

Fachartikel: Microservices at Scale  
mit Kubernetes und Istio

Veröffentlichung  
im Java aktuell

Ausgabe - 06/2018  
[www.ijug.eu](http://www.ijug.eu)



► Wie es aussieht, sind Microservices gekommen, um zu bleiben. Jedoch wurden die erhofften Vorteile einer agileren Entwicklung sowie einer verbesserten Wartbarkeit und Skalierung der einzelnen Komponenten häufig nicht erreicht. Der Grund dafür ist eine in vielen Aspekten signifikant erhöhte Komplexität. Um diese Problematik zumindest auf technischer Ebene zu adressieren, verschieben Kubernetes und Istio diese Komplexität in die Infrastruktur. Im Folgenden werden die Grundkonzepte beider Komponenten auf einfache Weise erklärt und es wird aufgezeigt, wie elegant teils hoch komplexe Problemstellungen gelöst werden können.

Getreu dem Motto „Divide and Conquer“ wurden in der Softwareentwicklung über viele Jahre unterschiedlichste Ansätze für die Aufteilung immer komplexerer Software-Systeme entwickelt. Im Java Umfeld sind hier vor allem Java EE oder OSGi zu nennen. Microservices führen diesen Ansatz konsequent weiter, indem einzelne Komponenten in dedizierte Prozesse ausgelagert werden und die Kommunikation zwischen den Komponenten ausschließlich über standardisierte Netzwerkprotokolle und APIs erfolgt. Was auf der einen Seite die bekannten Vorteile gegenüber einer monolithischen Architektur ausmacht, bringt auf der anderen Seite alle Herausforderungen eines verteilten Systems mit sich. Wir können uns also nicht mehr auf die umfangreichen Garantien unseres Application-Servers verlassen. Plötzlich müssen wir uns um viele Dinge selbst kümmern. Hier nur ein kleiner Ausschnitt:

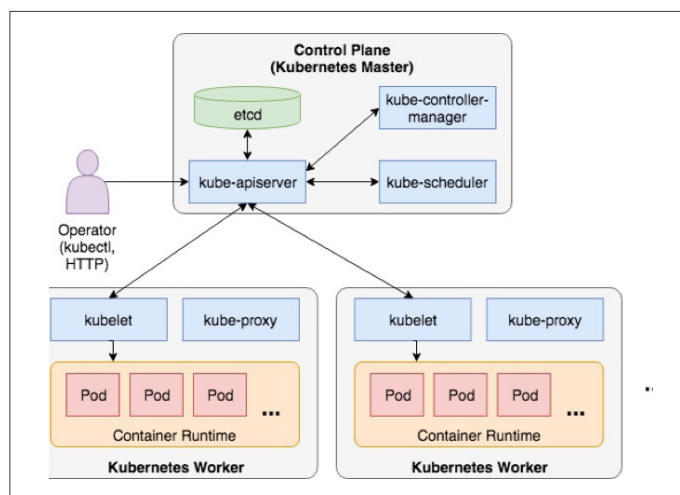
- Registrieren und Auffinden von Services
- Robuste und sichere Kommunikation zwischen Services
- Dynamische Skalierung einzelner Services je nach Auslastung
- Verwaltung von Konfigurationen und Geheimnissen (z.B. Zertifikate, Passwörter)
- Operative Transparenz (Metriken, Logs, Tracing)

### Schlüsseltechnologie Software-Container

Mit Docker-Containern wurde ein entscheidender Baustein für die praktische Umsetzung von Microservices einem breiten Nutzerkreis zugänglich gemacht. Software-Container entkoppeln die einzelnen Services vollständig vom zugrundeliegenden Betriebssystem und ermöglichen eine effiziente und in sich geschlossene und interoperable Paketierung und Auslieferung. Zum Verteilen dieser Images und zum Ausführen eines Containers ist letztendlich nur noch eine Container- Laufzeitumgebung wie z.B. „containerd“ (Docker) oder „cri-o“ notwendig. Die für die Umsetzung der Services verwendeten Programmiersprachen und Frameworks spielen bei der Auslieferung und Ausführung letztendlich keine Rolle mehr. Im Rahmen der „Open Container Initiative“ (OCI) entstehen nun offene Industriestandards rund um das Format sowie die Ausführung von Containern. Dies macht Software-Container zu einem entscheidenden Baustein auf dem Weg zu einer tragfähigen Microservices-Architektur. Ähnlich wie ein ISO-Frachtcontainer in der Schifffahrt, stellen Software-Container eine Schlüsseltechnologie zum Aufbau eines hoch effizienten und automatisierten Software Lebenszyklus dar.

