Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Jensen

Entwicklung und Vergleich einer Kubernetes Infrastruktur zu einer auf virtuellen Maschinen basierten Infrastruktur in der Cloud

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Jensen

Entwicklung und Vergleich einer Kubernetes Infrastruktur zu einer auf virtuellen Maschinen basierten Infrastruktur in der Cloud

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Lars Hamann

Eingereicht am: 11. Juli 2022

Daniel Jensen

Thema der Arbeit
Entwicklung und Vergleich einer Kubernetes Infrastruktur zu einer auf virtuellen Maschinen basierten Infrastruktur in der Cloud

Stichworte
AWS, Cloud, Virtuelle Maschine, Container, Kubernetes, EKS, Elastic Kubernetes Service

Kurzzusammenfassung
Container sind im Bereich der Infrastruktur-Technologie einer der größten Trends der letzten Jahre. Mit Containern mussten auch Lösungen gefunden werden diese im großen Stil zu verwalten. Gleichzeitig geht der Trend in der Organisation hin zu kleinen, Selbstorganisierten DevOps-Teams. Diese Arbeit untersucht die Frage, ob diese Trends zusammen funktionieren und im kleinstmöglichen Szenario noch sinnvoll sind. Dafür vergleicht sie den Container Ansatz mit einem auf virtuellen Maschinen basierten Ansatz.
Daniel Jensen

Title of the paper
Development of a Kubernetes based infrastructure and comparison to a virtual machine based one in the cloud

Keywords
AWS, cloud, virtual machine, container, Kubernetes, EKS, Elastic Kubernetes Service

Abstract
Containers have been one of the biggest trends in infrastructure technology in recent years. With containers, solutions also had to be found to manage them on a large scale. At the same time, the trend in the organization is towards small, self-organized DevOps teams. This paper examines the question of whether these trends work together and still make sense in the smallest possible scenario. To do this, it compares the container approach with an approach based on virtual machines.

Inhaltsverzeichnis

1 Einleitung

Wollte eine mittelgroße Firma in den frühen 2000ern einen Mailserver betreiben, musste ein Server gekauft werden. Anschließend musste der Server nur noch angeschlossen, installiert und konfiguriert werden.

Mit der immer schneller fortschreitenden Entwicklung der Mikrochips zu dieser Zeit, wurden die Prozessoren jedoch immer leistungsstärker. Um die gekaufte Hardware besser auszunutzen, wurde möglichst viel Software auf dem gleichen Server installiert.

Aus mehreren Gründen konnte dies zum Problem werden. Zum Beispiel könnten Laufzeitabhängigkeiten, wie Verschlüsselungsbibliotheken, von Software untereinander inkompatibel sein. Oder bei der Konfiguration des Servers werden Fehler gemacht und dadurch, ist der gesamte Server nicht mehr erreichbar. Wollte man nun auch noch Redundanz für seine Server bereitstellen, mussten die Administrator nun zwei oder mehrere Server parallel pflegen.

Um einige dieser Probleme anzugehen, haben Firmen wie Sun Microsystems, HP und IBM bereits in den späten 1990er Jahren Produkte mit Virtualisierung angeboten. Virtualisierung ermöglicht es, auf einem Host-System mit echter Hardware, mehrere Systeme mit virtueller Hardware zu betreiben. Diese virtuellen Systeme können flexibel auf den jeweilige Anwendungsfall zugeschnitten werden und so physikalische Hardware besser ausnutzen. Diese Systeme waren zu dieser Zeit jedoch noch entsprechend groß und teuer.

Mit dem Erscheinen von anderen Virtualisierungslösungen wie der Linux Kernel-based Virtual Machine in 2005 oder VMware's ESXi Platform in 2007 wurde diese Technologie auch für die breiteren Massen verfügbar.

Mit steigender popularität von "*Public Cloud*"-Anbietern wie *Amazon Web Services (AWS)* konnte dann jeder virtuelle Server im Rechenzentrum erstellen. Dies ermöglicht nicht nur, unterschiedliche Angebote wie Webserver und Mailserver auf unterschiedlichen virtuellen Rechnern laufen zu lassen, sondern auch die Portabilität eines Systems einfacher. Wo vorher eine Festplatte ausgebaut und wieder eingebaut werden musste, kann jetzt die gesamte virtuelle

Festplatte auf einmal auf einen anderen Server kopiert und dort wieder gestartet werden.

In 2012 ist mit dem Erscheinen von *Docker* eine weitere Technologie populär geworden: Container. Container versprechen, ähnlich wie Virtuelle Maschinen, die Separation von anderen Containern auf dem gleichen System. Sie können aber deutlich schneller gestartet werden und nutzen weniger Ressourcen auf dem Host-System als eine gleichwertige virtuelle Maschine.

Container haben damit ähnliche Vorteile wie virtuelle Maschinen, gleichen jedoch zusätzlich einige Nachteile dieser aus.

1.1 Motivation

Ich arbeite seit 2017 in meiner Firma in der Webentwicklung. Wir arbeiten in einem kleinen Team von etwa zehn Leuten an verschiedensten Projekten. Seitdem ich in diesem Team arbeite, hat sich die Infrastruktur, auf der unsere Projekte laufen, aber auch die Zuständigkeit für diese stark verändert.

Angefangen haben wir mit einigen Servern, die von einem Dienstleister gewartet und betrieben wurden. Mein Team und ich waren zu dieser Zeit nur für die Software verantwortlich. Dies hat sich 2018 geändert. Zu dieser Zeit hat sich die Firma entschlossen, einen großen Teil der Software in eine *Public Cloud*, namentlich *AWS*, zu migrieren.

Wir waren eine der ersten Abteilungen die ihre Projekte migriert haben. Zu dieser Zeit gab es bereits *Container* als Technologie. Das Team war aber zu dieser Zeit der Meinung, das diese noch nicht ausgereift ist. Aus diesem Grund haben wir entschieden, dass die Infrastrukturtechnologie für unsere Projekte, virtuelle Maschinen werden.

Wir haben seitdem mit 3-4 Kollegen (inklusive mir) die Infrastruktur stetig weiterentwickelt. Zeitweise ein Kollege in Vollzeit und bis zu drei je nach Bedarf.

1.2 Ziel der Arbeit

Seit unserer Entscheidung, nicht auf *Container* zu setzen, hat sich die Technologie weiterentwickelt. Zum Beispiel bieten alle großen "*Public Cloud*"-Anbieter Lösungen an, um containerbasierte Applikationen zu betreiben.

Ziel meiner Arbeit ist es, einen Prototyp für eine Container basierte Infrastruktur zu entwerfen und diesen mit der auf virtuellen Maschinen basierenden Lösung zu vergleichen. Ebenfalls soll eine Handlungsempfehlung abgegeben werden, ob eine Migration auf *Container* sinnvoll ist.

2 Grundlagen

2.1 Container

Die meisten nicht trivialen Programme heutzutage haben zur Laufzeit Abhängigkeiten. Dies kann zum Beispiel eine Runtime sein wie die JRE (Java Runtime Environment) oder shared librarys wie openssl. Damit eine Applikation erfolgreich ausgeführt werden kann, müssen alle Abhängigkeiten in den richtigen Versionen verfügbar sein und korrekt geladen werden. Wenn auf einem System nun zwei Applikationen von der gleichen shared library abhängen, jedoch in unterschiedlichen Versionen, gibt es ein Problem. Es müssen nun beide Versionen der Library installiert werden und beide Applikationen müssen wissen, wo sie ihre Version finden. Für einzelne Abhängigkeiten mag das noch simpel sein. In der Praxis hat sich dies jedoch als Problem herausgestellt. Zum einen sind oft diejenigen, die Systeme betreiben und Abhängigkeiten verwalten, nicht diejenigen, die die Anwendung entwickeln. Zum anderen muss der Entwickler nun die Testumgebung optimalerweise synchron mit der Produktionsumgebung halten. Passiert das nicht, könnten Fehler mit Abhängigkeiten erst in der Produktionsumgebung auffallen. (Matthias und Kane)

Container lösen das Problem, indem sie die Applikation zusammen mit allen Abhängigkeiten in ein Paket packen und dieses von der Container Runtime starten lassen.

Das macht einen Container zwar größer als eine nicht containerisierte Anwendung, aber dafür bringt sie die oben erwähnten Vorteile und ist trotzdem kleiner als eine gleichwertige Virtuelle Maschine.

