HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT
DRESDEN FAKULTÄT INFORMATIK/MATHEMATIK

BACHELORARBEIT

Komposition von Microservices mittels Containern auf Basis von Feature Branches

von

Kong Minh Cap

betreut von
Prof. Dr.-Ing. Jörg VOGT
und
Dipl.-Math. Christian SCHEER

eingereicht am
26. Juni 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In den letzten Jahren fand, nicht nur im Kontext von Enterprise Anwendungen, sondern in der Softwareentwicklung allgemein, ein Paradigmenwechsel statt. Es erfolgt eine immer stärkere Abkehr von der monolithischen Schichtenarchitektur hin zu einem rein service-orientierten Ansatz. Vor allem mit dem zunehmenden Anspruch der Skalierbarkeit und Flexibilität von Software, um dem Nutzerandrang oder den Performance-Vorstellungen des Kunden gerecht zu werden, findet die Umsetzung von Anwendungen auf Basis der Microservice-Philosophie immer mehr Einzug. Große Player wie Netflix, Amazon und Co. sind zwar die Vorreiter im produktiven Einsatz von Microservices, jedoch ziehen auch immer mehr kleinere und mittlere Projekte nach und erzielen damit beachtliche Erfolge. Mit der Popularität der Architektur steigt auch die Anzahl der Werkzeuge und Hilfsmittel, welche Abhilfe gegen die ebenso erhöhte Komplexität von verteilten Systemen schaffen. Zu diesen gehören vor allem die Software-Containerisierung und dessen Verwaltung durch ein Orchestrierungssystem. Es kommt zu immer schnelleren und häufigeren Anforderungsänderungen des Kunden, da dieser sich in vielen Branchen an einem dynamischen Markt bewegt. Aus diesem Grund findet Agilität nicht nur im Entwicklungsprozess, zum Beispiel durch Scrum, sondern nun auch in der Softwarearchitektur selber großen Anklang.

1.1 Zielstellung

Heutzutage bestehen komplexe Enterprise Anwendungen aus mehreren Einzelsystemen, welche zusammen getestet werden sollen. Um ein verteiltes System in seiner Gesamtheit testen zu können, ist es notwendig alle Teilsysteme verfügbar zu machen. Container erlauben es eine autarke Umgebung mit allen benötigten Drittsystemen, wie zum Beispiel Datenbanken und Message Brokern, für einen solchen Test schnell bereitzustellen. Im Rahmen dieser Arbeit soll ein Weg beleuchtet werden, wie und unter Nutzung welcher Technologie- und Designansätze man Services zusammenfügen sollte, sodass neue Versionen eines Services mit anderen Teilen oder gar der Gesamtanwendung zusammen getestet werden können. Dabei sollte sowohl das Testen als auch der Verteilvorgang automatisiert erfolgen.

1.2 Lösungsweg

Um die genannten Probleme zu lösen soll eine Continuous Delivery Pipeline entwickelt werden, die auf Basis von Feature-Branches die benötigten Umgebungen bereitstellt. Die Umgebungen sollen dabei unter Einsatz von Containern umgesetzt werden, welche sich in einem Kubernetes Cluster befinden. Diese Arbeit soll hierbei keine explizite Implementierung eines solchen Systems präsentieren, sondern vielmehr eine allgemeine Vorgehensweise zum erreichen der genannten Ziele aufzeigen, da die Umsetzung je nach eingesetzter Technologie anders aussieht.

2 Grundlagen

2.1 Git

Git ist ein von der Linux-Entwickler-Gemeinschaft und vor allem Linus Torvalds selber getriebenes und entwickeltes Versionsverwaltungssystem, welches vorerst dem Zweck diente die Versionskontrolle für den Linux-Kernel zu übernehmen [CSb]. Die Beliebtheit und der Einsatz von Git nahm über die Jahre immer mehr zu. Laut der Entwicklerumfrage von *Stackoverflow* [Sta17] lag der Anteil von Entwicklern, die im Jahr 2017, zum Verwalten ihres Quellcodes Git benutzten bei 69,2%. Somit ist Git das mit Abstand meist genutzte Versionsverwaltungssystem, gefolgt von Apache Subversion mit 9,1% und Microsoft Team Foundation Server mit 7,3%.
Im Folgenden sollen die Grundkonzepte und fortgeschrittenen Themen von Git beleuchtet werden, damit die spätere Anwendung auf die Welt der Microservices einfacher nachzuvollziehen ist.

2.1.1 Distributed Version Control Systems

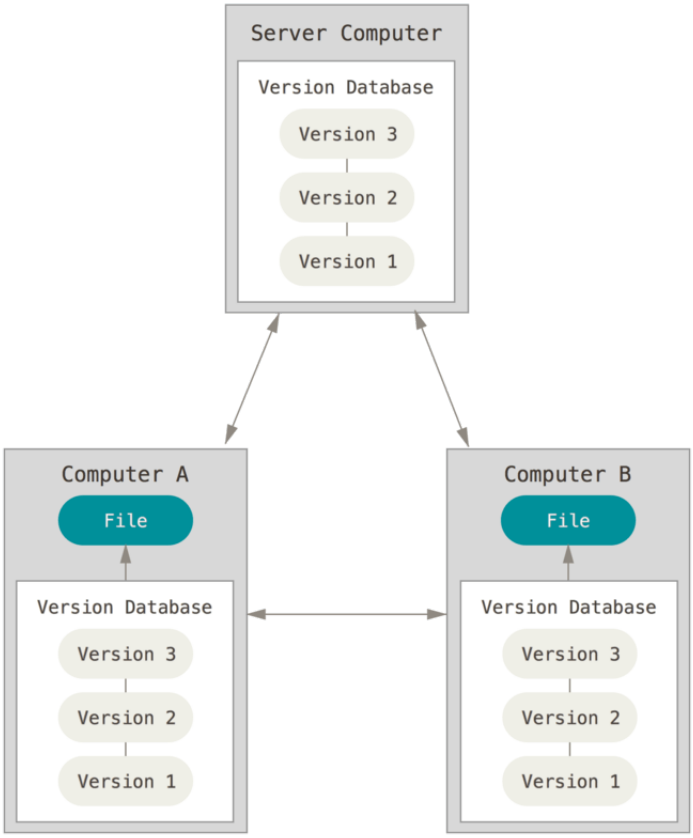

Abbildung 1: Verteiltes Versionsverwaltungssystem (nach [CSb])

Bei Git handelt es sich um ein sogenanntes verteiltes Versionsverwaltungssystem[CSb]. Mit so einem System erhält jeder Client nicht nur den neusten Stand der Dateien, sondern klont sich das ganze Repository mitsamt seiner Historie (siehe Abbildung 1). Somit ist jeder Klon ein Backup der Daten und das Repository wird als Single Point of Failure ausgeschlossen. Da alle Daten lokal vorhanden sind, können auch der Großteil von ausgeführten Operationen lokal erfolgen. Dies ermöglicht ein schnelles arbeiten ohne ständige Netzwerkabhängigkeit.

2.1.2 Commit

Die Git-Operation, die verwendet wird um Veränderungen an Projektdaten in die lokale Datenbank der Versionsverwaltung (auch Git-Repository genannt) zu speichern, lautet *Commit*. Jede Datei die von Git verwaltet wird kann sich gemäß [CSb, S. 15-16] in eine der drei Stadien befinden:

- Commited - die Datei wurde in der lokalen Datenbank gespeichert
- Modified - die Datei wurde verändert aber noch nicht in der lokalen Datenbank gespeichert
- Staged - Die Datei wurde ausgewählt um mit der nächsten Commit-Operation in der lokalen Datenbank gespeichert zu werden

Wie aus der Beschreibung zu entnehmen ist das *Commiten* eine lokale Operation und führt zu keinerlei Änderung bei entfernten Repositorys. Vor jedem Commit wird eine SHA-1 Prüfsumme auf Basis von verschiedenen Meta-Informationen, wie zum Beispiel der mitgegebenen Commit-Nachricht, dem Datum des Commits, etc. und dem Hash des gesamten Git-Verzeichnissbaumes, erzeugt [Bur14]. Diese Prüfsumme kann dann als eindeutige Identifikation für den Commit genutzt werden. So ist es beispielsweise möglich, gezielt zu einem Commit zurück zu springen.

2.1.3 Push / Pull

Entfernte Repositorys (engl. remote Repository) sind Versionen des verwalteten Projektes, welche über das Internet oder andere Netzwerke verfügbar gemacht werden [CSb, S. 49]. Diese Remotes sind für das Zusammenarbeiten mehrerer Entwickler oder das Arbeiten über verschiedener Maschinen hinweg notwendig. Nach dem Verknüpfen des lokalen Repositorys mit einem entfernten Repository werden mit dem *Push*-Befehl alle committeten Änderungen an den Remote übertragen. Kommt es hierbei zu einem Konflikt beim Zusammenführen des Quellcodes, so wird der Entwickler dazu aufgefordert diese aufzulösen. Das Gegenstück zum Push stellt die Pull-Operation dar. Mit diesem Befehl werden alle neuen Änderungen, die auf dem Server vorhanden sind, in die lokale Datenbank heruntergeladen und versucht in den bestehenden lokalen Code zu integrieren. Soll die Zusammenführung nicht erfolgen, so kann man stattdessen mit der Fetch-Operation nur den Download in das lokale Repository durchführen.

2.1.4 Tagging

Möchte man einen bestimmten Punkt in der Git-Historie markieren, so könnte man sich entweder den Commit-Hash dieser Stelle merken, oder aber man erstellt ein Git-Tag. Der Vorteil von Tags ist dabei, dass man für wichtige Ereignisse, zum Beispiel bei der Veröffentlichung (engl. release) einer neuen Version der Software, den Bezeichner für die Markierung selbst wählen kann. Neben der Bezeichnung können dem Tag des weiteren Kommentare hinzugefügt werden. Wie bei den Commit-Prüfsummen kann man unter der Angabe des Tag-Bezeichners zu dem Punkt des Codestandes springen, welcher markiert wird.

2.1.5 Branching / Merging

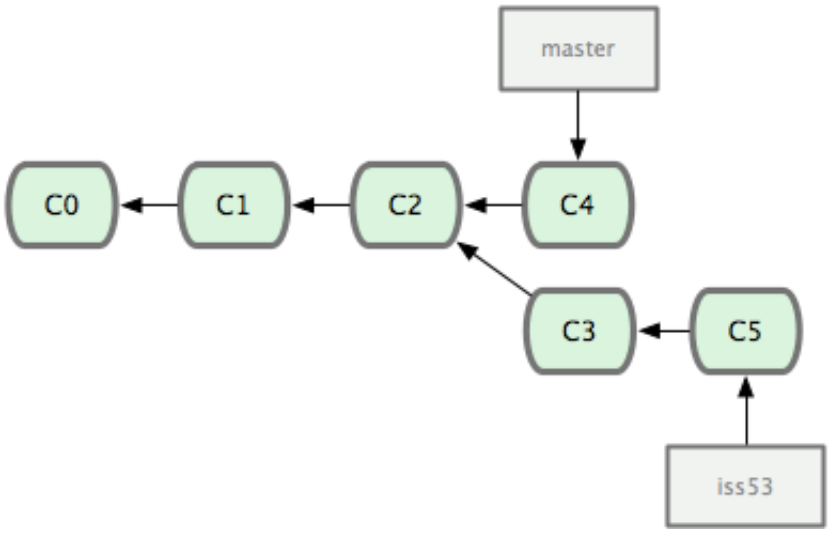

Abbildung 2: Auseinanderlaufende Codestände durch Branching (nach [CSa])

