Cloud-Native Programmiersprachen am Beispiel von Ballerina

Jan-Ole Hübner

***Zusammenfassung*—Moderne Anwendungen wandern immer weiter in „Die Cloud“. Dieses Paper beschäftigt sich mit der Bedeutung dieses Begriffes und betrachtet, wie sich die Anforderungen an eine Anwendung dadurch ändern. Es wird versucht grundlegende Begriffe zu erläutern und dem Leser einen Überblick über Anwendungsentwicklung für „Die Cloud“ zu geben.**
Hierzu werden am Beispiel der „cloud-nativen“ Programmiersprache Ballerina und eines Microservices, Konzepte und Denkweisen erläutert, die für eine solche Software nötig sind. Abschließend wird die Verwendbarkeit der Sprache für den produktiven Einsatz beurteilt.

***Index Terms*—Cloud-Computing, Cloud-Native, Ballerina, Programmiersprache**

I. Einleitung

War das Internet vor einigen Jahren noch neue Technologie, so ist es heute unerlässlich in allen Lebensbereichen. 2013 urteilte der Bundesgerichtshof sogar, dass der Zugriff auf das Internet ein Grundrecht sei. [Bun13] Vom Bestellen einer Pizza bis hin zur digitalen Patientenakte im Krankenhaus wird Software eingesetzt, die Daten generiert, mit ihnen arbeitet und vor allem mit anderen Systemen kommuniziert.
Die stark steigende Vernetzung von Systemen und die dauerhafte und günstige Verfügbarkeit des Internets führt zu neuen Anforderungen an Software, denn die Ausführung eines Programms findet immer weniger ausschließlich auf dem eigenen Rechner oder Server statt, sondern bei Dienstleistern, die Ihre Infrastruktur zur Verfügung stellen oder sich sogar komplett um die Ausführung von Code kümmern. Ein Entwickler hat keine Kontrolle mehr darüber, **wo** Code ausgeführt wird, sondern nur darüber, **wie**.
Die zentralen Punkte, auf die sich das *Cloud-Computing* konzentriert, sind Verfügbarkeit, Skalierbarkeit, sowie Kostenoptimierung. Der Fokus darauf führt zu weitreichenden Änderungen in allen Bereichen der Softwareentwicklung. An Stelle von monolithischer Software treten einzelne Services, die sich jeweils auf einen Teilbereich konzentrieren und miteinander kommunizieren, um ein Problem zu lösen. *Cloud-Computing* arbeitet also serviceorientiert.
Von der Konzeption bis hin zur Wartung findet ein Wandel statt, aus dem sich neue Architekturen und Vorgehensweisen ergeben. Die meisten Betrachtungen, die sich mit dem *Cloud-Computing* beschäftigen, konzentrieren sich dabei auf allgemeine Modelle und Architekturen, nicht aber auf die Werkzeuge mit denen ein Entwickler die Software erstellt - Programmiersprachen.
Mit dem *Cloud-Computing* tauchten auch die Begriffe der *Cloud-Nativen Programmiersprachen* sowie *Integration Languages* auf.
Dieses Paper betrachtet die Auswirkungen des *Cloud-Computing* auf Programmiersprachen und versucht am Beispiel der Sprache *Ballerina* einen Überblick darüber zu geben, was eine *Cloud-Native Programmiersprache* ausmacht.

JOH
März 2021

II. Überblick

Um zu verstehen was eine *Cloud-Native Programmiersprache* ausmacht, muss man sich zunächst mit der Definition des Begriffs *Cloud-Computing* auseinandersetzen. Bisher war von Anpassungen der Softwarearchitektur die Rede, um von den Vorteilen des *Cloud-Computing* profitieren zu können. Der Begriff umfasst jedoch mehr ein als nur den Aufbau der Software selbst. Es geht also nicht nur um Microservices.
Die beiden Begriffe *Cloud-Computing* und *Cloud-Native* lassen sich aufgrund der Komplexität des Themas nur schwer definieren und werden Heutzutage eher für Marketingzwecke verwendet. Allgemein geht es um die Bereitstellung von Ressourcen und damit verbundenen Dienstleistungen. Ein Anbieter stellt dem Kunden Ressourcen zur Verfügung. Der Kunde verwendet diese zur Ausführung seiner Anwendung und der Anbieter berechnet diese Ausführung. Dies hat für den Kunden mehrere Vorteile. Er muss sich nicht um Hardware und deren Erreichbarkeit kümmern, sondern überlässt dies dem Anbieter. Der Anbieter kann, da er mehrere Kunden bedient, seine Server effizienter auslasten und es sich deshalb leisten, dem Kunden nur die wirklich genutzten Ressourcen zu berechnen. Dies spart dem

Kunden Geld, da er nicht nur genauer abgerechnet werden kann, sondern unter Umständen auch Personal einsparen kann, das die Verwaltung von Hardware entfällt. *Cloud-Computing* bietet also Vorteile für alle Beteiligten. Aber wie?
Das NIST (National Institute of Standards and Technology) unterteilt *Cloud-Computing* in drei Bereiche: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) und Software as a Service (SaaS) [Ste20]. Sie stellen Unterschiedliche Abstraktionsebenen dar, die festlegen welche Bereiche in die Zuständigkeit des Anbieters fallen und welche in der Verantwortung des Kunden liegen. Abbildung 1 zeigt diese Zuständigkeiten.