Auch beim Thema Sicherheit haben Container einen Vorteil. Container sind in der Standard Konfiguration zunächst voneinander separiert. Sollen zwei Container miteinander Daten austauschen, muss das explizit definiert werden.

2.2 Container Orchestration

Mit Container Orchestration kann man das Organisationsproblem lösen, das sich aus der Scale-Out-Strategie (vgl. Starke (2015)) ergibt. Es ist relativ einfach, einige wenige Host-Systeme mit Containern zu pflegen. Aufwändiger wird es jedoch, sobald sehr viele Host-Systeme

daran beteiligt sind. Gerade im Cloud-Kontext lässt sich sehr einfach skalieren wodurch diese Probleme schneller auftreten.

Virtualisierungslösungen wie ESXi und Proxmox haben bereits seit einiger Zeit Möglichkeiten, mehrere Virtualisierungshosts zu einem Cluster zusammenschließen. Dies ermöglicht dann Features wie eine zentrale Übersicht über alle Cluster-Ressourcen oder automatische Migration zwischen Hosts. Einige dieser Lösungen übernehmen dabei nicht nur die Verwaltung der virtuellen Maschinen selbst, sondern auch die Verwaltung der Konnektivität zwischen diesen.//

In der Container-Welt gibt es ähnliche Produkte. *Kubernetes* ist die aktuell verbreiteste und umfangreichste Lösung Selbst beschreiben sie sich folgendermaßen:

> Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. (Kubernetes, b)

Als Open Source Projekt wird es von verschiedenen Organisationen mit einem etwas anderem Feature-Set angeboten. Zum Beispiel bietet RedHat eine Lösung namens OpenShift an, die eine Integration in das RedHat-Ökosystem bietet. Auch die meisten "*Public Cloud*"-Anbieter bieten eine Version an. Darunter auch *AWS* mit dem Service *Elastic Kubernetes Service (EKS)*.

Kubernetes hat sich für große Installationen und im Cloud-Umfeld als Unterbau weitgehend durchgesetzt. Jedoch gibt es noch einige andere Lösungen, die nicht auf *Kubernetes* aufsetzen. Beispiele dafür sind zum Beispiel die LXC-Container in Proxmox oder Nomad von Hashicorp. LXC-Container sind ein anderes Container-Format und sind nicht mit *Kubernetes* kompatibel. Nomad ist eine Lösung, die einfacher sein möchte als *Kubernetes*, sie lässt sich zum Beispiel als einzelnes Binary starten.

2.3 Infrastructure as Code

Der Begriff *Infrastructure as Code (IaC)* ist die Idee, IT-Ressourcen wie CPU-Ressourcen, Konfiguration oder Netzwerktopologie mit Code auszudrücken. Daraus ergeben sich mehrere Vorteile. Ein Vorteil ist die Reproduzierbarkeit. Mit dem Code lässt sich die Infrastruktur, die damit beschrieben wird, beliebig kopieren und gleichzeitig die Kopien aktuell halten. Damit vereinfacht *IaC* einfach eine Produktivumgebung und eine Testumgebung aufzusetzen und diese dabei so identisch wie möglich zu halten. Dadurch können Fehler, die mit der Infrastruktur zu tun haben, bereits in der Testumgebung erkannt und behoben werden.
Der zweite Vorteil ist die Dokumentation. Dadurch, dass die Definition der IT-Ressourcen

in Code abgelegt ist, haben die Entwickler nicht nur eine Übersicht über alle Hardware, die momentan im Einsatz ist. Auch kann durch ein *Version Control System (VCS)* und sinnvolle Commit-Nachrichten, eine dokumentierte Versionshistorie gepflegt werden.

Es gibt auf dem Markt verschiedene Produkte um diese Lösung zu nutzen. Zu den bekanntesten gehören Ansible, Puppet, Terraform und Cloudformation. Diese Lösungen unterscheiden sich in vielen Eigenschaften wie Dokumentation, Finanzierung, Preis und auch Beschreibungsstil. Beim Beschreibungsstil gibt es zwei Ansätze. Zum einen den imperativen Ansatz. Hierbei wird beschrieben, welche Änderungen an der Infrastruktur gemacht werden müssen, um den gewünschten Zielzustand zu erreichen. Zum anderen gibt es den deskriptiven Ansatz. Hierbei wird der gewünschte Zielzustand beschrieben und das entsprechende Tool sorgt dafür, das die Änderung am bestehenden System gemacht werden, um es in den neuen Zustand zu überführen.

Aufgrund seines deskriptiven Beschreibungsstils, der meiner Meinung nach als Dokumentation besser geeignet ist, und Vorerfahrung mit dem Produkt setze ich in dieser Arbeit auf das Tool *Terraform*.

3 Analyse

Im folgenden beschreibe ich die Anforderungen, die der Prototyp abdecken soll. Die Unterkapitel bauen aufeinander auf und soll meinen Entwicklungszyklus abbilden. Begonnen habe ich im Kapitel 3.1 mit den Anforderungen. Aufgrund meiner Vorerfahrung mit Agilen Arbeitsweisen, habe ich anschließend meine Arbeitspakete in Form von Storys aufgeteilt. Diese finden sich in Kapitel 3.2. Diese verfeinere ich in Kapitel 3.3 mit einigen Diagrammen. Abschließend gehe ich in 3.4 noch auf einige Designentscheidungen ein, die ich entweder in oder vor der Entwicklung getroffen habe.

3.1 Anforderungen

Wartungsarm

Damit die Infrastruktur von einem kleinen Team, wie wir es sind, betrieben werden kann, muss die Infrastruktur möglichst wartungsarm sein. Das bedeutet möglichst wenig manuellen Aufwand bei Updates und eine hohe Ausfallsicherheit.

Konsequente Nutzung von *IaC*

Die Reproduzierbarkeit und Dokumentation von *IaC* macht es erst möglich, komplexe Infrastruktur, die schnell mit neuen Features versorgt werden kann, zu betreiben.

Ausfallsicherheit bei Verlust einer *Availability Zone (AZ)*

Es ist Unternehmensrichtline, dass unsere Infrastruktur gegen den Ausfall einer einzelnen *AZ* abgesichert wird.

Anbindung von *On-Premise* Diensten

Einige Dienste werden noch in unseren eigenen Rechenzentren betreiben. Es muss möglich sein diese Dienste zu nutzen.

Nutzung von State of the Art Software

Es sollte Software genutzt werden, welche dem aktuellen Stand der Technik entspricht und nicht mittelfristig durch andere Software ersetzt werden muss.

Optimierung der Deployment-Zeit

Eine der zeitaufwändigsten Aufgaben ist es auf Prozesse in der Cloud zu warten. Ein solcher Prozess ist das Deployment von neuen Versionen. Egal ob man seine gerade geschriebene Software gerne auf dem Entwicklungsserver testen möchte oder ein Problem mit dem Deployment-Prozess debuggen muss, eine lange Deployment Zeit bindet Zeit von Entwicklern.

Das ist für das Unternehmen nicht nur teuer sondern auch frustrierend für den Entwickler.

3.2 User Storys

Story #1: Bootstrap	
Beschreibung: Erstellen der Arbeitsumgebung. Erstellung der Grundkomponenten.	Akzeptanzkriterien: • Es gibt ein Git-Repository für den Infrastruktur-Code. • In diesem Repository ist ein *Terraform*-Modul definiert, welches ein rudimentäres *EKS*-Cluster erstellt. • Der *Terraform*-State wird Remote gespeichert • Der *Terraform*-Code kann applyed werden. • Es kann per *kubectl* auf das Cluster zugegriffen werden.