Das Branching beschreibt den Vorgang des Abweichens von einem Entwicklungszweig und dem weiteren Arbeiten mit diesem Codestand ohne den ursprünglichen Zweig zu beeinflussen. Ein Branch wird dabei von einem benannten Pointer repräsentiert, welcher auf den letzten Commit des Zweiges zeigt. Da das Branchen und das Wechseln zwischen den Zweigen in Git sehr leichtgewichtig ist, wird der Entwickler dazu angehalten seinen Codestand oft zu verzweigen. In Abbildung 2 wird an einem Beispiel das Branching verdeutlicht. Es ist zu sehen, dass bis zum Commit *C2* der Entwicklungszweig nur aus dem Master-Branch bestand. Ab dem besagten Commit wurde ein Branch namens *iss53* abgezweigt und unter diesem die weiteren Änderungen *C3* und *C5* vorgenommen. In der Weile wurde auch der Hauptentwicklungszweig mit dem *C4* Commit weiter entwickelt.
Der Prozess welcher divergierende Codestände wieder zusammenführt, wird als Merging bezeichnet. Bei diesem kann es zu so genannten Merge-Konflikten kommen, diese entstehen beispielsweise, wenn eine Datei in zwei verschiedenen Zweigen an der selben stelle verändert wurde. Möchte man diese Zweige nun zusammenführen, so muss erst die Auflösung aller Konflikte erfolgen. Soll eine Abzweigung in einen Branch zurückgeführt werden, welcher sich in der Zwischenzeit nicht verändert hat, so kann und wird, wenn

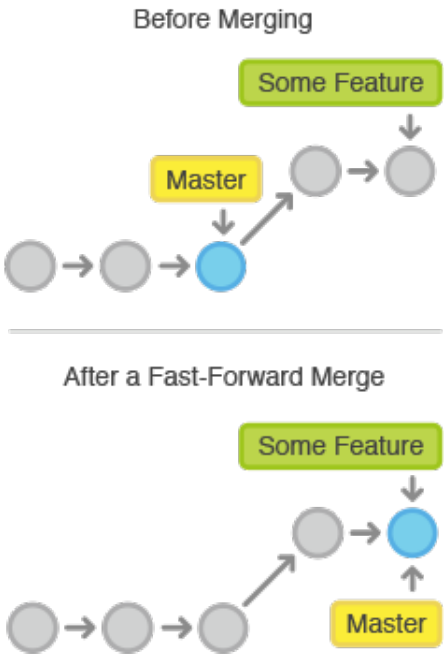

Abbildung 3: Fast-Forward-Merge (nach [Sas16])

nicht anders in Git konfiguriert, ein Fast-Forward-Merge durchgeführt [CSb, S. 70-71]. Bei diesem ist es Aufgrund des nicht Divergieren der Entwicklungsstränge möglich, den Branch-Pointer des Hauptzweiges (in Abbildung 3 als Master bezeichnet) auf den letzten Commit, des zu integrierenden Zweiges (in Abbildung 3 als Some Feature bezeichnet), zeigen zu lassen.

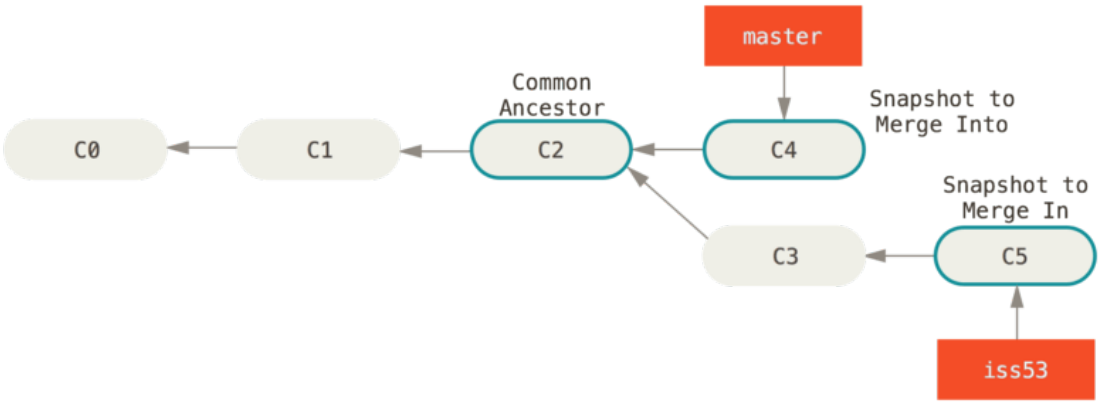

Abbildung 4: Verwendung von drei Schnappschüssen in einem Merge (nach [CSb, S. 73])

Das Mergen von Zweigen deren Codestände sich auseinander entwickelt haben (siehe Abbildung 2), können also nicht ohne weiteres mit einem Fast-Forward Merge vereint werden. Um dies zu bewerkstelligen muss Git einen so genannten Three-Way-Merge durchführen [CSb, S. 72-73]. Da der Commit des zu integrierenden Branches, wie in

Abbildung 4 zu sehen, kein direkter Nachfahre des Zweiges ist, in den der Merge durchgeführt werden soll, muss Git für die Zusammenführung weitere Operationen tätigen. Auf Grundlagen der Commits an den Enden der zu mergenden Zweige und dessen gemeinsamer Vorfahr-Commit erzeugt Git nun einen neuen Merge-Commit (in Abbildung 5 als C6 bezeichnet), der als Besonderheit mehrere Vorgänger-Commits (in Abbildung 5 als C4 und C5 zu sehen) besitzt.

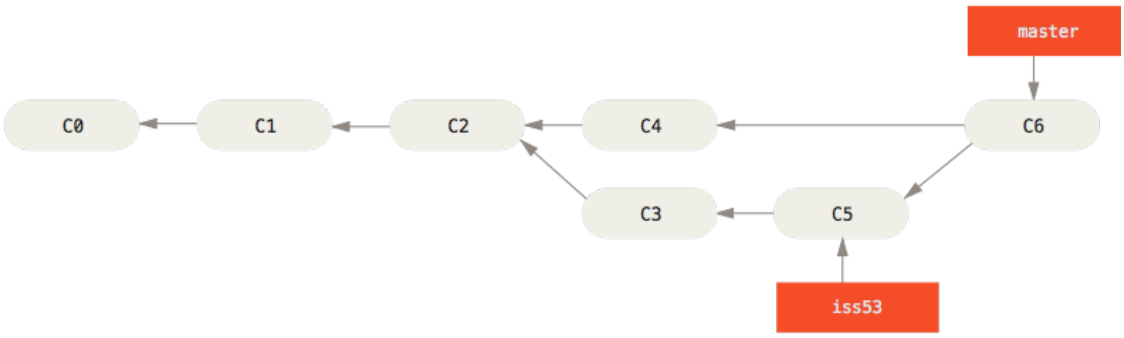

Abbildung 5: Git-Baum nach einem erfolgreichen Merge mit Merge-Commit (nach [CSb, S. 73])

2.1.6 Branching Strategien

Im Git-Versionsverwaltungssystem haben alle Branches den selben Stellenwert und funktionieren auf die selbe Weise. Um jedoch einen, für den Entwickler oder auch für automatisierte Build-und Deploy-Systeme, organisierten Arbeitsverlauf (engl. Workflow) zu erhalten, kann eine semantische Trennung der Branches in zwei größere Hauptgruppen vorgenommen werden.

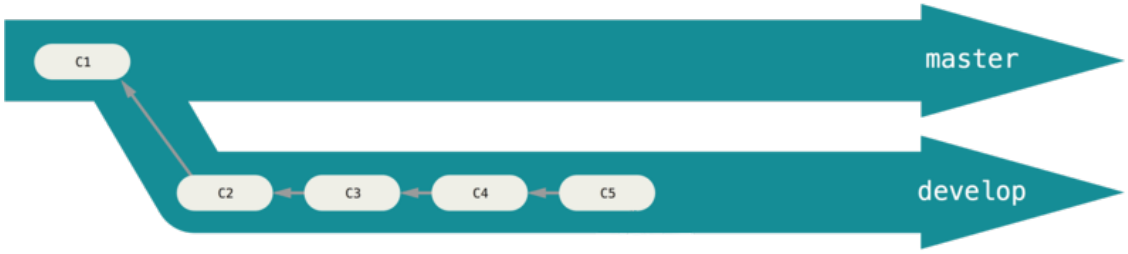

Abbildung 6: Long-Running-Branches (Vereinfachte Abbildung in Anlehnung an [CSb, S. 78])

Long-Running-Branches Wie der Name verlauten lässt sind Long-Running-Branches Zweige, die für einen langen Zeitraum beziehungsweise für die gesamte Dauer des Entwicklungsprozesses offen sind [CSb, S. 78]. Diese werden in der Regel dazu genutzt um verschiedene Entwicklungsstadien des Codes abzubilden. Die Abbildung 6 zeigt zum

Beispiel ein Vorgehen, bei dem der stabile Codestand im Master-Branch verbleibt. Auf diesem aufbauend zweigt sich der Develop-Branch ab, in dem die aktive Weiterentwicklung der Anwendung stattfindet. Wird ein Punkt erreicht, an dem der Entwicklungszweig, nach ergiebigen Tests und Code-Reviews als stabil genug empfunden, so kann eine Zurückführung in den Master-Branch erfolgen. Somit bildet dieser nun den neusten stabilen Stand ab und der Develop-Branch kann zur weiteren Entwicklung wiederverwendet werden. Ein Vorteil dieser Branching-Strategie ist, dass sich die verschiedene Zweige gut für das Bespielen unterschiedlicher Umgebungen eignen. So könnte man in dem vereinfachten Beispiel den Stand des Master-Branches in einer Produktivumgebung ausführen, während der Develop-Zweig in einer Entwicklungsumgebung läuft.

Feature- / Topic-Branches Im Gegensatz zu den oben erläuterten Long-Running Branches sind Feature- Branches, gemäß [CSb, S. 79], kurzlebige Zweige, welche nur zur Entwicklung einer spezifischen Funktionalität dienen. Diese Vorgehensweise erlaubt es verschiedene Funktionalitäten, in der Entwicklungsphase zu isolieren, und damit einfacher kontextabhängig testbar und überschaubar zu machen. Sobald das Feature fertig gestellt ist, wird der neue Code in den Ursprungszweig zurückgeführt und der Feature-Branch gelöscht. Der in Abbildung 4 als *iss53* bezeichnete Zweig ist ein solcher Feature-Branch, welcher in Abbildung 5 mit dem Hauptzweig zusammengeführt wird, und anschließend entfernt werden sollte.

2.1.7 Pull-Requests

Ein Pull-Request (kurz PR) ist eine Funktionalität, die von vielen Git-Repository-Hosting Anwendungen zur vereinfachten Zusammenarbeit zwischen Entwicklern, angeboten wird. Der Name setzt sich hierbei aus den englischen Begriffen *request*, für Anfrage und *pull*, bezeichnend für die in Unterunterabschnitt 2.1.3 beschriebene Operation. Der Anfragesteller eines PRs fordert also einen oder mehrere Entwickler dazu auf, seinen neuen Code zu "pullen". Die einfachste Form eines Pull-Requests stellt ein Mechanismus dar, welcher die Teammitglieder eines Entwicklers darüber informiert, dass dieser einen zur Integration bereiten Feature-Branch besitzt [Atl]. Um einen Pull-Request zu öffnen muss der Antragssteller jeweils das Quell-Repository und Quell-Zweig, sowie auch die dazugehörigen Ziele, in welche der Code integriert werden soll, angeben. Ist der Pull-Request erstellt worden, so werden alle von der Integration betroffenen Entwickler benachrichtigt und dazu aufgefordert eine Diskussion über die neue Funktionalität zu führen, bevor der Code wirklich integriert wird. Abhängig von der Qualität des eingereichten Codes, können die verantwortlichen Teammitglieder, eine Verbesserung anfordern und durch Kommentare die Defizite des Codes spezifizieren, die Verbesserung selber durchführen, die Integrationsanfrage ablehnen oder aber den Code zum Mergen freigeben.
Unabhängig von einem manuellen Codereview wird meistens ein erfolgreicher Statusbericht eines automatisierten Build-Prozesses als Voraussetzung für die Integration gestellt. Der sogenannte Pull-Request-Build (kurz PR-Build) wird in der Regel über Web-Hooks initiiert. Mithilfe dieser werden an einen Build-Server alle relevanten Meta-Daten, wie zum Beispiel die Adresse des Git-Repositorys oder die relevanten Ziel-und Quellinformation der Branches, an die entsprechend lauschenden Endpunkte übermittelt. Das daraus resultierende Auslösen des Buildes findet normalerweise beim Öffnen eines Pull-

Requestes oder bei der Veränderung der Quell- oder Zielbranches statt. Gemäß [CSb, S. 352-357] werden für das Ausführen von Skripten, unter Auftreten bestimmter Ereignisse, Git Hooks verwendet. Diese werden in zwei Gruppen kategorisiert:

- **clientseitige Hooks** - sind Hooks, die durch Operationen an dem lokalen Repository ausgelöst werden. So gibt es beispielsweise Client-Side Hooks welche eine Fehlermeldung ausgeben, wenn beim Committen die Nachricht fehlt, oder die E-mail-Adresse des Commit-Autoren nicht angegeben wurde.
- **serverseitige Hooks** - sind Hooks, die durch netzwerkabhängige Operationen ausgelöst werden. Das heißt das Auslösen dieser Hooks findet entweder vor oder nach einem Push statt. Die im oberen Abschnitt erwähnten Web-Hooks sind eine Spezialisierung, welche HTTP-Requests absenden.