### Service-Orchestrierung mit Kubernetes

Um in der Schifffahrtsanalogie zu bleiben, kann Kubernetes als Hafeninfrastruktur gesehen werden. Mit der allgemeinen Verfügbarkeit von Software-Containern dank Docker und dem Ziel das eigene Cloudgeschäft zu stärken, entschied sich Google 2014 Kernkonzepte seiner bis dato streng geheimen Container-Infrastruktur namens „Borg“ und „Omega“ im Rahmen des Open-Source Projektes Kubernetes öffentlich zu machen. Kubernetes ermöglicht es Software-Container dynamisch auf einem Cluster von bis zu 5.000 Servern auszuführen. Abbildung 1 gibt eine Übersicht der einzelnen Kubernetes-Komponenten und deren Zusammenspiel.



► Abbildung 1: Kubernetes High-Level Architektur

Die Server in einem Kubernetes Cluster können grundsätzlich in zwei Klassen unterteilt werden. „Master-Server“ dienen dem Management des Clusterzustandes, wohingegen „Worker“ für das Ausführen der eigentlichen Services zuständig sind. Die Kontrollschicht von Kubernetes besteht im Wesentlichen aus vier Komponenten.

#### kube-apiserver

Der API-Server bildet die zentrale Schnittstelle zwischen allen involvierten Komponenten sowie dem Nutzer. Er nimmt Zustandsbeschreibungen und Statusinformationen entgegen und benachrichtigt andere Komponenten über entsprechende Änderungen.

#### etcd

Etcd ist ein konsistenter und hochverfügbarer Key-Value-Store und wird vom API-Server für die Ablage des gesamten Clusterzustandes genutzt. Zugriff auf den etcd erfolgt ausschließlich über den API-Server.

## kube-scheduler

Der Scheduler überwacht den Clusterzustand auf neu eingestellte oder nicht zugewiesene Arbeitspakete und selektiert einen passenden Worker-Knoten. Bei der Auswahl werden eine Vielzahl unterschiedlicher Parameter berücksichtigt (z.B. Ressourcenanforderung, Hardware- und Software-Einschränkungen, affinity und anti-affinity Spezifikationen, usw.).

