleichzeitig auf die gleiche Ressource zugreifen [Red20a]. Es existieren unterschiedliche namespaces, wie beispielsweise *mount namespaces*, *network namespaces* und *pid namespaces*. Durch mount namespaces kann die Sicht auf das Dateisystem (englisch *filesystem*) für Prozesse beschränkt werden. Mit network namespaces lassen sich IP-Adressen, Ports und Routing Tabellen isolieren. PID (Prozess-ID) namespaces teilen den Identifikationsbereich. Ein Prozess hat dadurch zwei PIDs. Eine Identifikationsnummer auf dem Host-Betriebssystem und eine innerhalb des namespaces, welche sich beide unterscheiden können [Ker13].

Die cgroups erfüllen zwei Aufgaben. Sie erlauben das Gruppieren von Prozessen und können dynamisch Ressourcen für Gruppierungen begrenzen. Begrenzt werden können beispielsweise CPU Zeit und Speicher eines Prozesses. Möglich wird dies durch *subsystems*, auch *resource controller* genannt. Subsystems sind Bestandteil des Kernels und erlauben das Ändern von Prozesseigenschaften und Prozessverhalten in einer cgroups Gruppierung [Red20a].

Für die Sicherheit von Containerinstanzen ist seccomp-bpf verantwortlich. Es filtert und limitiert Systemaufrufe (*syscalls*) von Prozessen, sowie die mitgelieferten Parameter der Systemaufrufe. Seccomp-bpf ist ein wichtiger Baustein in der Isolation von Containern. Es verkleinert die Angriffsfläche des Kernels. Jedoch benutzen Container einen Kernel, sodass bei Ausnutzung von Sicherheitslücken des Kernels alle Container angreifbar sind [Ser18].

2.2.3 Ideale Virtualisierung für Serverless

Durch Serverless ist es möglich, dass eine Applikation in kürzester Zeit auf hunderte Instanzen hochskaliert. Vor allem bei FaaS Diensten, wo eine Funktion nur eine ganz spezifische Aufgabe tätigt, kann es vorkommen, dass eine Anwendung aus mehreren hunderten von kleinen Funktionen aufgebaut ist. Diese können alle separat skaliert werden. Traditionelle VMs bieten eine robuste Isolation, denn jede VMs hat einen eigenen Guest-Kernel und ist durch den Hypervisor abgekapselt vom Host-Kernel und anderen VMs. Sie haben jedoch einen

großen Startaufwand und sind für Serverless zu ressourcenintensiv [Ser18]. Container dagegen haben einen geringeren Startaufwand. Die Laufzeitumgebung der Anwendung lässt sich genaustens definieren, sodass keine unnötigen Ressourcen verschwendet werden. Aber Container teilen sich den Host-Kernel, sodass eine Sicherheitslücke im Kernel die Container angreifbar macht. Die ideale Virtualisierungsmethode für Serverless sollte gewisse Eigenschaften aufweisen [Ser18].

Isolation Mehrere Anwendungen, die gleichzeitig auf derselben Hardware laufen, sollten abgesichert und isoliert voneinander sein. Die Wahrscheinlichkeit auf Sicherheitslücken und unbefugten Zugriff soll verringert werden [Ser18].

Aufwand und Dichte Anders als traditionelle VMs sollte die ideale Virtualisierungsmethode Anwendungen nicht zusätzlich verlangsamen. Die sichere Umgebung sollte kompakt sein [Ser18].

Performance Die Leistung der Anwendungen sollten nicht beeinträchtigt werden. Zusätzlich sollte die Leistung auch konstant und unabhängig von den einzelnen Instanzen sein [Ser18].

Kompatibilität Anwendungen sollten die Möglich haben, ohne Einschränkungen beliebige Bibliotheken und Dateien zu verwenden. Anwendungen sollten nicht extra angepasst werden müssen, um von der Sicherheit und Isolation zu profitieren [Ser18].

Schnelles Umschalten Die Eigenschaft der Skalierung von Serverless sollte durch die Virtualisierungsmethode nicht beeinträchtigt werden. Mehrere Instanzen der Anwendung sollten zügig hoch- und runtergefahren werden können [Ser18].

Soft Allocation Da FaaS und Serverless Container Dienste auch nach Speicher, CPU und anderen Ressourcen konfiguriert werden können, muss die Virtualisierungsmethode eine Überprovisionierung diese Ressourcen erlauben. Jedoch sollte eine Instanz nur die nötigen Ressourcen in Anspruch nehmen [Ser18].

2.2.4 AWS Firecracker

Nach diesem theoretischen Ideal hat AWS mit Firecracker [Ser18] eine eigene Virtualisierungsmethode aufgebaut, um ihre serverlosen Dienste zu verbessern. Firecracker ist ein Typ 2 Hypervisor optimiert für Serverless und entwickelt in Rust[6]. Mithilfe der KVM im Linux-Kernel erzeugt es kompakte Virtual-Machines (*MicroVMs*). Möglich wird dies durch das Entfernen von Funktionen üblicher Hypervisor. Es werden nur wenige Geräte unterstützt. Geräte-Treiber für USB und GPUs wurden entfernt. Firecracker läuft als Prozess, der durch seccomp, cgroups und namespaces eingeschlossen ist und nur begrenzten Zugriff auf die Hardware und das Dateisystem hat. AWS hat Firecracker auch eine REST-API beigelegt. Mit ihr können die einzelnen MicroVMs besser gesteuert und verwaltet werden [Ser18].

2.2.5 Google gVisor

Auch Google hat mit gVisor bei der Virtualisierung ihre Infrastruktur nachgebessert. Google gVisor ist ein Applikationskernel [Goo20f]. Es reimplementiert viele Aspekte des Linux-Kernel in Go[7]. Es läuft als Prozess und nimmt den Platz des Guest-Kernels ein auf dem übliche Anwendungen laufen können, unter anderem auch isolierte Container. Ähnlich wie seccomp fängt gVisor alle Systemaufrufe der Anwendungen ab und versucht viele diese Aufrufe selbst zu bearbeiten. Nur wenige werden an den Host-Kernel weitergeleitet. Da gVisor nicht alle Systemaufrufe implementiert kann es für bestimmte Anwendungen zu Kompatibilitätsproblemen führen [Goo20f].

[6] `https://www.rust-lang.org`
[7] `https://golang.org`

Kapitel 3

Entwicklung

In diesem Kapitel wird die serverlose Anwendung entwickelt. Sie soll Benutzer ermöglichen Quellcode von Java Konsolenanwendungen und Gradle-Projekten in einer sicheren Umgebung zu bauen, zu starten und zu testen. Durch die Entwicklung sollen auch Vor- und Nachteil von Serverless besser eingeschätzt werden. Dieses Kapitel beinhaltet die Architektur und Implementierung der Anwendung. Davor werden Dienste und Werkzeuge vorgestellt, die für die Realisierung verwendet wurden.

Die serverlose Anwendung ermöglicht es einzelne Javadateien auszuführen. Werden zusätzliche JUnit5[1] Testdateien mitgeliefert, können auch die Tests gestartet werden. Das Resultat der Tests wird wieder an den Benutzer zurückgeliefert. Der Dienst kann auch von Studenten und Korrektoren benutzt werden. Da die Aufgaben des Moduls Programmierung der HHU Gradle benutzen, können auch Gradle-Projekte ausgeführt und getestet werden. Jedoch verwenden die Modulaufgaben keine JUnit Tests, sodass sie angepasst werden müssen. Außerdem wird für das Ausführen der Gradle-Projekte das Applikations-Plugin[2] vorausgesetzt. Die Anwendung kann vollständig durch die Kommandozeile angesprochen werden, denn sie stellt eine REST-API bereit. Jedoch wird auch ein Endpunkt zur Verfügung gestellt, der eine Webseite zurückliefert. Das Layout der Seite richtet sich grob nach dem Styleguide der Modularen-Online-Plattform für Studierende (MOPS)[3] der HHU. In ihr wird das Frontend eingebunden. Es kann von den Studenten als Entwicklungsumgebung benutzt werden, um kleinere Java-Programme zu entwickeln und anschließend zu testen. Jedoch liegt der Fokus

[1] `https://junit.org/junit5/docs/current/user-guide/`
[2] `https://docs.gradle.org/current/userguide/application_plugin.html`
[3] `https://mops.style`

der Arbeit auf der serverlosen Backend-Anwendung.