2.2 Kubernetes

Kubernetes (griech. für Steuermann oder Pilot) ist ein von Google designetes System für das automatisierte Verteilen, Skalieren und Verwalten containerisierter Anwendungen [Kub16a].Das System ist seit 2014 Open Source und wird nun von der *Cloud Native Computing Foundation* gepflegt. Laut deren Umfrage[Con18] hat sich Kubernetes als Standard für die Container-Orchestrierung umgesetzt und hat Konkurrenten wie zum Beispiel *Docker Swarm* so gut wie vollkommen verdrängt. Mit der zunehmenden, weltweiten Akzeptanz und Verwendung von Containern steigt auch dessen Einsatz, nicht nur als eigenständige Technologie, sondern auch als Teil einer größeren Containerisierungs-Strategie. In mitten des Container-Aufschwungs steht Docker, den derzeit Führenden in Sachen Software-Container-Platform. Laut Angaben von Docker CEO *Ben Golub*, die auf der *DockerCon 2017* getroffen wurde, gab es zu diesem Zeitpunkt: 14 Millionen Docker-Hosts, 900 Tausend Docker-Applikationen, 12 Milliarden Docker-Image-Pulls und 3330 Entwickler, welche am Docker-Projekt mitgewirkt haben [Gol17]. Das in 2013 erschienene Docker hat also in sehr kurzer Zeit, einen sehr großen Einfluss darauf genommen, wie heutzutage Software entwickelt wird. Anders als die Container-Orchestrierungen wie *Kubernetes* oder *Docker Swarm*, gibt es Software-Container an sich schon seit geraumer Zeit. Gemäß [Str16] fanden die Anfänge der Containerisierung schon 1979 mit *UNIX V7* statt. Was Docker aber von der damaligen und heutigen Konkurrenz absetzt, sind die diversen Werkzeuge und generelle Benutzerfreundlichkeit die sie eingeführt haben [MG, S. 12]. Die durch Container gebotene Isolation auf der Betriebssystemebene, bieten eine ideale Grundlage für Microservices (vgl. Abschnitt 3) und Continuous Integration/Continuous Delivery (vgl. Abschnitt 4). Mit dem immer mehr zunehmenden Einsatz dieser Technologien und fortlaufenden Adoption der damit verbundenen Denkweisen ist eine Strategie nötig, all diese Container und Microservices zu verwalten. Insbesondere, wenn Unternehmen und Organisationen hunderte oder gar tausende Container simultan betreiben wollen [Bai17, S. 13].
Die Grundlagen der dafür entworfene Lösung, die Container-Orchestrierung insbesondere Kubernetes, sollen im folgenden bekannt gemacht werden.

2.2.1 Definition Docker-Container

Gemäß [Doc] ist ein Container-Abbild ein leichtgewichtiges, eigenständiges, ausführbares Paket. In diesem befinden sich ein Stück Software und Alles was dazu benötigt wird, diese ausführen zu können. Beispielsweise gehören dazu der Quellcode, eine Laufzeitumgebung, Systembibliotheken und Konfigurationen. Unabhängig von ihrer Umgebung, sei es eine Windows- oder Linux basierte Anwendung, wird containerisierte Software immer gleich ausgeführt. Jeder Container isoliert seine innewohnende Software von äußeren Einflüssen, so können Entwicklungs- und Bereitstellungsumgebungen getrennt werden und Konflikte bei der Ausführung verschiedener Software auf der gleichen Infrastruktur gar nicht erst aufkommen.

2.2.2 Vergleich zwischen Containern und virtuellen Maschinen

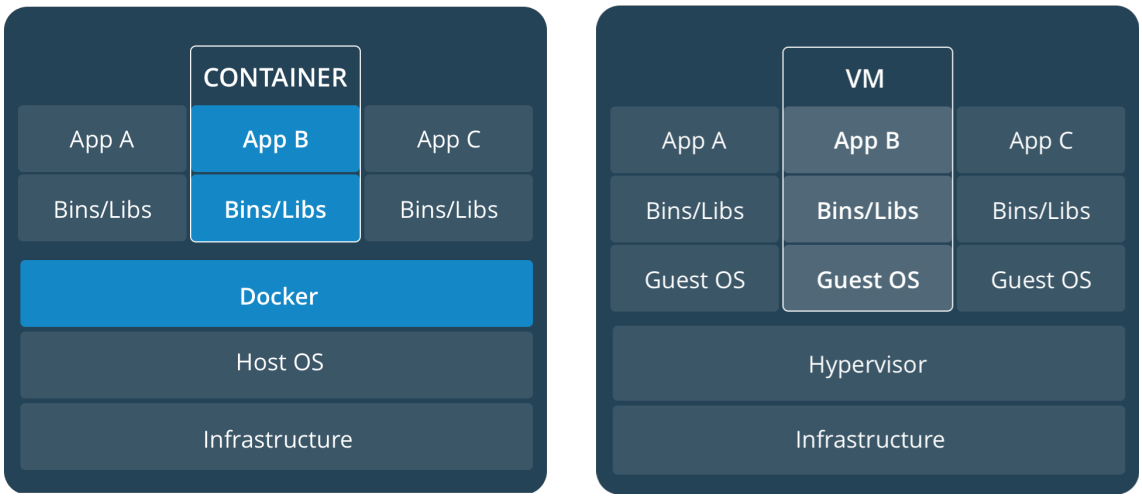

Abbildung 7: Vergleich der Architektur von Containern (links) und virtuellen Maschinen (rechts) (nach [Doc])

Container werden von vielen als leichtgewichtige virtuelle Maschinen (kurz VM) bezeichnet. Obwohl beide Technologien gemeinsame Merkmale teilen ist die darunter zugrunde liegende Architektur grundsätzlich verschieden [Doc16]. Die zwei größten Gemeinsamkeiten sind hierbei laut [Doc16], dass sowohl Container als auch VMs eine isolierte Umgebung zur Ausführung von Anwendungen bieten und beiden dieser Umgebungen als binäres Artefakt repräsentiert werden. Dies erlaubt es schnell zwischen verschiedenen Hosts zu wechseln. Der Unterschied liegt darin, dass bei VMs für jede virtualisierte Anwendung nicht nur die Applikation und dessen Abhängigkeiten an sich bezogen wird, sondern außerdem auch das ganze Gast-Betriebssystem (siehe Abbildung 7 rechts). Dabei abstrahieren VMs physische Hardware und erschaffen so viele verschiedene Maschinen, die mithilfe des Hypervisors auf einer echten Maschine ausgeführt wird [Doc]. Da jede der VMs ihre eigene Infrastruktur und Betriebssystem besitzt, kann das System in seiner Größe aufgebläht werden, wodurch mehr Speicherplatz benötigt wird und die Hochfahrzeiten leiden. Ein Docker-Container beinhaltet, wie in Abbildung 7 links zu sehen, nur die Anwendungen und deren benötigten Abhängigkeiten. Container sind dazu konzipiert eine geteilte Infrastruktur zu nutzen [Doc16]. Mehrere Container können so auf der selben Maschine laufen während sie den Kernel des Betriebssystems miteinander teilen, wobei jeder als abgeschirmter Prozess im Userspace ausgeführt wird[Doc]. Aus diesem Grund benötigen Container im Gegensatz zu VMs weniger Speicherplatz, sind portabler, effizienter und fast immer sofort einsatzbereit.

2.2.3 Aufbau eines Kubernetes Clusters

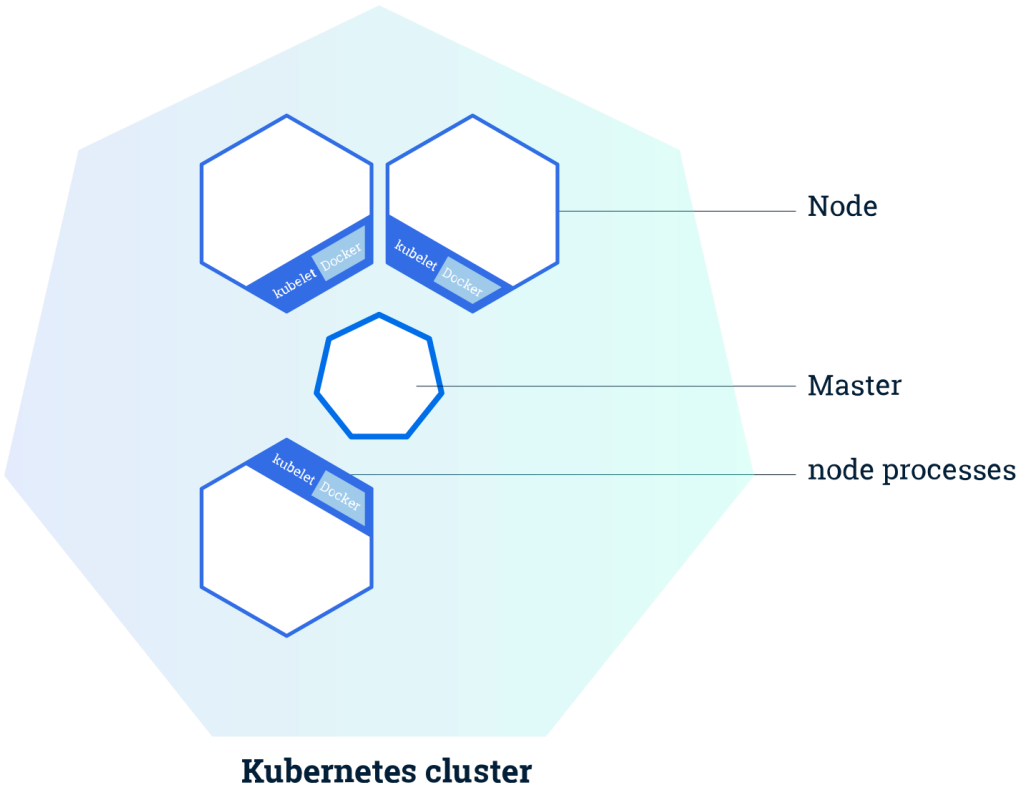

Abbildung 8: Grundlegende Bestandteile eines Kubernetes Clusters(nach [Per18c])

Um die Orchestrierung von Containern durchführen zu können, koordiniert Kubernetes ein Hochverfügbarkeits-Cluster (engl. high availability oder kurz HA) bestehend aus einem Verbund von Computern, welche als eine Einheit fungieren [Per18c]. Kubernetes ist also für das automatisierte Verteilen und Planen von Container-Applikationen zuständig. Wie Abbildung 8 zu entnehmen ist besteht ein Cluster aus zwei verschiedenen Elementtypen: dem Master, der die Koordination des Clusters übernimmt, und Nodes, welche die verschiedenen Maschinen, darstellen auf denen die Anwendungen ausgeführt werden. Die Nodes können dabei, sowohl virtueller und auch physischer Natur sein.
Der Kubernetes Master verwaltet alle den Cluster betreffenden Aktivitäten und ist dafür zuständig den gewünschten Zustand des Clusters herbeizuführen oder beizubehalten [Per18c]. Des weiteren ist er für die Verteilungsplanung der Applikationen und dem Ausrollen von dessen Updates verantwortlich. Die zur Verwaltung benötigten Prozesse laufen gewöhnlich auf einem Node im Cluster, können jedoch auch unter dem Nutzen einer höheren Erreichbarkeit repliziert werden.
Auf jedem Node wird, beim Eingliedern in das Cluster, eine sogenanntes *kubelet* installiert (siehe Abbildung 8 an jedem Node). Das kubelet ist ein Software-Agent, welcher über die vom Kubernetes Master bereitgestellte Kubernetes API kommuniziert. Selbige

API kann vom Endnutzer verwendet werden um mit dem Cluster zu interagieren. Eine direkte Interaktion mit den Nodes ist zwar möglich, sollte im Normalfall aber nicht nötig sein.

