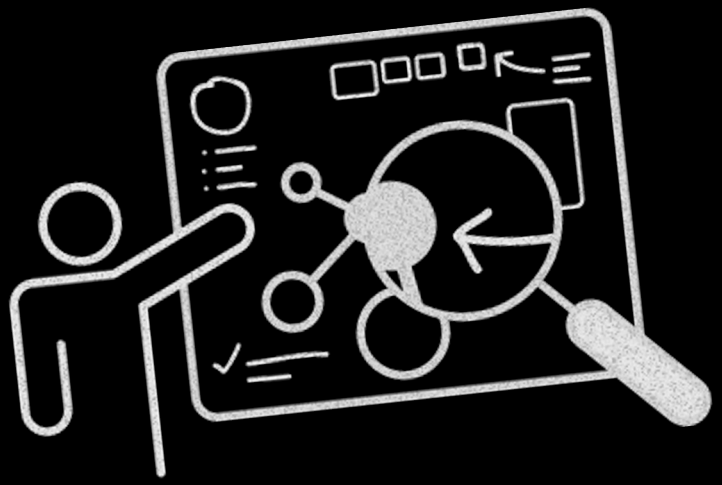


CLOUD-NATIVE COMPUTING

Unit 09:

Monitorbare Architekturen

Stand: 27.03.23

CLOUD NATIVE COMPUTING

PROF. DR.
NANE KRATZKE

KAPITEL 13

Beobachtbare Architekturen (Observability)

Nane Kratzke

Cloud-native Computing

Software Engineering von Diensten und Applikationen für die Cloud

284 Seiten. E-Book inside

€ 59,99. ISBN 978-3-446-46228-1

Weitere Informationen unter: **www.hanser-fachbuch.de** HANSER

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

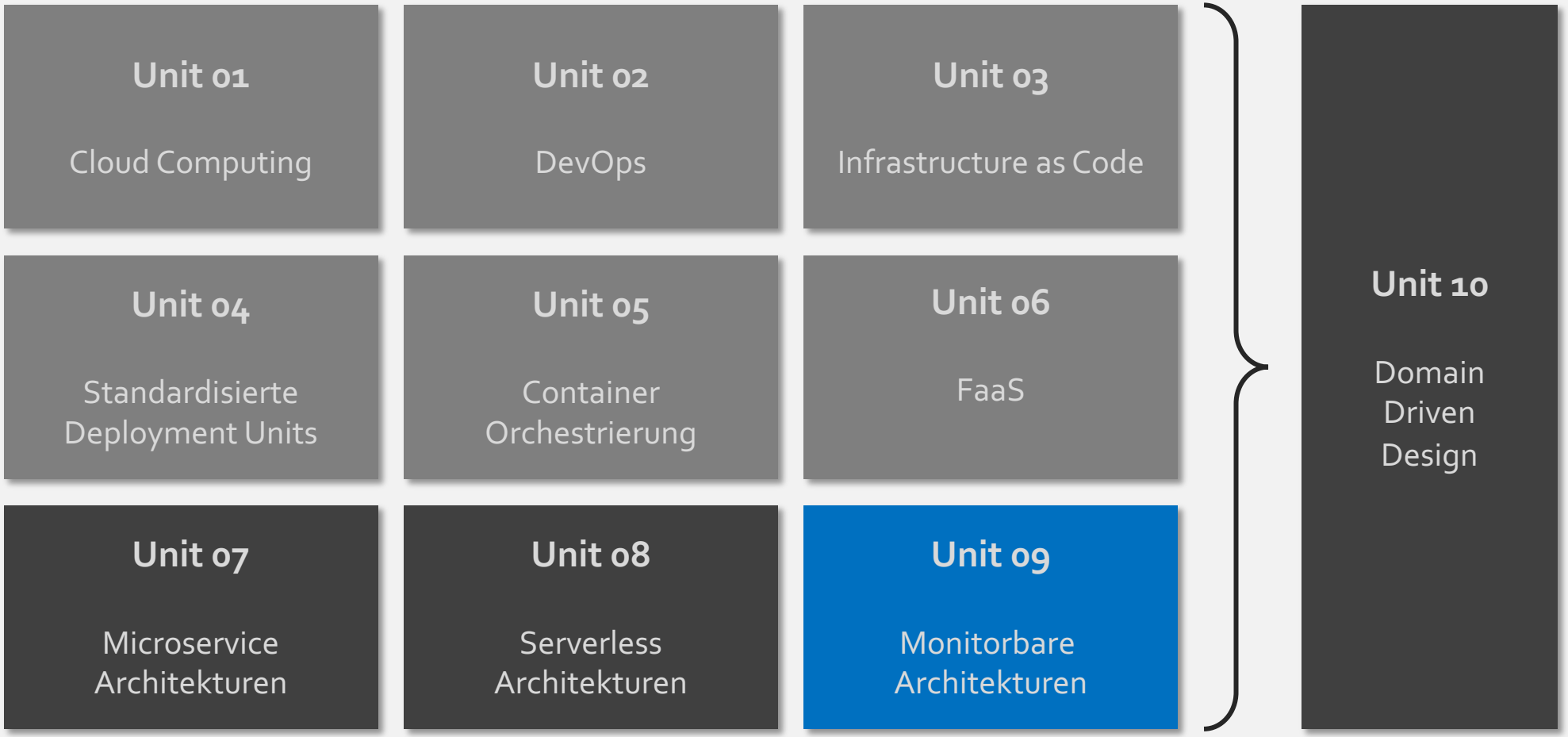
Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

INHALTSVERZEICHNIS

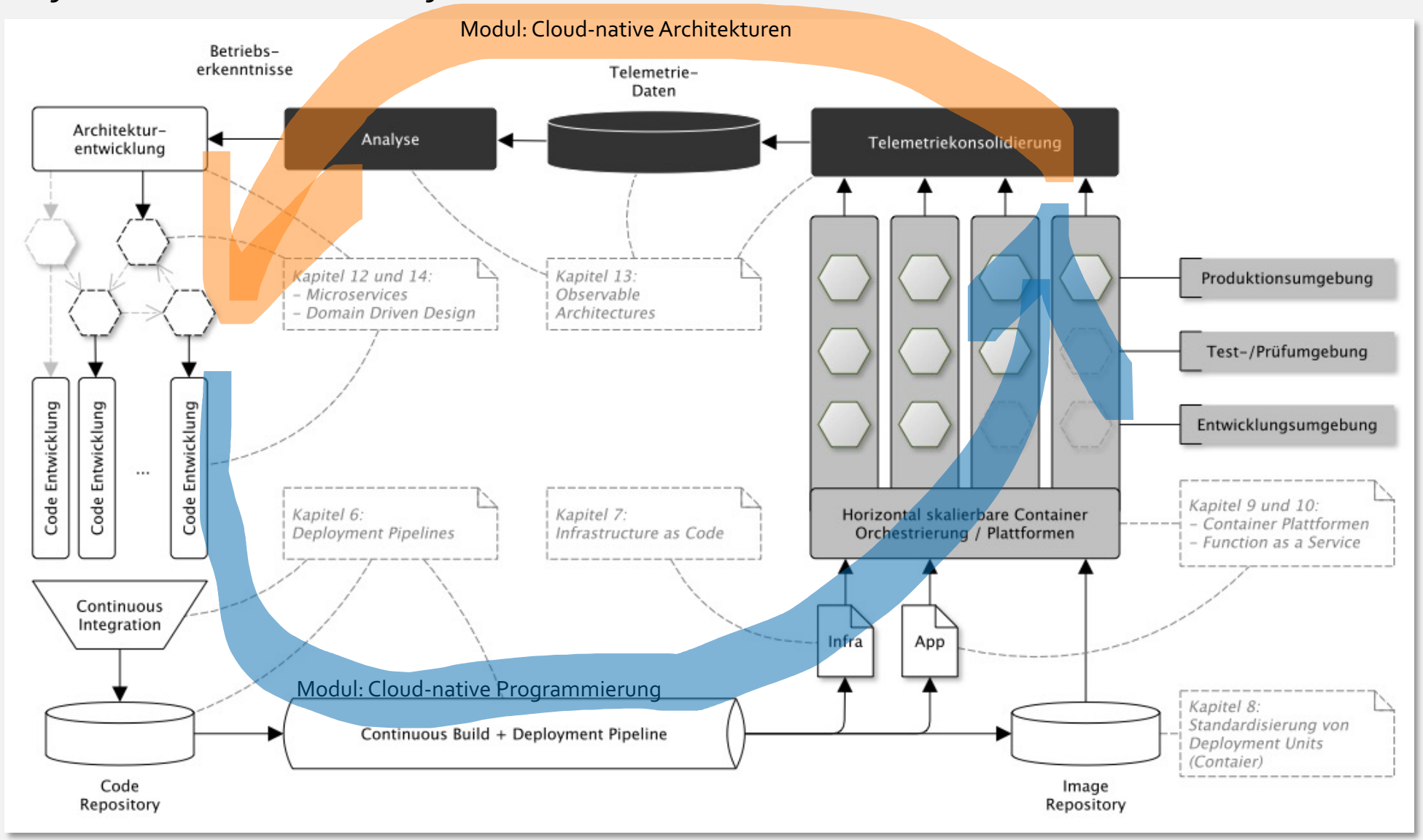
Überblick über Units und Themen dieses Moduls



DEVOPS

konforme Architekturen und Infrastrukturen

Sie erinnern sich an die Unit 2? DevOps? Wir sind jetzt quasi auf dem Rückweg!



Prinzipien des Flow
(durch Automatisierung und Plattform Entwicklung beschleunigen)

Prinzipien des Feedbacks
(durch Beobachtung Systeme verstehen und optimieren)

INHALTE

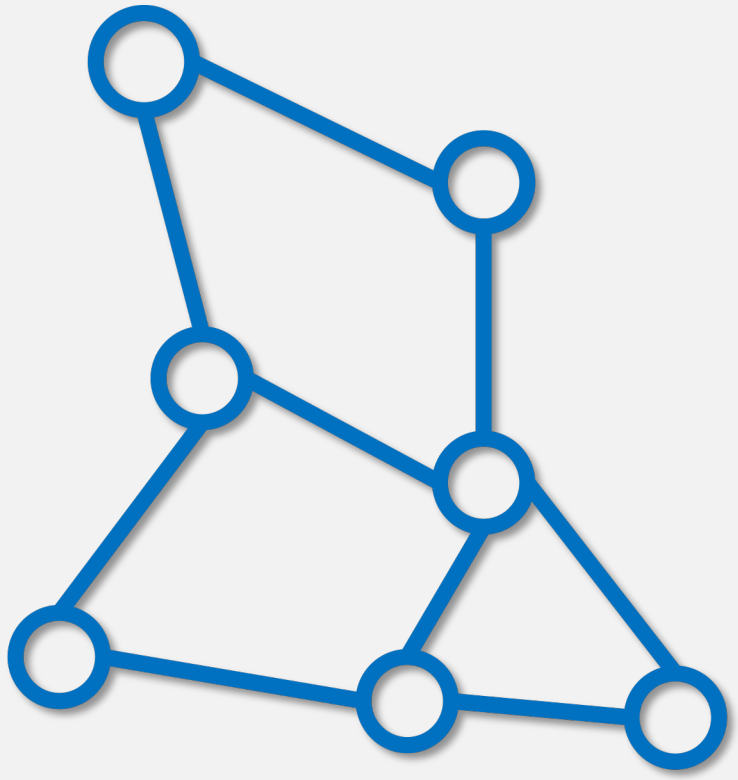
CLOUD NATIVE COMPUTING

Warum Service Meshs?
- Was findet sich in der Literatur?
- Was sagen die Anbieter?

Service Meshs
- Traffic Management
- Resilienz
- Sicherheit
- Beobachtbarkeit

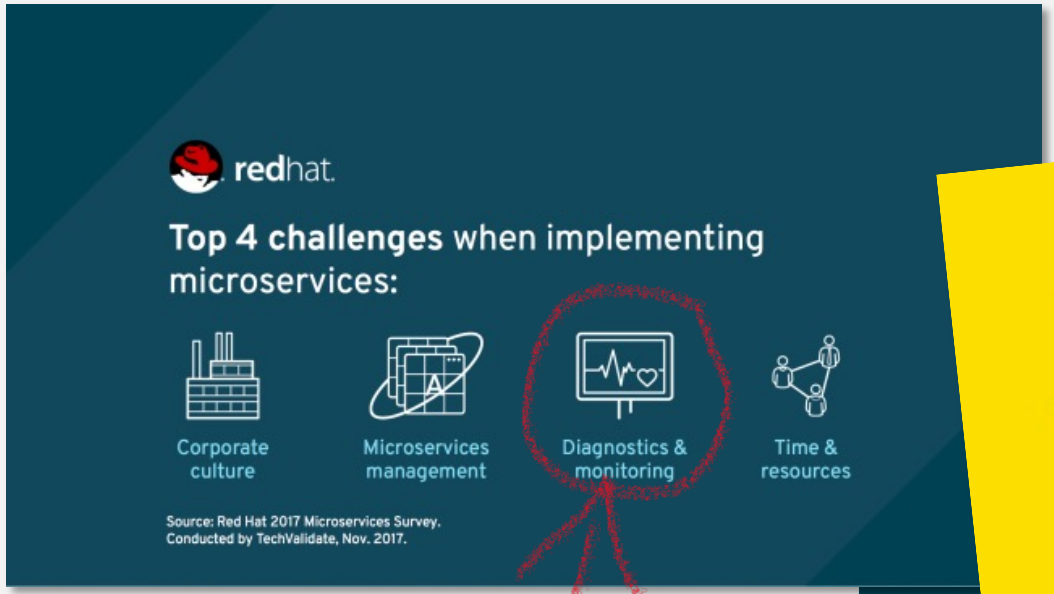
Fallstudie: Istio
- Traffic Management
- Security
- Observability