Story #2: EKS Autoscaling	
Beschreibung: Je nach Last der *Pods* skaliert das Cluster dynamisch die *Nodes*.	Akzeptanzkriterien: • Das automatische skalieren funktioniert • Das Scaling ist *AZ*-übergreifend • Das Cluster ist ausfallsicher

Story #3: Erstellen der *Container*-Registry	
Beschreibung: Erstellen der *Container*-Registry für die Applikationen.	Akzeptanzkriterien: • Es existiert ein *Container*-Registry • Das Cluster kann *Container* aus der Registry laden

Story #4: Erzeugen der *Container*	
Beschreibung: *Container* werden automatisiert gebaut und in die Registry hochgeladen.	Akzeptanzkriterien: • *Container* für unsere Applikationen werden gebaut • *Container* werden ins Registry hochgeladen • *Container* werden mit ihrer Version getagged

Story #5: Automatisierter Deployment Prozess	
Beschreibung: *Container* können automatisiert auf dem Cluster deployed werden.	Akzeptanzkriterien: • *Container* werden über einen automatisierten Prozess auf dem Cluster deployed • Es kann nachvollzogen werden, wer wann welches Deployment gestartet hat • Deployments können nur von Berechtigten ausgeführt werden

Story #6: Externe Erreichbarkeit über einen Load Balancer einrichten	
Beschreibung: Services können aus dem Internet erreichbar gemacht werden.	Akzeptanzkriterien: • Ein Service kann über ein *Ingress*-Objekt aus dem Internet erreichbar gemacht werden • Externe TLS-Zertifikate werden automatisch ausgestellt • Domains werden in Terraform verwaltet, können aber in Kubernetes angebunden werden

Story #7: Erreichbarkeit von Services außerhalb des Clusters	
Beschreibung: Externe Services wie *RDS*, Elastic Cache und *On-Premise* können erreicht und pro Applikation zugelassen werden.	Akzeptanzkriterien: • Externe Services können aus dem Cluster erreicht werden • Standardmäßig sind alle Verbindungen zu externen Services gesperrt

Story #8: Transport-Verschlüsselung innerhalb des Clusters	
Beschreibung: Die Kommunikation zwischen den Applikationen findet über HTTPS statt, um die Sicherheit dieser sicherzustellen.	Akzeptanzkriterien: • Services kommunizieren verschlüsselt untereinander • Services müssen selbst dafür nicht verändert werden

Story #9: Erstellen von Rollen für Entwickler	
Beschreibung: Es gibt eine Rolle, mit der Entwickler ihre Entwicklerarbeit tun können.	Akzeptanzkriterien: • Entwickler können: – Logs einsehen – Neue Applikationen erstellen – Die Applikationen aus dem Internet verfügbar machen – Applikationen löschen

Story #10: Kosten können auf Applikationen verteilt werden	
Beschreibung: Für das Reporting der Kosten müssen die Anteile an der Infrastruktur auf verschiedene Applikationen zuordenbar sein.	Akzeptanzkriterien: • Es gibt eine Möglichkeit, die Kosten des Clusters anteilig pro Applikation zu bestimmen

Story #11: Monitoring für Cluster Komponenten	
Beschreibung: Alle Applikationen im Cluster sollen automatisch überwacht werden können.	Akzeptanzkriterien: • Es gibt eine Prometheus- und Grafana-Instanz • Pods werden nach korrekter Konfiguration automatisch überwacht

Story #12: Logs werden nach *Cloudwatch* exportiert	
Beschreibung: Logs werden automatisch nach *Cloudwatch* exportiert und sind dort einsehbar. Wichtig für die *Splunk*-Integration.	Akzeptanzkriterien: • Logs aller Pods werden nach *Cloudwatch* exportiert • Logs der Cluster Komponenten werden nach *Cloudwatch* exportiert • Logs können von *Cloudwatch* nach *Splunk* exportiert werden

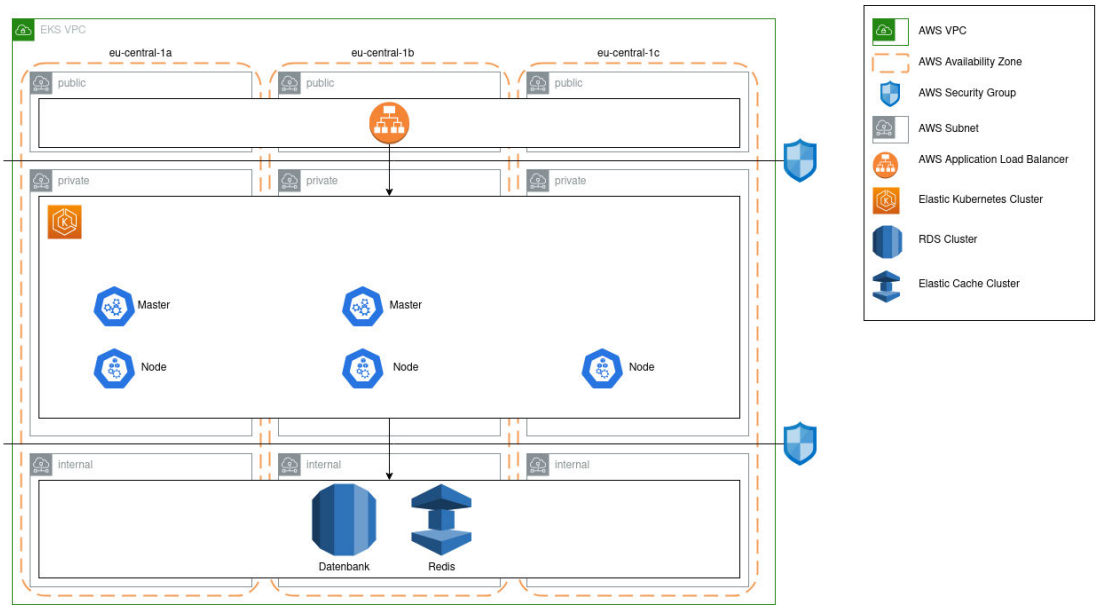

Abbildung 3.1: Übersicht über die Cluster Architektur

3.3 Architektur

Die Architektur des Clusters soll eine möglichst strikte Trennung zwischen Entwickler- und Operations-Rollen unterstützen. Im Folgenden werden der Begriff Developer für die Entwickler-Rolle und der Begriff Operator für die Betriebs-Rolle verwendet. Ich gehe auf die zwei Sichten ein, die die beiden Rollen jeweils auf die Infrastruktur haben.

3.3.1 Die Operator-Sicht

In Abb. 3.1 sieht man horizontal eine Einteilung in drei *AZs*. Vertikal hat jede *AZ* drei Zonen (vgl. Zonenkonzept Goll und Gmbh (2019)). Die public-zone ist einzig für Komponenten, die direkte Internet-Konnektivität benötigen. Hierzu gehört zum Beispiel der öffentliche *Load Balancer*, der für den *Ingress*-Traffic zuständig ist.
In der private-zone befinden sich die Komponenten des Clusters. Hierzu zählen vor allem die *Nodes*, auf denen später die *Container* laufen.
In der internal-zone befinden sich weitere Komponenten wie Datenbanken, Cache Server etc. Jede Schicht ist wie im klassischen Schichtenmodell auf Netzwerkebene durch *Security Groups* nur von ihrem unmittelbaren Vorgänger aus erreichbar.