2.2.4 Pods

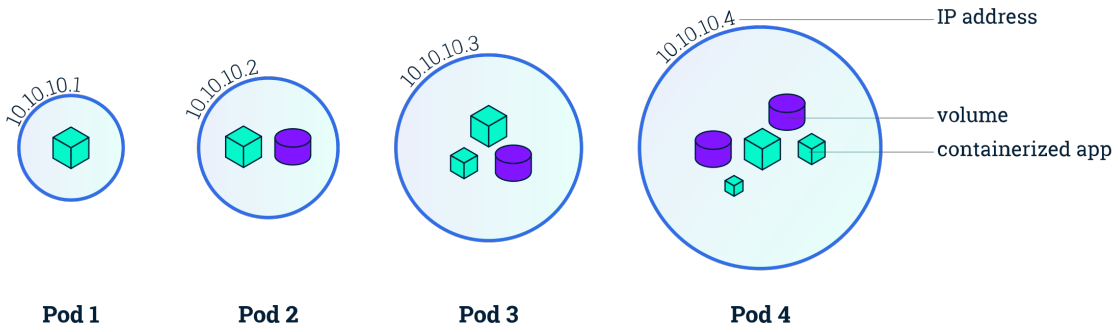

Abbildung 9: Verschiedene Kubernetes Pods(nach [Per18d])

Pods stellen die kleinste Einheit eines Kubernetes Clusters dar und repräsentieren einen laufenden Prozess [TPJ18b]. Innerhalb eines Pods ist, wie Abbildung 9 anhand der grünen Würfel zeigt, das Starten von mehreren Containern möglich, welche relativ eng miteinander gekoppelt sind. Diese teilen sich Ressourcen und Abhängigkeiten, wie zum Beispiel Speicher in Form von Volumen oder das selbe Netzwerk anhand der einzigartigen Cluster-IP-Adresse. Aufgrund der Zusammengehörigkeit werden Container eines Pods immer auf einem Node ausgeführt (siehe Abbildung 10). Soll die Anwendung horizontal skaliert werden, so findet eine Replikation der Pods statt. Jeder so erzeugte Pod stellt dann eine eigenständige Instanz der Anwendung dar.

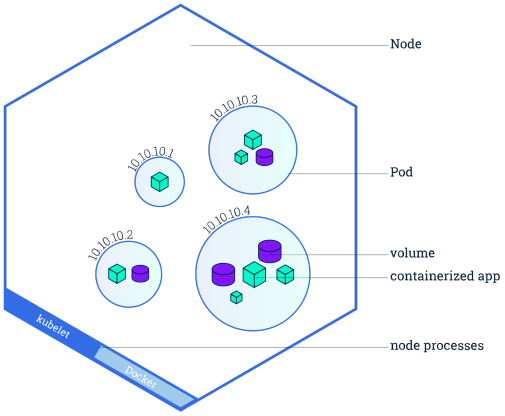

Abbildung 10: Pods werden auf der selben Node platziert(nach [Per18d])

2.2.5 Kubernetes-Deployments

Beginnend mit der Kubernetes Version 1.2 wurde das Deployment-Objekt hinzugefügt. Deployments erlauben es, Applikationen auszurollen, diesen Vorgang zu pausieren und wieder aufzunehmen, Anwendungen zu skalieren und Rolling Updates/Releases durchzuführen. Sie gewähren demnach eine feinere Kontrolle über den Verteilvorgang selbst [Bai17, vgl S. 124-133]. Die Konfiguration der Deployments finden deklarativ statt. Es wird ein Soll-Zustand beschrieben, den das Deployment versucht herbeizuführen. So legt das Deployment beispielsweise neue Repliken einer Anwendung an oder entfernt alle definierten Pods [TPJ+18a]. Wurde eine Instanz der zu verteilenden Anwendung, in Form eines Pods, angelegt, folgt eine Überwachung dessen durch den Deployment-Controller. Stürzt nun ein dem Deployment zugehöriger Pod, oder gar die gesamte Node ab oder wird entfernt, erfolgt eine Ersetzung durch den Controller. Dies bietet eine Lösung gegen das Problem des Maschinenversagens. Somit ist der Vorteil gegenüber Ansätzen die nicht aus dem Container-Orchestrierungs-Umfeld kommen der, dass Anwendungen nicht nur instanziiert, sondern auch am Laufen gehalten werden[Per18a].

2.2.6 Kubernetes-Services

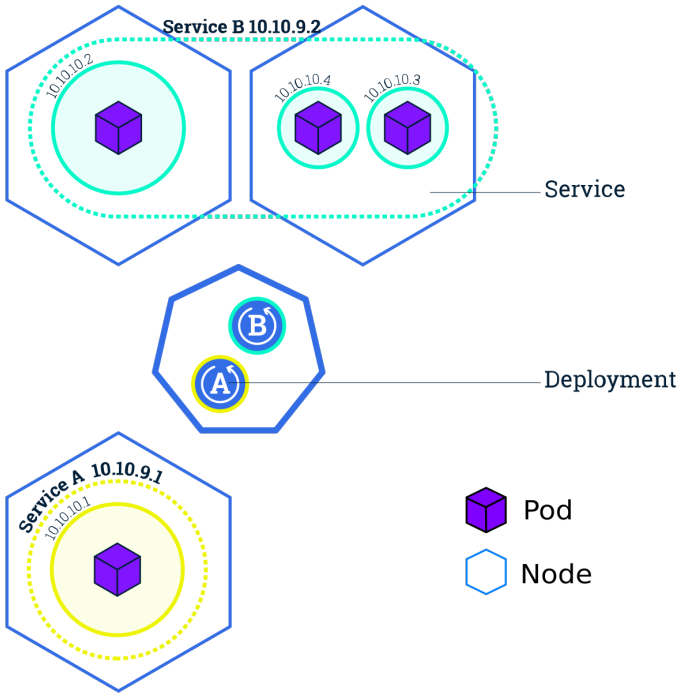

Abbildung 11: Pods werden durch Kubernetes-Services anhand einer Cluster-IP freigegeben (nach [Per18a])

Die in diesem Abschnitt benannten Kubernetes-Services, dürfen nicht mit den Services verwechselt werden, wie man sie im Kontext von Microservices verwendet. Um

Missverständnisse zu vermeiden, werden die Kubernetes-Objekte mit der Bezeichnung *Service*, im weiteren Verlauf dieser Arbeit als Kubernetes-Services angegeben. Alle anderen Nutzungen der Worte Microservice, Service oder Dienst beziehen sich auf die Teile einer Microservice-Anwendung.

Laut Aussagen der Kubernetes Dokumentation [TPJ+18c], sind Kubernetes-Services eine Abstraktion, welche eine Menge von Pods und eine Regel, um auf diese zuzugreifen, definiert. Durch Kubernetes-Services wird eine lose Koppelung, in Abbildung 11 an den eingrenzenden gestrichelten Linien zu erkennen, zwischen voneinander abhängigen Pods geschaffen. Damit nur bestimmte Pods von den Regeln eines Kubernetes-Services beeinflusst werden, wählt man diese über einen sogenannten *Label Selector* aus [Per18b] Labels sind Schlüssel-Werte Paare, die an jedes Kubernetes-Objekt angefügt werden können. Unter Nutzung der Labels kann man somit jeden Pod oder einer Gruppe von Pods einen selbst gewählten Bezeichner mit eigenen Kriterien, zur eindeutigen Identifikation, geben.
Wie in Unterunterabschnitt 2.2.4 beschrieben, besitzt jeder Pod eine einzigartige Cluster-IP-Adresse (siehe auch Abbildung 11). Auf diese kann jedoch, ohne die Nutzung eines Kubernetes-Services, nicht von Außen zugegriffen werden. Je nachdem welchen Typ der Kubernetes-Service besitzt, ändert sich das Verhalten der Freigabe. Gemäß [TPJ+18c] gibt es folgende Typen:

- **ClusterIP** ClusterIP ist der voreingestellte Service-Typ bei der Erstellung eines Kubernetes-Serivces. Der Service wird über eine Cluster-IP-Adresse angesprochen und kann somit nur innerhalb des Clusters kontaktiert werden.

- **NodePort** Die Freigabe des Kubernetes-Services findet beim NodePort über die IP-Adresse des Nodes statt. An diesem wird ein statischer Port geöffnet über den man dann in der Form *<NodeIP>:<NodePort>* zugreifen kann.

- **LoadBalancer** Dieser ServiceTyp kann nur genutzt werden, wenn der Bereitsteller des Kubernetes-Clusters den Einsatz eines externen Load-Balancers unterstützt. Ist dies der Fall, so wird dem Service die IP des Load-Balancers zugewiesen.

- **ExternalName** Der Kubernetes-Service wird über einen beliebigen Namen zugänglich gemacht, indem ein CNAME Eintrag mit dem Namen zurückgegeben wird.

3 Aufbau einer Microservice-Anwendung

3.1 Definition von Microservices

Der Begriff Microservice beschreibt die Verwendung eines bestimmten Architekturmusters in der Softwareentwicklung, bei dem eine einzelne, komplexe Anwendung aus einer Vielzahl von Einzelsystemen zusammengesetzt ist [FL14]. Diese Komponenten, genannt Microservices, Dienste oder Services sind autarke Einheiten, welche über leichtgewichtige Mechanismen und Schnittstellen, wie zum Beispiel einer RESTful API, kommunizieren. Jeder dieser Dienste laufen in ihrem eigenem Prozess und können somit separat auf virtuellen Maschinen und Containern ausgeführt werden.

3.2 Vergleich zwischen Microservices und Monolithen

Um Microservices besser verstehen zu können bietet sich ein Vergleich mit dessen architektonischen Gegenstück, dem monolithischen Stil, an. Ein Monolith ist eine Anwendung, die nur eine einzige ausführbare Datei besitzt und als Einheit konzipiert ist. Zwar kann eine Aufteilung der Anwendung über Klassen, Komponenten und Namensräume erfolgen, trotzdem bleibt die starre Kopplung zwischen den Teilen der Software bestehen und eine Modularisierung ist nur schwierig umzusetzen. Unternehmenssoftware, bei denen die Monolith-Architektur oft zum Einsatz kommt, bestehen gewöhnlich aus drei Teilen:

- Einem Präsentationsteil der die Nutzerschnittstelle, meist in Form von HTML-Seiten im Browser des Nutzers, darstellt.
- Einer Anwendungslogik, welche eine serverseitige Anwendung ist, die den eigentlichen Großteil der Funktionalität zur Verfügung stellt.
- Einer Datenbank, die benötigte Daten speichert und abrufbar macht.

Jegliche Änderung Einer dieser Teile zieht einen kompletten Build- und Verteilzyklus nach sich, denn es muss die gesamte Software neu paketiert, ausgeliefert und schließlich gestartet werden. Dies kann bei einem groß gewachsenem Softwareprojekt sehr viel Zeit und Rechenleistung in Anspruch nehmen. Die generelle Wart- und Erweiterbarkeit ist nur bedingt durchführbar, da durch die fehlende Modularisierung, nur mit erheblichen Aufwand Änderungen vollzogen werden können, die keine ungewünschte Beeinflussungen der restlichen Codebasis mit sich ziehen. Es ist außerdem nicht möglich einzelne Bereiche der Anwendung zu skalieren. Um erhöhte Last für einen Anwendungsfall zu bearbeiten muss die Gesamtanwendung skaliert werden.
Aufgrund der genannten Nachteile kam es in den letzten Jahren immer mehr zum Einsatz des Microservice Architekturmusters. Dieses löst viele genannte Probleme allein durch die Modularisierung der Anwendung und dessen Design als Sammlung von unabhängigen aber dennoch zusammengehörigen Diensten [FL14]. Es wird darauf hingearbeitet Dienste so zu entwickeln, dass sie zu jeder Zeit austauschbar sind. Da die Microservices einzeln ausführbare Anwendungen sind, muss eine Umstellung der Kommunikation innerhalb der Anwendung erfolgen. Bei einem Monolithen werden Methodenaufrufe aus dem Systemspeicher heraus getätigt. Eine Microservice-Applikation nutzt hingegen Service-Endpunkte, in den meisten Fällen sind dies die HTTP-Endpunkte einer RESTful

API, leichtgewichtiges Messaging oder oft auch eine Kombination beider. Die genannten Kommunikationsarten können sowohl zur Inter-Service-Kommunikation als auch zum Empfangen von Anfragen außerhalb des Systemes genutzt werden.