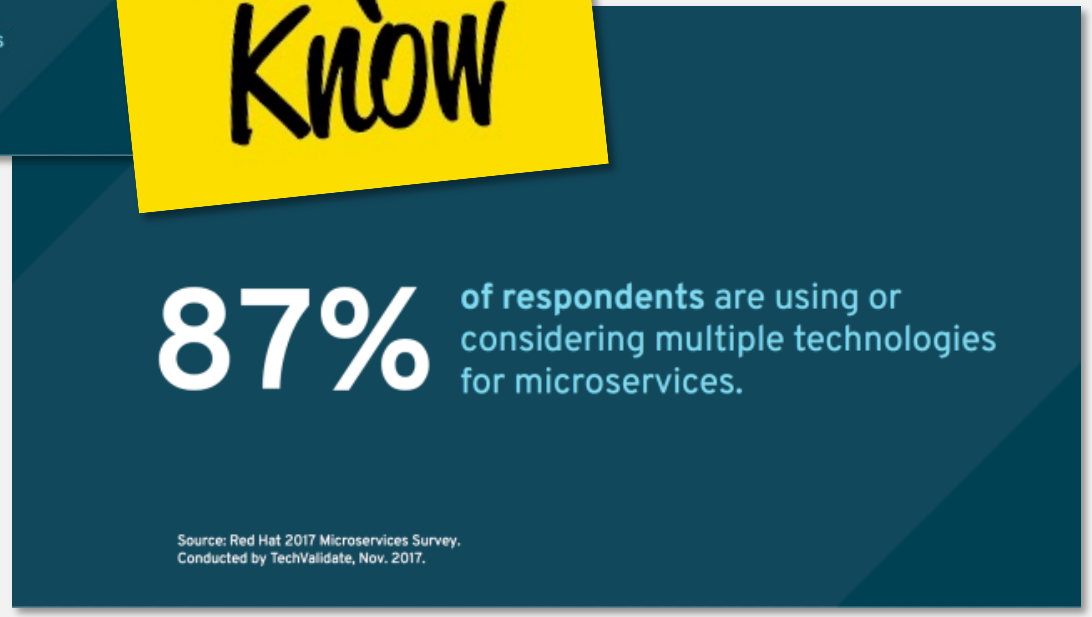
Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



TOP-4 PROBLEME MIT MICROSERVICES

Kultur, Zeit + Ressourcen , Microservices Management, Überwachung





CLOUD NATIVE COMPUTING

Gem. einer Studie von RedHat aus dem Jahre 2017.

Den Einfluss von Microservice Architekturen auf Kultur und Ressourceneinsatz (Conways-Law) haben wir in der Unit 2 (DevOps) und Unit 7 (Microservices) bereits betrachtet.

In dieser Unit geht es primär um Management + Überwachung von Microservice Arch. Im Sinne sogenannter Observable Architectures..

MICROSERVICES PAINS AND GAINS

Ergebnisse aus Industriellen Fall-Studien

CLOUD NATIVE COMPUTING

Spannend: Management wird sowohl auf der Positiv- wie Negativ-Seite gleich häufig in der Operation Phase genannt

Darum soll es in dieser Unit gehen

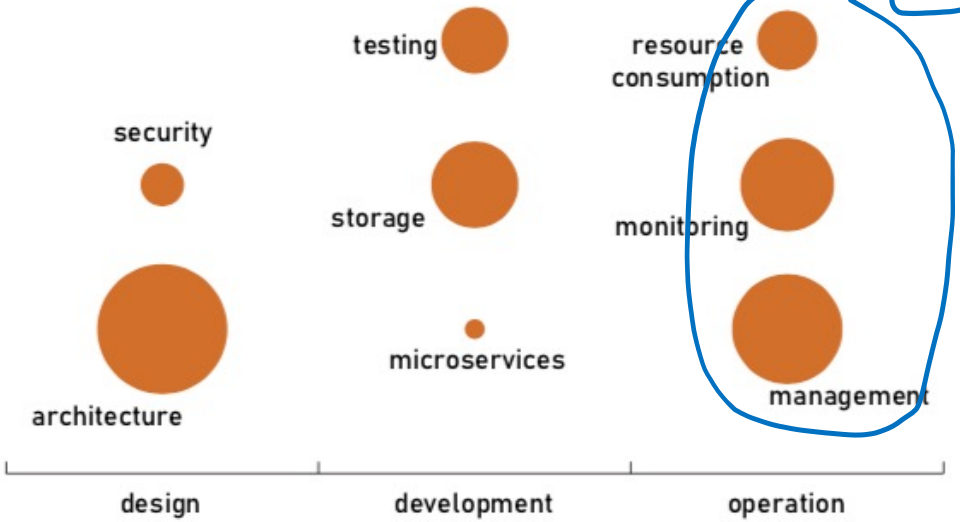


Figure 6: Coverage of the concerns in the taxonomy for pains of microservices. The size of each bubble is directly proportional to the number of selected industrial studies discussing a pain pertaining to the corresponding concern.

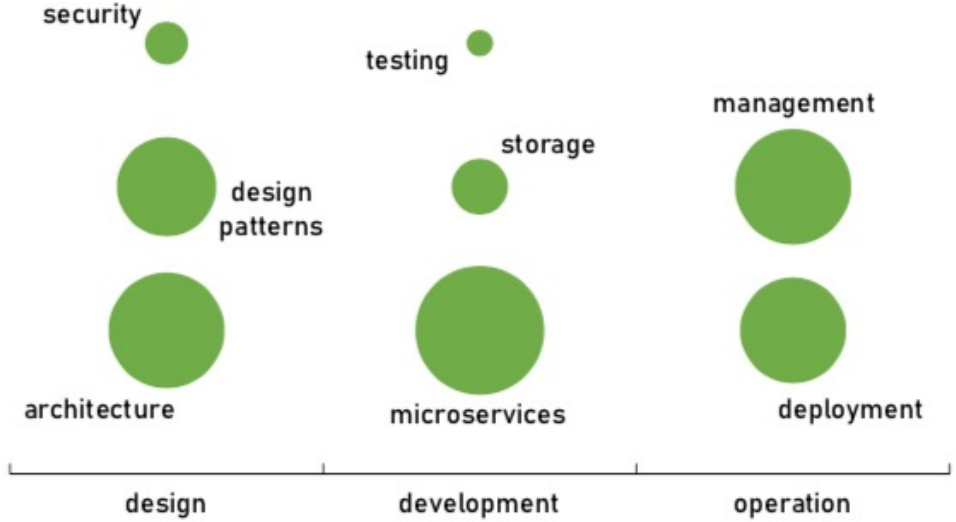


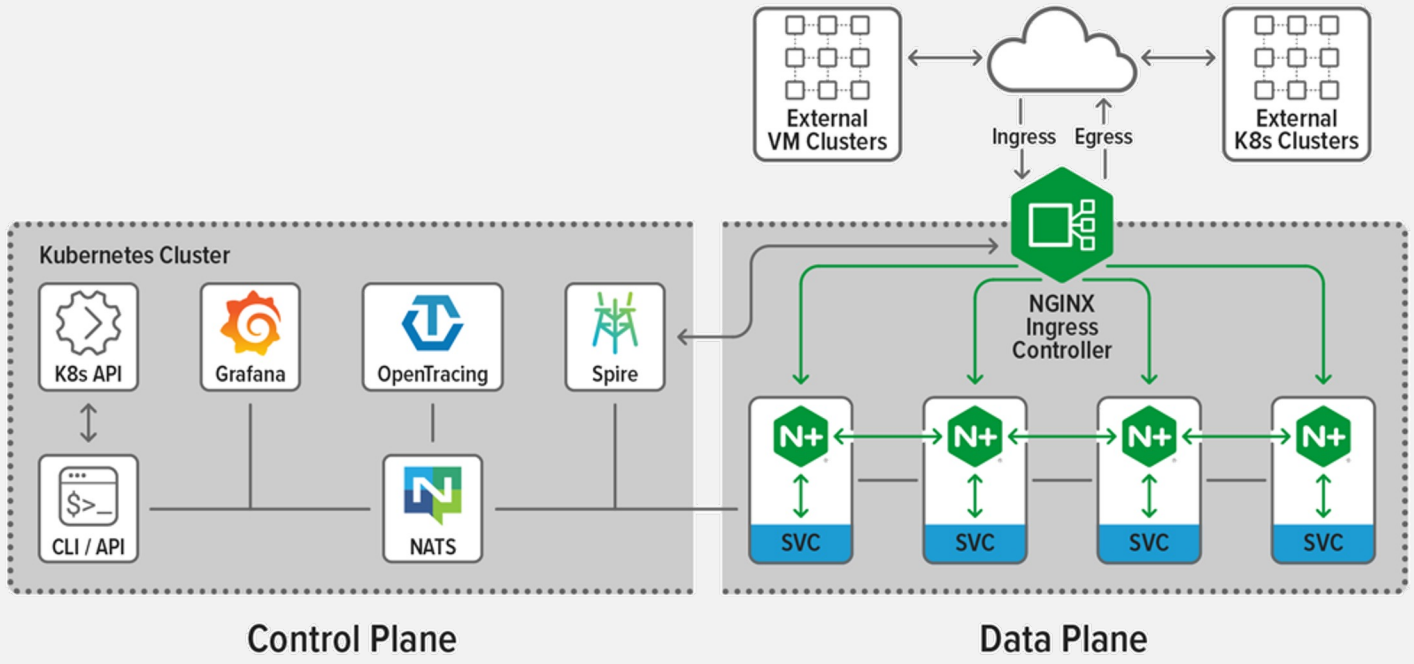
Figure 17: Coverage of the concerns in the taxonomy for gains of microservices. The size of each bubble is directly proportional to the number of selected industrial studies discussing a gain pertaining to the corresponding concern.

Quelle: *Soldani J, Tamburri DA, Van Den Heuvel WJ. "The pains and gains of microservices: A Systematic grey literature review", Journal of Systems and Software, Volume 146, 2018, Pages 215-232. DOI: 10.1016/j.jss.2018.09.082*

NGINX

Was sagen die Anbieter?

NGINX Service Mesh



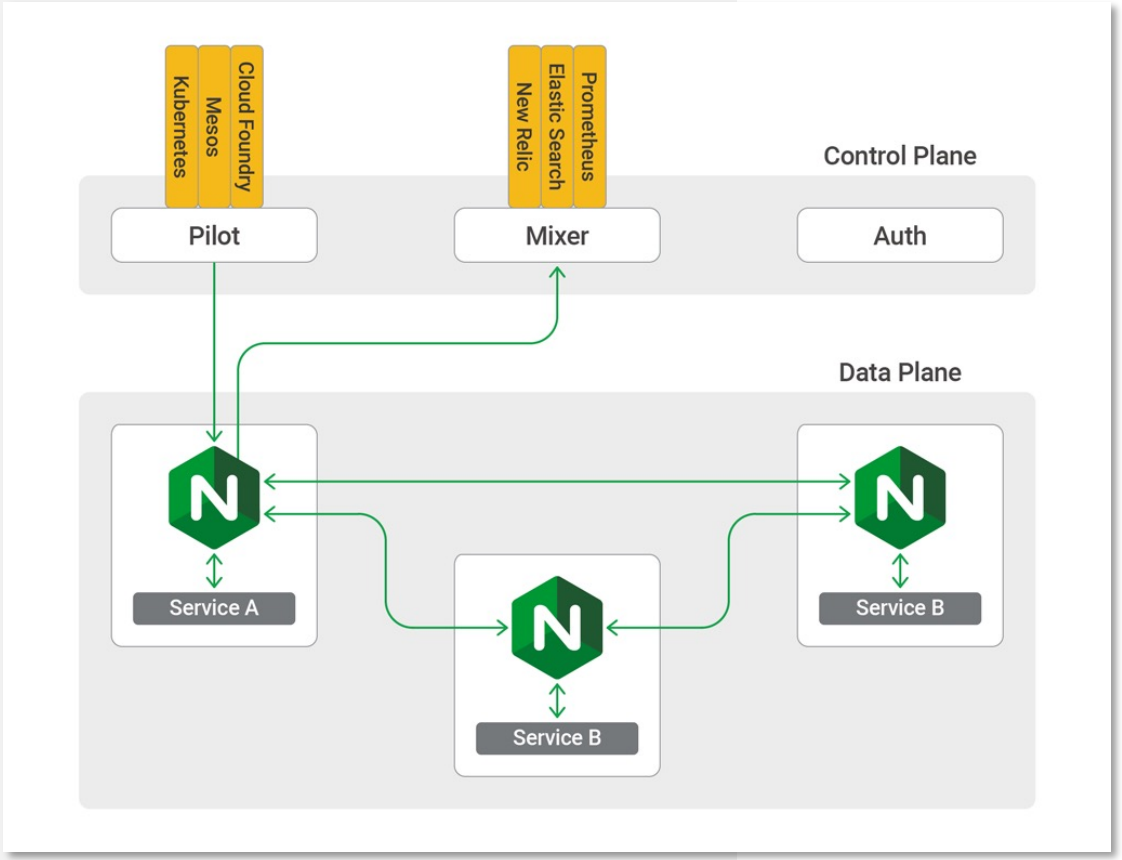
Adopting microservices methodologies comes with challenges as deployments scale and become more complex. Communication between the services is intricate, debugging problems can be harder, and more services imply more resources to manage.

Quelle: NGINX

NGINX FEATURES

Was sagen die Anbieter?

- **Sidecar proxy.** A *sidecar proxy* is a proxy instance that's dedicated to a specific service instance. It communicates with other sidecar proxies and is managed by the orchestration framework.
- **Service discovery.** When an instance needs to interact with a different service, it needs to find – discover – a healthy, available instance of the other service. The container management framework keeps a list of instances that are ready to receive requests.
- **Load balancing.** In a service mesh, load balancing works from the bottom up. The list of available instances maintained by the service mesh is stack-ranked to put the least busy instances – that's the load balancing part – at the top.
- **Encryption**. The service mesh can encrypt and decrypt requests and responses, removing that burden from each of the services. The service mesh can also improve performance by prioritizing the reuse of existing, persistent connections, reducing the need for the computationally expensive creation of new ones.
- **Authentication and authorization.** The service mesh can authorize and authenticate requests made from both outside and within the app, sending only validated requests to service instances.
- **Support for the circuit breaker pattern.** The service mesh can isolate unhealthy instances, then gradually brings them back into the healthy instance pool if warranted.