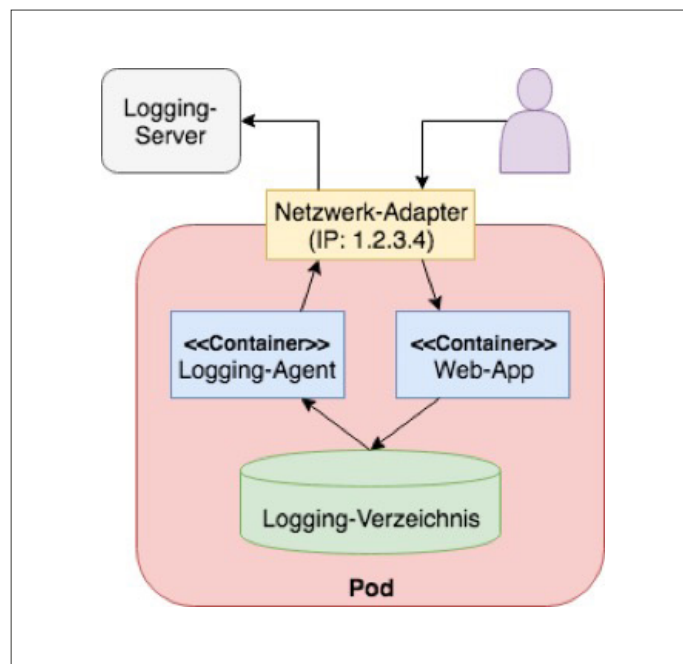
## kube-controller-manager

Der Controller-Manager enthält eine Reihe von unterschiedlichen Controllern, die auf Zustandsänderungen im Cluster reagieren. Dies können z.B. Änderungen des gewünschten Deployments durch den Nutzer (z.B. Skalierung, Versionsupdate) oder der Wegfall einzelner Knoten sein.

Die Worker-Knoten sind dagegen deutlich einfacher aufgebaut. Das kubelet dient dabei als Agent, der die Befehle des API-Servers entgegennimmt und entsprechend Container startet, konfiguriert und wieder stoppt. Hierfür interagiert das kubelet mit der eigentlichen Container-Runtime. Wo anfangs noch direkt Docker angebunden wurde, hat sich nun das „Container Runtime Interface“ (CRI) etabliert. CRI definiert eine offene und standardisierte Schnittstelle und wird unter anderem von Docker bzw. containerd, rkt und cri-o implementiert. Zusätzlich läuft auf jedem Knoten eine kube-proxy Instanz. Diese ermöglicht die Umsetzung eines Servicekonzeptes, indem Netzwerkroutern und Weiterleitungen zwischen einzelnen Containern im Cluster dynamisch konfiguriert werden.

Der Nutzer stellt nun seine Services als Container-Images bereit und beschreibt seine gewünschte Servicelandschaft deklarativ unter Verwendung der durch Kubernetes definierten API-Objekte. Neben unterschiedlichen Objekten zum Ausführen von Containern, bietet Kubernetes zudem Konzepte für die Verwaltung von Konfigurationen und Geheimnissen, Netzwerkconfiguration inklusive einfacher Lastverteilung sowie der Bereitstellung von persistentem Speicher. Einmal per API an Kubernetes übergeben, versuchen unterschiedliche Kontrollprozesse kontinuierlich den gewünschten Zustand herzustellen. Verfügbare Server werden auf ausreichende Ressourcen geprüft und bekommen entsprechend Arbeit zugewiesen. Die Arbeit wird dabei nicht, wie zu vermuten, als einzelner Container definiert, sondern als sogenannter „Pod“. Im einfachsten Falle enthält ein Pod auch nur genau einen Container (siehe Listing Pod).

Ein Pod kann sich auch aus mehreren Containern, die als eng gekoppelte Einheit zu verstehen sind, zusammensetzen. Container in einem Pod laufen immer gemeinsam auf einem Server, teilen sich ein Netzwerk-Interface und haben gleichermaßen Zugriff auf externe Mount-Punkte (siehe Abbildung 2). Pods werden im Kubernetes grundsätzlich als flüchtig betrachtet. Sie bekommen beim Starten eine zufällige IP im Pod-Netzwerk zugewiesen und können im Falle eines Fehlers oder Serverausfalls einfach an anderer Stelle im Cluster neugestartet werden.



➤ Abbildung 2: Kubernetes Pod

In der Praxis werden jedoch selten Pods durch den Benutzer erstellt. Um eine „Selbstheilung“ der Servicelandschaft zu erreichen, implementiert Kubernetes höherwertige Konzepte, die wiederum den Lebenszyklus eines Pods kontrollieren. Alle diese Konzepte folgen einem einheitlichen Muster. Der Nutzer definiert seinen gewünschten Zustand in einem passenden API-Objekt und ein zum Objekt passender Controller überwacht alle Anpassungen an seinen Instanzen und führt die notwendigen Aktionen aus, um den gewünschten Zielzustand zu erreichen. Im Standard bietet Kubernetes unter anderem folgende Controller an.