3.3 Wichtige Eigenschaften eines Microservices

Um die Komposition von Microservices besser bewältigen zu können, ist es zuerst wichtig die Grundbausteine eines solchen Systemes zu untersuchen. Deshalb werden in diesem Abschnitt der allgemeine Aufbau und bewährte Vorgehensweisen bei der Entwicklung eines Microservices beleuchtet. Es existieren viele verschiedene Definitionen von Microservices, welche aber Alle folgende Eigenschaften gemein haben.

3.3.1 Zustandslosigkeit

Bei dem Entwickeln einer Microservice-Anwendung ist zu beachten, dass der überwiegende Teil an Dienste zustandslos sein sollte. Falls es doch nötig ist Daten zu persistieren, so speichert man diese in zustandsbehafteten, unterstützenden Diensten [Wig11]. Es ist eine klare Trennung zwischen zustandslosen und zustandsbehafteten Services durchzuführen. Wird diese Bedingung erfüllt, so kann man einen der Hauptvorteile von Microservice-Architekturen, die Möglichkeit zur schnellen Skalierung, ausnutzen. In Verbindung mit der Containerisierung, ermöglicht die Unabhängigkeit eines zustandslosen Dienstes das starten mehrerer Instanzen auf vielen verschiedenen Maschinen. Die Skalierung findet nicht mehr auf der Maschinenebene beziehungsweise auf VM-Ebene statt, sondern vielmehr an den spezifischen Services an sich. Wie schon genannt ist aber nicht alles Zustandslos realisierbar. Dateisysteme und Datenbanken, die einer Applikation angehören, sind im Regelfall zustandsbehaftet implementiert. Dementsprechend sind diese sehr an bestimmte Hosts gebunden und somit schwieriger zu verteilen. Einen Lösungsansatz zu diesem Problem stellen beispielsweise die *Persistent Volumes* [TPM18] der Containerorchestrierung Kubernetes dar. Diese entsprechen Speicher auf einer separat angelegten Persistenzschicht, welche unabhängig vom Host-System genutzt werden können.

3.3.2 Dezentralisierte Abhängigkeiten

Die Grundprinzipien der Microservice-Architektur besagen, dass jeder Service eigenständig und abgeschlossen sein sollte. Insbesondere beim Einsatz von Containern stellen Services eine Einheit aus Betriebssystem, Frameworks, Laufzeitumgebungen und Abhängigkeiten [MSV16], wie zum Beispiel Programmbibliotheken, dar. Auch wenn zwei verschiedene Services dieselbe Abhängigkeit besitzen, sollte man davon absehen diese zentral zu hinterlegen und zwischen den Diensten zu teilen. Beschriebenes Vorgehen würde die Autonomie der Microservices beeinträchtigen und somit in den Punkten Schnelligkeit und Flexibilität behindern. Ein Aktualisierung der Abhängigkeiten auf eine neue Version kann zum Fehlschlagen eines Services führen, während der Andere ohne Probleme weiter funktioniert.
Es Sollte nie davon ausgegangen werden, dass eine Abhängigkeit auf dem Zielsystem vorhanden ist [Wig11]. Aus diesem Grund ist es wichtig, alle Abhängigkeiten explizit zu deklarieren. Die Arbeit mit Containern verhindert durch deren Abgeschlossenheit und

der dadurch benötigten eindeutigen Abhängigkeitsdeklaration einen Fall, bei dem es zu geteilten Abhängigkeiten kommt.

3.3.3 Host-Maschinen Unabhängigkeit der Services

Um die Verteilbarkeit der Anwendung zu gewährleisten, dürfen keine spezifischen Einstellungen bezüglich der Host-Maschinen getroffen werden. Das Festlegen von IP-Adressen und Ports des Hosts im Service verhindert die automatisierte Distribution auf die Knoten eines Clusters. Jeder Service sollte, mithilfe einer Containerorchestrierungsumgebung, auf jeden Knoten, der die vordefinierten Anforderungen erfüllt, verteilt werden können [MSV16]. Die Bedingungen entsprechen weniger Software Konfigurationen, sondern eher den vorhandenen Rechnerspezifikationen, wie zum Beispiel genügend freie Arbeits- oder Festplattenkapazität. Die Funktionsfähigkeit der Dienste sollte nicht Abhängig von dem System sein, auf dem sie eingesetzt werden [MSV16].

3.4 Komposition von Microservices

Obwohl die Autarkie von Microservices ihr prägendstes Merkmal darstellt, ist jeder Service nur Teil eines großen Ganzen. Erst wenn diese Services zusammen zum Einsatz kommen kann man von einer Anwendung sprechen. Im Folgenden werden Richtlinien und Umsetzungsvorschläge bei der Zusammenführung von Services angebracht, um die vielen Fallstricke der Architektur zu vermeiden.

3.4.1 Separate Codebasis für jeden Service

Bevor das Implementieren einer Applikation beginnen kann, sollte über die Strukturierung der Codebasis nachgedacht werden. Es gibt generell zwei Herangehensweisen ein Softwareprojekt in Git zu verwalten. Entweder werden alle Microservices in einem großen Repository gesammelt und durch den Einsatz von Unterverzeichnissen voneinander getrennt, oder es wird pro Dienst ein eigenes Repository angelegt. Der Nutzung entsprechend bezeichnet man diese Arten der Führung eines Versionverwaltungssystemes *Mono- und Multi-Repository*. Die Entwicklung von Microservices unter der Verwendung von separaten Code-Repositories bietet einige Vorteile gegenüber der Nutzung eines Mono-Repositorys, sobald eine Form der Automatisierung für das Verteilen und Testen der Dienste vorliegt. Im Folgenden sollen Gründe beleuchtet werden, warum ein Multi-Repository bevorzugt wird.

Spiegelt den Kerngedanken von Microservices wieder Das Aufteilen der Services in einzelne Repositories erzwingt gewissermaßen die in der Microservice-Architektur vorgesehene Modularität. So wird versichert, dass jedes Service-Repository einzeln klonbar und dessen Inhalt eigenständig kompilier- und lauffähig ist. Können diese Kriterien nicht erfüllt werden, so handelt es sich nicht um einen Service einer Microservice-Anwendung.

Leichtgewichtige Repositories Je größer das genutzte Repository ist, desto weniger performant wird das Git-Versionsverwaltungssystem. Hierbei unterscheidet [Sch16] nach verschiedenen Arten der Größe. Eine große Anzahl an Dateien hat dabei einen negativen Einfluss auf die Leistungsfähigkeit jeglicher Git-Operation. Sind stattdessen viele Commits und Branches vorhanden, so schlägt sich das überwiegend auf die Zeit nieder, die benötigt wird um ein solches Repository zu klonen. Besonders beim automatisierten bauen, verteilen und testen von Microservices, bei denen genannte Operationen häufig vorkommen ist somit eine große Zeitersparnis zu erzielen. Das Separieren der Codebasis auf Grundlagen der Service-Zugehörigkeit hilft dabei die genannten Nachteile zu vermeiden.

Minimierung von Konflikten bei der Codeintegration Die Verwaltung des Quellcodes wird im allgemeinen weniger komplex. Dies trifft vor allem zu, wenn die einzelnen Repositorys verschiedenen Entwicklerteams zugewiesen sind. Aufgrund der Isolation des Codes muss, wenn überhaupt, nur eine minimale Koordination zwischen den Repository-Zuständigen stattfinden. Somit wird auch eine Entkopplung der Abhängigkeit innerhalb eines Projektes und deren Teilteams eingeführt, was zu einer effizienteren und schnelleren Implementierung von Neuerungen führen kann.

3.4.2 Implementierung / Einsatz einer Service Discovery

Im Unterunterabschnitt 3.3.3 wird erläutert, warum Microservices ihren Host nicht selber spezifizieren sollten. Trotzdem muss es eine Möglichkeit geben, mit der sich Services gegenseitig finden und kommunizieren können. Dieses Konzept der automatischen Erkennung und Bekanntmachung von Diensten innerhalb eines Netzwerkes wird als Service Discovery bezeichnet. Weitere zu lösende Herausforderungen für eine solche, stellen gemäß [JMD+16] auch das Ermitteln des Gesundheitszustandes der Services und die Verkündung bei der Registrierung eines neuen Dienstes dar. In [JMD+16] werden verschiedene Formen der Service Discovery vorgestellt und erläutert. Unter diesen befinden sich beispielsweise *clientseitige Service Discoverys*.

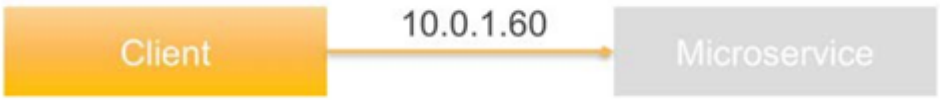

Abbildung 12: Clientseitige Service Discovery (nach [JMD+16])

Sie stellen die einfachsten Möglichkeit dar, eine Verbindung zwischen verschiedenen Diensten herzustellen. In Abbildung 12 wird verdeutlicht, dass diese Methode der Service Discovery eine Hartcodierung der IP-Adressen von den Services, innerhalb der Konfiguration des Clients vorsieht. Zwar kann die Adresse in einem Domain Name System hinterlegt und bei Bedarf abgerufen werden, eignet sich aber dennoch nicht für den Einsatz in einer Microservice-Architektur. Dieser Schluss kann Aufgrund der, durch die in Unterunterabschnitt 3.3.3 beschriebenen, erstrebten Eigenschaft gezogen werden. Sobald sich die IP-Adresse oder der Port sich durch eine Skalierung, Neustart oder Umkonfiguration ändert, muss eine Anpassung der Hostnamen in jedem Clienten vorgenommen werden, der mit den betroffenen Diensten in Kontakt steht. Dies sorgt selbst bei kleinen

Microservice-Anwendungen, die sowohl Client als auch Kommunikationsziel darstellen, für einen enormen Konfigurationsaufwand. Neben der clientseitigen Service Discovery nennt [JMD+16] unter anderem noch *Applikations-Load-Balancing basierte Service Discovery*, *Service Discovery unter Verwendung einer Schlüssel-Werte-Datenbank* und *Service Discovery unter Verwendung von Konfigurations-Management-Werkzeugen*.
Wird für Verteilung der Microservices die Container-Orchestrierungs-Umgebung Kubernetes genutzt, so stehen für Cluster-internen Container zwei weitere Arten zur Verfügung, mit denen Dienste einer Microservice-Anwendung effizienter gefunden werden können. Die Autoren von [TPJ+18c] benennen diese als:

Service Discovery unter Verwendung Umgebungsvariablen Gemäß [TPJ+18c] stellt Kubernetes für jeden Service zwei Umgebungsvariablen zur Verfügung, welche die IP-Adresse und den Port des Dienstes preisgeben. Über diese kann nach erfolgreicher Variablenauflösung auf den gewünschten Dienst zugegriffen werden. Die so entstandenen Host- und Port-Variablen bleiben weiter bestehen solang der zugehörige Service noch läuft.