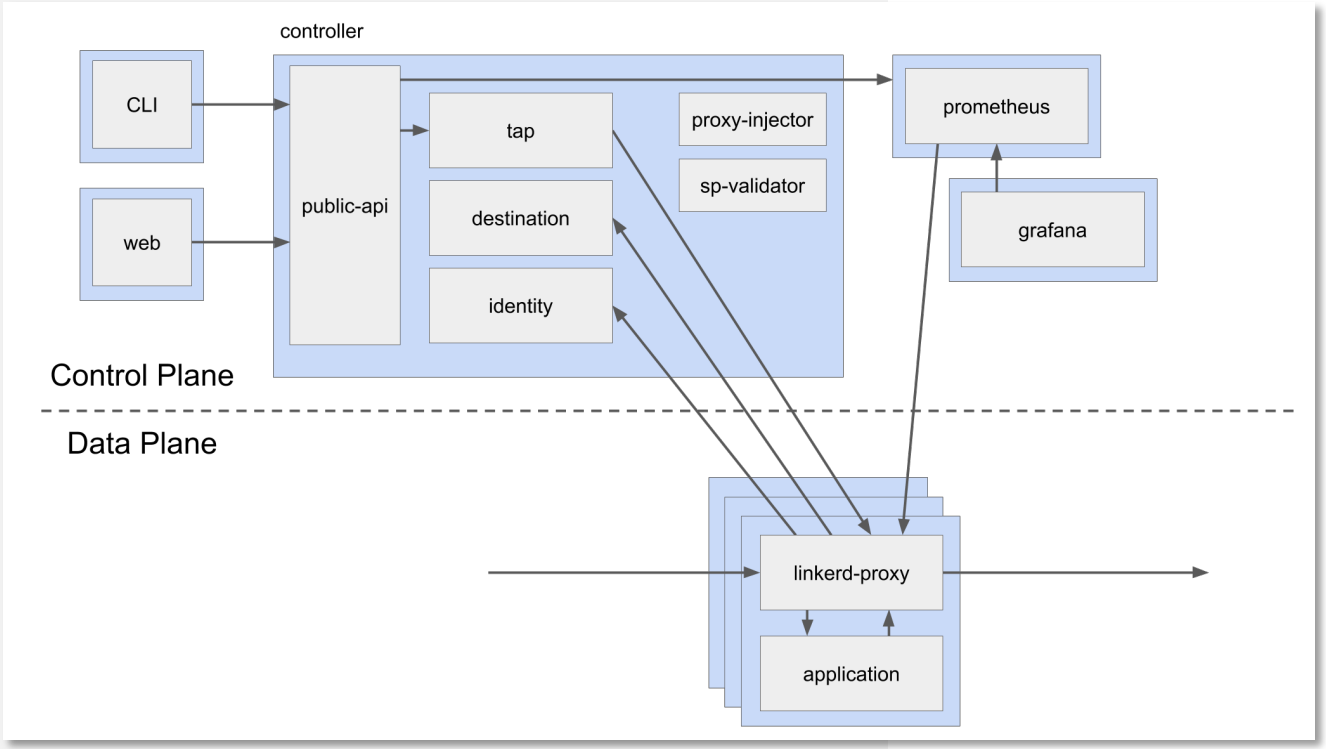


Quelle: NGINX

LINKERD

Was sagen die Anbieter?

- A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services.
- In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.
- Reliably delivering requests in a cloud native application can be incredibly complex.
- A service mesh manages this complexity with a wide array of powerful techniques: circuit-breaking, latency-aware load balancing, eventually consistent ("advisory") service discovery, retries, and deadlines.



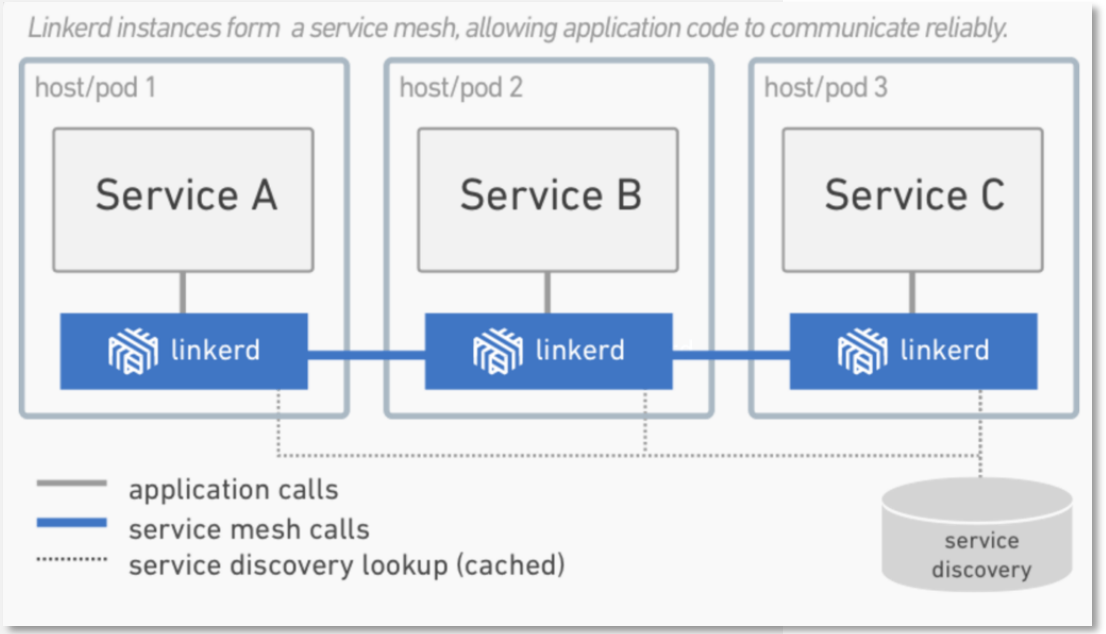
LINKERD

Quelle: Linkerd

LINKERD FEATURES

Was sagen die Anbieter?

- Linkerd applies dynamic routing rules to determine which service the requester intended. Should the request be routed to a service in production or in staging? To a service in a local datacenter or one in the cloud? To the most recent version of a service that's being tested or to an older one that's been vetted in production?
- Having found the correct destination, Linkerd retrieves the corresponding pool of instances from the relevant service discovery endpoint.
- Linkerd chooses the instance most likely to return a fast response based on a variety of factors, including its observed latency for recent requests.
- If the instance is down, unresponsive, or fails to process the request, Linkerd retries the request on another instance.
- If an instance is consistently returning errors, Linkerd evicts it from the load balancing pool, to be periodically retried later.
- If the deadline for the request has elapsed, Linkerd proactively fails the request (circuit-breaking) rather than adding load with further retries.
- Linkerd captures every aspect of the above behavior in the form of metrics and distributed tracing, which are emitted to a centralized metrics system.



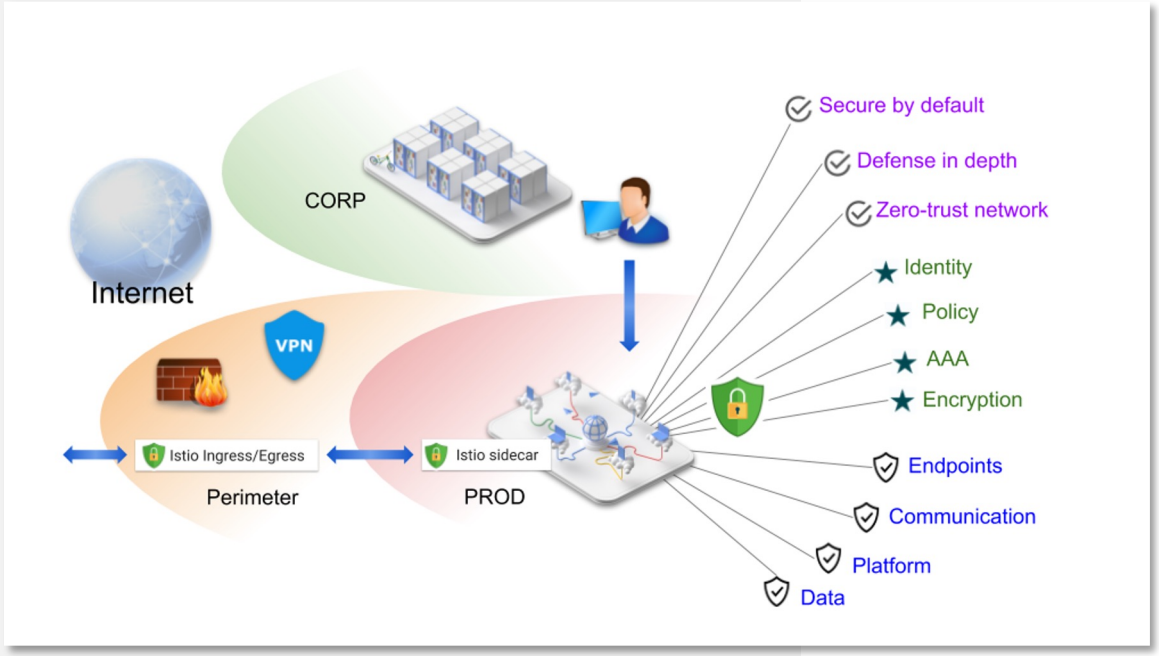
Quelle: Linkerd

ISTIO

Was sagen die Anbieter?

The term service mesh is used to describe the network of microservices that make up such applications and the interactions between them. As a service mesh grows, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary releases, rate limiting, access control, and end-to-end authentication.
Istio makes it easy to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, without any changes in service code. You can configure and manage Istio using its control plane functionality, which includes:

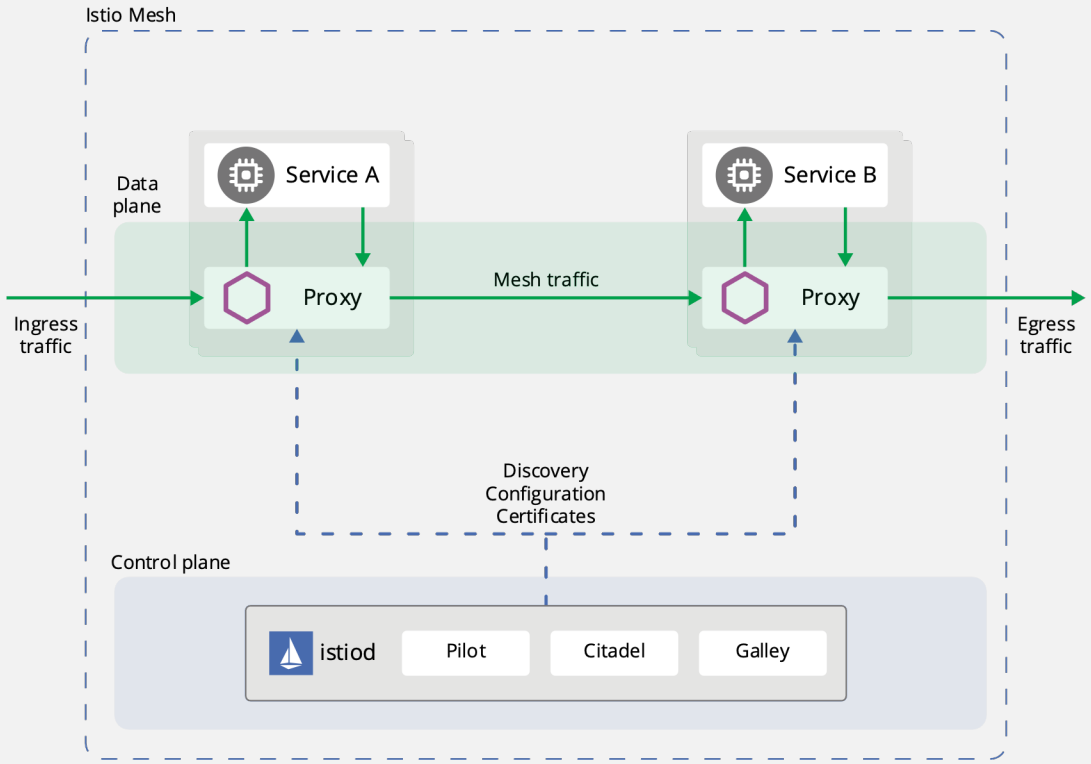
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.



Quelle: Istio

ISTIO FEATURES

Was sagen die Anbieter?



Traffic management:

- Istio's easy rules configuration and traffic routing lets you control the flow of traffic and API calls between services.
- Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries.
- It is possible to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits.

Security:

- Istio provides the underlying secure communication channel, and manages authentication, authorization, and encryption of service communication at scale.
- Service communications are secured by default, letting you enforce policies consistently across diverse protocols and runtimes.

Observability:

- Istio's robust tracing, monitoring, and logging give you deep insights into your service mesh deployment.
- Istio's Mixer component is responsible for policy controls and telemetry collection.
- It provides backend abstraction and intermediation, insulating the rest of Istio from the implementation details of individual infrastructure backends, and giving operators fine-grained control over all interactions between the mesh and infrastructure backends.

Quelle: Istio

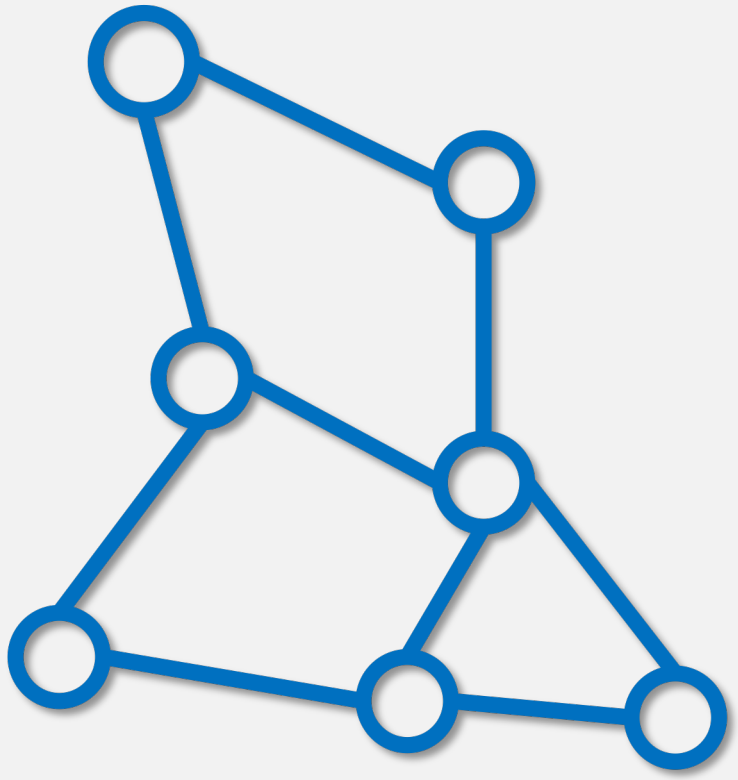
INHALTE

Warum Service Meshs?
- Was findet sich in der Literatur?
- Was sagen die Anbieter?