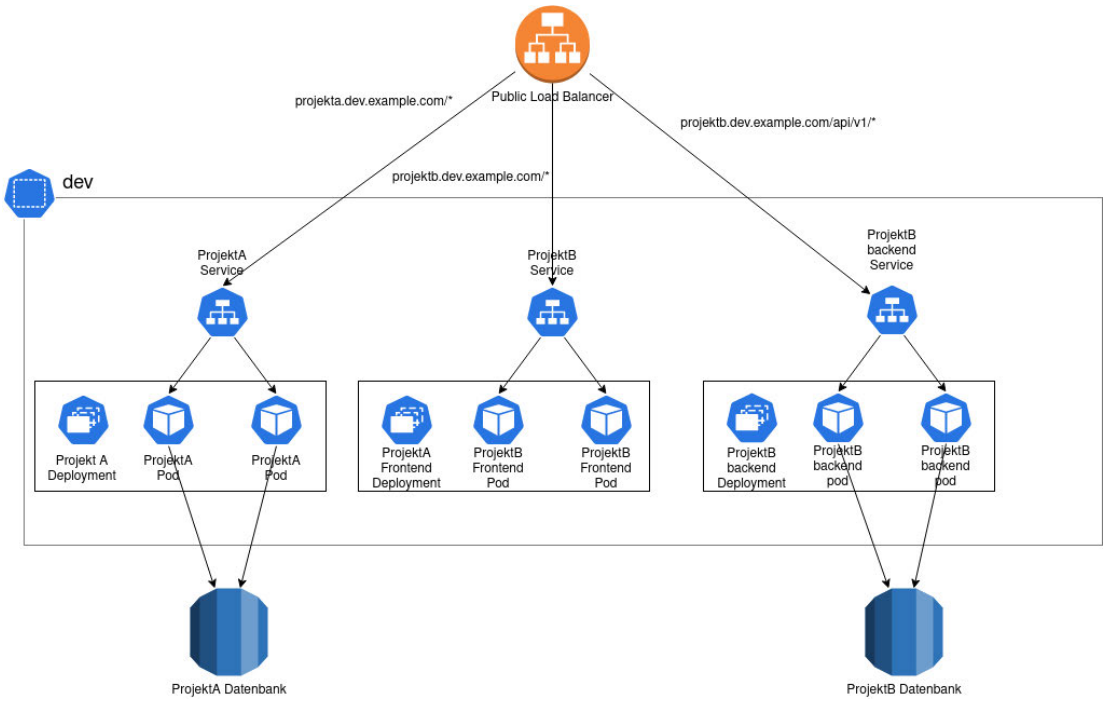

Abbildung 3.2: Beispielhafte Darstellung eines Namespaces

3.3.2 Die Developer Sicht

Innerhalb des Clusters wird für jede Entwicklungsumgebung ein *Namespace* bereitgestellt. Die Applikationen werden dann je nach Bedarf angeordnet und verbunden. In Abb. 3.2 ist dies an einem Beispiel demonstriert. "Projekt A" stellt hierbei eine Monolithische Applikation dar und "Projekt B" eine Applikation mit getrenntem Frontend und Backend.
Kommt nun *Traffic* für "Projekt A" beim *Load Balancer* an, wird der *Traffic* an den *"Projekt A"-Service* geleitet. Der *Service* leitet den *Traffic* anschließend an einen der *"Projekt A"-Pods* weiter, wo die Anfrage verarbeitet und gegebenenfalls in der *Datenbank* gespeichert wird.

3.3.3 Tooling

Die Trennung der beiden Rollen wird auch durch die Trennung des Toolings unterstützt. Das Haupttool für Operator ist *Terraform*. Hiermit wird das Cluster verwaltet und über das jeweilige *VCS* versioniert. Developer arbeiten vorwiegend mit *kubectl*. Damit haben sie Zugriff auf Logs und Performance-Daten außerdem können sie bei Bedarf debuggen.

3.4 Designentscheidungen

Im Folgenden gehe ich auf einige Designentscheidungen ein, die ich vor der Implementierung treffen musste. Einige Entscheidungen habe ich während der Entwicklung angepasst. Auf diese gehe ich in der Umsetzung genauer ein.

3.4.1 AWS Fargate vs. AWS EC2 Cluster

AWS bietet zwei verschiedene Betriebsmodi für Kubernetes Nodes an. Der erste Betriebsmodus *Fargate* nutzt von AWS gemanagte virtuelle Maschinen. Abgerechnet wird pro *Pod* und es kann immer nur ein *Pod* pro *Fargate* Instanz gestartet werden. In der Praxis bedeutet das, dass für jeden *Pod*, der über *Fargate* gestartet wird, ein eigenes *Node* in der Kubernetes API auftaucht. Über *Fargate* gestartete *Nodes* werden automatisch bei Bedarf hinzugefügt und sobald sie nicht mehr benötigt werden, wieder entfernt.
Der EC2-Betriebsmodus dagegen provisioniert eine virtuelle Maschine und erlaubt es, solange *Pods* auf den Instanzen zu starten, wie das *Node* Ressourcen für diese hat. Abgerechnet wird hier nach *Elastic Compute Cloud (EC2)* Instanz. Die *EC2*-Maschinen müssen jedoch vom Operator selbst gemanaged werden.
Ich habe mich für die *EC2* Variante entschieden da ich annehme, dass die Kosten niedriger sind und die Zeit bis ein *Fargate-Pod* bereit ist in meinen Tests etwa 30 Sekunden und mehr betrug.

3.4.2 Datenbank im Cluster vs. Außerhalb des Clusters

Eine Frage, die sich mir während der Planung gestellt hat, war die Frage, wie ich Datenbanken für die Applikationen im Cluster zur Verfügung stelle. Im Moment hat jedes Projekt sein eigenes *RDS*-Cluster mit jeweils einer R/W und einer RO Instanz. Denkbar wäre, die Datenbank aber auch im Cluster selbst zu hosten. *Kubernetes* bietet hierfür zum Beispiel ein Feature namens PersistentVolumeClaims an. Hierzu gibt es auch vorgefertigte Konfigurationen, die ein komplettes Postgres-Cluster aufsetzen können.
In den meisten *Public Clouds* wird ein *Software as a Service (SaaS)* Dienst mit den populärsten Datenbanken angeboten. Viele der Probleme, die man beim selbst-betrieb der Datenbank hat, werden dadurch gelöst.
Ein grundlegendes Prinzip von Kubernetes ist, dass der gesamte State sich in der Cluster-Datenbank befindet und das System aktiv versucht, Abweichungen vom Zielstatus zu erkennen und dahin zu transformieren. Aus diesem Grund sind *Pods* als wegwerfbar entworfen worden. Weicht ein *Pod* vom beschriebenen Verhalten ab, wird der *Pod* durch eine frische Kopie ersetzt.

Ein solcher Austausch kann auch passieren, wenn zum Beispiel eine Konfigurationsdatei aktualisiert wird, die der *Pod* nutzt. Dies ist nur möglich, weil die *Pods* als *Stateless* angelegt sind.
Datenbanken funktionieren deutlich anders. Sie sind auf das lokale Speichern von Daten ausgelegt. Sie lösen das Hochverfügbarkeitsproblem auf Applikations-Ebene und nicht auf Infrastruktur-Ebene.

3.4.3 Container Network Interface

Container Network Interface (CNI) ist ein Standard für das Management von Netzwerkschnittstellen in *Container*-Netzen. Er wird in *Kubernetes* (Kubernetes, a) dazu genutzt, um jedem *Pod* seine eigene IP zuzuweisen und Kommunikation zwischen allen Containern auf allen *Nodes* zu ermöglichen.
Es gibt verschiedene Implementationen dieses Standards mit unterschiedlichen Features. Das Plugin calico ist eines der bekanntesten Vertreter. Neben Standard-Features wie *Network Policys* bietet es einen *Border Gateway Protocoll (BGP)*-Stack der Routen des Clusters außerhalb des Clusters verkünden kann.
Ein anderes Plugin ist das amazon-vpc-cni. Dieses Plugin funktioniert nur auf *AWS* und bietet einige spezifische Features. Zum Beispiel werden Container IPs direkt über das *Virtual Private Cloud (VPC)* geroutet und sind damit direkt über *Security Groups* und Network *Access Control Lists (ACLs)* in *AWS* kontrollierbar. Hiervon verspreche ich mir Vorteile im Management von cluster-externen Services.
Aus diesem Grund und dem Support von *AWS* habe ich mich für das amazon-vpc-cni entschieden.

4 Umsetzung

Story 1

Für die Erstellung des *EKS* Clusters habe ich das offizielle *EKS* Terraform Modul genutzt. Der Terraform State wird in Gitlab gespeichert.

Story 2

Für die Umsetzung dieser Story wird zum einen auf das die *Auto Scaling Group (ASG)* des EKS-Terraform-Moduls zurückgegriffen. Zum anderen werden zwei weitere Services innerhalb des Clusters benötigt. Der erste Service ist der *Node Termination Handler*. Dieser sorgt dafür, dass bevor *Nodes* heruntergefahren werden, alle laufenden *Pods* auf andere *Nodes* umgezogen werden. Der zweite Service ist der *Cluster Autoscaler*. Dieser sorgt dafür, dass das Cluster immer genug *Nodes* hat um alle *Pods* zu starten. Die *ASG* sorgt dafür das die *Nodes* gleichmäßig auf alle *AZs* verteilt werden. Sind nicht mehr genug Ressourcen im Cluster verfügbar, startet der *Cluster Autoscaler* neue *Nodes*.

Story 3

Für jede Applikation im Cluster wurde ein *Elastic Container Registry (ECR)* Repository angelegt. Den *Nodes* werden über eine *Identity and Access Management (IAM)* Policy Rechte zugewiesen, um auf diese zuzugreifen.

Story 4

Es wird über eine *Gitlab CI* Pipeline ein Container für die Applikation gebaut und automatisiert im dazugehörigen *ECR* Repository gespeichert.

Story 5

Die *Manifests* werden im Gitlab Repository abgelegt und werden von da aus in das Cluster übernommen. Nur Personen mit entsprechenden Berechtigungen können Änderungen im Cluster vornehmen.