3.1 Dienste und Tools

Serverloser Dienst Bei dem serverlosen Dienst wird auf Google Cloud Run zurückgegriffen. Es bietet mehr Freiheit als FaaS Dienste der Cloud-Provider, da mittels Dockerfile die Laufzeitumgebung der Anwendung genaustens definiert und konfiguriert werden kann [Goo20a]. Auch Open-Source Plattformen wurden in Erwägung gezogen, jedoch aufgrund der etwas komplexeren Bereitstellung nicht ausgewählt.

Webserver Die Anwendung selbst wird mit dem Web-Framework Flask [The20c] und Python entwickelt. Flask ermöglicht die schnelle und unkomplizierte Bereitstellung einer REST-API. Sollte jedoch eine ähnliche Anwendung in Zukunft für Studenten und Korrektoren bereitgestellt werden soll, würden die üblichen Werkzeuge und Bibliotheken der Modularen-Plattform die Implementierung vereinfachen.

Frontend Web-Framework Für die grafische Entwicklungsumgebung wird mit Vue.js[4] ebenfalls ein Web-Framework verwendet. Das Frontend wird in der Programmiersprache JavaScript geschrieben. Es dient als erweiterte Funktionalität und gehört nicht zum Kern der in dieser Arbeit entwickelten serverlosen Anwendung. Jedoch bietet sie eine bequemere Interaktion mit der Anwendung als die Kommandozeile. Das Frontend wird mit Firebase Hosting, einem weiteren Google Dienst, bereitgestellt. Bei Firebase Hosting kann das sichere Hosten von statischen Webseiten mit wenigen Befehlen bewältigt werden [Goo20e].

3.2 Architektur

Die Architektur in Abbildung 3.1 besteht auf zwei Komponenten. Die Cloud Run Containerinstanzen bilden das serverlose Backend. Es werden Endpunkte bereitgestellt, die mit

[4] `https://vuejs.org`

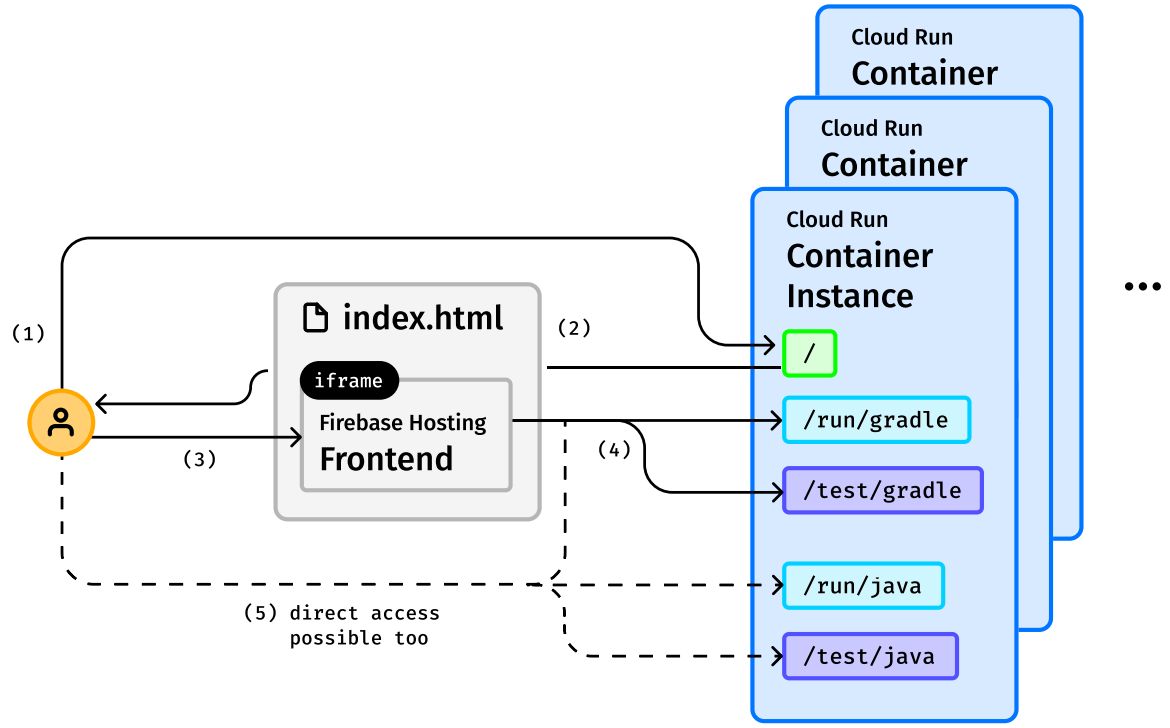

Abbildung 3.1: Architektur und Interaktion der Anwendung

HTTP-Anfragen[5] angesprochen werden können. Rufen Benutzer mittels einer HTTP-GET Anfrage den Root-Endpunkt auf **(1)** so wird die `index.html` Datei zurückgesendet **(2)**. Sie enthält die grafische Benutzeroberfläche. Das Layout richtet sich an dem Styleguide des MOPS. Durch ein HTML-iframe Element[6] wird das eigentliche Frontend eingebunden. Das Frontend dient als Entwicklungsumgebung, um Programme zu entwickeln. Die Aufteilung von Frontend und Backend ermöglicht eine einfache Verwaltung und Erweiterung beider Komponenten. Durch Interaktion mit der Entwicklungsumgebung **(3)** lassen sich einige der restlichen Endpunkte **(4)** aufrufen. Da das Backend eine REST-API bereitstellt, ist auch ein direkter Zugriff **(5)** auf alle Endpunkte über die Kommandozeile durch HTTP-POST Anfragen möglich.

[5] `https://tools.ietf.org/html/rfc2616`
[6] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe`

3.3 Serverless Backend

Das Backend wird durch Endpunkte angesprochen. Der Java Quellcode kann mittels HTTP-POST Anfrage an das Backend gesendet werden und wird dort durch das Java Development Kit beziehungsweise Gradle ausgeführt oder getestet.

Java Development Kit Für das Kompilieren und anschließende Ausführen der Javadateien wird Zulu OpenJDK 11 benutzt, da die Installation auf Alpine Linux-Betriebssystemen[7] verständlich dokumentiert ist [Azu20]. Das JDK stellt die zwei Befehle `javac` und `java` bereit. Übergeben Studenten oder Korrektoren ihren Java Quellcode mittels HTTP-POST Anfrage an das Backend, so muss es in der Lage sein, den Befehl `javac` aufzurufen. Denn dadurch können die Javadateien kompiliert werden. Nach der erfolgreichen Kompilation können durch den `java` Befehl die kompilierten Dateien ausgeführt werden. Um das Backend dies zu Ermöglichen muss das JDK in die Laufzeitumgebung eingepackt werden, damit die Anwendung während ihrer Ausführung die zwei Befehle aufrufen kann.

```
1  # Install Zulu11
2  RUN apk --no-cache add zulu11-jdk-headless
3  ENV JAVA_HOME=/usr/lib/jvm/default-jvm
4  ENV PATH="$JAVA_HOME/bin:${PATH}"
5
6  # Check if two commands are available
7  RUN which javac
8  RUN which java
```

Listing 3.1: Ausschnitt aus der Dockerfile. Herunterladen und Einrichten des JDK.

In Auflistung 3.1 ist ein Ausschnitt aus der Dockerfile zusehen, welche die Laufzeitumgebung der Cloud Run Containerinstanzen definiert. Das JDK wird heruntergeladen und installiert **(Zeile 2)**. Außerdem werden wichtige Umgebungsvariablen gesetzt **(Zeile 3, 4)**. Abschließend wird geprüft ob die beiden Befehle zur Verfügung stehen **(Zeile 7, 8)**.