Service Meshs
- Traffic Management
- Resilienz
- Sicherheit
- Beobachtbarkeit

Fallstudie: Istio
- Traffic Management
- Security
- Observability

Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



CLOUD NATIVE COMPUTING

SERVICE MESH

Ein Definitionsvorschlag

CLOUD NATIVE COMPUTING

Grund für Service Meshs:

- Die Entwicklung einzelner Microservices ist leicht.
- Der Betrieb einer Microservice Architektur ist jedoch nicht trivial.

A Service Mesh creates a network abstraction to simplify the managing of containerized applications and makes it easier to dynamically route, monitor and secure microservice-based applications.

Technischer Ansatz:

- Kanalisierung von Netzwerkinteraktionen zwischen Microservices durch Proxies.
- Proxies können dann Netzwerkverkehr verschlüsseln, analysieren, überwachen und routen.

Ein Service Mesh erstellt eine Netzwerkabstraktion, um die Verwaltung von Containeranwendungen mittels dynamischen Routing, Monitoring und Sichern von Microservice-Anwendungen zu vereinfachen.

SERVICE MESHS

Warum? Wieso? Weshalb?

Service Meshs erzeugen eine Netzwerkabstraktion für containerbasierte Applikationen und Dienste, um deren Management zu vereinfachen und Aspekte wie *Traffic Management*, *Zuverlässigkeit*, *Beobachtbarkeit und Sicherheit* innerhalb von Microservice-basierten Architekturen zu vereinfachen und querschnittlich zu isolieren.

Wozu eigentlich Service Meshs?

*Die Entwicklung und Betrieb eines einzelnen Microservice ist zwar einfach (→ **GAINS**), der Betrieb komplexer Service-of-Service Systeme bringt aber im Gegenzug diverse weitere Herausforderungen mit sich (→ **PAINS**). Service Meshs lösen diese Herausforderungen außerhalb einer Applikation in einer explizit dafür vorgesehenen Schicht.*

GAINS of Microservices

- Einfache Entwicklung (einzelner) Microservices
- Cloud-native und DevOps by design
- Unabhängiges Deployment (kein Downtime durch monolithische Rollouts)
- Einfache Integration in Continuous Integration und Continuous Deployment Pipelines

PAINS of Microservices

- Inhärente Komplexität
- Geschäftslogik ist über voneinander unabhängige Microservices verteilt
- Sicherheit (Zugriffskontrolle und vergrößerte Attack Surfaces)
- Verteiltes Logging, Traceability

CLOUD NATIVE COMPUTING

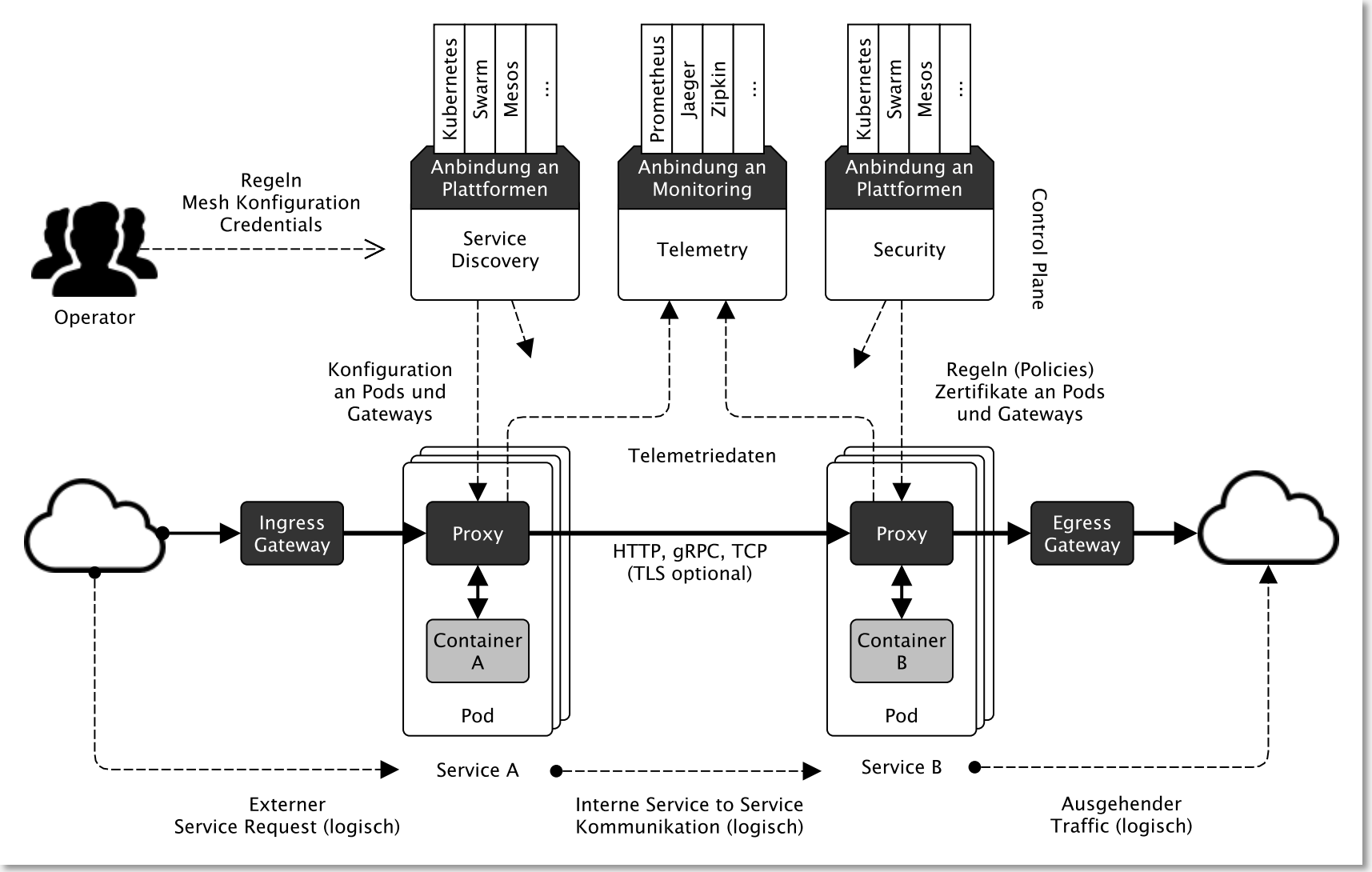
Sie erinnern sich:

Komplexität verschwindet nie!!!

Man kann Komplexität aber verschieben, verstecken, kapseln, isolieren, etc. um sie besser handhaben zu können.

SERVICE MESH

Architektur

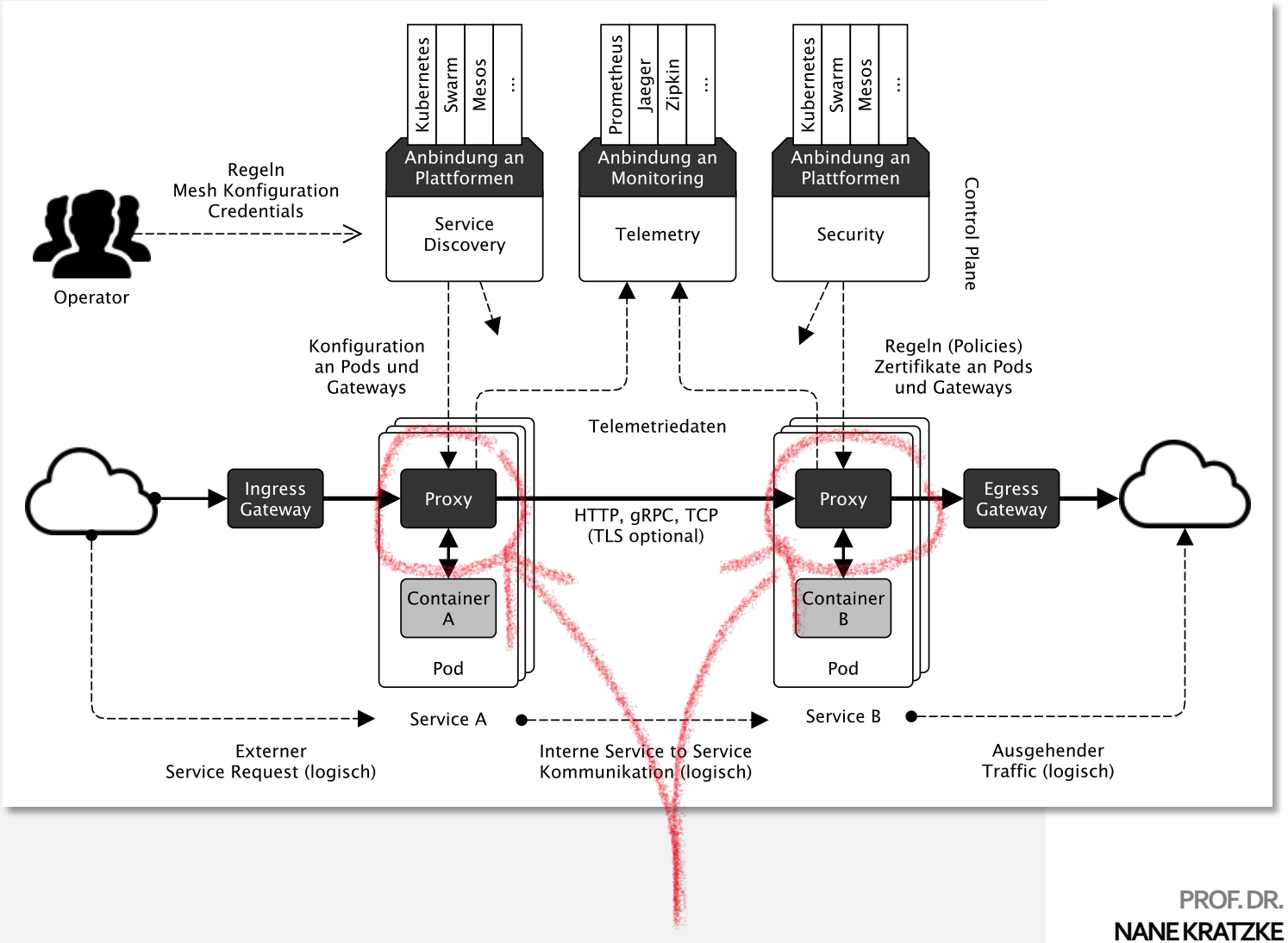


*Ein Service Mesh besteht aus einer Kontrollebene (**Control plane**), die **Proxies** (Sidecar Pattern) steuert. Die Proxies sind Serviceinstanzen zugeordnet (Sidecar Pattern) und spannen eine (ggf. verschlüsselte) Datenschicht (**Data plane**) für Services auf.*

SERVICE MESH

Sidecar-Pattern

Anwendungen und Dienste benötigen häufig zusammengehörige Funktionen, wie z. B. Überwachung, Protokollierung, Konfiguration, etc.. Solche kohärenten Peripherieaufgaben können als separate Komponenten oder Dienste implementiert werden. Werden diese Aufgaben in einem eigenen Prozess oder Container bereitgestellt, um Plattformdienste sprachübergreifend bereitzustellen, spricht man von einem Sidecar. Proxies in Service Meshs werden häufig als Sidecars in Containern bereitgestellt, die jeder Serviceinstanz zur Seite gestellt werden.



SERVICE MESH

Gängige Features

Traffic Management	Resiliency	Security	Observability
Request Routing	Timeouts	mTLS	Metrics
Load Balancing	Circuit Breaker	Role-Based Access Control	Logs
Traffic Shifting	Health Checks	Workload Identity	Traces
Traffic Mirroring	Retries	Authentication Policies	
Service Discovery	Rate Limiting	CORS Handling	
Ingress, Egress	Delay & Fault Injection	TLS Termination	
API Specification	Connection Pooling		
Multicluster Mesh			

SERVICE MESH

Wesentliche Komponenten und deren Verantwortlichkeiten in einem Service Mesh

Proxy

- Jeder Dienst kommuniziert über einen dedizierten Proxy (Sidecar Pattern).
- Die gesamte Kommunikation im Mesh erfolgt über diese Proxies.
- Proxies können so das Kommunikationsverhalten anreichern und z.B. Tracing- und Monitoring-Daten sammeln.
- Proxies lassen sich bequem von Orchestrierungs-Frameworks verwalten, um so z.B. bei jedem Deployment automatisch platziert zu werden.