## Deployment

Mittels eines Deployment-Objektes kann die genau Spezifikation eines Pods (Pod-Template) sowie die gewünschte Anzahl der Instanzen definiert werden. Der Deployment-Controller überwacht Änderungen an den Deployment-Objekten und stellt den gewünschten Zustand her. Hierfür implementiert der Deployment-Controller unter anderem Funktionen zum kontrollierten Skalieren sowie Aktualisieren (Rolling-Update) der einzelnen Pods. Primär wird das Deployment für zustandslose Services genutzt. Name und IP-Adressen der Pods werden jeweils zufällig gewählt.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: busybox:1.29.1
      command: ['sh', '-c', 'echo Hello K8s! && sleep 3600']
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80

```

## StatefulSet

Das StatefulSet ist speziell für das Deployment von Services mit eigenem Zustand konzipiert worden (z.B. Datenbanken). So gibt das StatefulSet im Vergleich zum Deployment zusätzliche Garantien hinsichtlich der Reihenfolge, in der Pods gestartet und gestoppt werden und folgt bei deren Benennung einem stabilen Namensmuster. Zudem stellt das StatefulSet sicher, dass persistenter Speicher immer wieder dem passenden Pod zugeordnet wird.

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: k8s.gcr.io/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "my-storage-class"
        resources:
          requests:
            storage: 1Gi

```

## DaemonSet

DaemonSets stellen sicher, dass auf jedem Knoten im Cluster genau eine Instanz des beschriebenen Pods läuft. Kommt ein neuer Knoten hinzu, wird automatisch ein passender Pod gestartet.

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
        - name: fluentd-elasticsearch
          image: k8s.gcr.io/fluentd-elasticsearch:1.20
          volumeMounts:
            - name: varlibdockercontainers
              mountPath: /var/lib/docker/containers
              readOnly: true
          terminationGracePeriodSeconds: 30
      volumes:
        - name: varlibdockercontainers
          hostPath:
            path: /var/lib/docker/containers

```

## Kubernetes Services

Wie bereits erwähnt, sind Pods sehr dynamisch, da sie jederzeit auf einen anderen Knoten verschoben werden können und ihre IP-Adressen dynamisch vergeben werden. Damit stellt sich die Frage, wie wir auf unsere Services zugreifen können. Hier setzt das Service-Konzept von Kubernetes an. Mittels des kube-proxies und eines Service-API-Objektes kann ein virtueller Service ähnlich einem klassischen Load-Balancer im Cluster definiert werden. Dieser bekommt einen stabilen DNS Namen und leitet Anfragen an einen seiner verfügbaren Endpoints weiter. Die Endpoints entsprechen hierbei einem Set von Pods, die mit Hilfe eines Label-Selectors ermittelt werden. Jeder Pod mit den passenden Labels wird automatisch in die Liste der Endpoints des Services aufgenommen. Kommen Pods hinzu oder fallen weg, wird die Liste dynamisch aktualisiert.