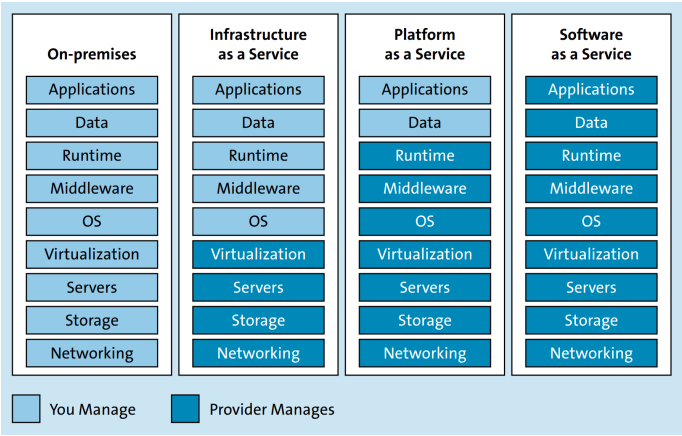

Abbildung 1. Zuständigkeiten bei den Verschiedenen Cloud-Computing Services [Ste20]

Bei Infrastructure as a Service stellt der Anbieter dem Kunden (virtualisierte) Hardware zur Verfügung. Diese kann meist bei Bedarf über eine API des Anbieters angefordert werden und ist innerhalb weniger Sekunden bis Minuten bereit. Der Kunde gibt schon hier die Kontrolle über die Hardware ab. Er kann nicht mehr entscheiden auf welchem physischen Server seine Software läuft. Dafür entfällt aber auch die Wartung der physischen Hardware sowie Hardwareausfälle oder Schäden. Um diese kümmert sich der Anbieter.
Der Kunde muss jedoch selbst sicherstellen, dass seine gewünschten Workloads ausgeführt werden und Fehlerfrei laufen, Platform as a Service bietet dem Kunden hier zusätzlich Software an, die dies erleichtert, so wird z.B. Software zur Verfügung gestellt, die dem Kunden bei der Einrichtung des Betriebssystems hilft oder es dem Kunden ermöglicht bestimmte Container zu starten. Die Grenzen sind hier fließend.
Software as a Service geht noch einen Schritt weiter. Der Kunde nutzt nur noch eine Software, die die gewünschte Aufgabe erledigt. Um die Verfügbarkeit kümmert sich der Anbieter.
PaaS und IaaS sind dabei für dieses Paper am relevantesten, da es auf diesen Ebenen auch um die Erstellung von Software und die Verwaltung der Daten geht.
Zusätzlich zu diesen Services findet seit neuestem auch Function as a Service (FaaS) immer mehr Verwendung. FaaS abstrahiert dabei soweit, dass der Entwickler nur noch einzelne Funktionen zur Verfügung stellt. Die Ausführung und Skalierung übernimmt komplett der Anbieter.
Der Begriff *Cloud-Native* schließt also auch die Art der Auslieferung der Software sowie den Betrieb ein. Hierzu werden meist Container verwendet. Diese bieten eine isolierte, virtualisierte Umgebung, die zum Einen ein schnelles und einfaches Starten ermöglicht und zum Anderen aufgrund der Virtualisierung identisches Verhalten auf unterschiedlicher Hardware garantiert. Diese Container werden dann meist von Tools wie Kubernetes, Mesos oder Docker Swarm verwaltet. Deren Aufgabe ist das starten neuer Instanzen. [Kra18]
Das Ziel ist eine „Welt der APIs“. Vom Anfordern neuer Ressourcen bis hin zum Verwalten von Nutzerdaten soll alles in Software möglich sein. Es wird also versucht von der Ebene der Hardware und deren Verwaltung zu abstrahieren.

Aus diesen Punkten wird ersichtlich, warum eine Anpassung der Software von Nöten ist. Die Software muss bestimmte Rahmenbedingen erfüllen, damit der *Cloud-Computing* Dienstleister seine Aufgaben optimal erfüllen kann. Ein Faktor der sich aus diesen Punkten ergibt, ist dass ein für *Cloud-Computing* optimierter Microservice **zustandslos** sein sollte, da ein Speichern, Laden und gegebenenfalls Übertragen des Zustandes einer Instanz viele Probleme mit sich bringt, die die horizontale Skalierbarkeit erschweren.
Der Anbieter Heroku veröffentlichte 2011 die ”12-Faktor-Methodik”, welche 12 Regeln aufstellt, die sicherstellen sollen, dass eine Anwendung von *Cloud-Computing* profitieren kann.[Wig17] Sie lauten:

- Codebase - Es existiert nur eine Codebasis und diese wird immer von einem Versionskontrollsystem verwaltet. Dies ermöglicht eine Überwachung der Aktivitäten und hilft bei der Automatisierung von Deployments. So kann zum Beispiel bei jedem Commit ein Deployment automatisiert gestartet werden. Trotz vieler Deployments und unterschiedlicher Versionen bleibt die Codebasis dieselbe.