[7] `https://alpinelinux.org`

JUnit Console Launcher Da der Endpunkt `/test/java` einzelne Javadateien und keine vollständigen Gradle-Projekte entgegennimmt, müssen die mitgelieferten Tests manuell gestartet werden. Es wird der JUnit Console Launcher benutzt. Mit diesem Kommandozeilen-Programm können JUnit Tests ausgeführt werden. Hierfür wird das eigenständige Console Launcher Jar-Archiv[8] beim Ausführen des `java` Befehls mit eingebunden [The20a]. Um dies zu ermöglichen, muss es der Anwendung zur Laufzeit bereitstehen. Deshalb wird auch das Jar-Archiv in die Laufzeitumgebung verpackt.

Gradle Damit Gradle-Projekte gestartet und getestet werden können wird, wie in Auflistung 3.2 zu sehen, Gradle in die Laufzeitumgebung eingepackt. Es reicht aus Gradle herunterzuladen **(Zeile 2)** und zu entpacken **(Zeile 3)**. Somit hat die Anwendung während der Laufzeit Zugriff auf den `gradle` Befehl. Die Anwendung braucht nur noch den `gradle test` oder `gradle run` Befehl aufrufen und das Gradle-Projekt kann getestet oder ausgeführt werden [Gra20a].

```
1  # Get Gradle distribution zip
2  RUN wget https://downloads.gradle-dn.com/distributions/gradle
       -6.3-bin.zip
3  RUN unzip gradle-6.3-bin.zip
```

Listing 3.2: Ausschnitt aus der Dockerfile. Herunterladen und Entpacken von Gradle.

3.3.1 Endpunkte

Die Endpunkte sind mit `/run` und `/test` in zwei Gruppen aufgeteilt. Die erste Gruppe ist für das Kompilieren und anschließende Ausführen von Java Quellcode zuständig. Die zweite Gruppe ist für das Ausführen von JUnit-Tests verantwortlich. Die Anwendung sendet das Resultat der Ausführung als eine HTML-Datei zurück. Für die Gradle Endpunkte kann alternativ mit dem zusätzlichen Query-String[9] *?return=json* auch ein JSON-Objekt zurückgesendet werden. Außerdem kann mit einem Query-String der Form *?args1=Value* Argumente an die JDK und Gradle Befehle übermittelt werden.

[8] `https://de.wikipedia.org/wiki/Java_Archive`
[9] `https://wiki.selfhtml.org/wiki/Query_String`

Ausführen und Testen einzelner Javadateien Ein beispielhafter Aufruf des `/run/java` Endpunktes mit curl[10] ist in Auflistung 3.3 dargestellt. Die einzelnen Javadateien werden im Entity-Body[11] der Anfrage mitgeliefert **(Zeile 2, 3)**. Zusätzlich muss der Name der Klasse angegeben werden, in der die Hauptmethode definiert ist **(Zeile 1)**. Dieser Endpunkt soll den Benutzern ermöglichen ihren Quellcode schnell und unkompliziert auszuführen.

```
1  curl -F main_file=MyMainClass \
2    -F file=@MyMainClass.java \
3    -F file=@Calculator.java \
4    https://ba-serverless-testing-t6e6p4w6oa-ew.a.run.app/run/java
```

Listing 3.3: HTTP-POST Anfrage an `/run/java` um einzelne Javadateien zu kompilieren und auszuführen.

Für das Ausführen von JUnit-Tests müssen die Testdateien, wie in Auflistung 3.4, mitgeliefert werden **(Zeile 1)**. Zusätzlich muss der Endpunkt zu `/test/java` umgeändert werden **(Zeile 2)**.

```
1  curl -F file=@Calculator.java -F file=@CalculatorTest.java \
2    https://ba-serverless-testing-t6e6p4w6oa-ew.a.run.app/test/
     java
```

Listing 3.4: HTTP-POST Anfrage an `/test/java` um einzelne JUnit-Tests auszuführen.

Ausführen und Testen eines Gradle-Projekts Um ein Gradle-Projekt über dem Endpunkt `run/gradle` auszuführen, muss das Projekt vorher in ein Zip-Archiv komprimiert werden. Erst dann kann es der HTTP-POST Anfrage beigelegt **(Zeile 1)** und an das Backend versendet werden.

```
1  curl -F file=@gradle_project.zip \
2    https://ba-serverless-testing-t6e6p4w6oa-ew.a.run.app/run/
     gradle
```

Listing 3.5: HTTP-POST Anfrage an `/run/gradle` um ein Gradle-Projekt auszuführen.

[10] `https://curl.haxx.se`
[11] `https://tools.ietf.org/html/rfc2616#section-7.2`

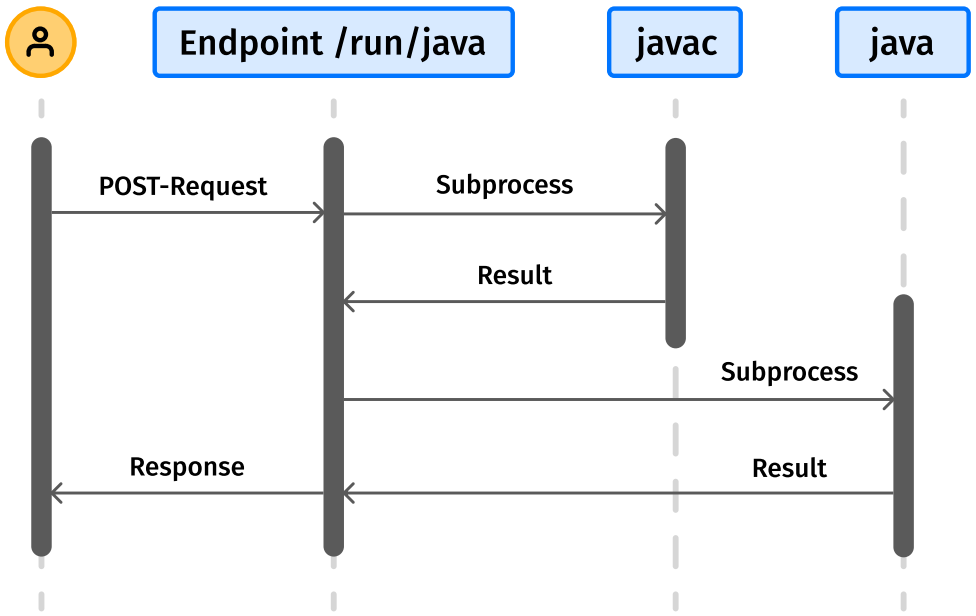

Abbildung 3.2: Ablauf einer Anfrage an /run/java Endpunkt.

Sollen enthaltene JUnit-Tests ausgeführt werden, muss nur der Endpunkt des Befehls zu `/test/gradle` abgeändert werden.

Einbinden des Frontends Es gibt noch einen weiteren Endpunkt, welcher nicht primär für das Testen und Ausführen von Java Quellcode zuständig ist. Durch eine GET-Anfrage an den Root-Endpunkt wird eine grafische Benutzeroberfläche zurückgesendet. Vor Beginn der Ausarbeitung war als eine mögliche Erweiterung definiert, dass die serverlose Anwendung an das MOPS System der HHU angebunden werden soll. Da das Einbinden an das System nicht das zentrale Thema der Arbeit ist, wurde nur das Layout nachgebaut. Jedoch wurde zusätzlich eine Entwicklungsumgebung gebaut. Die Entwicklungsumgebung ist das eigentliche Frontend und wird durch einen HTML-iframe in das nachgebaute MOPS Layout eingebunden. Somit kann sie auch vollkommen eigenständig benutzt werden.

3.3.2 Ablauf

In Abbildung 3.2 ist der Ablauf einer Anfrage an den Endpunkt `/run/java` abgebildet. Nachdem das Backend die Anfrage entgegennimmt, werden die enthaltenen Dateien in einem temporären Bereich gesichert. Von dort aus wird durch das Subprocess-Modul von Python

der `javac` Befehl aufgerufen. Mit dem Subprocess-Modul lassen sich zusätzliche Prozesse starten und verwalten [Pyt20a]. Wenn das Kompilieren des Quellcodes erfolgreich ist, wird mit dem `java` Befehl das Programm gestartet. Die Standardausgabe des Programms wird eingefangen und dem Benutzer als Antwort zurückgesendet. Für die beiden Gradle Endpunkte ist der Ablauf ähnlich. Es müssen die beiden JDK Befehle durch den `gradle` Befehl ausgetauscht werden. Soll das übermittelte Gradle-Projekt gebaut und ausgeführt werden, wird der `gradle run` Befehl durch das Subprocess-Modul aufgerufen. Soll es gebaut und getestet werden, wird der `gradle test` Befehl aufgerufen.