```

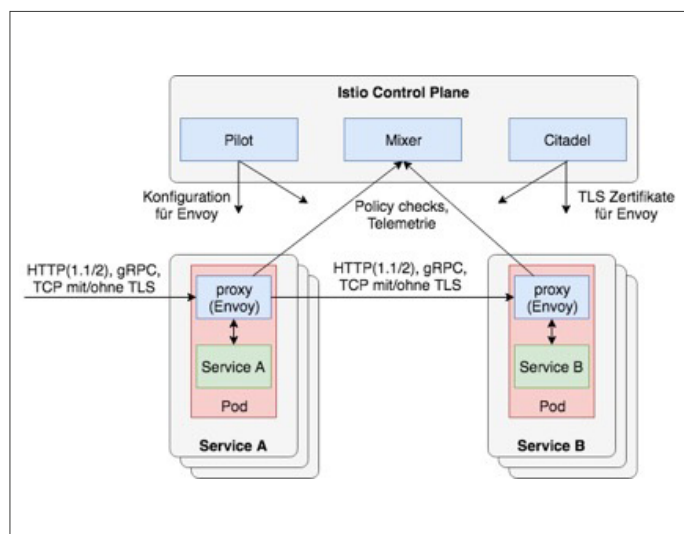
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376

```

Zusammenfassend bietet uns Kubernetes eine Plattform, um unsere Services in Form von Containern deklarativ auf einem Cluster von Servern laufen zu lassen. Updates der Services sowie das Hoch- und Runterskalieren von Instanzen übernehmen die unterschiedlichen Controller automatisch für uns. Das Service-Konzept sowie Funktionen zum Verteilen von Konfigurationen oder Geheimnissen implementieren weitere wichtige Bestandteile einer Microservices-Architektur. Weiterhin fehlen jedoch Konzepte für eine robuste und sichere Kommunikation und die operative Transparenz wird durch Kubernetes auch nur rudimentär adressiert.

## Istio Service-Mesh

Das was uns in Kubernetes noch an Fähigkeiten hinsichtlich einer verteilten Microservices-Architektur fehlt, adressiert das Open-Source Service-Mesh Istio. Wie die Bezeichnung „Service-Mesh“ schon vermuten lässt, betrachtet Istio das gesamte Netz einer verteilten Service-Landschaft und stellt eine ganze Reihe an querschnittlichen Funktionen bereit. Istio ermöglicht die Kommunikation zwischen Services sehr feingranular und dynamisch zu steuern, abzusichern und zu überwachen, ohne dabei die Implementierung der einzelnen Services anzupassen. Möglich wird dies durch den Einsatz eines speziellen „Sidecar“-Proxys namens „Envoy“. Dabei erhält jeder Service sozusagen als Beiwagen einen eigenen Proxy, der jegliche eingehende und ausgehende Kommunikation abfängt und anhand von zentral verwalteten Konfigurationen und Regeln weiterleitet (siehe Abbildung 3).



➤ Abbildung 3: Istio Architektur