Story 6

Für die externe Erreichbarkeit wird ein weiterer Service innerhalb des Clusters deployed: Der sogenannte *Load Balancer Controller*. Dieser überwacht *Ingress* Objekte in Kubernetes und richtet bei Bedarf *AWS Load Balancer* ein und konfiguriert diese so, dass diese extern erreichbar sind. TLS-Zertifikate werden automatisch in *AWS* gesucht und angewendet.

Story 7

Cluster-externe Services wie zum Beispiel ein *RDS* Cluster können über den entsprechenden DNS-Namen aufgelöst und auf diese zugegriffen werden. Wenn gewünscht kann dies auch über *Security Groups* eingeschränkt werden. Kommunikation im Cluster ist standardmäßig über eine *Network Policy* eingeschränkt und muss explizit freigegeben werden.

Story 8

Diese Story wurde nicht umgesetzt und ist somit für spätere Verbesserungen vorgesehen. Im Kapitel 6 gehe ich hierauf genauer ein.

Story 9

Über eine bestimmte Konfiguration kann einem *AWS*-Benutzer eine Kubernetes-Rolle zugewiesen werden. Dadurch können Entwicklern entsprechende Rechte eingeräumt werden.

Story 10

Der Ressourcen-Verbrauch kann über die Metrics API von Kubernetes abgerufen und überwacht werden. Zum Beispiel kann hierfür Prometheus genutzt werden.

Story 11

Prometheus kann einfach über das prometheus-operator Projekt installiert werden. Dieses bietet neben einem *Prometheus*- und einem *Grafana*-Deployment auch eine direkte Integration der Kubernetes-Komponenten.

Story 12

Die Pod Logs können über Tools wie zum Beispiel *Fluentd* weitergeleitet werden. *Fluentd* kann wiederum die Logdateien an beliebige andere Services weiterleiten. Hierzu gehören zum Beispiel *Splunk* und *Cloudwatch*.

5 Evaluation

5.1 Beschreibung des Vergleichsobjekts

Im folgenden beschreibe ich die aktuelle Version der Architektur, um sie mit dem Prototyp zu vergleichen.

5.1.1 Architektur

Wie in Abb. 5.1 zu sehen, ähnelt die Struktur dem Prototyp. Es gibt drei Zonen in drei *AZs*. Daraus resultieren insgesamt neun Subnetze. *Ingress* kommt nur über den *Application Load Balancer (ALB)* im "public" Subnetz. Von dort aus wird der *Traffic* in das "private" Subnetz geleitet. Im einfachen Fall verarbeitet eine der *EC2*-Maschinen diesen und kann anschließend beispielsweise auf das *Redis*-Cluster zugreifen.
Wenn die Applikation noch von anderen internen Services abhängt, erfolgt die Kommunikation mit diesen über einen internen *Load Balancer*.

5.1.2 Deployment

In der aktuellen Infrastruktur nutzen wir *Blue Green Deployment*. Beim Deployment wird also eine Kopie von der *ASG* gemacht und die neuen *EC2* Maschinen mit dem neuen *Artefakt* gestartet. Sind alle neuen Instanzen gestartet und bereit, wird der *Traffic* langsam auf die neuen Instanzen umgeschaltet. Anschließend wird der *Traffic* aller alten Instanzen abgeschaltet und die alten Versionen gelöscht.

5.1.3 Kosten

Die Kosten der Infrastruktur teilen sich auf zwei Bereiche auf. Zum einen die fixen Kosten, dazu gehören unter anderem *Load Balancer*, geteilte Services, *Artefakt* Speicher und andere Posten, die entweder gar nicht oder nur schwer bestimmten Einzelprojekten zugeordnet werden können.

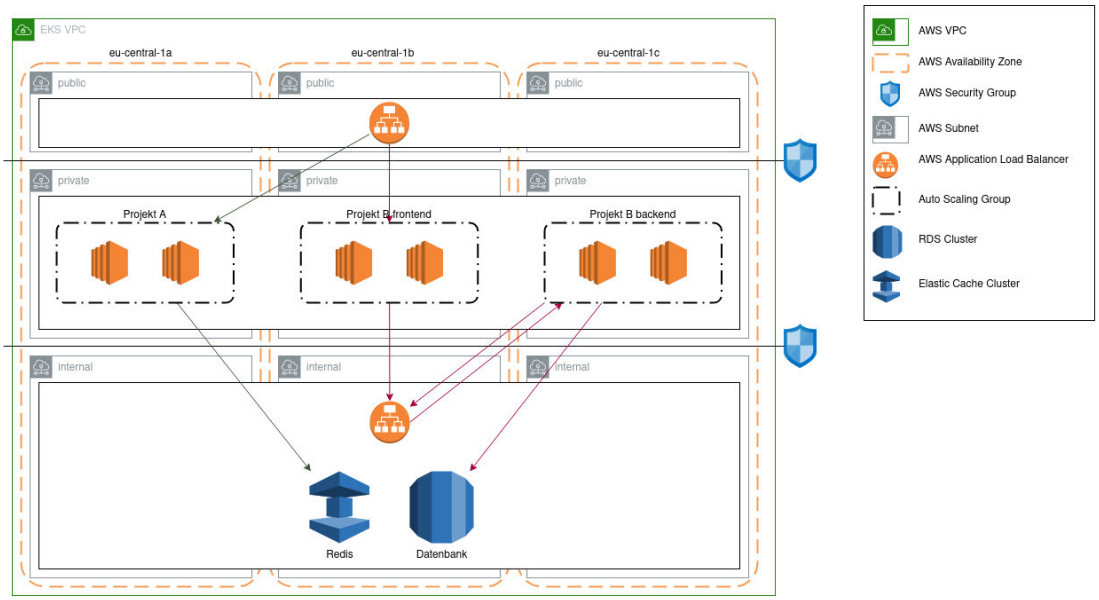

Abbildung 5.1: Aktuelle Infrastruktur Architektur

Zum anderen gibt es Kosten, die direkt auf Einzelprojekte verteilt werden können. Hierzu gehören zum Beispiel die Kosten für Datenbanken und *EC2*-Maschinen, welche für dieses Projekt erstellt werden.
Zur Berechnung der Betriebskosten für die Einzelprojekte werden die Kosten, die auf diese eindeutig umlegbar sind, verwendet. Die Gemeinkosten werden gleichmäßig auf alle Projekte verteilt. Bei der Betrachtung der realen Kosten, in der auf virtuellen Maschinen basierten Infrastruktur, können nur knapp 8% (siehe 5.2) der Kosten auf konkrete Projekte aufgeteilt werden.

5.2 Metriken

Im Folgenden stelle ich einige Metriken vor, mit denen ich die Komplexität der zugrundeliegenden Infrastrukturen bewerten und vergleichen werde.

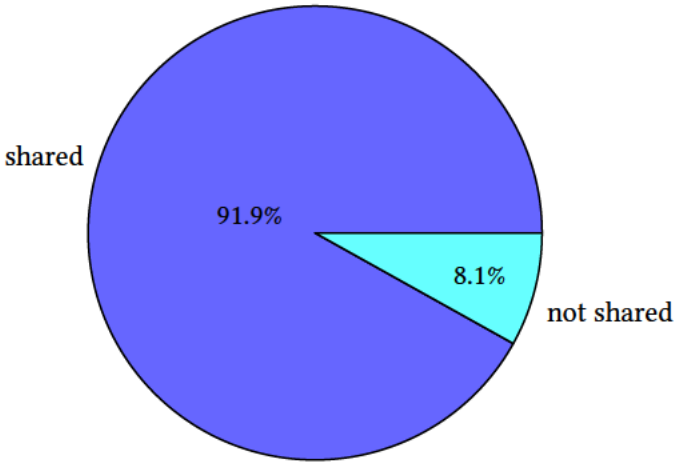

Abbildung 5.2: Monatliche Kosten der VM Infrastruktur nach Typ

5.2.1 Menge an Infrastruktur-Code

In Informatiker-Kreisen gelten die *Lines of Code* als gute Metrik für Qualität. In diesem Kontext halte ich sie jedoch für eine gute Metrik für Komplexität. Je mehr Code in einer Code-Basis vorhanden ist, desto mehr Fehler hat der Code potenziell.
Um beide Codebasen möglichst vergleichbar zu machen, beschränke ich den Vergleich auf einen kleinen Teil des Quellcodes. Hierzu gehört alles, was benötigt wird, um die Applikation auf einem Server auszuführen und zu deployen. Hierzu gehört der gesamte Terraform-Code, der mit *Load Balancers*, *Redis*, Datenbank, *VPC* und der Applikation selbst zu tun hat. Leerzeilen und Kommentare werden nicht mitgezählt.