3.4 Frontend

Für das Erstellen der Entwicklungsumgebung wurde das Open Source Framework *Vue.js* und der Texteditor *Monaco Editor* verwendet. Monaco Editor wird unter anderem bei VS Code verwendet. Es stellt einige übliche Funktionen, wie Syntax-Highlighting [Mic20] bereit. Jedoch wurde für eine angenehmere Benutzererfahrung eine zusätzliche Projektansicht entwickelt, sodass Benutzer zwischen unterschiedlichen Dateien wechseln können. Das Frontend setzt sich aus vier Komponenten zusammen. Die erste Komponente beinhaltet die Buttons für das Senden des Gradle-Projekts an das Backend. Die zweite Komponente ist die Projektansicht. Das Projekt wird als Baumstruktur dargestellt. Außerdem können Dateien und Ordner erstellt werden. Die dritte Komponente ist der Editor. Hier können Benutzer ihren Quellcode schreiben. Nach dem Ausführen oder Testen wird die Antwort des Backends in der Konsole, der vierten Komponente, ausgegeben. Das Frontend stellt den Benutzern das Grundgerüst eines Gradle-Projekts bereit. Benutzer können durch das Erstellen von neuen Dateien und Ordnern ihre Anwendungslogik in das Projekt einfügen und das Gerüst ausbauen. Das Frontend bietet die nötigsten Funktionen, um Benutzern zu ermöglichen ihren Quellcode zu starten und zu testen. Es ist ein erster Ansatz einer Entwicklungsumgebung und ist noch sehr ausbaufähig. Es fehlen noch wichtige Eigenschaften von Webanwendungen, wie das temporäre Speichern des eingetippten Quellcodes nach der Aktualisierung der Webseite.

Das Frontend wird durch Google Firebase Hosting, zur Verfügung gestellt. Mit wenigen Befehlen ermöglicht der Dienst das unkomplizierte Bereitstellen von statische Webseiten.

Unter anderem ist SSL[12] vorkonfiguriert, sodass beim Besuchen der Webseite eine sichere Verbindung aufgebaut wird [Goo20e].

[12] https://tools.ietf.org/html/rfc6101

Kapitel 4

Probleme und Entscheidungen

Während der Entwicklung von Backend und Frontend sind einige Probleme aufgetreten, die zum Umdenken einiger Funktionen in beiden Komponenten verneigt haben. Außerdem wurden bei der Architektur und Funktionsweise der Anwendung kritische Entscheidungen getroffen.

4.1 JUnit Tests

Für das Testen von einzelnen Javadateien oder Gradle-Projekten werden JUnit5 Tests vorausgesetzt. JUnit Tests lassen sie sich leicht in ein Java-Projekt integrieren und ausführen. Für das Ausführen von Tests der Übungsaufgaben des Moduls Programmierung hat die Entscheidung jedoch einige Nachteile. Neben einer älteren Gradle und Java Version verwenden die Übungsaufgaben aus dem Wintersemester 2018 keine JUnit Tests, sondern es werden mehrere Gradle Tasks[1] erstellt. Es werden nicht die Rückgabewerte und Zustände der Funktionen getestet, sondern der Zustand der Standardausgabe nach dem Ausführen des Programms. Die Übungsaufgaben müssten also angepasst werden, um von der entwickelten serverlosen Anwendung getestet werden zu können. Jedoch wird durch JUnit die Anwendung auch außerhalb der Universität nutzbar und leichter erweiterbar.

[1] `https://docs.gradle.org/current/dsl/org.gradle.api.Task.html`

4.2 Gradle Wrapper

Um ein Gradle-Projekte auszuführen und zu testen, gibt es unter anderem zwei Möglichkeiten. Einem Gradle-Projekt kann der Gradle-Wrapper (Wrapper) beigelegt sein. Dieser definiert welche Version von Gradle benutzt werden soll und lädt sich Gradle für das Bauen des Projekts selbstständig herunter. Normalerweise ist es empfohlen den Wrapper zu benutzen, da dieser das Bauen des Projektes standardisiert und somit weniger Fehler entstehen [Gra20b]. Jedoch hat sich beim ersten Ansatz der Implementierung herausgestellt, dass der Wrapper für das serverlose Backend etwas ungünstig ist. Bei einer Anfrage, in der eine neue Cloud Run Containerinstanz gestartet wird, verschwendet das Herunterladen von Gradle unnötig Zeit und Rechenressourcen, denn die verbrauchte Zeit und der Rechenaufwand werden bei serverlosen Diensten in Zahlung gestellt [Bal+17]. Zusätzlich müssen Benutzer warten bis die Anwendung mit dem eigentlichen Ausführen und Testen des Gradle-Projekts weitermacht. Besser ist es Gradle in die Laufzeitumgebung einzubinden. Dadurch muss Gradle nicht immer wieder heruntergeladen werden. Dies hat jedoch den Nachteil, dass die auszuführenden Programme der Benutzer kompatibel mit der verwendeten Gradle Version sein müssen.

4.3 Google Cloud Run Konfiguration

Google Cloud Run ermöglicht die Konfiguration von Ressourcen der Containerinstanzen. Es können unter anderem die Anzahl an zugewiesenen CPUs [Goo20d] und Speicher [Goo20c] angepasst werden. Den Containerinstanzen werden zwei vCPUs sowie zwei Gibibyte[2] an Speicher zugewiesenen, da bei einer geringeren Einstellung Gradle nicht genug Speicher zur Verfügung steht und Anfragen nicht bearbeitet werden. Der Nachteil ist, dass der Dienst dadurch teurer wird, da die Konfiguration der Ressourcen in den Kosten mit einberechnet wird [Goo20b].

[2] https://en.wikipedia.org/wiki/Gibibyte

4.4 Entfernung von Endpunkte

Während der Entwicklung des Frontends wurde schnell ersichtlich, dass die Erstellung eines vollständigen Gradle-Projekts nicht immer benutzerfreundlich ist. In der Projektansicht verschwenden die Dateien und Ordner viel Platz. Außerdem können Benutzer nicht frei über ihre Ordnerstruktur entscheiden. Deshalb sollten die zusätzlichen Endpunkte `/run/zip` und `/test/zip` bereitgestellt werden. Sie sollten das Ausführen und Testen von Java-Projekten mit beliebiger Ordnerstruktur ermöglichen. Während der Endpunkt `/run/zip` sich leicht implementieren ließ, kamen beim Kompilieren und Ausführen der JUnit Tests Probleme hervor. Der erste Ansatz war es alle Ordner und Unterordner des Projekts zu durchlaufen, um alle Dateien mit der Dateiendung *.java* einzusammeln. Dateien mit *Test* im Namen wurden als JUnit Testdateien gewertet. Jedoch ist dieser Ansatz auch nicht benutzerfreundlich. Während Benutzer bei einem Gradle-Projekt keine freie Ordnerstruktur festlegen können, können sie bei dieser Methode nicht frei über Dateinamen entscheiden.

Eine bessere Lösung wäre es durch den Inhalt der Dateien zu ermitteln, welche von ihnen JUnit Tests enthalten. Aufgrund der Unvollständigkeit des `/test/zip` Endpunkts wurde der implementierte `/run/zip` Endpunkt entfernt. Dadurch müssen Benutzer im Frontend zwar ein vollständiges Gradle-Projekt erstellen, um ihre Anwendung zu testen, jedoch wird die Benutzererfahrung klar und einheitlich. Als zusätzliche Hilfe wird den Benutzern im Frontend das Grundgerüst eines Gradle-Projekts automatisch bereitgestellt.