Istio integriert und profitiert dabei sehr stark von Kubernetes, auch wenn andere Service-Deployments möglich sind (z.B. Consul oder Eureka auf VMs). Das bereits beschriebene Pod-Konzept von Kubernetes kann beispielsweise exzellent für die Umsetzung des Sidecar-Proxies genutzt werden. Seit Kubernetes 1.9 muss der Proxy nicht einmal mehr explizit in die Pod-Spezifikation aufgenommen werden. Mit Hilfe eines sogenannten „Admission Controllers“ wird der Pod beim Anlegen automatisch modifiziert und um den Proxy erweitert. Innerhalb des Pods wird per IP-Tables der gesamte Netzwerkverkehr auf den Proxy umgeleitet. Für den Entwickler eines Services ist damit das Service-Mesh vollkommen transparent.

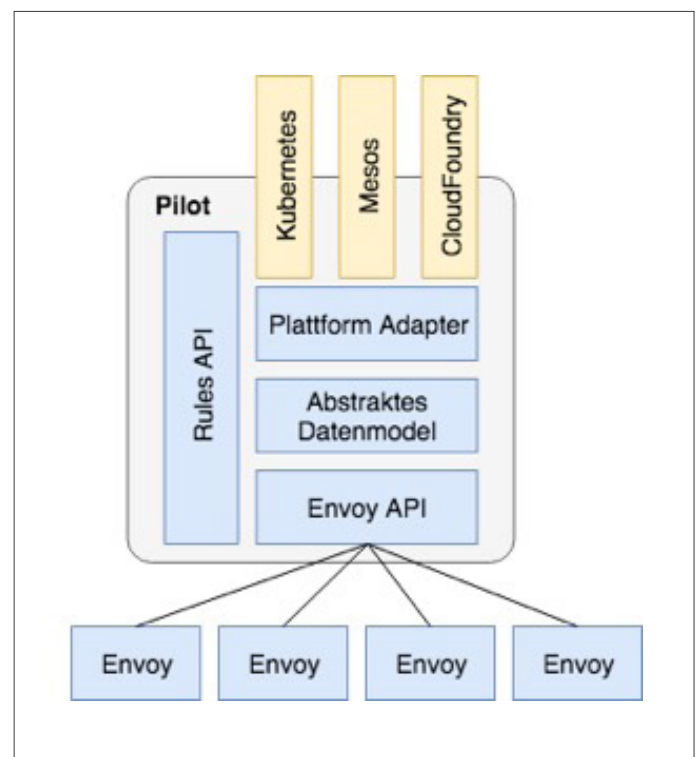
Der in Istio eingesetzte Open-Source Proxy Envoy wurde ursprünglich vom Fahrdienstleister Lyft speziell für ein solches Szenario entwickelt. Envoy ist in C++ geschrieben und weist, neben einer sehr hohen Performance, einen relativ geringen Speicherbedarf auf. Envoy verfügt über eine umfangreiche Konfigurations-API und beherrscht eine ganze Reihe an erweiterten Lastverteilungsfunktionen:

- Automatisches Wiederholen von Aufrufen
- Circuit-Breaking
- Globale Bandbreitenbeschränkung
- Zonenbasierte Lastverteilung

Gesteuert werden die einzelnen Envoy-Instanzen, ähnlich wie bei Kubernetes, durch eine zentrale Kontrolleinheit. Diese setzt sich bei Istio aus den im Folgenden beschriebenen Komponenten „Pilot“, „Mixer“ und „Citadel“ zusammen.

## Pilot

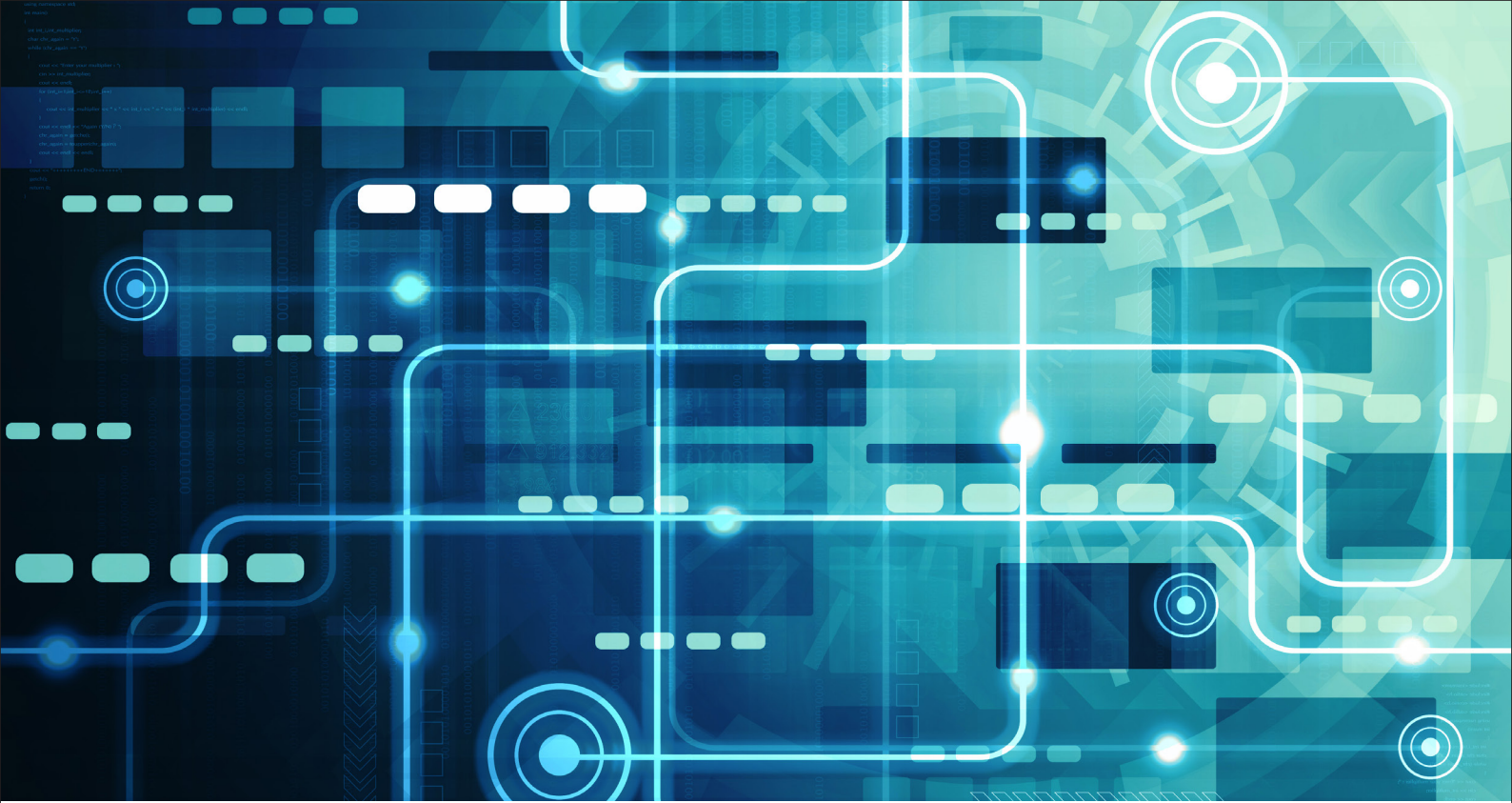
Pilot ist für die Konfiguration der Envoy-Instanzen zuständig und integriert sich in die zugrundeliegende Service-Plattform (siehe Abbildung 4).



➤ Abbildung 4: Istio Pilot