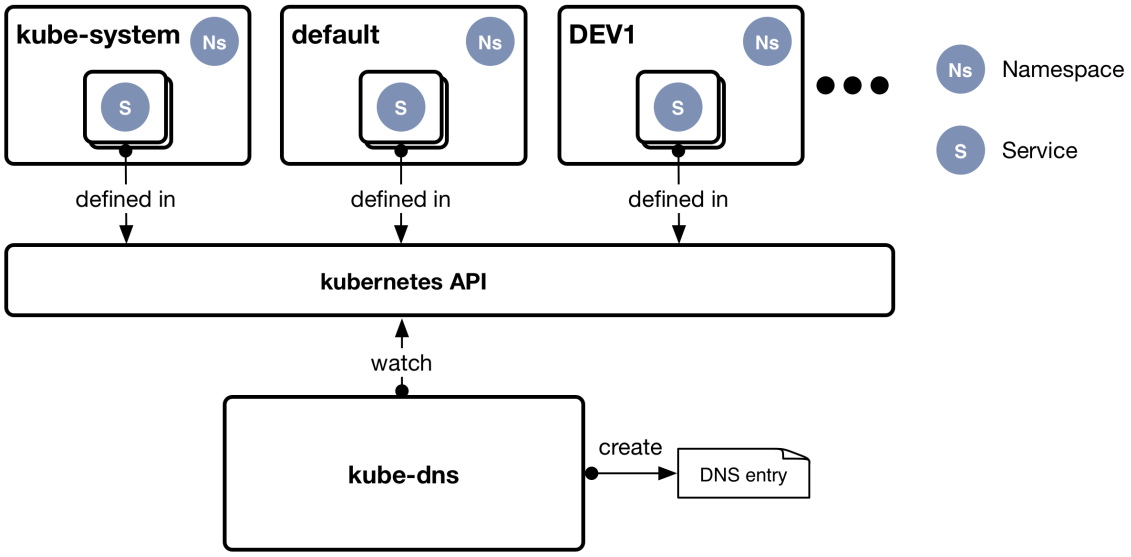

Abbildung 13: Überwachung der Kubernetes API auf Erstellung neuer Services und Einteilung in verschiedene Namensräume (nach [Dru15])

Service Discovery unter Verwendung von Kubebernetes DNS Die Zweite Methode der Service-Auffindung findet mithilfe des sogenannten *Kubernetes DNS* Services statt. Laut [TP16], [TPJ+18c] und unter Betrachtung des Quellcodes unter [Kub16b] ist *Kubernetes DNS* ein DNS-Server, der innerhalb eines Kubernetes Clusters platziert werden kann. Dieser Server überwacht die Kubernetes API auf die Erstellung neuer Kube-Services und registriert für jeden dieser, DNS-Einträge, wie in Abbildung 13 zu sehen ist. [TPJ+18c]. Über die interne Namensauflösung kann jeder Kube-Service und somit jede Website oder Dienst, der sich in einem Pod des Clusters befindet, erreichbar gemacht werden. Ein weiterer Vorteil dieser Vorgehensweise ist, dass man die Namensauflösung auf bestimmte Namensräume beschränken kann [Dru15]. Somit müssen keine um-

gebungsspezifischen Angaben innerhalb der Anwendung getätigt werden. Dies verstärkt die Host-Unabhängigkeit (vgl. Unterunterabschnitt 3.3.3). Außerdem wird damit auch die ungewollte Kommunikation zwischen Anwendungsteilen in verschiedenen Umgebungen verhindert. Dies bedeutet, dass beispielsweise ein Dienst der sich im Test befindet niemals mit einem Service in der Produktion interagieren kann, wenn dies nicht vorgesehen ist.

3.4.3 Kommunikation der Services

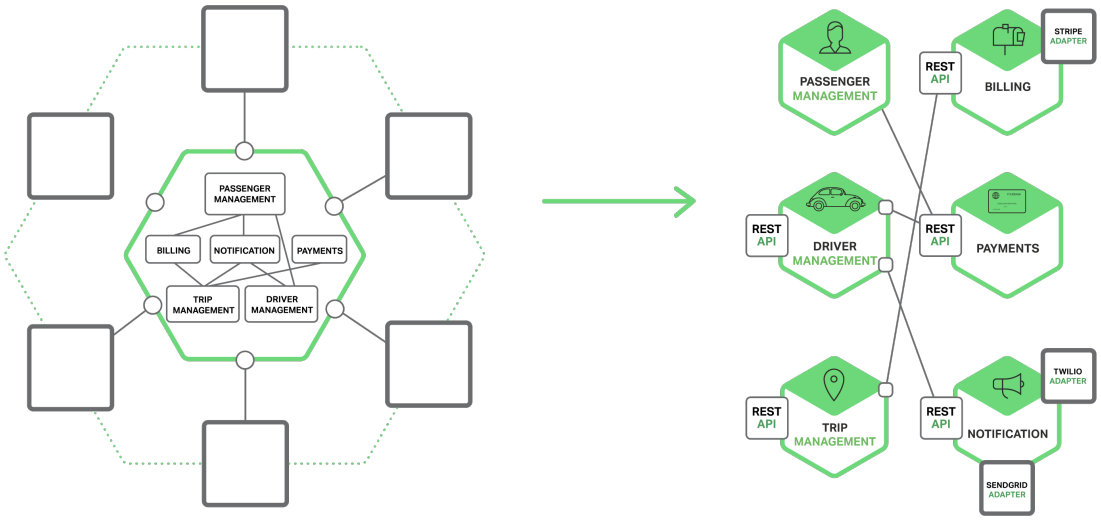

Abbildung 14: Vergleich der Kommunikationsmechanismen von monolitischen Anwendungen (links) und Microservice-Anwendungen (rechts) (nach [Ric15])

Microservice-Anwendungen sind verteilte Systeme, die auf verschiedenen Maschinen ausgeführt werden. Aus diesem Grund kann keine direkte Interaktion zwischen den Komponenten, zum Beispiel durch Methodenaufrufe auf Ebene der Programmiersprache, stattfinden. Gemäß [Wig11] und [Ric15] ist jede Instanz eines Services ein Prozess. Daraus schlussfolgernd muss auch die Kommunikation über eine sogenannte Interprozesskommunikation (IPC) erfolgen. Der Unterschied zwischen Komponenten-Interaktion in einer Anwendung mit monolithischer Architektur und einer Microservice-Architektur wird in Abbildung 14 nochmals verdeutlicht. Um den richtigen IPC-Mechanismus auszuwählen, muss für den Anwendungsfall spezifisch geplant werden, wie die Services miteinander interagieren sollen. Richardson [Ric15] kategorisiert dafür verschiedene Interaktions-Stile anhand von zwei Hauptmerkmalen. Der erste Betrachtungspunkt sagt aus, von wie vielen Service-Instanzen eine Client-Anfrage angenommen und verarbeitet wird. Dabei unterscheidet man nur zwischen Interaktionen mit einer One-To-One-Beziehung, bei dem jede Anfrage von genau einer Service-Instanz empfangen wird, oder One-To-Many-Interaktionen, wobei Anfragen von mehreren Service Instanzen weiterverarbeitet werden können. Das zweite Merkmal beschreibt die Synchronität der Wechselbeziehung. Handelt es sich um eine synchrone Interaktion, so wird vom Client eine zeitnahe Antwort vom Service erwartet. Während des Wartens blockiert der Client, und wird erst nach Erhalt einer Antwort oder einem Timeout wieder freigegeben. Findet die Interaktion asynchron statt,

so blockiert der Client nicht und die Antwort des Services wird, wenn überhaupt, nicht sofort angefordert. Unter Kombination dieser Eigenschaften erhält man die in Tabelle 1

	One-to-One	One-to-Many
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Tabelle 1: Interprozesskommunikations-Stile (nach [Ric15])

zusammengetragenen Interaktions-Stile. Diese werden in [Ric15] wie folgt beschrieben:

- **Request/response** - Der Client stellt eine Anfrage an einen Service und erwartet eine zeitnahe Antwort. Im Zeitraum des Wartens kann es zum Blockieren des Clients kommen.
- **Notification** - Der Client stellt eine Anfrage an einen Service, erwartet jedoch keine Antwort. Der verarbeitende Service nimmt die Anfrage entgegen, sendet aber keine Antwort zurück.
- **Publish/subscribe** - Der Client verschickt eine Benachrichtigung, welche von keiner oder mehreren Services konsumiert werden kann.
- **Request/async response** - Der Client stellt eine Anfrage an einen Service mit der Annahme, dass dieser möglicherweise keine Antwort gibt. Deshalb kommt es nicht zum Blockieren des Clients. Die Antwort des Services erfolgt dabei asynchron.
- **Publish/async responses** - Der Client stellt eine Anfrage und wartet auf eine asynchrone Antwort von keinem oder mehreren Services.

Services nutzen meist eine Kombination dieser IPC-Mechanismen, um ihre Aufgabe zu erfüllen. Die Rolle des Clients muss sich hierbei nicht unbedingt auf einen Anfragesteller außerhalb des Microservice-Systems beziehen, sondern kann auch von den einzelnen Diensten selber übernommen werden.
Je nachdem ob die Kommunikation synchron oder asynchron verläuft, stehen verschiedene IPC-Technologien zur Verfügung. Soll die Interaktion zwischen den Services synchron stattfinden, so wird in dem Großteil der Anwendungsfälle eine RESTful API genutzt. Bei einer solchen steht die Manipulation von Ressourcen, welche meistens Geschäftsobjekt wie zum Beispiel einen Kunden oder Nutzer darstellen, im Vordergrund. Der Client stellt eine, in fast allen Fällen, HTTP-Anfrage mit einem bestimmten HTTP-Verb, welches Auskunft darüber gibt, wie die Ressource behandelt werden soll. So kann man beispielsweise mit dem GET-Verb die Repräsentation einer Ressource, zum Beispiel in Form eines JSON-Objektes, anfordern. Auch bei einem Fehlschlag, wird eine Antwort über genau solch einen zurückgesendet, da der Client in jedem Fall auf eine Antwort wartet.
Wird ein asynchroner Kommunikationsansatz verfolgt, so geschieht dies in der Regel über Messaging. Bei der Nachricht-basierten Kommunikation interagieren Prozesse, indem sie asynchron Nachrichten miteinander austauschen. Wird vom Client eine Antwortnachricht erwartet, so wird diese in einer separaten Nachricht zurückgesendet. Eine solche Nachricht besteht dabei aus vielen Nachrichtenköpfen, welche Metadaten, wie zum Beispiel

den Absender enthalten, und dem eigentlichen Nachrichteninhalt im Nachrichtenkörper. Der Austausch findet auf sogenannten Kanälen (engl. channels) statt, dabei unterscheidet man zwischen point-to-point und publish-subscribe Kanälen. Bei einem Point-to-point-Kanal gibt es genau einen Konsumenten, welcher die Nachricht empfängt. Dahingegen wird die Nachricht bei einem Publish-subscribe-Kanal von jedem Konsumenten, der sich an den Kanal gebunden hat, empfangen.

4 Testen der Anwendung

Mircorservice-Anwendungen benötigen aufgrund ihrer verteilten Natur, andere beziehungsweise modifizierte Teststrategien, als man es von dem Testen monolithischer Software kennt. So versteht man je nach Kontext unter einem Integrationstest gänzlich verschiedene Dinge. Aus diesem Grund werden die Teststrategien im Bezug auf Microservices nach [Cle14] wie folgt definiert:

- **Unit-Test** Unit-Tests testen die kleinsten testbaren Codebestandteile einer Anwendung darauf, ob sie sich erwartungsgemäß verhalten.
- **Komponententest / Component Testing** Komponententests testen Dienste in ihrer Gesamtheit, betrachten dabei jedoch nicht die Abhängigkeiten zu anderen Services.
- **Integrationstest** Integrationstests testen einen Service im Zusammenspiel mit anderen Services.

Zusätzlich werden in [Cle14] noch Contract-Testing und End-to-End-Tests definiert. Im Rahmen dieser Arbeit wird nicht weiter auf diese Teststrategien eingegangen.

4.1 Continuous Integration

Zur Gewährleistung des problemlosen Zusammenspiels von neu implementierten Funktionalitäten und Verbesserungen mit der bestehenden Codebasis, ist es notwendig häufig und kontinuierlich zu testen. Um Dies ohne eine Beeinträchtigung des Arbeitsflusses zu vollziehen, muss sowohl die Integration des neuen Codeanteil als auch das Testen des daraus resultierenden Moduls mittels Komponententests automatisiert erfolgen. Ein Entwickler sollte nur dann eingreifen müssen falls Konflikte beim Integrieren auftreten, Komponententests nicht mit Erfolg ausgeführt wurden oder ein Fehler im Build-Prozess auftritt. Als Continuous Integration wird genau solch eine Vorgehensweise bezeichnet, bei dem der Code mehrmals täglich in ein gemeinsames, vom Projektteam geteiltes, Repository eingecheckt und anschließend von einem automatisierten Build verifiziert wird [Fow06]. Diese Art der Praxis ermöglicht es Probleme schnell zu erfassen. Die Gesamtheit der notwendigen Zwischenschritte um am Ende einen sauberen integrierten Code zu erhalten wird demnach Continuous Integration Pipeline (kurz CI Pipeline) genannt.