4.5 Zustandslosigkeit

Bereits vor Beginn der Arbeit war eine mögliche Schwierigkeit der Anwendung bekannt. Da die Anwendung einen serverlosen Dienst benutzt, kann mit den auszuführenden Programmen nicht interagiert werden. Die Containerinstanzen sind zustandslos. Anfragen werden nicht immer von der gleichen Instanz bearbeitet. Alle Programme, die vom Benutzer eine Eingabe einfordern, können nicht erfolgreich ausgeführt werden. Alternativ kann der Programmzustand in eine Datenbank ausgelagert werden und bei der nächsten Interaktion eingelesen werden. Jedoch wurde beschlossen die Funktion nicht in dieser Arbeit zu implementieren. Sie kann als mögliche Erweiterung entwickelt werden.

Kapitel 5

Evaluation

In diesem Kapitel wird geprüft, ob die serverlose Anwendung für das Ausführen und Testen von einfachen Java Programmen verwendet werden kann oder wo noch nachgebessert werden muss.

Für das Testen und die Analyse der serverlosen Anwendung werden zwei unterschiedliche Gradle-Projekte und drei Javadateien verwendet. Die Gradle-Projekte verwenden das JUnit5 Starter Gradle als Vorlage [The20b]. Die Messungen selbst werden primär durch das Ausführen von entwickelten Skripten ermittelt, jedoch zum Teil manuell ausgewertet und protokolliert.

Das erste Beispielprojekt ist eine einfache Hello World Applikation. Das Programm gibt den Text *„Hello World“* in der Standardausgabe aus. Bei dem zweiten Projekt wird die Fibonacci-Folge[1] berechnet. Durch die beigelegten JUnit Tests wird stichprobenartig geprüft, ob die Folge richtig berechnet wurde. Während die beiden Beispielprojekte für das Testen der Gradle Endpunkte zuständig sind, werden die Javadateien für das Analysieren der Java Endpunkte verwendet.

[1] `https://de.wikipedia.org/wiki/Fibonacci-Folge`

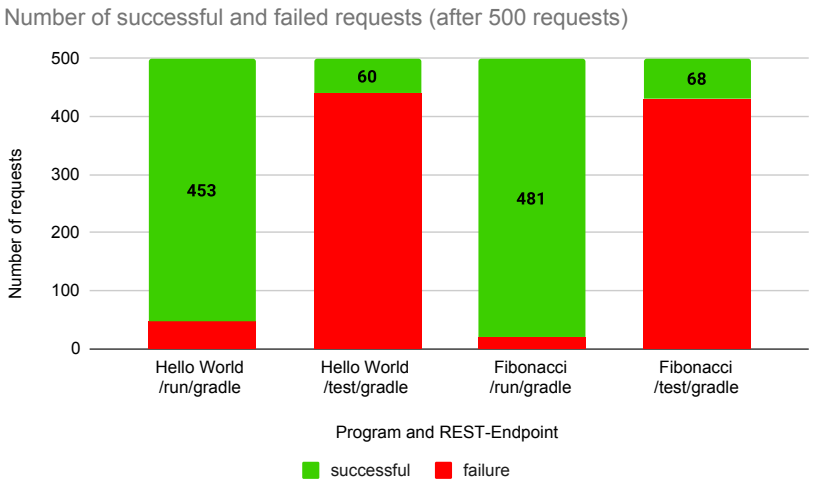

Abbildung 5.1: Anzahl der erfolgreichen und fehlerhaften Antworten nach insgesamt 500 Anfragen (bei 10 Anfragen pro Sekunde)

5.1 Google Cloud Run Konfiguration

Für die Analyse werden den Cloud Run Containerinstanzen zwei vCPUs sowie zwei Gibibyte Speicher zugewiesen. Anfragen werden nach maximal 300 Sekunden automatisch abgebrochen. Die maximale Anzahl an gleichzeitigen Anfragen, welche eine Containerinstanz entgegennehmen kann, wird auf 80 eingestellt. Außerdem wird die maximale Anzahl an Containerinstanzen auf 1000 begrenzt.

5.2 Probleme der Gradle Endpunkte

Da die Anwendung auf dem lokalen System bei vereinzelten Tests des `/test/gradle` Endpunkts oft mehrere Sekunden für die Bearbeitung brauchte, wurde in der ersten Testreihe schlicht die Anzahl an erfolgreichen und fehlerhaften Antworten betrachtet. Die Endpunkte `/run/gradle` und `/test/gradle` beider Beispielprojekte wurden insgesamt 500 Mal aufgerufen. Es wurden 10 Anfragen pro Sekunde gesendet. In Abbildung 5.1 ist deutlich zu sehen, dass die `/test/gradle` Endpunkte tatsächlich noch große Probleme aufweisen. Während die beiden `/run/gradle` Endpunkte die Mehrheit der Anfragen erfolgreich beantworten, werden mehr als 85% der Anfragen an die Gradle Test-Endpunkte nicht er-

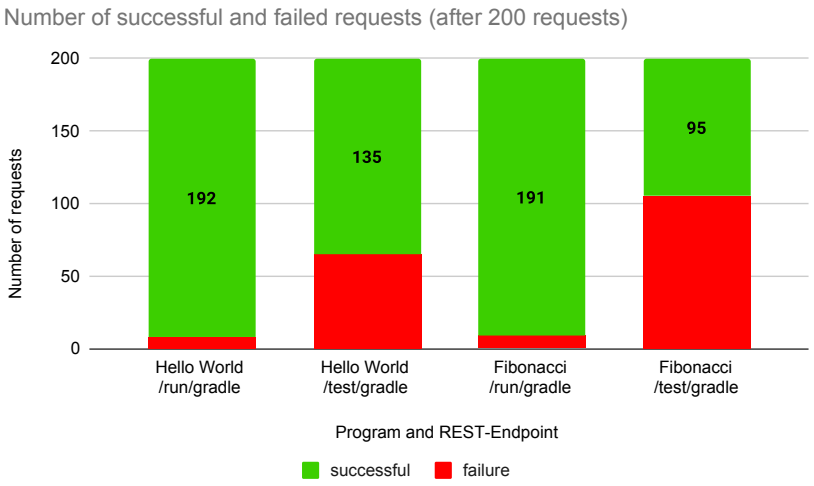

Abbildung 5.2: Anzahl der erfolgreichen und fehlerhaften Antworten nach insgesamt 200 Anfragen (bei 10 Anfragen pro Sekunde)

folgreich beantwortet. Es wird größtenteils die wenig aussagende Fehlermeldung *„Service Unavailable“* als Antwort zurückgesendet, was die Fehlersuche erschwert.

In erster Annahme wurde vermutetet, dass der serverlose Dienst Schwierigkeiten hat die Anwendung zu Skalieren. Um zu Prüfen, ob eine Minderung der Gesamtaufrufe eine Verbesserung herbeiführt, wurden die Endpunkte bei der zweiten Testreihe nur 200 Mal aufgerufen. Auch diesmal wieder mit 10 Anfragen pro Sekunde. Zwar ist in Abbildung 5.2 eine Verbesserung zu sehen, aber nach einem Blick auf das Log des serverlosen Dienstes wurden Fehlermeldungen beim Löschen der temporären Ordner angezeigt. Die gesendeten Zip-Archive bei Anfragen werden in diesen Ordner entpackt und abgespeichert. Für das Erzeugen der Ordner wird Pythons *tempfile.TemporaryDirectory* Funktion verwendet. Die Funktion sollte nach dem Austritt aus dem Kontext temporäre Ordner automatisch löschen [Pyt20b].