Er definiert ein einheitliches Datenmodell für die Beschreibung der Servicekonfigurationen und eine entsprechende API für die Pflege. Der jeweilige Plattform-Adapter realisiert die Persistenz der Konfigurationen und interagiert mit der Service-Discovery der Plattform, um Informationen über die aktuelle Servicelandschaft zu erlangen. Im Falle von Kubernetes werden alle Istio-Konfigurationen über den kube-apiserver im etcd abgelegt. Möglich wird dies durch ein weiteres Konzept von Kubernetes, die „Custom Resource Definition“ (CRD). Zudem überwacht Pilot die Kubernetes Service-Objekte, welche als Basis für die erweiterten Istio Servicekonfigurationen dienen und die letztendlich die Endpunkte der eigentlichen Service-Pods enthalten. Per Konfiguration können so eine Vielzahl an Funktionen unabhängig vom eigentlichen Applikations-Code realisiert werden:





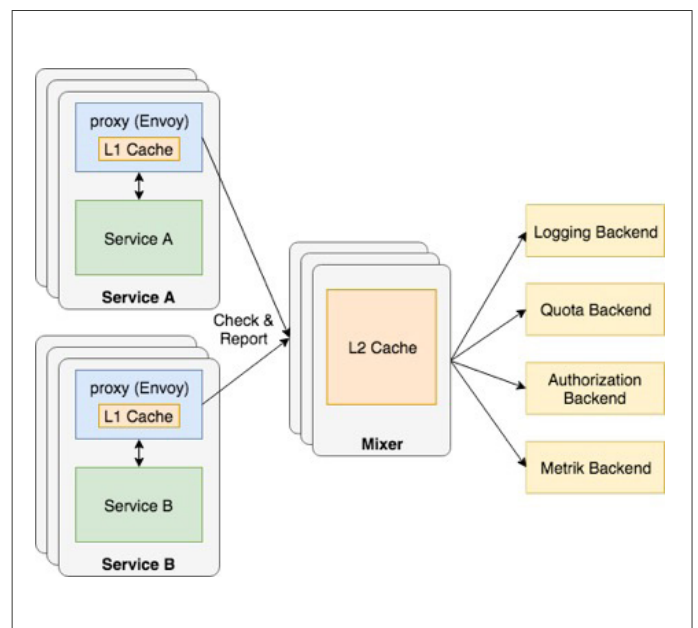
- Anteilige Verteilung von Anfragen auf unterschiedliche Service-Versionen (z.B. Canary-Releases, schrittweiser Rollout, usw.)
- Verteilung von Anfragen auf unterschiedliche Service-Versionen anhand von Inhalten (z.B. A/B-Tests)
- Fehlerbehandlung (z.B. Timeouts, Retries, Circuit-Breaking)
- Fehlerinjizierung zum Testen der korrekten Fehlerbehandlung

Das folgende Listing zeigt wie eine neue Service-Version dediziert für einen User aktiviert werden kann. Es handelt sich hierbei um ein sehr einfaches Beispiel, welches jedoch die Mächtigkeit von Istio bereits vermuten lässt.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        cookie:
          regex: "^(_.*)?(user=jason)(;.*)*?$"
      route:
      - destination:
          host: reviews
          subset: v2
    - route:
      - destination:
          host: reviews
          subset: v1
```