- Abhängigkeiten deklarieren - Da ein Entwickler, der seine Anwendung in der Cloud betreibt, die Umgebung in der sie ausgeführt wird nicht kennt, darf er nie davon ausgehen, dass Abhängigkeiten immer vorhanden sind. Er muss diese also deklarieren, damit sie gegebenenfalls installiert werden können.

- Konfiguration in Umgebungsvariablen ablegen - Denn die Codebasis ist für alle Deployments dieselbe. Die Konfiguration unterscheidet sich unter Umständen jedoch stark.

- Lokale Ressourcen und externe Dienste (wie Datenbanken) werden gleich behandelt.

- Build, release, run - Ein Deploy ist immer in diese drei Phasen unterteilt

- Zustandslosigkeit

- Bindung an Ports

- Nebenläufigkeit - Anwendungen die diesem Prinzip folgen, skalieren durch zusätzliche Prozesse. Sie überlassen dem Betriebsystem die Verwaltung und skalieren, da sie zustandslos sind.

- Einweggebrauch - Prozesse einer App sollten jederzeit weggeworfen und neu gestartet werden können, damit eine optimale Verfügbarkeit garantiert ist.

- Dev-Prod-Vergleichbarkeit - verschiedene Deployments einer App sollten so ähnlich sein wie möglich, um gleiches Verhalten garantieren zu können.

- Logs sollten nicht in Dateien abgelegt werden, sondern in Streams.

- Administrationsaufgaben als einmalig betrachten - Administrative Aufgaben in einer 12-Faktor Anwendung sollten nie als regelmäßige Aufgaben betrachtet werden. Eine Wartung der Umgebung ist nicht notwendig. Tritt ein Fehler auf, so wird ein neuer Prozess gestartet. Administrative Aufgaben wie zum Beispiel der Export einer Datenbank sollten daher immer als einmalig betrachtet werden.

Folgt eine Anwendung diesen Prinzipien, kann sie als *Cloud-Native* bezeichnet werden. *Cloud-Native Programmiersprachen* haben das Ziel, dem Entwickler das Erfüllen dieser Randbedingungen zu erleichtern. Sie ändern beispielsweise den Umgang mit einzelnen Ressourcen wie Threads oder dem Netzwerk. An eine Cloud-Native Programmiersprache ergeben sich also folgende Anforderungen:

- Programme arbeiten parallelisiert und nicht nur auf mehreren Prozessorkernen sondern auch über verschiedene Systeme hinweg.

- Programme stellen einen Webservice zur Verfügung und kommunizieren mit anderen Services.

- Durch die Kommunikation mit anderen Services müssen Daten übertragen werden. Diese sind häufig strukturiert und liegen nicht als reiner Bytestream vor. Das Parsen dieser Daten ist also notwendig. Der häufige Datenaustausch erfordert ebenfalls eine einfache Möglichkeit, Daten inhaltlich zu überprüfen, da die Daten von externen Services stammen.

- Performance ist ein zentraler Punkt bei den Anforderungen an eine Cloud-Native Sprache. Anbieter, die den Code ausführen, rechnen die Ausführung anhand der Ausführungszeit und den angeforderten Ressourcen ab.

Viele Programmiersprachen stellen zur Lösung dieser Probleme Bibliotheken zur Verfügung, während Cloud-Native Sprachen deren Lösung direkt in die Sprache integrieren.

III. Ballerina

Ballerina ist eine Programmiersprache, die sich selbst als *Cloud-Native* bezeichnet. Laut eigener Aussage steht die Microservice-Architektur und damit der Fokus auf Netzwerk und eventbasierte Anwendungen im Vordergrund. [WSO21a]
Es wird also versucht, den Entwickler bei der Erstellung eine 12-Faktor Anwendung zu unterstützen. Da die 12 Faktoren sich nicht nur auf das reine Produzieren von Code beziehen, sondern auch Vorgaben zur Auslieferung und dem Betrieb machen, unterstützt Ballerina den Entwickler auch in diesen Bereichen.

A. DevOps & Tooling

Der Begriff *DevOps* setzt sich zusammen aus den Begriffen *Development* und *Operations*. *DevOps* bezieht

sich also auf das Zusammenspiel von Entwicklung und Betrieb. Änderungen sollen schnell und direkt - meist automatisiert - getestet und integriert werden. (*Continuous Integration/Continuous Delivery*). Zusätzlich zu den bereits erwähnten Anforderungen versucht Ballerina mit verschiedenen Funktionen den Ablauf der agilen Softwareentwicklung und der Teamarbeit zu erleichtern und *Development* und *Operations* näher aneinander zu bringen. Dazu bietet Ballerina zum Beispiel die Möglichkeit, lauffähige Container-Images als Teil des Build-Vorgangs zu erzeugen. Es bedarf dazu keiner komplizierten Konfiguration sondern lediglich einiger Annotationen.
In Ballerina ist es ebenfalls möglich aus dem Code ein Sequenzdiagramm zu erzeugen. Dies bietet den