Auch die beiden Endpunkte `/run/java` und `/test/java` wurden auf den Fehler geprüft. Ähnlich wie zuvor wurden die Endpunkte mit 10 Anfragen pro Sekunde angesprochen, bis 500 Anfragen gestellt wurden. In Abbildung 5.3 sieht man, dass alle Anfragen erfolgreich und korrekt beantwortet wurden. Interessanterweise verwenden die beiden Java Endpunkte die gleiche Python Funktion. Als weitere Fehlerquelle wird das Speichern und Entpacken des Zip-Archivs vermutet. Das ist der größte Unterschied zu den Java Endpunkten und könnte möglicherweise der Grund sein, wieso der Fehler bei den Java Endpunkte nicht auftritt. Für eine produktionsreife Bereitstellung der Anwendung muss der Fehler gefunden werden, da

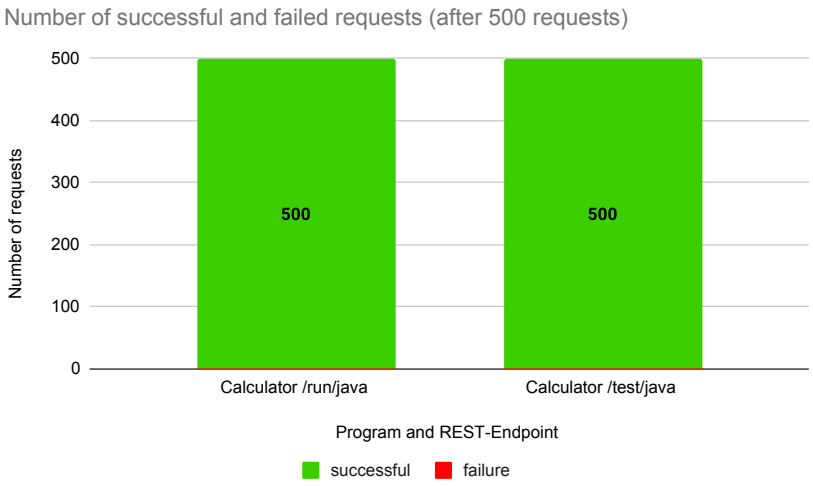

Abbildung 5.3: Anzahl der erfolgreichen und fehlerhaften Antworten nach insgesamt 500 Anfragen an Java Endpunkte

bei großer Nachfrage die zahlreichen fehlerhaften Antworten keine gute Benutzererfahrung bieten.

5.3 Antwortzeiten

Eine gute Benutzererfahrung setzt auch voraus, dass die Anwendung die überreichten Programme in angemessener Zeit ausführt und testet. Somit sollten die Antwortzeiten möglichst gering sein. Aus diesem Grund wurden die durchschnittlichen Antwortzeiten der Endpunkte analysiert. Bei dieser Testreihe wurden nur erfolgreiche Antworten der vorherigen Testreihen mit in die Berechnung eingeschlossen. In Abbildung 5.4 werden die wenigen erfolgreichen Antworten der `/test/gradle` betrachtet. Für das Testen der beiden Beispielprojekte vergehen im Durchschnitt mehr als 25 Sekunden.

Bei den Antworten der beiden `/run/gradle` Endpunkte sind nur kleine Verbesserungen zu sehen. Wie in Abbildung 5.5 zu sehen ist, vergehen im Schnitt 20 bis 25 Sekunden bis eine Antwort ankommt. Dies ist immer noch zu viel für eine produktionsreife Bereitstellung.

Abschließend wurden auch die Antwortzeiten der Java Endpunkte analysiert. Die Java Endpunkte könnten vielleicht zeigen, dass die lange Wartezeit der vorherigen Endpunkte durch

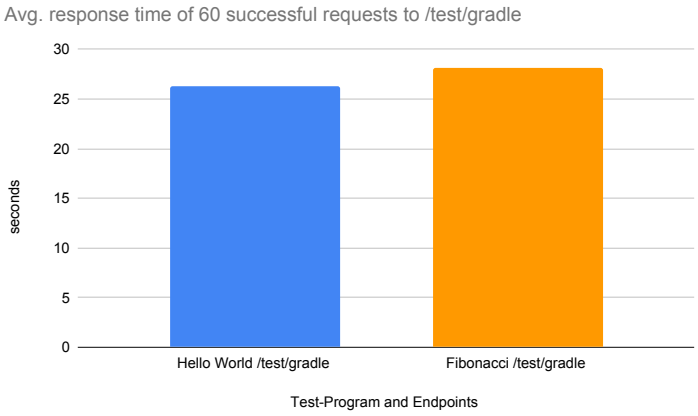

Abbildung 5.4: Durchschnittliche Antwortzeit von 60 Anfragen an /test/gradle

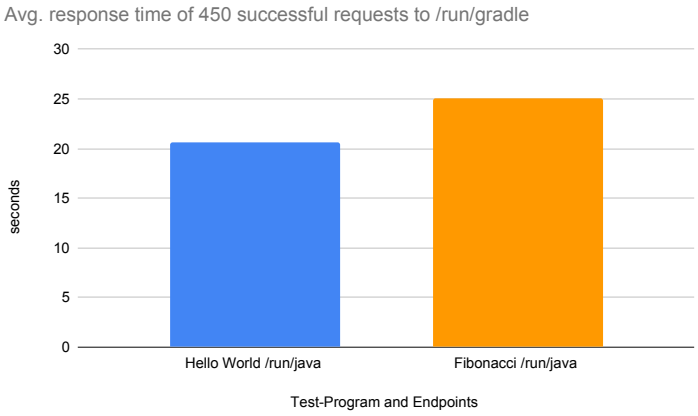

Abbildung 5.5: Durchschnittliche Antwortzeit von 450 Anfragen an /run/gradle

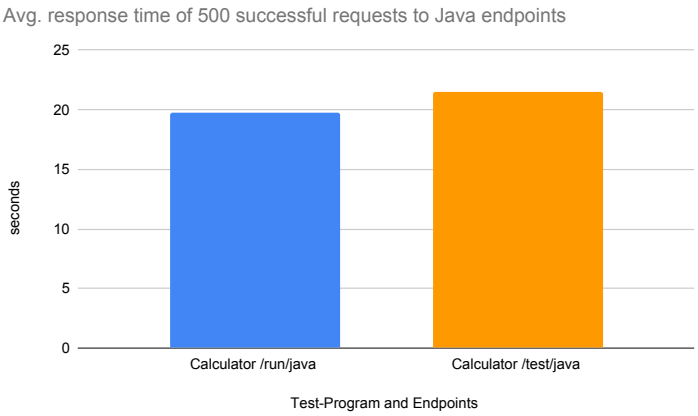

Abbildung 5.6: Durchschnittliche Antwortzeit von 500 Anfragen an /run/java und /test/java

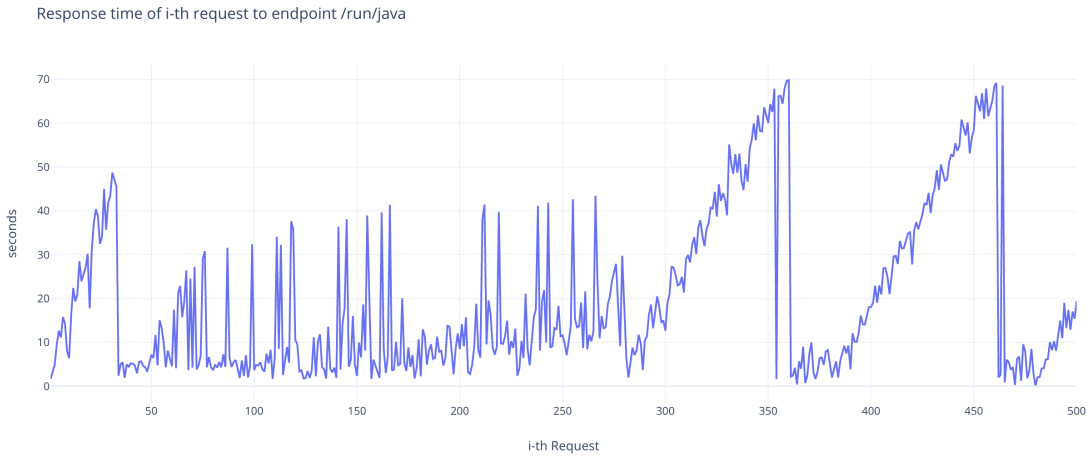

Abbildung 5.7: Antwortzeit des i-ten Aufrufs von /run/java

den bereits erwähnten Fehler auftritt. Bei vereinzelten Tests auf dem eigenen lokalen System wurden die Anfragen sehr zügig bearbeitet. Abbildung 5.6 zeigt jedoch an, dass auch hier im Durchschnitt gut 20 Sekunden gewartet werden muss. Ein möglicher Grund dafür könnte das konservative Starten neuer Containerinstanz von Seiten Googles sein. Nach näherem Betrachten der einzelnen Anfragen ist in Abbildung 5.7 zu sehen, dass vor allem ab der dreihundertsten Anfrage viel Zeit vergeht. Es sieht so aus, als ob der Google Dienst zu Beginn genug Containerinstanz startet, um die Nachfrage zu bewältigen. Jedoch stagniert das Hochfahren neuer Instanzen bis zu einem gewissen Punkt. Möglicherweise geht Google davon aus, dass die meisten Anwendungen auf den Containerinstanzen nur kurzlebig sind. Deshalb könnte die Skalierung zu Beginn höher sein. Läuft die Anwendung jedoch länger, könnte der Skalierungsfaktor verkleinert werden.