## Mixer

Vor und nach jedem Request eines Services konsultieren die Envoy-Instanzen den Istio-Mixer. Vor dem Request prüft Mixer beispielsweise Zugriffsrechte, Quotas oder beliebige weitere Richtlinien. Nach dem Request nimmt Mixer diverse Telemetrie Daten entgegen und gibt diese an externe Dienste weiter (siehe Abbildung 5).



➤ Abbildung 5: Istio Mixer

Mixer wurde für diesen Zweck hochgradig erweiterbar konzipiert. Viele in Istio enthaltene Funktionen, wie „Role-Based-Access-Control“ (RBAC) zwischen Services oder die Anbindung des Monitoring-Systems „Prometheus“ sind als Mixer-Plugins realisiert. Falls notwendig, können aber auch eigene Systeme angebunden werden.

Da Mixer in dem Gesamtsystem eine sehr zentrale Rolle einnimmt, wurde bei dessen Design vor allem auf Zuverlässigkeit und Latenz geachtet. Neben einem zweistufigen Cache-Konzept ist Mixer „stateless“ implementiert und kann somit einfach horizontal skaliert werden. Jede einzelne Mixer-Instanz ist zudem auf eine Verfügbarkeit von > 99,999% ausgelegt.

## **Citadel**

Citadel ist zuständig für eine starke Authentifikation zwischen einzelnen Services sowie dem Benutzer. Istio nutzt beidseitiges TLS (mutual TLS) für die Absicherung der „Service-zu-Service“ Kommunikation. Hierfür stellt Citadel unter anderem die notwendigen Schlüssel-Paare passend zu den Services zur Verfügung. Diese werden automatisch über Kubernetes-Mechanismen in die einzelnen Service-Pods transportiert und dort von Envoy verwendet. Es müssen also nicht mehr manuell Zertifikate verteilt und in den einzelnen Services eingebunden werden. Da die Verschlüsselung erst durch Envoy vorgenommen wird, hat Envoy zudem jederzeit Zugriff auf den Inhalt und kann entsprechende Routing-Regeln anwenden. Für die Benutzer-Authentifikation steht aktuell OAuth 2 mittels JWT zur Verfügung.

## **Fazit**

Wie wir gesehen haben, ermöglicht uns der Einsatz von Kubernetes und Istio einen großen Teil der Komplexität einer Microservices-Architektur in die Infrastruktur zu verlagern und gleichzeitig das volle Potential einer solchen Architektur zu nutzen. Die für die Entwicklung von fachlichen Services zuständigen Mitarbeiter können sich so voll und ganz auf die Schaffung von Mehrwerten für das Unternehmen konzentrieren. Kombiniert mit organisatorischen Konzepten wie DevOps und einem hohen Grad an Automatisierung, kann so letztendlich die erhoffte Geschwindigkeit und Flexibilität in der IT-Organisation erreicht werden.



## Autor & Ansprechpartner

---

Florian Assmus

Managing IT-Consultant   Chief Architect

[florian.assmus@prodyna.com](mailto:florian.assmus@prodyna.com)

PRODYNA SE

Ludwig-Erhard-Str. 12-14   65760 Eschborn

T +49 69 597 724 - 0   F +49 69 597 724 - 700

[info@prodyna.com](mailto:info@prodyna.com)   [prodyna.com](http://prodyna.com)