Control plane

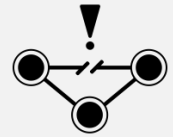
- Verwaltung von Routen und Regeln
- Konfiguration von Timeouts, Retries, etc.
- Service Discovery
- Access Control
- Policy Control
- Telemetriedaten

Data plane

- Eingehende (ingress) und ausgehende (egress) Kommunikation
- Service-to-Service Kommunikation
- Kommunikation über Proxies
- Transparent für verwaltete Dienste

SERVICE MESH

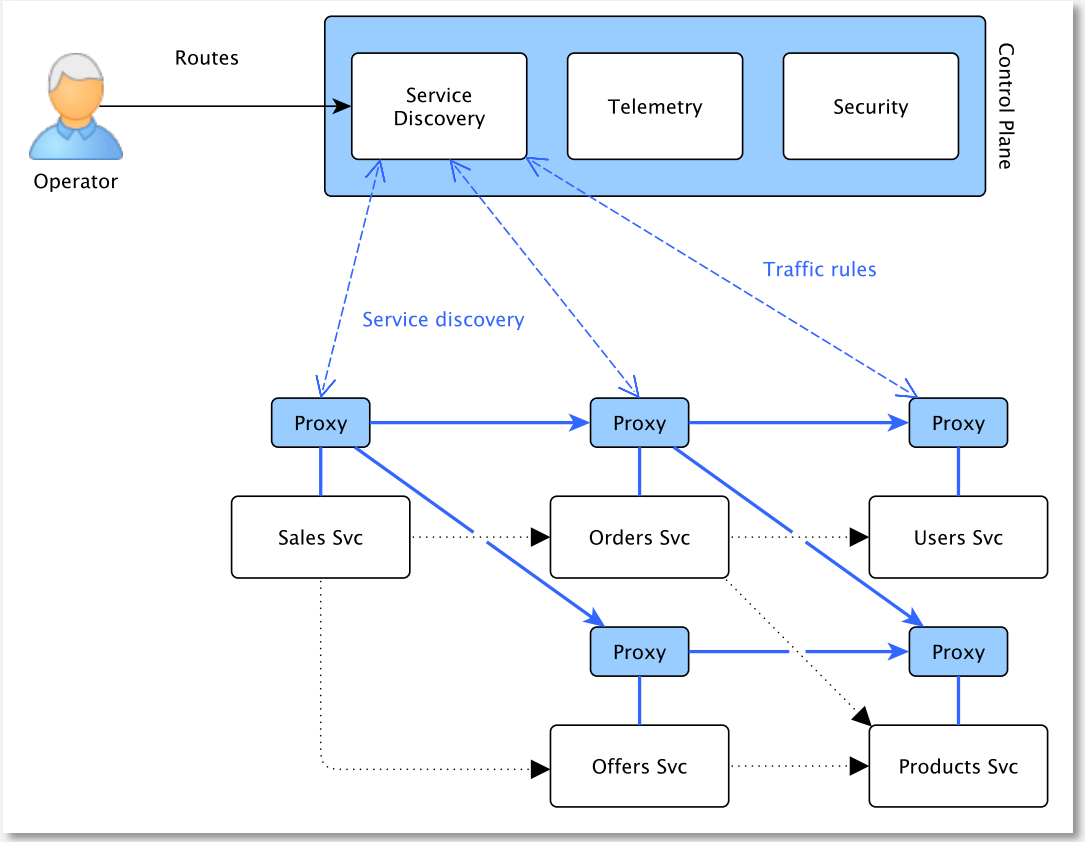
Zuverlässigkeitsmechanismen



CLOUD NATIVE COMPUTING

Service Meshs stellen vielfältige Zuverlässigkeitsmechanismen bereit, die nicht im Sevicecode realisiert werden müssen:

- Last- und Zustands-adaptives Service Discovery
- mit einem selbstheilendem System Design
- ermöglicht intelligentes (adaptives) Routing,
- A/B tests, Canary Deployments, schrittweises Rollout, etc.
- und Traffic-Splitting/Steering

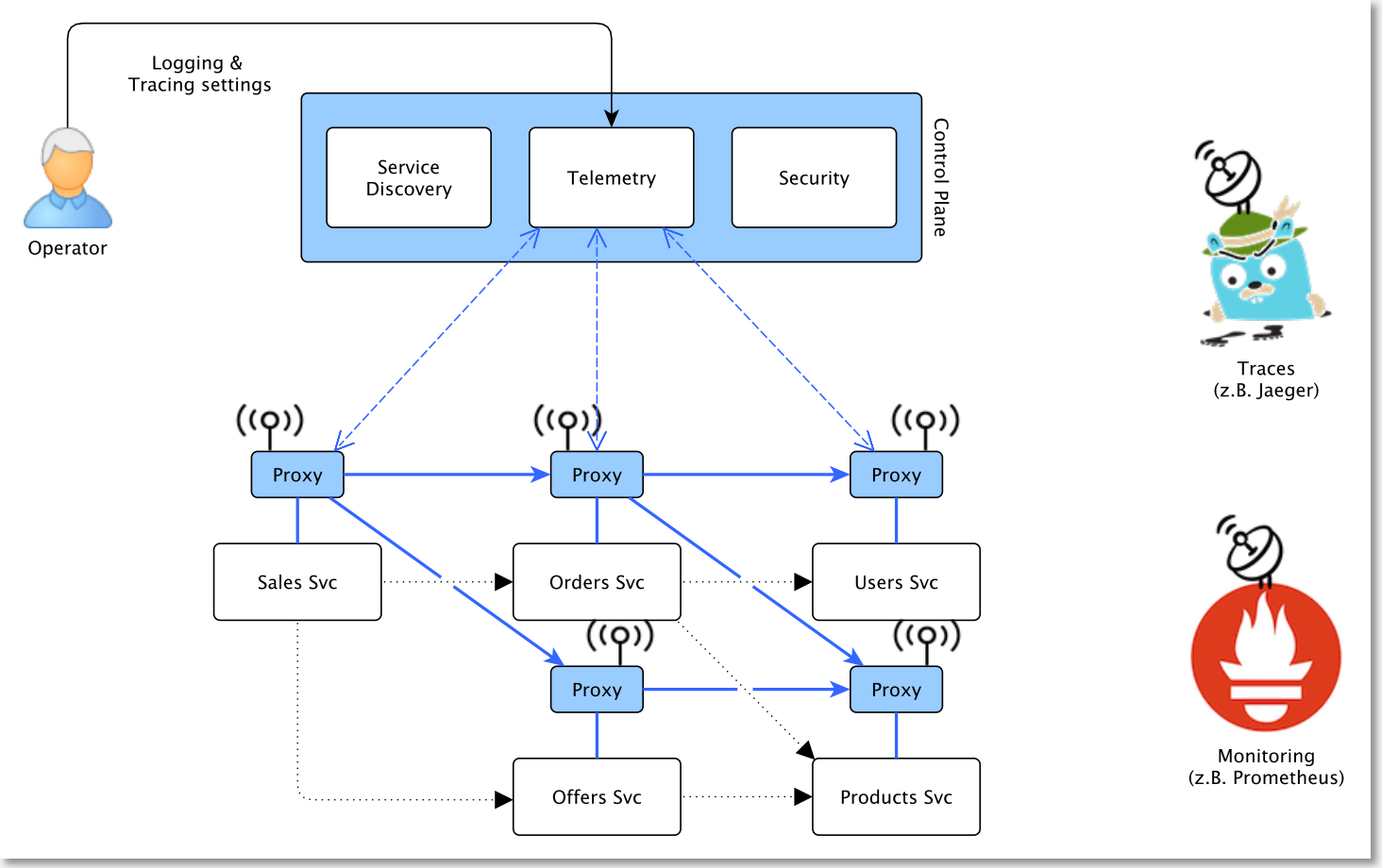


Resilienz-Patterns

- Circuit-breaker
- Timeouts + Retries
- Recovery
- Latenz-basiertes Load Balancing
- Health-aware Service Discovery

SERVICE MESH

Telemetrie und Beobachtbarkeit



Telemetriedaten werden transparent durch die Proxies des Service Meshs gesammelt um Latenz-, Zustands- und Volumen-basiertes Load Balancing und Traffic-Management zu ermöglichen. Dies ermöglicht Resilienz und Beobachtbarkeit.

Merke:
Telemetrie umfasst

- *Tracing*
- *Monitoring*
- *Logging*

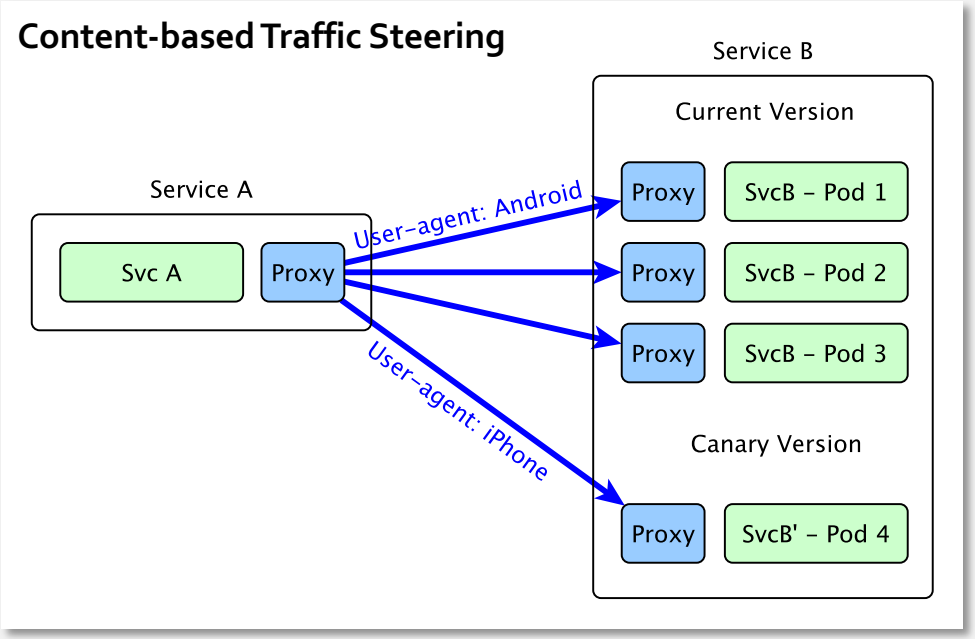
SERVICE MESH

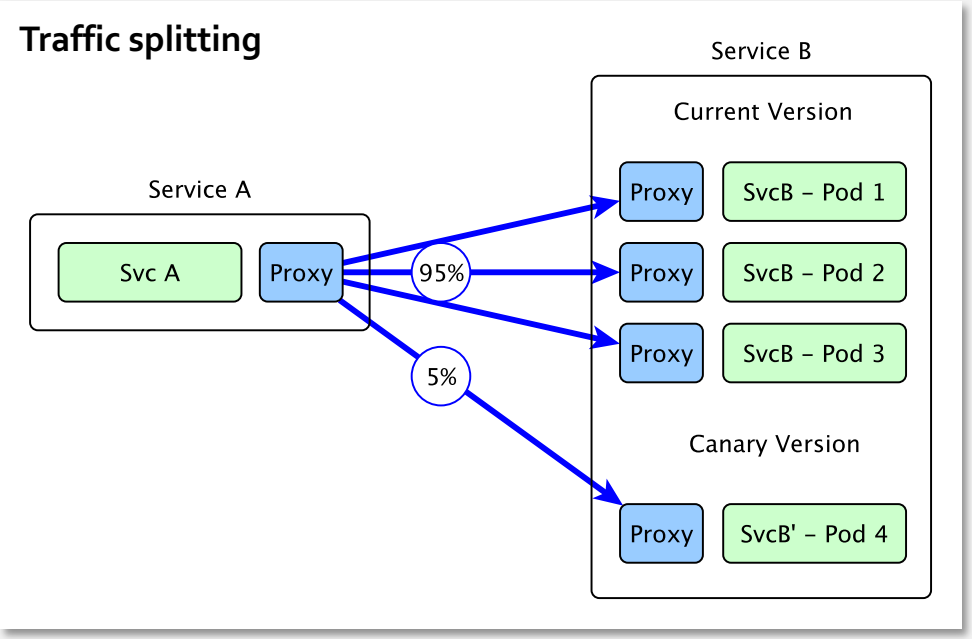
Traffic Management

CLOUD NATIVE COMPUTING

Service Meshs ermöglichen u.a. folgende Formen des Traffic Managements

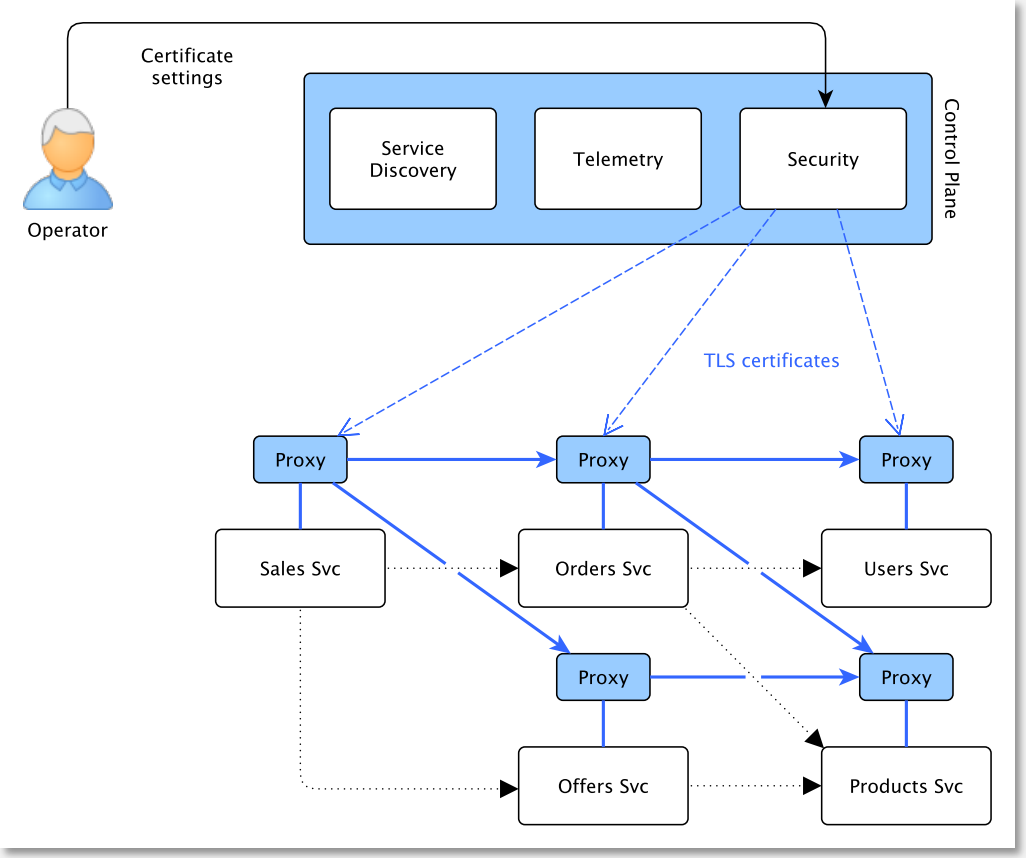
- Dynamisches Regel-basiertes Routing
- Regel- und Volumen-basiertes Traffic Splitting
- Rate Limit
- Quotas





SERVICE MESH

Sicherheit (Security)



Sicherheitsmechanismen von Service Meshs umfassen oft

- Authentifizierung
- Authorisation
- Verschlüsselung der Service Communication
- TLS Endpunkt Termination
- Transparentes Zertifikat Handling (TLS)

Diese unterstützen dabei folgende Aspekte einer Service Architektur

- Secure by Default Design
- Service-to-Service und End-User Authentifizierung
- Identitäts- und Credential Management
- Transparente Verschlüsselung der Kommunikation im Service-Mesh
- Setzt Kommunikationsregeln flexibler auf Basis von Serviceidentitäten (logisch) anstatt Network Controls (physisch) durch.