5.4 Modulaufgaben Programmierung WS 2018

Es wurden auch vereinzelt Aufgaben des Moduls Programmierung aus dem Wintersemester 2018 der HHU getestet. Die Aufgaben benutzen für das Testen Gradle Tasks. Sie testen auch nicht Funktionszustände und Rückgabewerte, sondern es wird die Ausgabe auf der Standardausgabe kontrolliert. Da jedoch bei der ausgearbeiteten Anwendung JUnit Tests vorausgesetzt werden, müssen die Gradle Tasks erst umgewandelt werden. Dies ist jedoch recht trivial, solange man auch bei den JUnit Tests nur die Ausgabe auf der Standardausgabe testet. Es müssen meist nur kleinere syntaktische Änderungen durchgeführt werden. Einige umge-

schriebene Tests sind dem Quellcode im GitLab der Arbeitsgruppe Rechnernetze der HHU beigelegt.

Kapitel 6

Ähnliche Anwendungen

6.1 Repl.it

Repl.it ist eine Entwicklungsumgebung im Browser [Rep20]. Es können entweder vordefinierte Umgebungen gestartet oder ein Projekt aus einer GitHub Repository importiert werden. Bei den Umgebungen kann aus unterschiedlichen Programmiersprachen und Frameworks ausgewählt werden. Die Projektstruktur und der Quellcode werden dann in einer grafischen Oberfläche angezeigt. Zusätzlich wird eine Konsole bereitgestellt, wodurch die verbundene Maschine in der Cloud kontrolliert werden kann. In der kostenlosen Version werden 500 Megabyte an Arbeitsspeicher und Festplattenspeicher, sowie 0,2 bis 0,5 vCPUs zur Verfügung gestellt. Anders als die ausgearbeitete Anwendung wird also eine konstante Verbindung zu einer VM in der Cloud bereitgestellt. Repl.it hat den großen Vorteil, dass der Quellcode bereits auf dem System liegt, wodurch die Ausführung schneller läuft als bei dem serverlosen Dienst. Bei der ausgearbeiteten serverlosen Anwendung müssen die Programme immer wieder zu einem Zip-Archiv komprimiert und versendet werden. Repl.it bietet noch weitere Funktionen, wie kollaboratives Arbeiten [Rep20]. Es ist eine sehr gute und günstige Alternative und kann als Vorlage für Verbesserungen der selbst entwickelten Anwendung dienen.

6.2 GitHub Codespaces

Ein recht neuer Dienst ist *GitHub Codespaces*. Es handelt sich auch hier um eine Entwicklungsumgebung in der Cloud [Git20b]. Es können GitHub Repositories importiert und im Browser oder in VS Code geöffnet werden. Die Entwicklungsumgebung basiert auf VS Code und bietet dadurch zahlreiche Funktionen [Git20a]. Der Dienst ist zum Zeitpunkt dieser Arbeit nur in begrenzter Public Beta, sodass er nicht ausprobiert und weiter verglichen werden kann.

Kapitel 7

Fazit

Im Rahmen dieser Arbeit wurde eine serverlose Anwendung entwickelt, die es Benutzern ermöglicht einfache Java Konsolenanwendungen in einer sicheren Umgebung auszuführen und zu testen. Sie lässt sich primär durch eine REST-API steuern, kann aber auch zum Teil über die grafische Entwicklungsumgebung angesprochen werden. Um einen Überblick über das relativ neue Konzept des Serverless-Computing zu geben, wurden die wichtigsten Eigenschaften, sowie Vor- und Nachteile erläutert.

Das Serverless Prinzip ist für die Anwendung nicht ganz geeignet. Während der Evaluation der Anwendung wurden einige Probleme ersichtlich. Die Anwendung weist bei den Gradle Endpunkten noch einen Fehler auf, sodass bei großer Nachfrage viele Anfragen nicht erfolgreich beantwortet werden. Der Vorteil der einfachen Skalierung von Serverless wird hier nicht ausgenutzt und muss noch verbessert werden. Die Antwortzeit der Anwendung ist noch zu groß, um eine angenehme Benutzererfahrung zu bieten. Das auszuführende Programm muss bei jeglichen Änderungen komprimiert und immer wieder an den Dienst versendet werden. Außerdem sind serverlose Anwendungen zustandslos. Eine Interaktion mit den ausgeführten Programmen wird nur durch Umstände möglich. Anders ist dies bei der konstanten Verbindung zu einer VM in der Cloud. Der Quellcode kann auf der Maschine liegen und muss deshalb nicht immer wieder versendet werden. Durch die direkte Verbindung können auch mit den ausgeführten Programmen interagiert werden. Der Sicherheitsaspekt der Arbeit ist auch mit einer VM basierten Cloud Lösung gegeben, ähnlich wie Repl.it oder GitHub Codespaces. Die Entwicklung der Applikation hat deutlich gemacht, dass die Serverless Prinzipien nicht für alle Anwendungen vorteilhaft sind.

Der Dienst hat aber nicht nur Nachteile. Bei serverlose Anwendungen wird nur der eigentliche Verbrauch in Zahlung gestellt. Da die entwickelte Anwendung auch auf null aktive Instanzen herunterskalieren kann, fallen bei nicht Benutzen der Anwendung keine kosten an. Hier hat die serverlose Anwendung einen Vorteil gegenüber VM basierten Lösungen, wie Repl.it oder GitHub Codespaces. Nach Anpassung der Tests kann die entwickelte Anwendung auch für das Ausführen und Testen von einigen Übungsaufgaben des Moduls Programmierung benutzt werden. Als Erweiterung kann das Ausführen von Gradle Tasks implementiert werden.

Das entwickelte Frontend ist ein erster Ansatz und noch sehr ausbaufähig. Um eine vollständige Entwicklungsumgebung zu bieten, müssen noch nützliche Funktionen implementiert werden. Da das Backend eine REST-API bereitstellt, können alternativ auch etablierte Texteditoren benutzt werden und ein passendes Plugin entwickelt werden. Das hätte den großen Vorteil, dass Benutzer ihren Quellcode in einer familiären Umgebung und mit nützlichen Werkzeugen entwickeln, wie es bei GitHub Codespaces der Fall ist.

Literatur

[AC17] Gojko Adzic und Robert Chatley. „Serverless computing: economic and architectural impact“. In: Unpublished, Aug. 2017. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117767.

[Ama20] Amazon Web Services. *AWS Lambda - Preise*. 20. Aug. 2020. URL: https://aws.amazon.com/de/lambda/pricing/ (besucht am 20. 08. 2020).

[Apa19] Apache Software Foundation. *System overview*. 14. Aug. 2019. URL: https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works (besucht am 20. 08. 2020).

[Apa20] Apache Software Foundation. *OpenWhisk Actions*. 11. Feb. 2020. URL: https://github.com/apache/openwhisk/blob/master/docs/actions.md (besucht am 20. 08. 2020).

[Azu20] Azul Systems. *Installation on Alpine Linux Using an APK Repository*. 20. Aug. 2020. URL: https://docs.azul.com/zulu/zuludocs/ZuluUserGuide/InstallingZulu/InstallOnLinuxUsingAPKRepository.htm (besucht am 20. 08. 2020).