Infrastruktur Typ	Zeilen Terraform Code
EC2	2582
Kubernetes	983

Tabelle 5.1: Anzahl der Zeilen Terraform-Code der jeweiligen Infrastruktur

Wie in der Tabelle 5.1 zu sehen, beläuft sich die Menge an Zeilen für die Infrastruktur auf etwa das Zweieinhalb-fache. Dafür gibt es mehrere Gründe. Zum einen hat die *EC2*-basierte Infrastruktur einige Features, die der Prototyp nicht hat. Zum anderen ist im Prototyp die Trennung zwischen Applikation und Infrastruktur an einer anderen Stelle. In der *EC2*-basierten Infrastruktur existiert ein *Terraform*-Modul, das als Blaupause für das Deployment einzelner Applikationen dient. In der *Kubernetes*-basierten Infrastruktur ist die

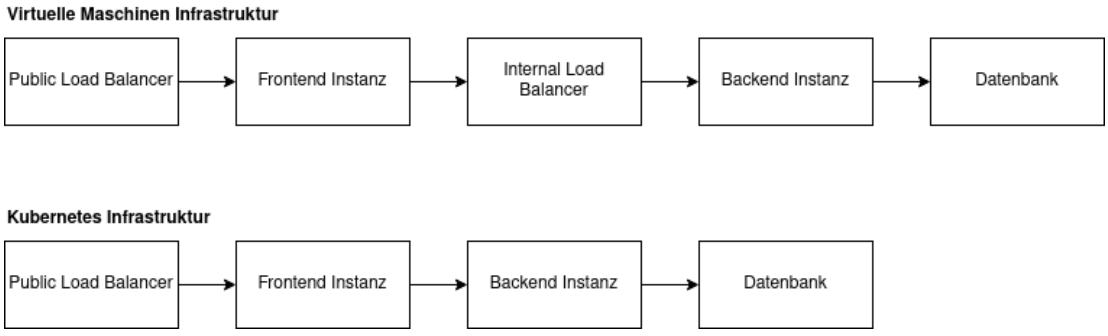

Abbildung 5.3: Aktive Komponenten die ein Request durch die Infrastrukturen passieren muss

Konfiguration und das Deployment nicht Teil der Infrastruktur. Es ist Teil der Applikation und damit in der Hand der Entwickler.
Der aber wohl größte Einflussfaktor ist, dass viele Mechanismen, die in *AWS* konfiguriert werden müssten, in *Kubernetes* vordefiniert sind. Einer dieser Punkte ist zum Beispiel das Deployment. In der *EC2*-Infrastruktur entspricht die Konfiguration des Deployments etwa 200 Zeilen *Terraform*-Code. In der *Kubernetes*-Infrastruktur gibt es keinen expliziten *Terraform*-Code, der das Deployment konfiguriert. Die Deployment-Methode ist dort schon durch *Kubernetes* vorgegeben.

5.2.2 Anzahl der Hops

Bei der Fehleranalyse ist es wichtig, den Weg, den ein Request durch die Infrastruktur nimmt nachzuvollziehen und mögliche Verarbeitungsfehler aufzudecken. Moderne Infrastrukturen haben jedoch zwischen dem Kunden und der Applikation selbst mehrere *Reverse Proxys* und *Load Balancer*. An jeder dieser Zwischenstationen kann ein Request theoretisch auf Probleme stoßen.
Aus diesem Grund nutze ich hier die Anzahl der Applikationen, die ein Request passieren muss, als Metrik für Komplexität. Umso mehr Applikationen desto komplexer die Infrastruktur. Dabei zähle ich nur Komponenten, die oberhalb von Layer 3 aus dem OSI-Schichtenmodell arbeiten. (vgl. selflinux.de)
Gemessen habe ich am Beispiel von “Projekt B” (vgl. 5.1 und 3.2) aus der Beschreibung der Infrastruktur. Dies bedeutet konkret den Weg den ein Request von einem Frontend über ein Backend bis in die Datenbank nehmen muss.

EC2		0,00 USD	363,64 USD
Kubernetes		0,00 USD	353,68 USD

Abbildung 5.4: Kostenberechnung im AWS pricing calculator

Wie in Abb. 5.3 zu sehen muss ein Request in der *Kubernetes*-Infrastruktur eine aktive Komponente weniger passieren. Durch die Eigenschaften vom Kubernetes Service, eine Cluster-IP-Adresse bereitzustellen, ist durch die Einsparung des internen Load Balancers kein Verbindungsabbruch zu erwarten.

5.2.3 Kosten

Die Kosten von Infrastruktur waren in klassischen Rechenzentren oft als Fixkosten zu betrachten. Einen Server in einem Rechenzentrum an- oder auszuschalten hatte dort nur wenig Auswirkungen. Mit der Migration in die Cloud hat sich das geändert. "*Public Cloud*"-Anbieter schreiben sekundengenaue Rechnungen für die Nutzung ihrer Serverkapazitäten. Das macht es möglich, Kosten detaillierter zu analysieren und zu optimieren. Um die Kosten sinnvoll zu vergleichen, beschränke ich mich auf einen kleinen Teil der Infrastruktur.
Aus rechtlichen Gründen sind die folgenden Zahlen keine echten, sondern wurden mit dem AWS pricing calculator ermittelt. Ich habe jedoch die Kosten mit den echten Zahlen validiert. Wie in 5.4 zu sehen, sind die Kosten bei dem in Kapitel 3.3 vorgestellten Modell mit zwei Projekten in etwa gleich. Jedoch entwickeln sich die Kosten mit steigender Service-Anzahl zugunsten der Kubernetes Infrastruktur. Dies liegt daran, dass die Container weniger Overhead haben und die Kosten dadurch besser skalieren.

5.2.4 Deployment-Zeit

Eine niedrige Deployment-Zeit ist in der Entwicklung ein Vorteil. Zum einen bekommt der Operator schneller Feedback, ob das Deployment erfolgreich war oder nicht. Bei einem defektem Deployment kann er anschließend auch schneller eine wieder funktionierende Applikationsversion deployen. Zum anderen lassen sich einige Probleme nicht oder nur

schwer auf einem Entwicklerrechner nachstellen. Hier kann das Testen in einer Testumgebung mit schneller Iterationszeit eine Einsparung von Entwicklerressourcen bedeuten.

Infrastruktur Typ	Minimale Zeit	Maximale Zeit	Durchschnitts Zeit
EC2	6m 6s	11m	8m 30s
Kubernetes	10s	20s	14s

Tabelle 5.2: Mittlere, Minimale und Maximale Deployment Zeiten in den jeweiligen Infrastrukturen

In 5.2 sieht man die durchschnittlichen Deployment-Zeiten auf den jeweiligen Infrastrukturen. Daraus lässt sich erkennen, dass die *Kubernetes*-Infrastruktur deutlich schneller deployen kann. Dies liegt zum einen an der Deployment-Technik. Kubernetes unterstützt nativ nur *Rollout Deployment*. Die *EC2*-Version setzt auf ein *Blue Green Deployment*. Zum anderen setzt *Kubernetes* auf einen Abbau aller Verbindungen auf Applikationsebene. *Kubernetes* geht davon aus, dass, sobald alle *Container* im *Pod* beendet sind, die *Container* gelöscht werden können. *Kubernetes* sorgt nicht dafür, dass vorher alle Verbindungen abgebaut worden sind.
Ganz im Gegenteil zum *ALB* von *AWS*. Dieser nutzt ein Feature namens *Connection Draining*, welches dafür sorgt, dass bestehende Verbindungen zu den einzelnen Instanzen gehalten werden und regulär beendet werden. Neue Verbindungen werden währenddessen bereits von einer neueren Version der Applikation behandelt.

5.3 Umfrage

Die Erhebung von Metriken ist für eine möglichst objektive Betrachtung der Infrastrukturen wichtig und richtig. Genauso wichtig ist aber die Meinung derjenigen die die Infrastruktur entwickeln und betreiben. Deshalb habe ich eine Umfrage in meinem Team durchgeführt, um ein Meinungsbild über die Technologie und meinen Prototypen zu bekommen.
Im Rahmen eines Workshops habe ich die Technologie und meinen Prototyp vorgestellt und anschließend die folgenden Fragen gestellt.

What do you think about using Kubernetes for our Infrastructure?