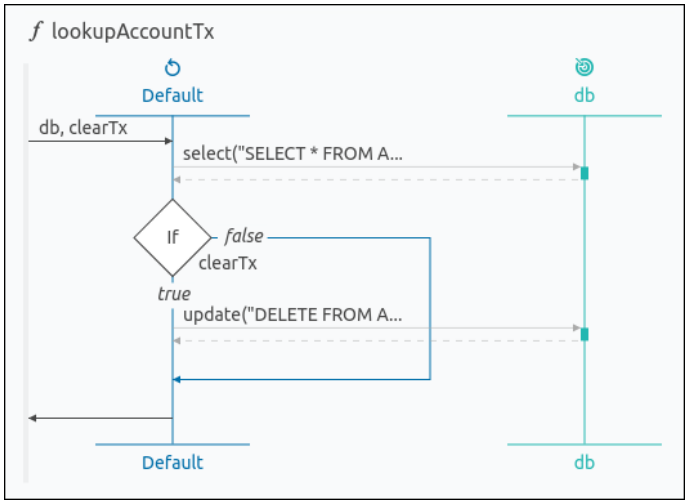

Abbildung 2. Automatisch erzeugtes Sequenzdiagramm [WSO21a]

Entwicklern unter Anderem einen Überblick über den Ablauf der Kommunikation mit anderen Services und kann so z.B. dazu beitragen, blockierende Anfragen zu erkennen und die Performance zu verbessern. In Bezug auf die 12 Faktoren einer *Cloud-Native* Anwendung bietet Ballerina aber noch mehr. Ein eigenes Package Management ist enthalten. Abhängigkeiten lassen sich in einem Ballerina-Projekt deklarieren und Ballerina sorgt dafür, dass diese zur Ausführung vorhanden sind.

B. Die Sprache

Ballerina verwendet die *Ballerina Virtual Machine* (BVM) zur Ausführung von Programmen. Der Compiler erzeugt Ballerina Bytecode, welcher dann in der BVM ausgeführt wird. Für die Übersetzung des Codes in Bytecode erzeugt der Compiler zunächst einen *Abstract Syntax Tree (AST)*. Hierbei handelt es sich um eine Baumstruktur mit deren Hilfe der Compiler die syntaktische Korrektheit des Programms überprüfen kann.

Eine Stärke von Ballerina ist die einfache Parallelisierbarkeit, sowie nicht blockierende Netzwerkanfragen. Die Grundlage dafür wird genau hier geschaffen. Der Compiler baut die Baumstruktur des Programms auf und optimiert ihn anschließend basierend auf der Ausführungsreihenfolge der Anweisungen. Dadurch wird ein flexibler Umgang mit Threads ermöglicht, da eine schnelle Freigabe der Threads möglich ist. [Ora19]
Die Sprache verfolgt dabei keinen klassischen objektorientierten Ansatz. Das Typensystem ist strukturell. In Java bilden vom Entwickler erstellte Typen Objekte. Diese Objekte nutzen Vererbung um die Beziehungen verschiedener Typen zu modellieren. In Ballerina besitzt jeder Wert eine eigene Form. Ein Typ besteht aus einer Menge an Werten. Werte wiederum besitzen immer eine Form. [WSO21b] Um zu verstehen was die Form eines Wertes genau ist, sind in Tabelle I die verschiedenen Arten von Typen aufgelistet die Ballerina kennt. Es werden grundsätzlich

Tabelle I
TYPEN IN BALLERINA

Typ	Werte
Basic Types	Bool, Float
Structured Values	Maps, List
Sequence Values	XML, String
Behavioral Values	Function, Object

vier Arten von Typen unterschieden. Diese bauen aufeinander auf. *Basic Types* sind Werte, welche sich nicht aus anderen Werten zusammensetzen wie zum Beispiel ein boolescher Wert oder eine Gleitkommazahl. *Structured Values* wie Listen oder Maps dienen der Strukturierung von *Basic Types* und *Sequence Values* können *Basic Types* und *Structured Values* kombinieren um komplexere Strukturen wie XML abzubilden.
Zu guter Letzt gibt es die *Behavioral Values*. Diese dienen, wie der Name vermuten lässt, zur Beschreibung von Verhalten. Zu diesen Werten zählen beispielsweise Funktionen. Funktionen können also in Variablen übergeben werden.