[Bal+17] Ioana Baldini u. a. „Serverless Computing: Current Trends and Open Problems“. In: *CoRR* abs/1706.03178 (2017). DOI: 10.1007/978-981-10-5026-8_1. arXiv: 1706.03178. URL: http://arxiv.org/abs/1706.03178.

[Com19] Council Innovation Committee. *Demystifying NoOps and Serverless Computing*. Techn. Ber. The Chief Information Officers Council, 2019. URL: https://www.cio.gov/resources/Demystifying%20NoOps%20and%20Serverless%20Computing_FINAL.pdf (besucht am 20. 08. 2020).

[Doc20] Docker. *Docker overview*. 20. Aug. 2020. URL: https://docs.docker.com/get-started/overview/ (besucht am 20. 08. 2020).

[Fou18] Cloud Native Computing Foundation. *CNCF Serverless Whitepaper v1.0*. Techn. Ber. Cloud Native Computing Foundation, 2018. URL: https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf (besucht am 20. 08. 2020).

[Git20a] Github, Inc. *About Codespaces*. 20. Aug. 2020. URL: https://docs.github.com/en/github/developing-online-with-codespaces/about-codespaces (besucht am 20. 08. 2020).

[Git20b] Github, Inc. *Codespaces*. 20. Aug. 2020. URL: https://github.com/features/codespaces (besucht am 20. 08. 2020).

[Goo20a] Google, Inc. *Cloud Run*. 20. Aug. 2020. URL: https://cloud.google.com/run?hl=de (besucht am 20. 08. 2020).

[Goo20b] Google, Inc. *Cloud Run: Container sekundenschnell in der Produktion*. 20. Aug. 2020. URL: https://cloud.google.com/run?hl=de#section-13 (besucht am 20. 08. 2020).

[Goo20c] Google, Inc. *Configuring Memory Limits*. 20. Aug. 2020. URL: https://cloud.google.com/run/docs/configuring/memory-limits (besucht am 20. 08. 2020).

[Goo20d] Google, Inc. *CPU allocation*. 20. Aug. 2020. URL: https://cloud.google.com/run/docs/configuring/cpu (besucht am 20. 08. 2020).

[Goo20e] Google, Inc. *Firebase-Hosting*. 20. Aug. 2020. URL: https://firebase.google.com/docs/hosting (besucht am 20. 08. 2020).

[Goo20f] Google, Inc. *What is gVisor?* 20. Aug. 2020. URL: https://gvisor.dev/docs/ (besucht am 20. 08. 2020).

[Goo20g] Google, Inc. *Zeitüberschreitung bei Anfrage festlegen*. 20. Aug. 2020. URL: https://cloud.google.com/run/docs/configuring/request-timeout?hl=de (besucht am 20. 08. 2020).

[Gra20a] Gradle, Inc. *Command-Line Interface*. 20. Aug. 2020. URL: https://docs.gradle.org/current/userguide/command_line_interface.html (besucht am 20. 08. 2020).

[Gra20b] Gradle, Inc. *The Gradle Wrapper*. 20. Aug. 2020. URL: https://docs.gradle.org/current/userguide/gradle_wrapper.html (besucht am 20. 08. 2020).

[IBM20] IBM Cloud Education. *Hypervisors*. 20. Aug. 2020. URL: https://www.ibm.com/cloud/learn/hypervisors#toc-type-1-vs--Ik2a8-2y (besucht am 20. 08. 2020).

[Ker13] Kerrisk, Michael. *Namespaces in operation, part 1: namespaces overview*. 4. Jan. 2013. URL: https://lwn.net/Articles/531114/ (besucht am 20. 08. 2020).

[Mic20] Microsoft. *Monaco Editor*. 20. Aug. 2020. URL: https://microsoft.github.io/monaco-editor/ (besucht am 20. 08. 2020).

[Mik20] Mikhail Shilkov. *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP*. 26. Apr. 2020. URL: https://mikhail.io/serverless/coldstarts/big3/ (besucht am 20. 08. 2020).

[MIT20] MITRE Corporation. *JRE Security Vulnerabilities Published in 2019*. 20. Aug. 2020. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-19117/year-2019/Oracle-JRE.html (besucht am 20. 08. 2020).

[MR17] John Chapin Mike Roberts. „Understanding the Latest Advances in Cloud and Service-Based Architecture“. In: *What Is Serverless?* Hrsg. von Brian Foster. O'Reilly Media, Inc., 2017. ISBN: 978-1-491-98416-1.

[Ora20] Oracle. *Java Platform Standard Edition 8 Documentation*. 20. Aug. 2020. URL: https://docs.oracle.com/javase/8/docs/ (besucht am 20. 08. 2020).

[Pyt20a] Python Software Foundation. *subprocess — Subprocess management*. 20. Aug. 2020. URL: https://docs.python.org/3/library/subprocess.html (besucht am 20. 08. 2020).

[Pyt20b] Python Software Foundation. *tempfile — Generate temporary files and directories*. 20. Aug. 2020. URL: https://docs.python.org/3.7/library/tempfile.html (besucht am 20. 08. 2020).

[Rai19] Raina, Shashi. *Serverless Containers are the Future of Container Infrastructure*. 14. Okt. 2019. URL: https://aws.amazon.com/de/blogs/apn/serverless-containers-are-the-future-of-container-infrastructure/ (besucht am 20. 08. 2020).

[Red20a] RedHat. *Chapter 1. Introduction to Linux Containers*. 20. Aug. 2020. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers (besucht am 20. 08. 2020).

[Red20b] RedHat. *Containers and Images*. 20. Aug. 2020. URL: https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/containers_and_images.html (besucht am 20. 08. 2020).

[Red20c] RedHat. *What is a hypervisor?* 20. Aug. 2020. URL: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor (besucht am 20. 08. 2020).

[Red20d] RedHat. *What is a virtual machine*. 20. Aug. 2020. URL: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine (besucht am 20. 08. 2020).

[Red20e] RedHat. *What is KVM?* 20. Aug. 2020. URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM (besucht am 20. 08. 2020).

[Red20f] RedHat. *What's a Linux container?* 20. Aug. 2020. URL: https://www.redhat.com/en/topics/containers/whats-a-linux-container (besucht am 20. 08. 2020).

[Rep20] Repl.it. *Repl.it - The collaborative browser based IDE*. 20. Aug. 2020. URL: https://repl.it/ (besucht am 20. 08. 2020).

[Ser18] Amazon Web Services. *Firecracker: Lightweight Virtualization for Serverless Applications*. https://assets.amazon.science/96/c6/302e527240a3b1f86c8 firecracker-lightweight-virtualization-for-serverless-applications.pdf. 2018. (Besucht am 20. 08. 2020).

[SKM19] Hossein Shafiei, Ahmad Khonsari und Payam Mousavi. *Serverless Computing: A Survey of Opportunities, Challenges and Applications*. Techn. Ber. 2019. arXiv: 1911.01296 [cs.NI]. URL: https://arxiv.org/pdf/1911.01296.pdf (besucht am 20. 08. 2020).

[SMM18] Josef Spillner, Cristian Mateos und David A. Monge. „FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC“. In: *High Performance Computing*. Hrsg. von Esteban Mocskos und Sergio Nesmachnow. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-73353-1. DOI: https://doi.org/10.1007/978-3-319-73353-1_11.

[The20a] The JUnit Team. *JUnit 5 User Guide*. 21. März 2020. URL: https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher (besucht am 20. 08. 2020).

[The20b] The JUnit Team. *junit5-jupiter-starter-gradle*. 25. Juli 2020. URL: https://github.com/junit-team/junit5-samples/tree/main/junit5-jupiter-starter-gradle (besucht am 20. 08. 2020).

[The20c] The Pallets Projects. *Flask*. 20. Aug. 2020. URL: https://palletsprojects.com/p/flask/ (besucht am 20. 08. 2020).

[Ubu20] Ubuntu Manpage. *namespaces - overview of Linux namespaces*. 20. Aug. 2020. URL: http://manpages.ubuntu.com/manpages/focal/man7/namespaces.7.html (besucht am 20. 08. 2020).

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 20. August 2020 Sirat Ahmadi