Die Antworten bewegen sich in einem breiten Spannungsfeld. Die einen wünschen sich diese Technologie einzusetzen und sehen Kubernetes sogar als strategische Konzernplatform. Andere sehen die technischen Vorteile, haben aber Bedenken, ob die Vorteile einen kompletten Neubau der Infrastruktur rechtfertigen. Einige Kollegen sehen keinerlei Vorteile in *Kubernetes* sehen und die Software für zu komplex halten.

Do you think Kubernetes reduces complexity?

Die meisten Kollegen sind der Meinung, dass durch *Kubernetes* die Komplexität des Gesamtsystems zunehmen würde. Einige weisen jedoch darauf hin, dass die Komplexität für Entwickler abnehmen würde.

Do you think Kubernetes would help empower other members of the team to work on operations?

Die Antworten zu dieser Fragen lassen sich grob in zwei Lager einteilen. Einige Kollegen halten es für möglich, dass sich die Akzeptanz durch Tools wie *kubectl* erhöht. Sie rechnen allerdings auch nur mit Interesse für den Betrieb der Applikationen, nicht des Clusters.
Die Kollegen, die bereits Erfahrung mit dem Betrieb von Applikationen haben, gehen dagegen davon aus, dass der Technologiewechsel kein weiteres Interesse für Operations-Themen wecken wird.

6 Ausblick

Im Folgenden gehe ich auf ein paar Themen ein, die mir während der Entwicklung begegnet sind. Diese sollten noch weiter evaluiert werden.

Sidecar Pattern

Wie im Kapitel Umsetzung bereits gesagt, habe ich die Story zur Transportverschlüsselung nicht umgesetzt. Eine Möglichkeit diese relativ einfach umzusetzen, ist das Sidecar Pattern. Dabei wird ein *Container* neben jedem *Pod* deployed, der für die eingehende und ausgehende Kommunikation des *Pods* zuständig ist. Der angehängte Container übernimmt dann die TLS-Termination und auch Aufgaben wie Retry und Monitoring. Eine beliebte Lösung im Kubernetes-Umfeld ist Istio.

Fargate vs. EC2 in Bezug auf Kosten und Deployment-Zeit

Wie in AWS Fargate vs. AWS EC2 Cluster bereits erwähnt kann *EKS* in zwei verschiedene Betriebsmodi genutzt werden. Zum einen der *EC2*-Betriebsmodus und der *Fargate*-Betriebsmodus. *Fargate* hat dabei den Vorteil, das keine eigenen *EC2*-Maschinen gebraucht werden und damit auch kein *Cluster Autoscaler*.
Bei meinen Experimenten sind mir jedoch auch einige Nachteile aufgefallen. Zum Einen hatte ich immer wieder Netzwerkprobleme, da die Container nicht direkt im eigenen *VPC* laufen, sondern von *AWS* betrieben werden. Zum Anderen ist die Zeit, bis diese lauffähig sind, deutlich erhöht.

Andere *CNIs*

In meinem Prototyp habe ich mich auf das offiziel unterstützte *CNI* von *AWS* selbst konzentriert. Nach der Umsetzung bin ich jedoch auf einige Nachteile gestoßen. Zum Beispiel unterstützt das *CNI* nur eine bestimmte Anzahl von Containern pro *Node*. Wie viele, hat dabei mit der Instanzgröße zu tun.

Eventuell gibt es andere *CNIs* ohne diese Einschränkungen.

7 Fazit

Nach abschließender Betrachtung sprechen meine erhobenen Metriken entweder dafür beziehungsweise nicht explizit dagegen auf Kubernetes als Technologie zu setzen. Die Menge an Infrastruktur-Code (5.2.1) würde sich drastisch verringern. Auch die Deployment-Zeit (5.2.4) und die Anzahl der Hops (5.2.2) sprechen für eine Weiterverfolgung der vorgestellten Lösung. Auch die meisten meiner Kollegen haben Interesse, mit dieser Technologie zu experimentieren und die Vorteile auszuspielen.
Dagegen sprechen die Bedenken der Operations-Kollegen bezüglich der Stabilität eines solchen Clusters und der Technologie im Allgemeinen. Ebenfalls dagegen spricht, dass wir im Moment eine Infrastruktur haben, welche funktioniert und in der wenig Probleme im Betrieb auftreten. An einigen Stellen ist sie komplizierter, als sie sein müsste und teilweise auch schlecht dokumentiert. Jedoch hat sie alle Features, die wir im Moment benötigen und ist auch nach einigen Jahren immer noch gut erweiter- und pflegbar.
Aus diesen Gründen kann ich keine Empfehlung dafür aussprechen, die bestehende Infrastruktur ohne weiteren Grund auszutauschen. Sollten in Zukunft größere Änderungen nötig sein, sollte der Wechsel jedoch in Betracht gezogen werden. Für neue Teams, die eigene Infrastruktur aufbauen wollen, empfehle ich, Kubernetes in Betracht zu ziehen.
Wenn sich so mehrere Abteilungen finden, lassen sich Synergieeffekte mit der Zusammenlegung von mehreren Clustern erreichen.

Literaturverzeichnis

[Goll und Gmbh 2019] Goll, Joachim ; Gmbh, Springer Fachmedien W.: *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik Strategien für schwach gekoppelte, korrekte und stabile Software.* Wiesbaden, Germany Springer Vieweg, 2019. – 32 ff. S

[Kubernetes a] Kubernetes: *Network plugins.* – URL https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/. – (letzter Zugriff: 1.7.2022)

[Kubernetes b] Kubernetes: *Production-Grade Container Orchestration.* – URL https://kubernetes.io/. – (letzter Zugriff: 30.6.2022)

[Matthias und Kane] Matthias, Karl ; Kane, Sean P.: *Docker Praxiseinstieg.* 2. mitp-Verlag. – URL http://www.content-select.com/index.php?id=bib_view&ean=9783958459397. – ISBN 9783958459397

[selflinux.de] selflinux.de. – URL https://www.selflinux.de/selflinux/html/osi02.html#d89e71. – (letzter Zugriff: 1.7.2022)

[Starke 2015] Starke, Gernot: *Effektive Softwarearchitekturen Ein praktischer Leitfaden.* München Carl Hanser Verlag Gmbh & Co. Kg, 2015. – 297–299 S

Glossar

Access Control List (ACL) ist im Netzwerkbereich eine Technologie, auf Paketebene Filter zu formulieren, die bestimmten Traffic zulassen oder blockieren. Im Gegensatz zu Secruity Groups sind diese jedoch stateless.. 16, 35

Amazon Web Services (AWS) ist der führende Anbieter im Bereich Cloud-Computing und wurde 2006 als Tochter von Amazon gegründet. 1, 2, 5, 16, 18, 23, 25, 27, 31–35

Application Load Balancer (ALB) ist ein Produkt aus dem EC2-Service von AWS. Ein ALB arbeitet wie ein klassischer Reverse Proxy auf Basis des HTTP Protokolls. Als Ziel des ALBs können virtuelle Maschinen aber auch Container oder andere AWS-Services sein. 20, 25, 33

Artefakt ist ein Paket, welches genutzt werden kann, um eine Applikation auf einem Server zu deployen. 20

Auto Scaling Group (ASG) ist ein Konzept, um mehrere virtuelle Maschinen gleichen Typs auf *AWS* zu verwalten. Ein typischer Anwendungsfall ist die Bereitstellung einer horizontal skalierenden Applikation. 17, 20

Availability Zone (AZ) ist ein Konstrukt um die Rechenzentren von AWS zu unterteilen. Das Rechenzentrum in Frankfurt ist zum Beispiel in drei AZ's unterteilt. Die einzelnen AZs sind so gebaut, dass sie unabhängig voneinander lauffähig sind. 7, 9, 13, 17, 20

Blue Green Deployment ist eine Deployment Methode bei der zunächst eine Kopie der aktuell laufenden Instanzen erstellt wird. Anschließend wird auf der Kopie die neue Version der Applikation installiert. Sobald dies abgeschlossen ist, wird der *Traffic* auf die neuen Instanzen umgeleitet und die alte Version wird gelöscht . 20, 25

Border Gateway Protocoll (BGP) ist ein Protokoll zum Austausch von Routen zwischen autonomen Systemen. 16

Cloudwatch ist ein Service von *AWS* und wird für die Verarbeitung von Log-Dateien und Metriken genutzt . 12, 19

Cluster Autoscaler ist eine Software, die von *AWS* entwickelt wird und es ermöglicht, ein Kubernetes-Cluster zu skalieren . 17, 27