> A type denotes a set of shapes. Subtyping in Ballerina is semantic: a type S is a subtype of type T if the set of shapes denoted by S is a subset of the set of shapes denoted by T. Every value has a corresponding shape. A shape is specific to a basic type: if two values have different basic types, then they have different shapes. [WSO21b]

Die Form eines Wertes ist durch seinen *Basic Type* definiert. Haben zwei Werte einen unterschiedlichen *Basic Type*, so ist ihre Form unterschiedlich. Damit lässt sich nun das Konzept *Subtyping* definieren (Siehe III-B). Ein Typ ist dann ein *Subtype* eines eines anderen Typen, wenn deren Formen kompatibel sind. Als konkretes Beispiel dient Abbildung 3. Hier wäre es möglich einen Studenten mittels einer Referenz des Typen Person zu manipulieren, denn ein Student enthält mindestens dieselben Werte wie eine Person. Student ist also ein *Subtype* von Person, da er lediglich spezifischer ist. Im Unterschied zur Vererbung muss dies aber nicht explizit festgelegt werden. Dies ermöglich einen flexiblen Umgang mit Daten.

```
type Person record {
    string name;
    int age;
};

type Student record {
    string name;
    int age;
    string school;
};
```

Abbildung 3. *Subtyping in Ballerina*

Ballerina bietet dem Entwickler eine Menge Funktionen, die den Umgang mit Kommunikation und Parallelität erleichtern. Dazu zählt unter Anderem der Umgang mit Threads. In Ballerina gibt es für Threads Worker-Objekte.
Worker können innerhalb einer Funktion verwendet werden. Sie werden dann bei Aufruf dieser Funktion automatisch initialisiert und gestartet. Anschließend kann noch innerhalb der Funktion darauf gewartet werden, dass beide Worker ihre Aufgaben erledigt haben.

```
function doStuff(){

    worker alice {
        ("Ballerina", "is awesome") -> bob;
    }
    worker bob {
        (string, string) fromAlice;
        fromAlice <- alice;
    }
    results = wait {bob, alice};
}
```

Abbildung 4. Worker und deren Kommunikation in Ballerina

Worker können bei Bedarf genau wie eine Funktion Werte zurück geben. Diese Rückgabewerte werden dann Zusammengetragen und stehen dem Entwickler zur Verfügung.
Andere Sprachen verwenden hier Konzepte wie *Futures*. Hier muss der Entwickler selbst überprüfen, ob ein Wert bereits vorhanden ist. Dies kann schnell zu Fehlern im Programm führen, wenn der Entwickler die *Futures* oder die aus Ihnen resultierenden Fehler (z.B. bei einem Timeout einer Anfrage) nicht richtig behandelt.
Die Verwendung eines Workers in einer Funktion ist besonders im Kontext Cloud-Nativer Anwendungen sinnvoll, da einzelne Funktionen eines Services eventbasiert aufgerufen werden. Eine Funktion stellt also oft einen einzelnen Endpunkt dar, der eine einzelne Anfrage beantwortet.
Abbildung 4 zeigt zwei Worker, die Nachrichten austauschen. Hier kümmert sich Ballerina automatisch um die Ausführungsreihenfolge der Worker, sodass angefragte Werte immer verfügbar sind. In Abbildung 4 ist ebenfalls ein weiterer syntaktischer Vorteil von Ballerina zu erkennen. Es wird zwischen lokalen- und remote-Aufrufen unterschieden. Der Pfeil (-> / <-) signalisiert dem Entwickler eine Kommunikation mit einem entfernten Objekt oder Endpunkt. Mit einem Punkt hingegen werden lokale Aufrufe getätigt.
Aber nicht jeder Worker startet auch gleich einen eigenen Thread. Ballerina unterscheidet zwischen Threads und sogenannten *Strands*. Ein *Strand (Strang)* ist eine Art leichtgewichtiger Thread. Der Unterschied besteht darin, dass Strands in der Voreinstellung einen Thread teilen. Es kann also immer nur ein *Strand* gleichzeitig ausgeführt werden. Der Sinn dieser Strands besteht hier darin, reines Warten zu verhindern. Wartet ein Strand beispielsweise auf die Antwort eines Servers zu seiner Anfrage, so kann währenddessen ein anderer Strand seine Aufgaben erledigen. Dies führt zu effizienterer Nutzung vorhandener Ressourcen.
Ein weiterer zentraler Punkt von Ballerina ist der Umgang mit Typen und Fehlern. Ballerina unterstützt sogenannte *Union Types*. Diese ermöglichen es, dass eine Variable eine Menge an Typen haben kann. So ist es z.B. möglich dass eine Netwerkanfrage, die ein JSON-Objekt liefern soll, durch beispielsweise einen Timeout einen Fehler verursacht die Rückgabe kann also entweder ein JSON-Objekt sein oder ein Fehler (error). Die Menge der erwarteten Typen kann in Ballerina mit dem Zeichen '|' angegeben werden. Erwartet der Entwickler also einen Fehler, oder ein JSON-Onjekt, so ist der passende Typ '*error|json*'. Wichtig ist, dass der tatsächliche Wert natürlich immer nur einen der angegebenen Typen haben kann. Zusätzliche Typen müssen durch zusätzliche