SERVICE MESH

Beispielprodukte

Kuma

Traefik

NGINX Service Mesh

Istio

Linkerd

Consul

Auf Layer5.io finden Sie eine Übersicht von vielen weiteren Service Meshs.

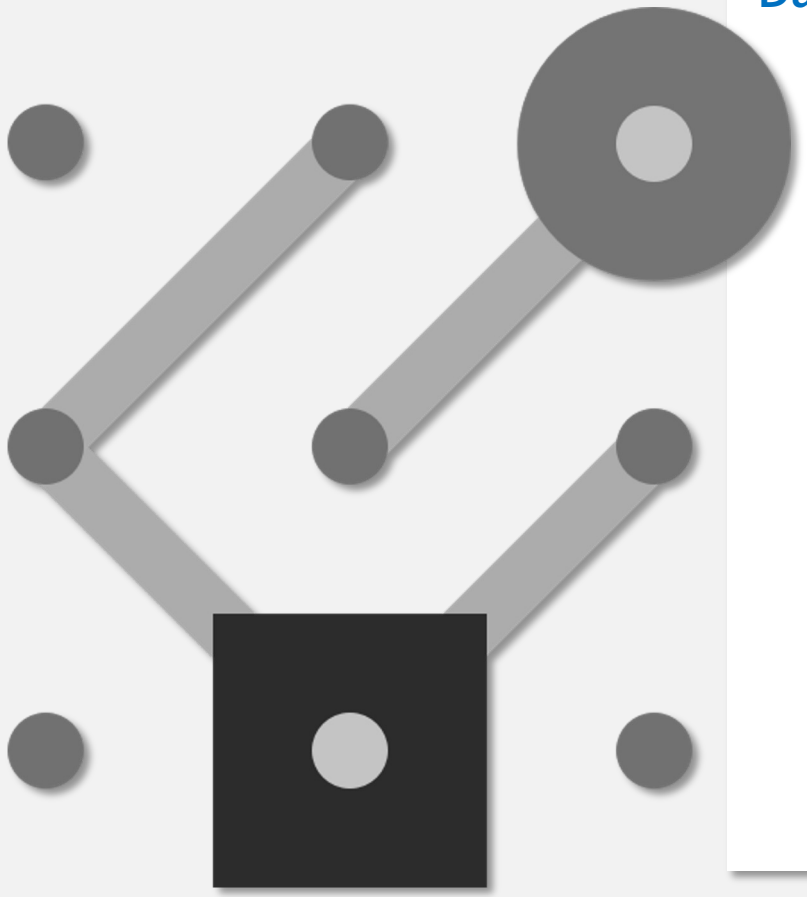
Es steht zu erwarten, dass sich dieser Bereich in den kommenden Jahren konsolidieren wird und stärker in Container Plattformen wie K8s integriert werden wird.

Mit dem SMI Standard (Service Mesh Interface) existiert ein erster Standardentwurf in diesem Bereich.

SERVICE MESH INTERFACE

Der SMI Standard

Das Service Mesh Interface (SMI) ist eine Spezifikation für Service Meshes, die auf Kubernetes ausgeführt werden. Es definiert einen gemeinsamen Standard, der von einer Vielzahl von Anbietern implementiert werden kann. Dies ermöglicht sowohl eine Standardisierung für Endbenutzer als auch Innovationen durch Anbieter von Service Mesh-Technologie. SMI ermöglicht Flexibilität und Interoperabilität und deckt die gängigsten Service-Mesh-Funktionen ab.



Das SMI ist:

Eine Standardschnittstelle für Kubernetes Service-Meshes und den grundlegenden Funktionsumfang von Service-Meshes definiert:

- ***Traffic Policy*** - Anwenden von Richtlinien wie Identität und Transportverschlüsselung zwischen Diensten
- ***Traffic Telemetry*** - Erfassen wichtiger Messdaten wie Fehlerrate und Latenz zwischen Diensten
- ***Traffic Management*** - Verschieben von Datenverkehr zwischen verschiedenen Diensten

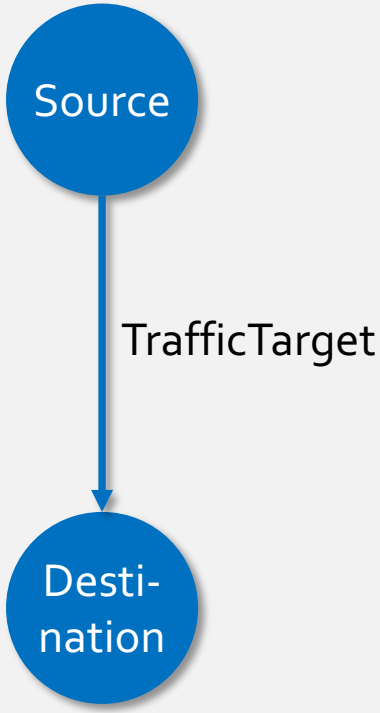
Quelle: https://smi-spec.io

DER SMI STANDARD

Traffic Access Control (I)

Mittels SMI können Zugriffsrichtlinien für Anwendungen definiert werden. Die Zugriffskontrolle ist additiv. Der gesamte Datenverkehr wird standardmäßig verweigert, es sei denn es werden Zugriffsregeln explizit ergänzt.

Ein TrafficTarget ordnet eine Reihe von Traffic Definitions (Regeln) einer Dienstidentität zu, die einer Gruppe von Pods zugeordnet ist. Der Zugriff wird über referenzierte TrafficSpecs und eine Liste von Quelldienstidentitäten gesteuert. Jeder Pod, der versucht, eine Verbindung herzustellen und nicht in der definierten Liste der Quellen enthalten ist, wird abgelehnt.
Der Zugriff wird basierend auf der Dienstidentität (Kubernetes-Serviceaccounts) gesteuert.



Regeln sind Traffic Definitions, die definieren, wie zulässiger Verkehr für bestimmte Protokolle aussehen darf. Ein gültiges TrafficTarget muss ein Ziel, mindestens eine Regel und mindestens eine Quelle angeben.

Quelle: https://smi-spec.io

DER SMI STANDARD

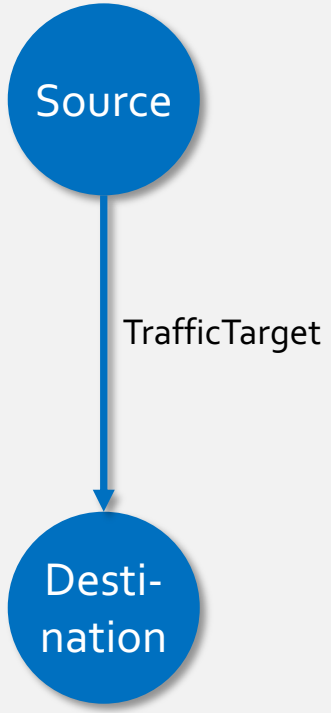
Traffic Access Control (II)

```
kind: TCPRoute
metadata:
  name: the-routes
spec:
  matches:
    ports: [8080]
---
kind: HTTPRouteGroup
metadata:
  name: the-routes
spec:
  matches:
  - name: metrics
    pathRegex: "/metrics"
    methods: ["GET"]
  - name: everything
    pathRegex: ".*"
    methods: ["*"]
```

Routes

```
kind: TrafficTarget
metadata:
  name: path-specific
  namespace: default
spec:
  destination:
    kind: ServiceAccount
    name: service-a
    namespace: default
  rules:
  - kind: TCPRoute
    name: the-routes
  - kind: HTTPRouteGroup
    name: the-routes
    matches: ["metrics"]
  sources:
  - kind: ServiceAccount
    name: prometheus
    namespace: default
```

Zugriffsregel



Dieses Beispiel zeigt einen häufigen Anwendungsfall, den Zugriff auf Metriken so einzuschränken, dass diese nur von Prometheus abgefragt werden dürfen.

Quelle: https://smi-spec.io

DER SMI STANDARD

Traffic Metrics

Diese Teil-Spezifikation des SMI-Standards beschreibt einen gemeinsamen Integrationspunkt für Tools, die Metriken des HTTP-Verkehrs auswerten (bspw. HPA-Skalierung).

Metriken sind dabei immer einer Ressource zugeordnet. Dies können Pods sowie übergeordnete Konzepte wie Namespaces, Deployments oder Services sein. Metriken sind dabei entweder der Kubernetes-Ressource zugeordnet, die den gemessenen Datenverkehr generiert (request) oder bedient (response).

```
kind: TrafficMetrics
resource:
  name: foo-775b9cbd88-ntxsl
  namespace: foobar
  kind: Pod
edge:
  direction: to
  side: client
  resource:
   name: baz-577db7d977-lsk2q
   namespace: foobar
   kind: Pod
timestamp: 2019-04-08T22:25:55Z
window: 30s
metrics:
  - name: p99_response_latency
    unit: seconds
    value: 10m
  - name: p90_response_latency
    unit: seconds
    value: 10m
  - name: p50_response_latency
    unit: seconds
    value: 10m
  - name: success_count
    value: 100
  - name: failure_count
    value: 100
```

Derartige Metriken werden im laufenden Betrieb aus den Sidecar-Proxies erhoben.

Quelle: https://smi-spec.io

DER SMI STANDARD

Traffic Specs

Diese Teil-Spezifikation beschreibt eine Reihe von Ressourcen, mit denen Datenverkehr protokollspezifisch selektiert werden kann. Es wird zusammen mit der Zugriffskontrolle und anderen Richtlinien verwendet, um zu definieren, was mit bestimmten Arten von Verkehr geschehen soll, wenn dieser durch das Netz fließt.

Jede Ressource in dieser Spezifikation soll 1: 1 mit einem bestimmten Protokoll übereinstimmen. Auf diese Weise können Benutzer den Datenverkehr protokollspezifisch definieren.

```
kind: HTTPRouteGroup
metadata:
  name: the-routes
  spec:
  matches:
  - name: metrics
    pathRegex: "/metrics"
    methods: ["GET"]
- name: health
    pathRegex: "/ping"
    methods: ["*"]
```

Beispiel-Ressource um den HTTP-Verkehr zu beschreiben. Es werden die Routen aufgelistet, die von einer Anwendung bedient werden können.

Derartige Specs werden von anderen Regeln (z.B. Traffic Access) genutzt und machen alleine vorkommend keinen Sinn.

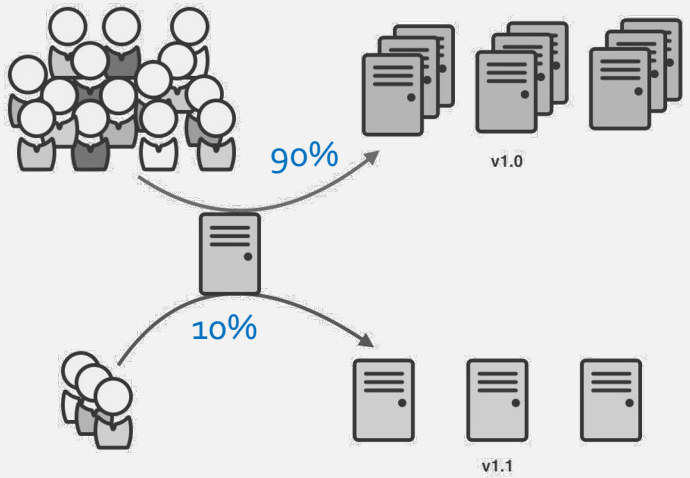
Quelle: https://smi-spec.io

DER SMI STANDARD

Traffic Split (Canary Releases)

Diese Teil-Spezifikation definiert die TrafficSplit-Ressource, mit der der prozentuale Anteil des Datenverkehrs zwischen verschiedenen Services schrittweise gesteuert werden kann. Es wird von Clients wie Ingress-Controllern oder Service-Mesh-Sidecars verwendet, um den ausgehenden Datenverkehr auf verschiedene Ziele aufzuteilen.