4.2 Continuous Delivery

Die Continuous Delivery ist eine Erweiterung der Continuous Integration, bei der sichergestellt wird, dass aus allen akzeptierten Codeänderungen eine verteilbare Softwareeinheit entsteht. Zu jeder Zeit sollte sich die Software in einem Zustand befinden, sodass sie in Produktion gehen kann. Ebenso soll es möglich sein zwischen den so erzeugten Versionen zu differenzieren und je nach Bedarf selektiv bereitzustellen [Bit15]. Analog zur CI Pipeline bezeichnet man den Weg und die erforderlichen Abläufe, um von einem Push auf das Versionsverwaltungssystem bis zum Einsatz der Anwendung auf einem Server zu kommen, als Continuous Integration/Continuous Delivery Pipeline (kurz CI/CD Pipeline).

4.3 Aufbau einer CI/CD Pipeline unter Nutzung von Containern

Um Microservices automatisiert, sowohl auf Komponenten-Ebene als auch integrativ, zu testen, ist die Entwicklung einer CI/CD Pipeline von Vorteil. Dabei wird für jedes Repository ein separater Pull-Request-Build und CI Pipeline erzeugt. Nachdem der neu hinzukommende Code auf die Erfüllung der Voraussetzungen zur Integration in die bestehende Codebasis geprüft wurde, werden die nächsten Schritte der Abfolge eingeleitet. Die Voraussetzungen sind gewöhnlich ein fehlerloses Kompilieren der Anwendung, erfolgreiche Ausführung dessen definierter Unit-/Komponenten-Tests und außerdem ein Code-Review von einem anderen Entwickler. Darauf folgend wird mit Hilfe des resultierenden Artefakts ein Container Abbild erstellt. Das entstandene Image hinterlegt man dann in einer Container Registry. Diesen Eintrag kann nun genutzt werden, um die Verteilung der Anwendung mittels einer Container-Orchestrierungs-Umgebung, zum Beispiel Kubernetes, durchzuführen.

4.4 Detailierter Verlauf einer Continuous Delivery Pipeline

4.4.1 Öffnen eines Pull-Requests zur Integration neuer Features

Der Entwickler kann sowohl nach Entwicklungsabschluss als auch während des Entwicklungsprozesses einer neuen Funktionalität eine Integrationsanfrage stellen. Letzteres hat den Vorteil, dass schon frühzeitig eventuelle Konflikte erkannt und vermieden werden können. Die gängige Vorgehensweise sieht es vor, dass der Entwickler vom Develop-Branch des Versionsverwaltungssystem abzweigt und auf diesem so genannten Feature-Branch seine Codeänderungen implementiert. Auf diesen wird ein Pull-Request zur Zusammenführung des Feature-Branches mit dem bestehenden Develop-Branches geöffnet.

4.4.2 Auflösen von Merge-Konflikten und Code-Review

Kann die Versionsverwaltung entstehende Konflikte nicht automatisch auflösen, dann werden die betroffenen Entwickler darüber benachrichtigt. Es muss nun eine manuelle Auflösung der Merge-Konflikte erfolgen. Im einfachsten Fall muss nur eine Wahl zwischen dem bestehenden und neuen Codestand, an der Konfliktstelle, getroffen werden. Für kritischere Bereiche müssen gegebenenfalls einzelne Zeilen im Quellcode editiert werden, um einen konfliktfreien Zustand zu erreichen. Schon in diesem Stadium wird nach jeder Änderung ein Build-Prozess angestoßen, welcher auf Fehler bei der Kompilierung, des Abhängigkeitsmanagement und der Unit-Tests aufmerksam machen soll. Sind diese Schritte geschehen, dann ist die erste Bedingung für die Weiterverarbeitung in der Pipeline erfüllt. Den anderen Teil stellen hierbei Code-Reviews dar. Bei diesen wird von einem oder mehreren Prüfern manuell auf Fehler untersucht.

4.4.3 Kompilieren des Codes und Erstellen der Artefakte

Wurde die Bestätigung des Systems und der Prüfer gegeben, dann kann die Integration der neuen Funktionalitäten in den stabilen Entwicklungszweig eingeleitet werden. Hier finden letztendlich nochmal die Kompilierung des betreffenden Quellcodes und das Bauen eines Artefaktes statt. Die daraus entstehenden binären Dateien sollten zusätzlich in einem Binary Repository hinterlegt werden.

4.4.4 Erstellen eines Container-Images

Damit die Verteilung der Anwendung über eine Container-Orchestrierung erfolgen kann, aber auch um das schnelle Hochfahren zum lokalen Tests auf Entwicklerrechnern zu ermöglichen, muss aus den Produkten des vorherigen Schrittes ein Container-Image erstellt werden. Dazu verwendet man meist in einer Datei, z.B. Dockerfile bei Docker-Containern, hinterlegte Instruktionsabfolgen, welche die benötigten Befehle beschreiben, um aus den erzeugten Artefakten ein Container-Abbild zusammensetzen zu können. Solch ein Image basiert dabei immer auf einem Grundimage, welches normalerweise eine Minimalinstallation eines Betriebssystemes und von der Anwendung benötigte Abhängigkeiten, zum Beispiel die Java Laufzeitumgebung bei Java-Applikationen, beinhaltet. Dieser Basis wird neben den Anwendungsartefakten noch weitere Konfigurationen hinzugefügt, beispielsweise benötigte Umgebungsvariablen oder Anweisungen für Logging-Ausgaben zur einfacheren Fehleranalyse. Aus diesen Informationen kann das Build-System ein leichtgewichtiges, auf den Anwendungsfall zugeschnittenes Container-Image erzeugen. Um ein Container-Abbild verteilen zu können, ist dessen Einreichung in eine Container-Registry notwendig. Diese stehen öffentlich, wie zum Beispiel bei *Docker Hub*, zur Verfügung. Für den privaten Gebrauch ist es möglich eine eigene Registrierungsdatenbank aufzusetzen.

4.4.5 Verteilung des Container-Images in ein Kubernetes-Cluster

Die CI/CD Pipeline kommuniziert mit dem Cluster über den sogenannten Kubernetes API Server. Diese Schnittstelle nimmt Befehle entgegen, die den Zustand des Clusters definieren sollen. Die Pipeline teilt dem API Server mit, welche Images aus der Container-Registry bezogen werden sollen um anschließend daraus Container bzw. Pods zu erstellen. Nach Übergabe der Vorschriften an den API-Server stellen Controller des Clusters den gewünschten Zustand her. Dazu startet dieser, die deklarierten Microservices anhand der nun vorhandenen Pods. Je nach Vorgabe wird die Anwendung noch für verschiedene Netzwerkbereiche, wie zum Beipiel nur innerhalb des Clusters oder über eine öffentliche IP, zugänglich gemacht.

4.5 Herausforderungen bei der Entwicklung von CI/CD Piplines für Microservices mit verteilten Repositorys

Neben den Vorteilen, wie zum Beispiel automatisiertes Testen, erhöhte Produktivität, leichtere Vorhersehbarkeit durch das schnelle Erfassen von Problemen, gibt es auch einige Herausforderungen bei dem Entwickeln und Planen einer CI/CD Pipeline.
Beispielsweise wird der Build-Prozess für den einzelnen Microservice einfacher, da der Quellcode und alle notwendigen Abhängigkeiten und Ressourcen separat vorliegen oder gezielt bezogen werden können. Die Komplexität erhöht sich aber stattdessen auf Seiten des gesamten Buildsystems, weil nicht jede Anwendung den selben Erstellungsprozess besitzt und somit von der Pipeline differenziert werden muss. Das System sollte daher mit einem großes Maß an Flexibilität entworfen werden, damit vom einfachsten Front-End Deployment, der nur das Verteilen von statischen HTML-Dateien benötigt, bis hin zum komplexesten Anwendungsfall abgedeckt wird. Aufgrund dessen kann dies zu komplizierten und somit schwer wartbaren Build-Scripten führen.

Das Aufteilen der Microservices in einzelne Quellcodeverzeichnisse gegenüber der Verwendung eines monolithischen Repository hat den Vorteil, dass nicht bei jeder Codeänderung die Gesamtheit der Anwendungen kompiliert, in Pakete gebündelt, getestet und verteilt werden müssen. Durch die Trennung, wird der Prozess nur für die Anwendungsteile durchgeführt, die auch tatsächlich betroffen sind. Das Gliedern der Services hat jedoch auch zufolge, dass Änderungen, die mehrere Bereiche der Software betreffen, zu Blockaden führen können und somit den gesamten Arbeitsfluss stören. So kann es zum Beispiel Konflikte geben, wenn ein Service Objekte in die Datenbank persistieren will, die Datenmodelle besagter Objekte im Zuge einer Codeintegration verändert wurden. Idealerweise wären für solch eine Situation separate Datenbanken für jeden Service vorhanden, was jedoch nicht in jedem Fall machbar ist. Im Bezug auf die Continuous Delivery Pipeline sind also im schlimmsten Fall mehrere zusammengehörige Datenbankschema-Aktualisierungen durchzuführen. Um das Problem zu vermeiden sollte ein Service für einen Komponententest seine Abhängigkeiten zu zustandsbehafteten Dingen (Datenbank etc.) im Test mitbringen.
Im Allgemeinen muss sich also die Frage gestellt werden, wie man am besten Abhängigkeiten zwischen Komponenten verwalten kann, welche sich schnell, häufig und unabhängig voneinander weiter entwickeln. Deshalb ist es eine Notwendigkeit, dass eine strikte Versionierung durchgesetzt wird. Jeder Service muss klar definieren welche anderen Dienste und vor allem in welcher Version, dieser ohne Einschränkungen zusammen funktioniert. Ohne den Einsatz eines einheitlichen Versionierungssystems ist das Testen oder gar das Verteilen einer funktionierenden Teil- oder Gesamtanwendung so gut wie unmöglich. Ein Beispiel für eine benötigte Versionierung stellt die Automatisierung von PR-Builds dar. Wird der Name des entsprechenden Feature-Branches vom Pull-Request als Versionsnummer für den PR-Build gewählt, so stellt sich die Frage: Wie wird die zugehörige, schon verteilte Anwendung, bei einem Push neuen Codes auf den Feature-Branches aktualisiert? Ohne eine Versionsänderung des Images bezieht Kubernetes auch nichts Neues aus der Registry, da so keine Änderung wahrgenommen wird. Eine Lösung dafür wäre es, die Buildnummer mit an die Version anzuhängen, welche sich somit bei jeder Aktualisierung und dem nach sich ziehenden Bauen des PR-Builds ändert.
Die Herausforderung beim Testen von Microservices im Kontext einer CI/CD Pipeline ist es eine Balance zwischen dem Hochfahren aller Anwendungsteile und der Isolierung eines einzelnen Services zu finden. Obwohl ein Microservice in der Lage ist unabhängig von anderen Diensten zu operieren, hat ein Test ohne Einbeziehung der zugehörigen Services keinen Wert, wenn die Funktionalität im zukünftigen Gesamtsystem betrachtet werden soll. Dahingegen ist das Starten und Verteilen von Komponenten die in keinen Bezug zu dem zu testenden Microservice haben ineffizient und macht den Sinn der Modularisierung zunichte.

4.6 Integratives Testen der Microservices

Komponententests alleine geben keine Versicherung für das reibungslose Zusammenspiel des Gesamtsystemes ab. Zwar wurde das Verhalten der einzelnen Microservices in Unabhängigkeit geprüft, es kann jedoch noch keine Aussage darüber getroffen werden was passiert, sobald diese Dienste zusammen arbeiten oder mit entfernten Abhängigkeiten interagieren müssen. Für genau solche Fälle sind Integrationstests zuständig. Beispielsweise kann man damit die korrekte Persistenz in eine Datenbank, nach Ausführung einer Funktion in der Anwendung überprüfen. Um die Kompatibilität aller Neuerungen effektiv und in ihrer Gesamtheit testen zu können, empfiehlt es sich in zwei Schritten vorzugehen.