Anfragen eliminiert werden. Eine Funktion, die eine Zeichenkette als Parameter erwartet, wird *error|string* nicht akzeptieren. Die Überprüfung auf Fehler ist notwendig, damit das Programm fehlerfrei kompiliert.
Dies sorgt für eine bessere Übersicht über mögliche Datentypen und resultiert in besserer Fehlerbehandlung.
Ein Microservice arbeitet wie bereits erwähnt mit den Daten anderer, externer Services. Diesen externen Services sollte nicht vertraut werden. Um zum Beispiel SQL-Injections vorzubeugen bietet Ballerina hierzu *Taint Checking*. Bei der Kompilierung wird überprüft ob Daten Benutzereingaben erlauben, oder von andern Services stammen. Die Werte werden dann als *Tainted* (verschmutzt) markiert. Ist dies der Fall, kann der Entwickler sie im Programm erst verwenden, nachdem er die nötigen inhaltlichen Überprüfungen durchgeführt hat. Dazu markiert er den Wert mit *<@untainted>* als vertrauenswürdig.
Es sei an dieser Stelle allerdings angemerkt, dass das *Taint Checking* in der Version, die für dieses Paper verwendet wurde, deaktiviert ist.
Eine weitere Besonderheit Ballerinas ist die Erweiterbarkeit der Sprache. Da Ballerina selbst in Java geschrieben ist, besteht die Möglichkeit Erweiterungen für den Compiler in Java zu schreiben.

IV. Anwendung der Lösung

Betrachten wir nun diese Konzepte in der Praxis. Hierzu verwenden wir die Version **Swan Lake Alpha 2 (2.0)**. Die Angabe der Version ist in diesem Fall sehr wichtig. Ballerina ist ein relativ junges Projekt. Die Version 1.0 ist erst seit Ende 2019 verfügbar. Dies führt dazu, dass grundlegende Konzepte der Sprache noch häufig überarbeitet werden. Mehr dazu in Abschnitt V.
Ein Service ist in Ballerina schnell erstellt. Abbildung

```
1  import ballerina/io;
2  import ballerina/http;
3
4  service / on new http:Listener(9090) {
5      resource function get service/[string parameter1]
6                                  (http:Caller caller,http:Request req) returns error? {
7          error? response = caller->respond(parameter1);
8      }
9  }
```

Abbildung 5. Minimaler HTTP-Service mit parametrisiertem Endpunkt.

5 zeigt einen minimalen HTTP-Service mit einem parametrisiertem Endpunkt. Dafür bedarf es weniger als 10 Zeilen Code. Viele der bereits erläuterten Funktion sind in diesen wenigen Zeilen sichtbar. Ein Service kann auf sämtlichen Listener-Objekten erstellt werden. Diese wiederum werden in eigenen Modulen definiert. Dadurch wird die Erweiterbarkeit sichergestellt. Möchte man z.B, statt HTTP einen Service für Apache Kafka erstellen, muss in der Theorie lediglich das Listener-Objekt ausgetauscht werden und die passenden Funktionen werden weiterhin eventbasiert aufgerufen. Dies lässt sich natürlich nicht nur auf Kommunikation über das Netzwerk anwenden. Es lassen sich auf die selbe Weise auch Listener integrieren, die auf lokale Events hören. Abbildung 6 zeigt die Definition eines Dateisystem-

```
1  listener file:Listener inFolder = new ({
2      path: "/home/ballerina/observed-dir",
3      recursive: false
4  });
```

Abbildung 6. Listener für Dateisystem-Events

Listeners. Der Syntaktische Aufbau eines Services bleibt genau wie in Abbildung 5 gezeigt. Natürlich muss hier aber beachtet werden, dass Anpassungen des Services in den meisten Fällen trotzdem notwendig sind, da jeder Service auf andere Events hört. Ein Listener, der auf Events des Dateisystems hört, kann natürlich nicht mit HTTP-spezifischen Begriffen wie GET oder POST umgehen. Umgekehrt wird ein Dateisystem-Event wie *onCreate* nicht in einem HTTP-Service ausgelöst werden.
Ein solcher Aufbau eines Services bietet dennoch Vorteile. Der offensichtlichste ist hier der sehr einfache Wechsel zwischen technisch ähnlichen Protokollen. In den meisten Fällen dürfte zum Beispiel bei einem Wechsel von HTTP auf HTTPS ein einfacher Austausch des Listeners genügen.
Weiterhin bietet ein solcher Aufbau die Möglichkeit einer weiteren Abstraktionsebene. Services wie Twitter können in einem eigenen Modul Events oder Aktionen definieren und so komplett von der Nutzung der API abstrahieren.
Der Entwickler kann also die ihm vertraute Sprache verwenden, ohne sich mit der API auseinandersetzen zu müssen.
Ballerina ermöglicht es dadurch Plattformen mehr als ein Protokoll zu betrachten.
Im Unterschied zu anderen Programmiersprachen, die diese Funktionalitäten über eigene Bibliotheken ebenfalls anbieten, ist der Ansatz von Ballerina mit deutlich weniger Aufwand für den Entwickler verbunden. Denn bei eigenen Bibliotheken kommt es oft zu einer langen Einarbeitungsphase, da diese unter Umständen einer eigenen Struktur folgt. Die Integration in die Syntax der Sprache minimiert die Einarbeitungszeit.
In dem in Abbildung 5 gezeigten Service ist auch die Fehlerbehandlung in Ballerina im Ansatz zu erkennen.