Integrationen können diese Ressource verwenden, um Canary Versionen für neue Softwareversionen zu orchestrieren. Die Ressource selbst ist keine vollständige Lösung, da es eine Art Controller geben muss, der die zeitliche Verschiebung des Datenverkehrs verwaltet. Die Gewichtung des Verkehrs zwischen verschiedenen Services kann auch in Szenarien jenseits von Canary Releases genutzt werden.




```
kind: TrafficSplit
metadata:
  name: canary
spec:
  # The root service that clients use to connect
  service: website
  # Services inside the namespace
  backends:
  - service: website-v1.0
    weight: 90
  - service: website-v1.1
    weight: 10
```

Beispiel-Ressource um den HTTP-Verkehr zwischen zwei Versionen eines Dienstes aufzuteilen. 90% des Traffics gehen an die Version v1.0. 10% des Traffics gehen an die Version v1.1 (Canary Release)

Quelle: https://smi-spec.io

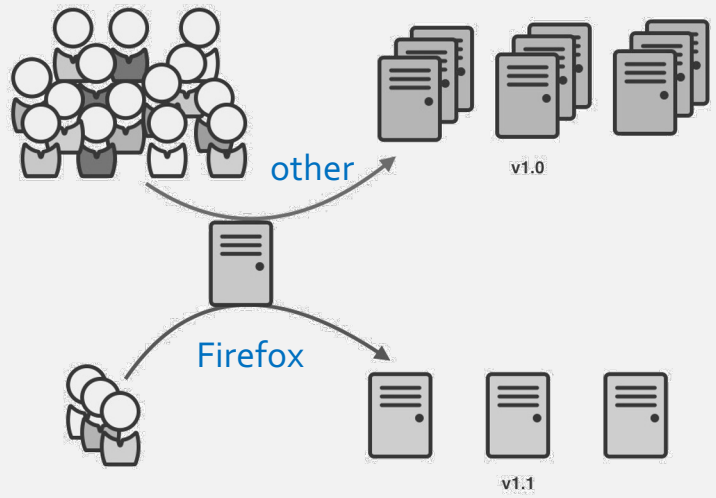
DER SMI STANDARD

Traffic Split (A/B Testszenarien)

Um A / B-Testszenarien zu berücksichtigen, kann ein Traffic Split anhand von HTTP-Headerfilter vorgenommen werden, um einen bestimmten Benutzeranteil an ein Backend weiterleiten, während alle anderen Benutzer (Kontrollgruppe) , die nicht zu diesem Segment gehören, an das Standarddienst-Backend weitergeleitet werden. Hierzu können HTTP-Header-Filter können mithilfe der HTTPRouteGroup-API definiert werden mittels derer Traffic Splits anhand von HTTP-Routen vornehmen zu können.

```
kind: HTTPRouteGroup
metadata:
  name: ab-test
matches:
- name: firefox-users
  headers:
  - user-agent: ".*Firefox.*"
```

Route um Nutzer anhand ihres Browser (user-agent) selektieren zu können, hier werden alle Firefox Nutzer selektiert.




```
kind: TrafficSplit
metadata:
  name: ab-test
spec:
  service: website
  matches:
  - kind: HTTPRouteGroup
    name: ab-test
  backends:
  - service: website-v1.0
    weight: 0
  - service: website-v1.1
    weight: 100
```

Hier werden alle Firefox Nutzer auf die v1.1 des Website Services gerouted.

Quelle: https://smi-spec.io

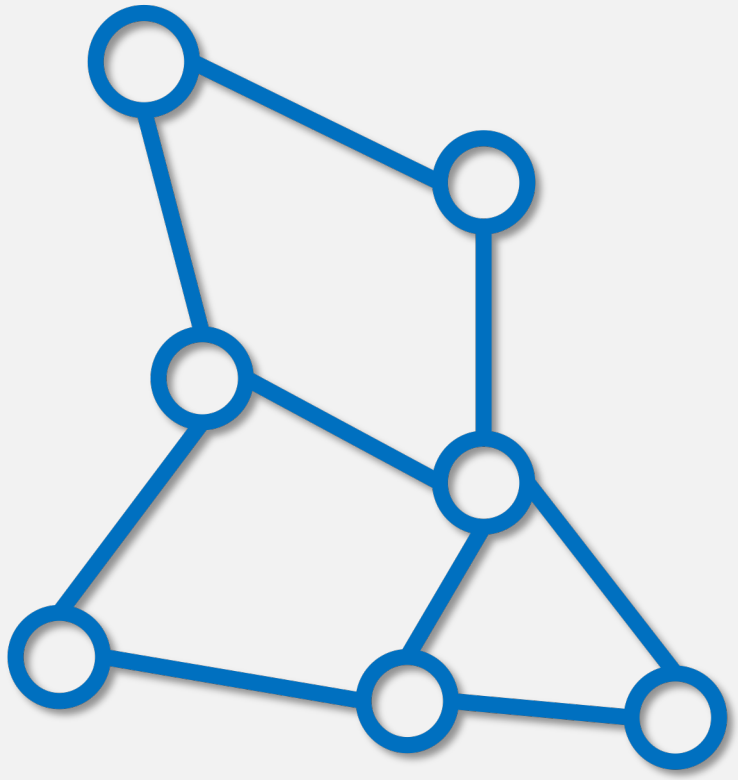
INHALTE

Warum Service Meshs?
- Was findet sich in der Literatur?
- Was sagen die Anbieter?

Service Meshs
- Traffic Management
- Resilienz
- Sicherheit
- Beobachtbarkeit

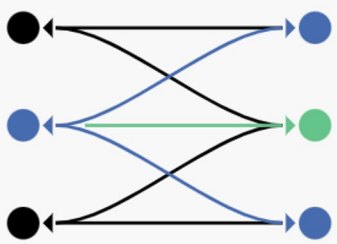
Fallstudie: Istio
- Traffic Management
- Security
- Observability

Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



ISTIO

Typvertreter für Service Meshs

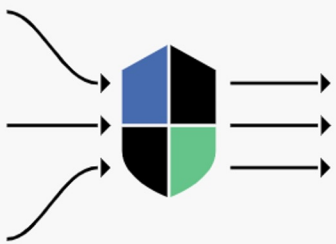


Connect

Intelligently control the flow of traffic and API calls between services, conduct a range of tests, and upgrade gradually with red/black deployments.

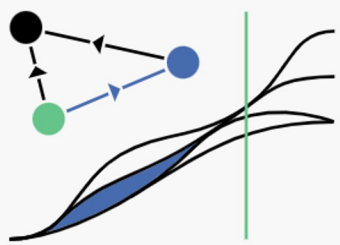
Secure

Automatically secure your services through managed authentication, authorization, and encryption of communication between services.



Control

Apply policies and ensure that they're enforced, and that resources are fairly distributed among consumers.



Observe

See what's happening with rich automatic tracing, monitoring, and logging of all your services.

Quelle: https://istio.io

CLOUD NATIVE COMPUTING

Merke:

Die Kernfunktionalitäten eines Service Mesh bestehen darin, in bestehende Service-of-Service Systeme,

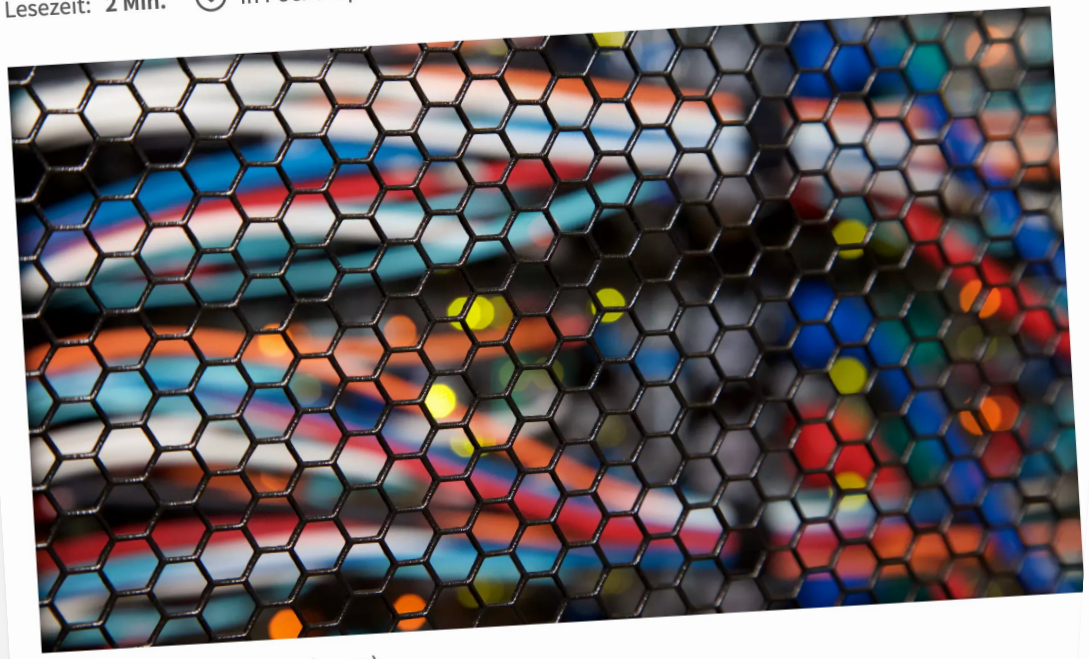
- *Traffic Management-,*
- *Security-,*
- *und Observability-*

Funktionalitäten zu injizieren, ohne dafür bestehenden Service Code anfassen zu müssen.

Google übergibt Istio an die Cloud Native Computing Foundation

Das Service Mesh Istio schlüpft als eines der letzten großen von Google initiierten Container-Projekte unter den Schirm der CNCF.

Lesezeit: 2 Min. In Pocket speichern



(Bild: lensmen/Shutterstock.com)

26.04.2022 11:13 Uhr | iX Magazin

Von Ulrich Wolf

Google Cloud

Blog What's New Product News Solutions & Technologies Topics CIOs & IT leaders Contact Sales Get started for free

OPEN SOURCE

The next step for Istio and cloud-native open source

Chen Goldberg
VP of Engineering
April 25, 2022

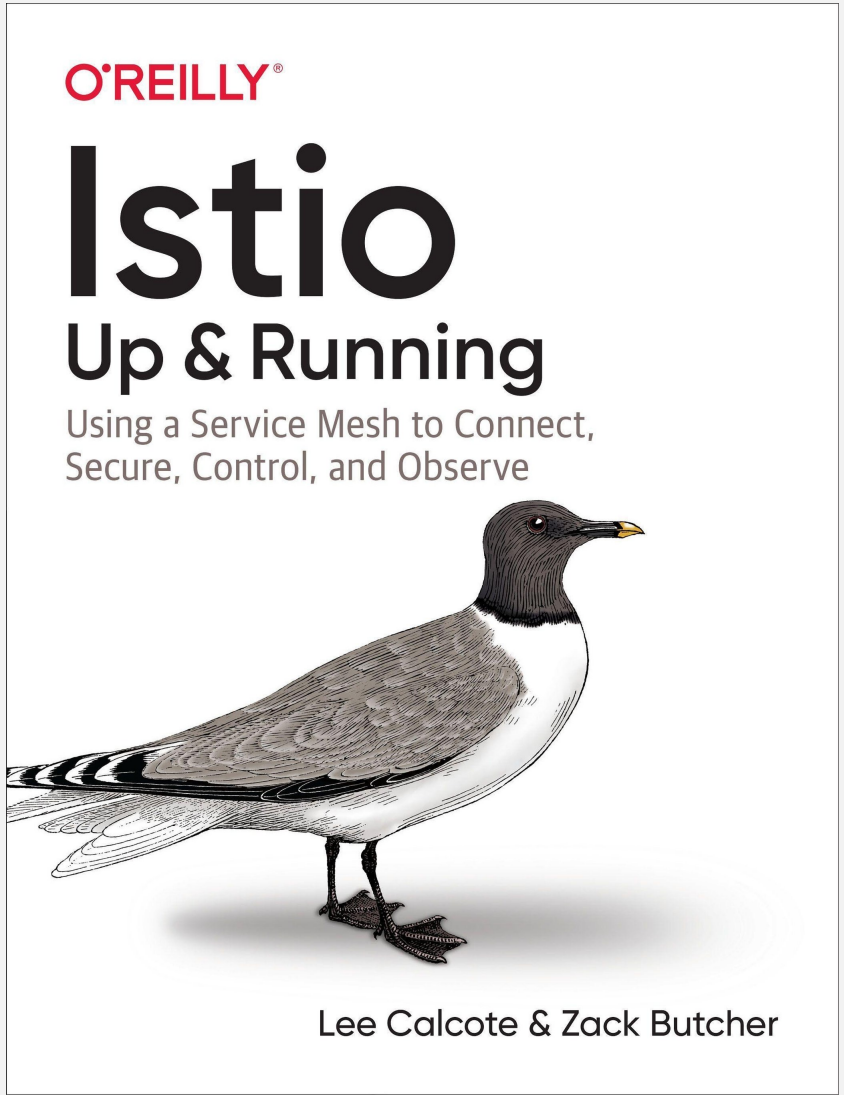
Today we are excited to announce that Google and the Istio Steering Committee have submitted the Istio project for consideration as an incubating project within the Cloud Native Computing Foundation (CNCF). This is a significant milestone for Istio and its community, and we are thrilled to reach this next step in the evolution of the project.

Google and Istio

Google originated the Istio project, which alongside Kubernetes and Knative, is a critical part of cloud-native infrastructure. The Istio project has found success and maturity in its current model — being adopted by hundreds of organizations and bringing in over 4,000 developers for IstioCon.

ISTIO

Typvertreter für Service Meshs

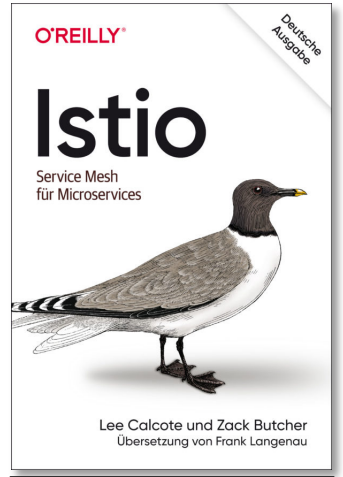


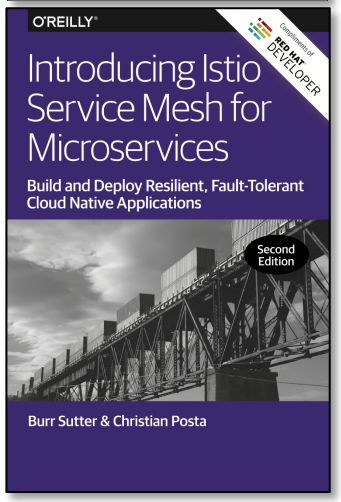
Wir betrachten am konkreten Beispiel von Istio den Funktionsumfang (und die damit einhergehende Komplexität) von typischen Open Source Service Meshs.

Die Funktionalitäten von Istio sind so umfangreich, dass hier nicht auf alle Einzelheiten eingegangen werden kann, sondern nur eine Auswahl von Fähigkeiten angerissen wird, die den grundsätzlichen Überlegungen des SMI Standards folgen und teilweise auch in Labs praktisch vertieft werden.