Connection Draining wird ein Vorgang bezeichnet, bei dem ein Load Balancer das Abschalten einer konkreten Instanz verzögert und dabei alle Verbindungen von Clients zu dieser Instanz nach und nach abbaut. Es wird genutzt, um keine plötzlichen Verbindungsabbrüche für Clients zu provozieren . 25

Container ist ein Prozess, der durch verschiedene Techniken von anderen Prozessen auf dem gleichen System isoliert wird.. 2, 3, 9, 10, 13, 16, 25, 27

Container Network Interface (CNI) ist eine Schnittstelle für die Netzwerkkonfiguration von Containern.. 16, 27, 28

Docker ist die bekannteste Lösung um containerisierte Applikationen zu erstellen und laufen zu lassen . 2

Elastic Compute Cloud (EC2) ist einer der ältesten Services von AWS. Er umfasst die Bereitstellung von virtuellen Maschinen, Block Storage, Netzwerk Ressourcen und Load Balancern. 15, 20–23, 25, 27

Elastic Container Registry (ECR) ist ein SaaS-Produkt von *AWS*, um Container zu speichern. 17

Elastic Kubernetes Service (EKS) ist ein SaaS-Produkt von *AWS*, um Kubernetes-Cluster bereitzustellen.. 5, 8, 17, 27, 32

Fargate ist ein Service von *AWS*, um ohne eigene Server, containerisierte Applikation laufen zu lassen. Abgerechnet wird nach Ressourcen und pro Container. Zusammen mit *EKS* können auch Kubernetes-Ressourcen über Fargate betrieben werden. 15, 27

Fluentd ist eine Software zum Sammeln und Weiterleiten von Log-Dateien. 19

Gitlab CI ist eine Lösung, um automatisiert Tests oder Deployments für ein Git-Repository auszuführen . 17

Grafana ist eine Software, um Daten in Graphen anzuzeigen. Häufig wird sie genutzt, um Applikationsmetriken zusammen mit Prometheus auszuwerten. 19

Identity and Access Management (IAM) ist ein Service, um Nutzer und Rechte auf *AWS* zu verwalten. 17

Infrastructure as Code (IaC) ist ein Software Pattern, um Recheninfrastruktur mit typischen Softwareentwicklungs-Tools zu organisieren und zu verwalten. Der Vorteil ist, dass man die Infrastruktur weitgehend automatisieren kann und mit Versionskontrollsystemen wie Git nachvollziehbar ablegt. 5, 7, 35

Ingress ist der *Traffic*, der von außerhalb in die eigene Infrastruktur herein kommt. 10, 13, 18, 20

kubectl ist das Haupttool, um Objekte innerhalb des Kubernetes-Clusters zu verwalten. Es können damit unter anderem Deployments gestartet, neue Container erstellt und Load Balancer konfiguriert werden . 8, 14, 26

Kubernetes (oder auch K8s) ist eine Software-Lösung, um containerisierte Applikationen über mehrere Host-Computer zu verteilen und zu organisieren. Kubernetes kümmert sich selbständig um die Bereitstellung und Lauffähigkeit, der im Cluster bereitgestellten Applikationen. Hierzu wird der Soll-Zustand der Applikationen in *Manifests* definiert und auf das Cluster angewendet . 5, 15, 16, 22–26

Load Balancer sind Applikationen, die Traffic, zum Beispiel vom Kunden, auf mehrere Instanzen einer Applikation verteilen. In AWS existieren zwei Typen von Load Balancern. Zum einen der *ALB*, der auf HTTP-Ebene arbeitet und der Network Load Balancer, der auf TCP-Ebene arbeitet . 13, 14, 18, 20, 22, 23

Load Balancer Controller ist eine Software innerhalb des Kubernetes-Clusters, die externe Load Balancer je nach Bedarf provisioniert und konfiguriert. 18

Manifest ist ein Konzept von Kubernetes und beschreibt ein Objekt darin. Häufig werden diese Manifeste in YAML geschrieben. 18, 33

Namespace im Kubernetes-Kontext ist ein Konzept, um andere Objekte in Kubernetes zu ordnen. Die meisten Objekte müssen einem Namespace zugeordnet werden . 14

Network Policy ist im Kubernetes-Kontext ein Möglichkeit, die Konnektivität zwischen Pods und Services einzuschränken. 16, 18

Node ist im Kubernetes-Kontext die Bezeichnung für einen Server auf dem Container gestartet werden können. Bei dem Server kann es sich um virtualisierte oder auch physische Computer handeln . 9, 13, 15–17, 27, 34

Node Termination Handler ist ein Programm, das innerhalb eines Kubernetes-Clusters auf *AWS* läuft. Es sorgt dafür, dass *Pods* die auf *Nodes* laufen, die heruntergefahren werden sollen, vorher auf andere *Nodes* verteilt werden . 17

On-Premise werden häufig im *Public Cloud* Umfeld Systeme oder Dienste genannt, die in einem privaten Rechenzentrum betrieben werden. Warum ein System nicht in der Cloud betrieben wird, kann verschiedene Gründe haben. Einer könnte sein, dass das System noch nicht migriert wurde oder besonderen Datenschutz Ansprüchen genügen muss, die in einer *Public Cloud* nicht erfüllt werden können . 7, 10

Pod ist im Kubernetes-Kontext die Bezeichnung für die kleinste Einheit, die Arbeit verrichten kann. Sie kann aus einem oder mehreren Containern bestehen . 9, 14–17, 25, 27, 34

Prometheus ist eine Software, um Applikationsmetriken zu erfassen und zu speichern . 19

Public Cloud Anbieter verkaufen Rechenzentrumsleistungen als Produkt. Die bekanntesten Anbieter sind *AWS*, Google Cloud und Microsoft Azure . 1, 2, 5, 15, 24, 34

Redis ist ein in Memory Key Value Store und wird zum Beipiel für Caching und Speichern von Sessions genutzt. 20, 22

Relational Database Service (RDS) ist ein SaaS Produkt von *AWS* für relationale Datenbanken. 10, 15, 18

Reverse Proxy ist ein Software, die als aktive Komponente zwischen zwei Applikationen eingesetzt werden kann. Damit wird zum Beispiel Load Balancing, TLS-Termination oder die Manipulation von Requests erreicht . 23

Rollout Deployment ist eine Deployment-Methode, bei der einzelne Instanzen nach und nach ersetzt werden. Sollen beispielsweise drei Instanzen durch eine neue Version ersetzt werden, wird zunächst eine neue Instanz mit der neuen Version gestartet und anschließend eine der alten abgeschaltet. Dies wird wiederholt bis alle alten Instanzen abgeschaltet sind . 25

Security Group ist ein Konzept aus dem Netzwerkbereich und ermöglicht die Einschränkung von Kommunikation zwischen Netzwerkkomponenten auf IP-, Port- und Protokoll-Ebene. 13, 16, 18

Software as a Service (SaaS) beschreibt Software, die von einem Anbieter entwickelt und betrieben wird. Der Kunde zahlt für die Nutzung eine Gebühr. 15

Splunk ist eine Software für die Auswertung von Log-Dateien und wird häufig in größeren Unternehmen eingesetzt . 12, 19

Stateless bezeichnet die Abwesenheit von Daten, die gespeichert werden müssen. Im Bezug auf Kubernetes meint dies zum Beispiel, dass ein Applikationen in Containern nicht davon ausgehen können, dass, Dateien die im Dateisystem abgelegt wurden, nach einem Neustart noch vorhanden sind. . 16

Terraform ist ein Tool, das *IaC* ermöglicht. Terraform wird von HashiCorp entwickelt und ermöglicht es Anbieter agnostisch Cloud-Infrastruktur zu konfigurieren. Um die Software zu nutzen, bedarf es nur einer Schnittstelle zwischen Terraform und dem Cloud-Anbieter . 6, 8, 14, 22, 23

Traffic bezeichnet Datenverkehr in Informationsnetzen. 14, 20, 31, 33

Version Control System (VCS) ermöglicht das Aufzeichnen und die Nachverfolgung von Code-Änderungen. Der bekannteste Vertreter ist das Tool Git. 6, 14

Virtual Private Cloud (VPC) ist ein Service von *AWS* um eigene private Netze zu erstellen. Hierzu gehören die Verwaltung von Subnetzen, Routing-Tabellen und Netzwerk *ACLs*. 16, 22, 27

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 11. Juli 2022 Daniel Jensen