Die Angabe *returns error?* zeigt einen sogenannten *Optional Type*. Dabei handelt es sich um die Kurzschreibweise von *error|()*. Wobei *()* gleichbedeutend mit Null ist (nil in Ballerina). Optional Types verhindern Null-Reference Fehler, da der Inhalt wie erwähnt überprüft werden muss, bevor der Wert verwendet werden kann.
Innerhalb des Services befinden sich Ressourcen. Eine Ressource ist eine Funktion, die einem Endpunkt zugeordnet ist.
Innerhalb dieser Funktionen können nun die in Abbildung 4 bereits gezeigten Worker verwendet werden. Zu sehen ist dies bereits in Abbildung 4.
Von einem solchen Service kann nun mittels einer einfachen Annotation (zu sehen in Abbildung 7) erstellen lassen. Sämtliche in der Abbildung sichtbaren Werte sind dabei optional, dienen aber der weiteren Automatisierung.

```
1  @docker:Config {
2      push: true,
3      registry: "index.docker.io/$env{DOCKER_USERNAME}",
4      name: "helloworld",
5      tag: "v1.0.0",
6      username: "$env{DOCKER_USERNAME}",
7      password: "$env{DOCKER_PASSWORD}"
8  }
9  service / on new http:Listener(9090) {
```

Abbildung 7. Annotation für ein automatisch generiertes Docker Image

V. Grenzen und Probleme

Wie jedes Werkzeug hat auch Ballerina Grenzen. Ballerina ist, wie in diesem Paper gezeigt wurde, speziell für Netzwerk und Kommunikation ausgelegt. Seine stärken zeigt Ballerina bei der Entwicklung von netzwerkbasierter Middleware. REST-APIs sind schnell erstellt. Verlässt man aber diesen Bereich und will zum Beispiel eine komplexe Anwendung für den PC entwerfen, die nicht unbedingt im Kontext von *Cloud-Computing* arbeitet, stößt Ballerina an Grenzen, die den „Developer First"-Ansatz bröckeln lassen. Das in III-B erläuterte Typensystem, verleitet dazu, die internen Datenstrukturen des Programms zu vernachlässigen. Die Modellierung von Beziehungen zwischen Typen ist gerade für einen Entwickler mit einem Hintergrund in der Objektorientierung unübersichtlich und bedarf einer längeren Einarbeitung.
Das Hauptargument für Ballerina - nämlich dass es keiner externen Bibliotheken bedarf - ist ebenfalls nur begrenzt anwendbar. Will man beispielsweise auf eine Anfrage in einem Service mit HTML antworten, so muss dies manuell geschrieben und anschließend der Header der Antwort manipuliert werden, da diese Funktionalität nicht integriert ist. Als Alternative bleiben hier dann nur externe Bibliotheken.
Das größte Problem von Ballerina ist allerdings die Unbeständigkeit der Sprache. Diese ist dem geringen Alter des Projektes geschuldet. Gemeint sind damit sehr häufige, tiefgreifende Änderungen von unter Anderem der Syntax, oder auch das Deaktivieren ganzer Funktionalitäten. Letzteres betrifft momentan zum Beispiel das Erzeugen von Sequenzdiagrammen. In der für dieses Arbeit verwendeten Version *Swan Lake Alpha 2* ist dies aktuell nicht möglich. Ältere Versionen verwenden aber häufig eine sehr stark geänderte Syntax. Ein Wechseln der Version ist also mit nicht unerheblichem Aufwand verbunden.
Die Definition eine Services wie in Abbildung 5 und seinen Endpunkten erfolgte zum Beispiel in der letzten verfügbaren Version (1.2) größtenteils über Annotationen, anstatt im Kopf der Funktion.

VI. Andere Lösungen

Ballerina ist ein weiteres Werkzeug unter Vielen. Es soll hier nicht vergessen werden einige davon zu erwähnen und einen kurzen Überblick darüber zu geben, welche Alternativen existieren und was diese bieten.
Die wohl offensichtlichste Alternative zu Ballerina ist Go. Go ist Ballerina sehr ähnlich. Der Fokus liegt ebenfalls auf Parallelität. Passende Konzepte für die hier erwähnten *Strands* und *Worker* lassen sich auch in Go wiederfinden (*Goroutines* und *Channels*). Nach Services jedoch sucht man in Go vergebens. Hier sind Frameworks oder Bibliotheken nötig. Dafür bietet Go den Vorteil der Stabilität und Verlässlichkeit. Die Sprache ist zu Einen deutlich älter, zum Anderen besitzt sie eine größere Nutzerbasis. Dies dürfte die Entwicklung erleichtern und sehr Vorteilhaft für den produktiven Einsatz sein, da die Stabilität dann ein zentraler Punkt ist.
Beide Sprachen werden kompiliert. Da Ballerina aber auf Java basiert, wird hier Bytecode für die JVM erzeugt.