4.6.1 Schritt 1

Als erstes wird der aktuell betrachtete Service aus dem entsprechenden Feature-Branch des Pull-Requestes in das Cluster verteilt. Danach folgen die anderen Services aus den stabilen Integrationszweigen. Darauf folgend kann das integrative Testen durchgeführt werden. Die Verteilung dieser Menge an Container muss mit einer Kennzeichnung der Zusammengehörigkeit erfolgen, was zum Beispiel durch die Verwendung gleicher Tags umgesetzt werden kann. Dies dient dazu während den Tests Ausgabeprotokolle einfacher zuordnen und analysieren zu können, und nach dem Testen das restlose Entfernen der benötigten Microservices zu ermöglichen.

4.6.2 Schritt 2

Der zweite Teil dient dazu Features zu testen, die mehrere Services beeinflussen. Auch hierfür wird der aktuell zu testende Service aus dem zugehörigen Feature-Branch in das Cluster verteilt. Der Unterschied zum ersten Schritt besteht hierbei jedoch, dass alle anderen Services aus einem zu definierenden Zweig hervorgehen. Somit kann man zum Beispiel die Interaktion mit Services aus anderen Feature-Branches und der Komplettanwendung testen. Eine weiter Möglichkeit ist das Nutzen eines Paketmanagement-Werkzeugs. Bei Kubernetes heißt dieses Tool Helm. Mit Hilfe dessen können komplexe Kubernetes-Applikationen installiert und verwaltet werden. So definiert man in einer so genannten Chart-Datei die gewünschten Anwendungen und deren Standardkonfiguration. Über die Chart können alle Services nun als Einheit verteilt, geändert, gestoppt und auch wieder entfernt werden. Mit Helm kann man dann zum Beispiel Situationen lösen, bei denen man in zwei verschiedene Repositories, Pull-Requests besitzt, welche aber zusammen getestet werden sollen. So definiert man die Anwendungen aus den Feature-Branches der Pull-Requests, sowie alle anderen Teile der Applikation aus den stabilen zweigen, als eine Einheit, welche zusammen hochgefahren und getestet wird.

Für Integrativtests gibt es viele verschiedene Werkzeuge, welche auch automatisiert werden können. Bei Microservices spielen in der Regel automatisierte HTTP-Requests eine Rolle, da diese in den meisten Fällen als RESTful-Services konzipiert sind. Die detaillierte Betrachtung der Verwendung solcher Werkzeuge, wie JMeter oder Postman, gehören jedoch nicht in den Rahmen dieser Arbeit.

5 Ergebnisse und Bewertung

Das Testen von Microservice-Anwendungen muss unter einer komplett anderen Strategie und Vorgehensweise durchgeführt werden, da sich durch ihre Verteiltheit, nicht alle relevanten Bestandteile an einem Ort befinden. Zur Minderung der Komplexität und um die Entwicklung eines verteilten Systemes zu erleichtern, gibt es schon heute verschiedene Plattformen wie Docker und Kubernetes, mit deren Hilfe der Weg für Cloud Computing geebnet wird. Um diese Hilfsmittel effektiv nutzen zu können ist es notwendig, dass die Entwicklung der Services schon mit dem Gedanken eines automatisierten Deployments erfolgt. Die Zustandslosigkeit und Verwendung von dezentralisierten Abhängigkeiten waren von Anfang an Grundpfeiler eines robusten Microservices. Doch mit dem Einzug der Containerisierung muss auch eine Entkopplung von spezifischen Host-Maschinen stattfinden. Der dadurch entstehende Abstraktionsgrad ist ohne ein sauberes und organisiertes Softwaredesign schwierig zu bewältigen. Deshalb ist es für das Einhalten der Ordnung von großer Bedeutung, für jeden Service ein eigenes Repository zur Verfügung zu stellen. Die Kommunikation zwischen Services stellt, aufgrund deren Isolation und Anonymität, ein weiteres Problem dar. Dieses wird durch die ausgeklügelte DNS basierte Kubernetes Service Discovery und dem Einsatz einer durchdachten Kombination von synchronen und asynchronen Interprozesskommunikation-Technologien gelöst. Die Rolle des verbindenden Elements zwischen Entwicklern und einem laufenden Gesamtsystem, wird von einer Continuous Inetegration/Continuous Delivery Pipeline übernommen. Wenn richtig und in enger Zusammenarbeit mit dem Versionsverwaltungssystem, Software-Container-Umgebung und Container-Orchestrierungssystem implementiert, übernimmt diese Pipeline von der Kommunikation zwischen Teammitgliedern, bis hin zum Verteilen und Testen der ausführbaren Software, alle benötigten Zwischenschritte. Das integrative Testen von Microservices an sich sollte dabei in zwei Schritten erfolgen. Diese unterscheiden sich größtenteils darin, wie viele andere Elemente vom getesteten Service beeinflusst werden. Sobald beide Testszenarien erfolgreich durchlaufen sind, kann man von einem harmonischen Zusammenspiel der Dienste ausgehen. Die Entwickler sollten keine Gedanken mehr an die Infrastruktur aufbringen müssen, sondern sich voll und ganz auf die Entwicklung neuer Funktionalitäten und Verbesserung der vorhandenen Bestandteile konzentrieren.

6 Schlussbemerkungen und Ausblick

Trotz der vielen Vorteile der Microservice-Architektur und Container-Orchestrierung, befinden sich beide Ansätze noch in einer frühen Phase ihrer Entwicklung. Deshalb entspringen viele Herausforderungen daraus, dass für den erfolgreichen Entwurf einer Microservice-Anwendung, eine Umstellung stattfinden muss, wie Software entwickelt wird. Da sich in den letzten Jahren ein Standard bezüglich der Containerisierung und Container Orchestrierung in Form von Docker und Kubernetes durchgesetzt hat, kann man in naher Zukunft neue Ideen und Vorschläge, nicht nur der Entwickler, sondern auch der rasant wachsenden Community dieser Plattformen, erwarten.

Literatur

[Atl] ATLASSIAN: *Making a Pull Request*. – https://www.atlassian.com/git/tutorials/making-a-pull-request [Online: Stand 13. Juni 2018]

[Bai17] BAIER, Jonathan: *Getting Started with Kubernetes - Second Edition: Orchestrate and manage large-scale Docker deployments*. 2017

[Bit15] BITTNER, Peter: *UNTERSCHIEDE ZWISCHEN CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY UND CONTINUOUS DEPLOYMENT*. 2015. – https://www.scrum.de/unterschiede-zwischen-continuous-integration-continuous-delivery-und-continuous-deployment/ [Online: Stand 28. Mai 2018]

[Bur14] BURGDORF, Christoph: *THE ANATOMY OF A GIT COMMIT*. 2014. – https://blog.thoughtram.io/git/2014/11/18/the-anatomy-of-a-git-commit.html [Online: Stand 13. Juni 2018]

[Cle14] CLEMSON, Toby: *Testing Strategies in a Microservice Architecture*. 2014. – https://martinfowler.com/articles/microservice-testing/ [Online: Stand 13. Juni 2018]

[Con18] CONWAY, Sarah: *Survey Shows Kubernetes Leading as Orchestration Platform*. 2018. – https://www.cncf.io/blog/2017/06/28/survey-shows-kubernetes-leading-orchestration-platform/ [Online: Stand 22. Juni 2018]

[CSa] CHACON, Scott ; STRAUB, Ben: *Pro Git - 1st Edition*

[CSb] CHACON, Scott ; STRAUB, Ben: *Pro Git - 2nd Edition*

[Doc] DOCKER INC.: *WHAT IS A CONTAINER*. – https://www.docker.com/what-container [Online: Stand 16. Juni 2018]

[Doc16] DOCKER INC.: *Docker for the Virtualization Admin*. 2016

[Dru15] DRURY, Des: *Kubernetes DNS Service Deep Dive - Part 1*. 2015. – https://desdrury.com/kubernetes_dns_part_1/ [Online: Stand 09. Juni 2018]

[FL14] FOWLER, Martin ; LEWIS, James: *Microservices a definition of this new architectural term*. 2014. – https://www.martinfowler.com/articles/microservices.html [Online: Stand 16. Juni 2018]

[Fow06] FOWLER, Martin: *Continuous Integration*. 2006. – https://martinfowler.com/articles/continuousIntegration.html [Online: Stand 28. Mai 2018]

[Gol17] GOLUB, Ben: *DockerCon 2017 - General Session Day 1*. 2017. – https://www.slideshare.net/Docker/dockercon-2017-general-session-day-1-ben-golub [Online: Stand 16. Juni 2018]

[JMD+16] JUNG, Matthias ; MÖLLERING, Sascha ; DALBHANJAN, Peter ; CHAPMAN, Peter ; KASSEN, Christoph: *Service Discovery*. 2016. – https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/service-discovery.html [Online: Stand 09. Juni 2018]

[Kub16a] KUBERNETES AUTHORS: *Kubernetes DNS service*. 2016. – https://kubernetes.io/ [Online: Stand 09. Juni 2018]

[Kub16b] KUBERNETES AUTHORS: *Kubernetes DNS service*. 2016. – https://github.com/kubernetes/dns [Online: Stand 09. Juni 2018]

[MG] MCKENDRICK, Russ ; GALLAGHER, Scott: *Mastering Docker - Second Edition*

[MSV16] MSV, Janakiram: *THE TEN COMMANDMENTS OF MICROSERVICES*. 2016. – https://thenewstack.io/ten-commandments-microservices/ [Online: Stand 09. Juni 2018]

[Per18a] PERRY, Steve: *Deployment*. 2018. – https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/ [Online: Stand 20. Juni 2018]

[Per18b] PERRY, Steve: *Using a Service to Expose Your App*. 2018. – https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ [Online: Stand 16. Juni 2018]

[Per18c] PERRY, Steve: *Using Minikube to Create a Cluster*. 2018. – https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/ [Online: Stand 16. Juni 2018]

[Per18d] PERRY, Steve: *Viewing Pods and Nodes*. 2018. – https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ [Online: Stand 16. Juni 2018]

[Ric15] RICHARDSON, Chris: *Building Microservices: Inter-Process Communication in a Microservices Architecture*. 2015. – https://www.nginx.com/blog/building-microservices-inter-process-communication/ [Online: Stand 09. Juni 2018]

[Sas16] SASIDHARAN, Divya: *To Squash or Not to Squash?* 2016. – https://seesparkbox.com/foundry/to_squash_or_not_to_squash, https://www.atlassian.com/pt/git/tutorial/git-branches#merge [Online: Stand 13. Juni 2018]

[Sch16] SCHNEIDER, Lars: *Large Git Repositories*. 2016. – https://larsxschneider.github.io/2016/09/21/large-git-repos [Online: Stand 12. Juni 2018]

[Sta17] STACKOVERFLOW: *Developer Survey Results 2017*. 2017. – https://insights.stackoverflow.com/survey/2017 [Online: Stand 13. Juni 2018]

[Str16] STROTMANN, Joerg: *INFOGRAPHIC: A BRIEF HISTORY OF CONTAINERIZATION*. 2016. – https://www.plesk.com/blog/business-industry/infographic-brief-history-linux-containerization/ [Online: Stand 16. Juni 2018]

[TP16] TOKUDA, Takuya ; PEDERSEN, Bjørn E.: *DNS for Services and Pods*. 2016. – https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ [Online: Stand 09. Juni 2018]

[TPJ+18a] TOKUDA, Takuya ; PEDERSEN, Bjørn E. ; JÄGER, Andreas u. a.: *Deployment*. 2018. – https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ [Online: Stand 20. Juni 2018]

[TPJ18b] TOKUDA, Takuya ; PEDERSEN, Bjørn E. ; JONES, Misty-Stanley: *Pod Overview*. 2018. – https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/ [Online: Stand 16. Juni 2018]

[TPJ+18c] TOKUDA, Takuya ; PEDERSEN, Bjørn E. ; JONES, Misty-Stanley ; LIU, Guang Y. u. a.: *Services*. 2018. – https://kubernetes.io/docs/concepts/services-networking/service/ [Online: Stand 09. Juni 2018]

[TPM18] TOKUDA, Takuya ; PEDERSEN, Bjørn E. ; MACEACHERN, Doug: *Persistent Volumes*. 2018. – https://kubernetes.io/docs/concepts/storage/persistent-volumes/ [Online: Stand 09. Juni 2018]

[Wig11] WIGGINS, Adam: *The Twelve-Factor App*. 2011. – https://12factor.net/processes [Online: Stand 09. Juni 2018]

Selbständigkeitserklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Dresden, den 26.06.2018

Kong Minh Cap