Eine weitere Alternative ist Node.js. Node.js ist eine Laufzeitumgebung für JavaScript basierend auf der *V8 JavaScript Engine* von Google. Hiermit lassen sich ebenfalls plattformübergreifend eventbasierte Microservices erstellen. Man ist hier jedoch immer abhängig von der Laufzeitumgebung. Eine eventbasierte Anwendung in JavaScript kann je nach gewünschter Komplexität allerdings schnell unübersichtlich werden, da der Entwickler tiefgreifende Konzepte wie *Callbacks* beherrschen muss, um effizient zu arbeiten. Vorteile sind auch hier die große Nutzerbasis und weite Verbreitung.

Eine letzte hier betrachtete Alternative stellt Python in Kombination mit Frameworks wie *Flask* dar. Der größte Nachteil dieses Ansatzes ist die Komplexität im Bereich der Nebenläufigkeit. Threads müssen bei diesem Ansatz meist vom Entwickler selbst verwaltet - das heißt erstellt, gestartet, gegebenenfalls gestoppt und gelöscht werden. Eventuelle Kommunikation zwischen Threads muss der Entwickler ebenfalls selbst übernehmen. Dafür bietet die interpretierte Sprache eine geringe Einstiegshürde, dann sie bietet eine sehr leicht verständliche Syntax und ist in den Möglichkeiten deutlich weniger eingeschränkt als Ballerina.
Da Python aber eine interpretierte Sprache ist und Frameworks wie *Flask* von Dritten entwickelt werden - dadurch also weniger spezialisiert sind - ist dies bei einem Fokus auf die Performanz vermutlich nicht die optimale Lösung, bietet jedoch einen guten Einstieg in das Thema.

VII. Fazit

Ballerina ist eine junge Programmiersprache, die ein sehr spezielles Problem löst - Das schnelle und einfache Erstellen von Microservices. Dafür bietet sie Konzepte, die den Umgang mit Ressourcen wie dem Netzwerk erleichtern und es dem Entwickler ermöglichen, den Überblick zu behalten. Es handelt sich wie erwähnt jedoch um ein junges Projekt, dass sich in einer sehr frühen Phase der Entwicklung befindet. Das hat zur Folge, dass es häufig zur Änderung zentraler Konzepte kommt, was wiederum dazu führt, dass ein produktiver Einsatz nur schwer zu rechtfertigen ist. Essentielle Funktionalitäten oder Konzepte wie das Generieren von Sequenzdiagrammen oder das *Taint Checking* werden häufig geändert, oder gar ganz deaktiviert.
Ballerina zeigt jedoch auf dennoch beeindruckende Weise, dass es das, wofür es gedacht ist, effizient und schnell löst. Auch wenn der starke Fokus auf Microservices momentan dafür sorgt, dass die Sprache für alle anderen Probleme nur schwer zu verwenden ist, zeigt Ballerina, dass es auf einem guten Weg ist.
Zusammenfassend lassen sich alle Probleme, die sich bei der Verwendung ergeben auf das geringe Alter des Projekts, das relativ kleine Team (im März 2021 gerade einmal 306 Mitwirkende auf GitHub verglichen mit 1625 bei Go) und die daraus resultierende, kleine Nutzerbasis zurückführen. Da Microservices in Zukunft vermutlich weiterhin an Bedeutung gewinnen werden, ist zu erwarten, dass auch Ballerina an Relevanz gewinnt, denn mit fortschreitender Zeit wird die Sprache sich soweit entwickeln, dass die Syntax weitgehend stabil bleibt und nicht jedes Update Anpassungen des Entwicklers erfordert. Aktuell jedoch, ist die Sprache nicht mehr als ein Vorreiter im Bereich Cloud, den es zu beobachten gilt.

Literatur

[Bun13] Bundesgerichtshof. *Urteil BGH, 24.01.2013 - III ZR 98/12*. 24. Jan. 2013. URL: https://juris.bundesgerichtshof.de/cgi-bin/rechtsprechung/document.py?Gericht=bgh&Art=en&nr=63259&pos=0&anz=1.

[Kra18] Nane Kratzke. “A Brief History of Cloud Application Architectures”. In: *Applied Sciences* 8.8 (2018). ISSN: 2076-3417. DOI: 10.3390/app8081368. URL: https://www.mdpi.com/2076-3417/8/8/1368.

[Ora19] Andy Oram. *Ballerina: A Language for Network-Distributed Applicationsn*. O’Reilly Media, 2019. ISBN: 9781492061151.

[Ste20] D. Stender. *Cloud-Infrastrukturen: Das Handbuch für DevOps-Teams und Administratoren*. Rheinwerk Verlag, 2020. ISBN: 9783836269490.

[Wig17] Adam Wiggins. *The Twelve-Factor App*. 2017. URL: https://12factor.net/.

[WSO21a] WSO2. *Ballerina*. 24. Feb. 2021. URL: https://ballerina.io/.

[WSO21b] WSO2. *Ballerina Language Secification*. 2021. URL: https://ballerina.io/spec/lang/2020R1/#values_types.