Istio geht aber weit über den SMI Standard hinaus. Dies hat Vor- und Nachteile. Wer sich daher intensiver in die Materie Istio (oder Service Meshs im Allgemeinen) einarbeiten möchte, sei daher auf die Literatur und die offizielle und umfangreiche Online-Dokumentation verwiesen.

https://istio.io/latest/docs



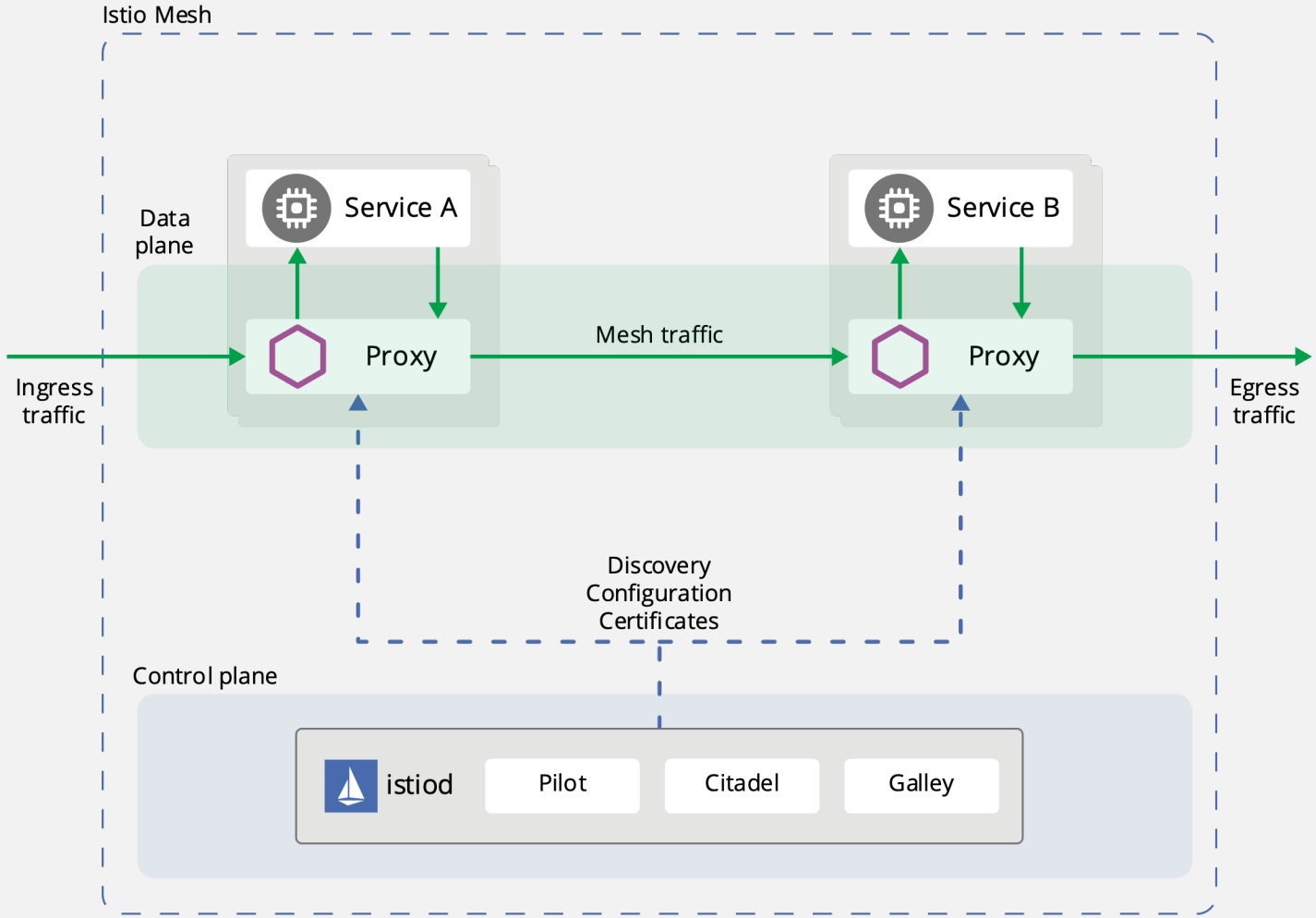


ISTIO

Typvertreter für Service Meshs

Istio befasst sich mit Aspekten, die beim Übergang monolithischer Anwendungen zu einer verteilten Microservice-Architektur entstehen.

Istio ermöglicht das Management und die Introspektion von Servicenetzen, indem Loadbalancing, Service-zu-Service-Authentifizierung, Überwachung und mehr in bestehende Deployments mittels Sidecar-Proxies injeziert werden können, ohne Codeänderungen am eigentlichen Servicecode vornehmen zu müssen.



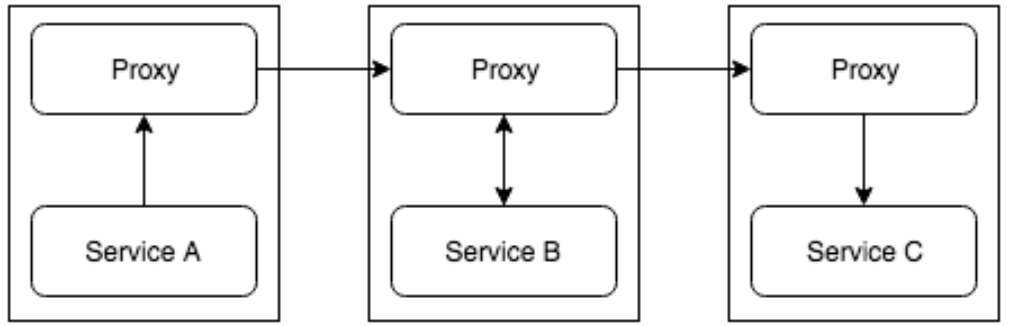
Quelle: istio.io

ISTIO

Nutzung des Sidecar Patterns

Istio (wie alle anderen Service Meshs auch) injiziert jedem Pod einen spezielles Sidecar-Proxy, das die gesamte Netzwerkkommunikation zwischen Microservices abfängt.

Auf diese Weise kann Istio den gesamten Live Traffic zwischen Microservices mitschneiden und analysieren und den Traffic über eine Control Plane steuern.

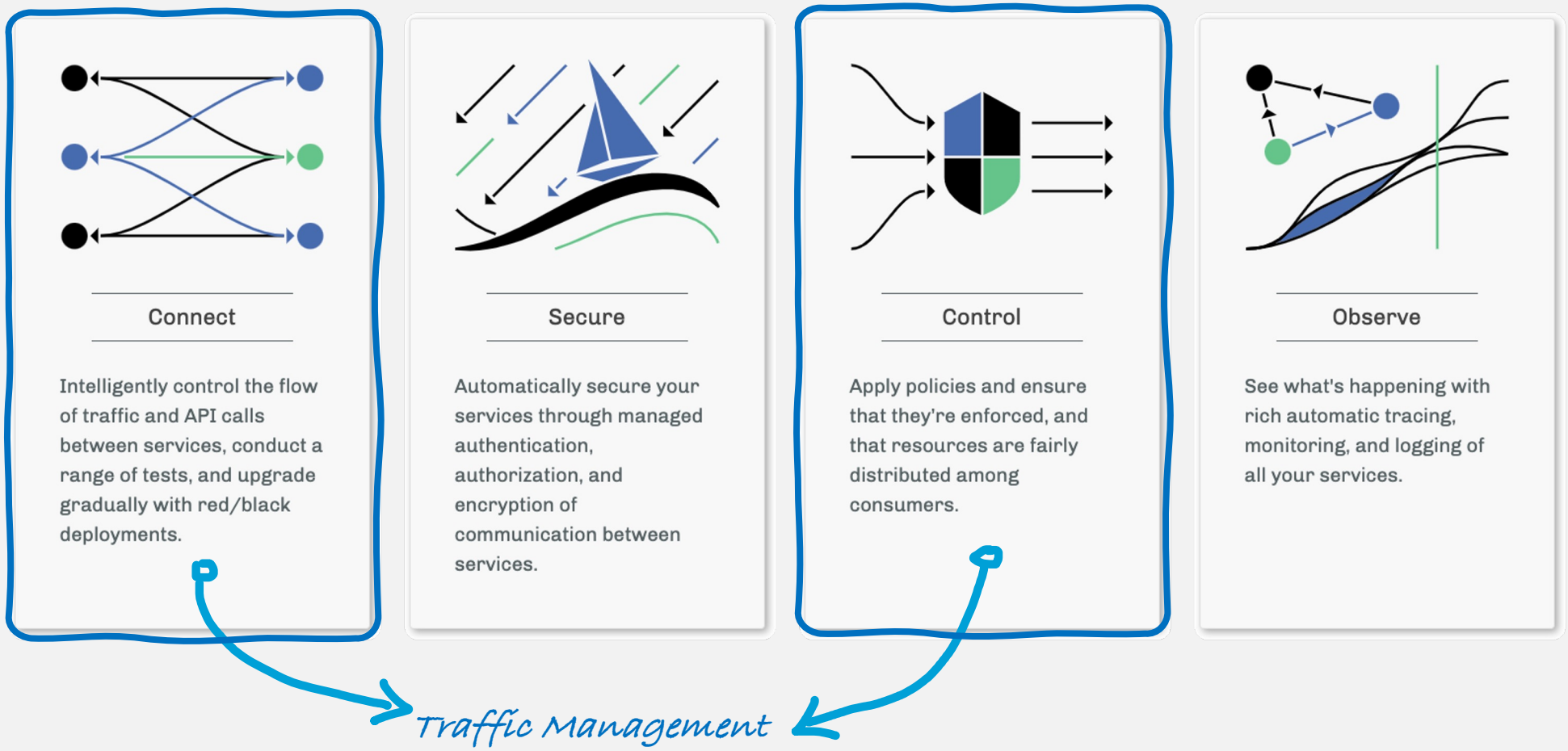


- Loadbalancing für HTTP-, gRPC-, WebSocket- und TCP-Verkehr.
- Steuerung des Verkehrsverhaltens mit umfangreichen Routing-Regeln, Wiederholungsversuchen, Failovers und Fehlerinjektion.
- Management von Traffic Policies, die Zugriffskontrollen, Ratenbeschränkungen und Kontingente regeln.
- Automatische Metriken, Protokolle und Traces für den gesamten Datenverkehr innerhalb eines Clusters, einschließlich Clusterein- und -ausgang.
- Sichere Service-zu-Service-Kommunikation mittels Authentifizierung und Autorisierung.

Quelle: istio.io

ISTIO

Typvertreter für Service Meshs



Quelle: https://istio.io

CLOUD NATIVE COMPUTING

Merke:

Die Kernfunktionalitäten eines Service Mesh bestehen darin, in bestehende Service-of-Service Systeme,

- *Traffic Management-,*
- *Security-,*
- *und Observability-*

Funktionalitäten zu injizieren, ohne dafür bestehenden Service Code anfassen zu müssen.

ISTIO

Traffic Management

Mit Traffic-Routing-Regeln lässt sich der Fluss des Datenverkehrs und der API-Aufrufe zwischen Services steuern. Istio vereinfacht die Konfiguration von Service-Level-Eigenschaften wie Circuit Breakers, Timeouts und Retries und erleichtert das Einrichten wichtiger Aufgaben wie A/B-Tests, Canary-Rollouts und gestaffelte Rollouts mit prozentualer Traffic-Aufteilung. Außerdem bietet es Out-of-Box-Failure-Recovery-Funktionen, die dazu beitragen, Services robuster gegen Ausfälle von Upstream-Services oder des Netzwerks zu machen.

Das Traffic Management von Istio stützt sich auf den Envoy-Proxy. Der gesamte Datenverkehr, den Services im Mesh senden und empfangen (Datenebene), wird über Envoy geleitet, sodass der Datenverkehr im Mesh leicht kontrolliert und überwacht werden kann, ohne Änderungen an Services vornehmen zu müssen.

Regeln werden mithilfe von **Kubernetes Custom Resource Definitions (CRDs)** spezifiziert, die sich Kubernetes-üblich mittels YAML ausdrücken lassen.

Mittels diesen Ressourcen lässt sich ein Istio Service Mesh definieren:

- Virtual Services
- Destination Rules
- Gateways
- Service Entries
- Sidecars

Quelle: istio.io

ISTIO

Traffic Management (Virtual Services)

Beispiel eines Nutzer-basierten Routings

Mehr dazu im Lab !!!

Virtual Services sind zusammen mit Destination Rules die wichtigsten Bausteine der Traffic-Routing-Funktionalität von Istio. Mit einem virtuellen Service lässt sich konfigurieren, wie Requests an einen Service innerhalb eines Istio-Service-Meshs weitergeleitet werden. Jeder virtuelle Service besteht aus einem Satz von Routing-Regeln, die der Reihe nach ausgewertet werden, so dass Istio jeden Request an den virtuellen Service mit einem bestimmten realen Service innerhalb des Meshes abgleichen kann.

Virtuelle Services entkoppeln Consuming Services von Providing Services durch einen Routing Mittler. Mittels virtuellen Services hat man so umfangreiche Möglichkeiten, verschiedene Regeln für das Routing von Datenverkehr in einem Mesh festzulegen.

Ein typischer Anwendungsfall ist das Umleiten von Datenverkehr an verschiedene Versionen eines Services, die als Serviceuntergruppen angegeben werden. Consuming Services senden Anfragen an den virtuellen Service-Host, als ob es sich um eine einzelne Entität handeln würde, und der Proxy leitet den Verkehr dann je nach den Regeln für den virtuellen Service an die verschiedenen Versionen weiter. Solche Regeln können bspw. sein

- "20 % der Anrufe gehen an die neue Version"
- *"Anrufe von diesen Benutzern gehen an Version 2".*

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
      end-user:
        exact: jason
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v3
```

Kubernetes Service-Name

Definition von Subsets (nächste Folie)

Routing-Regeln werden in sequentieller Reihenfolge von oben nach unten ausgewertet, wobei die erste Regel in der Definition des virtuellen Services die höchste Priorität erhält.

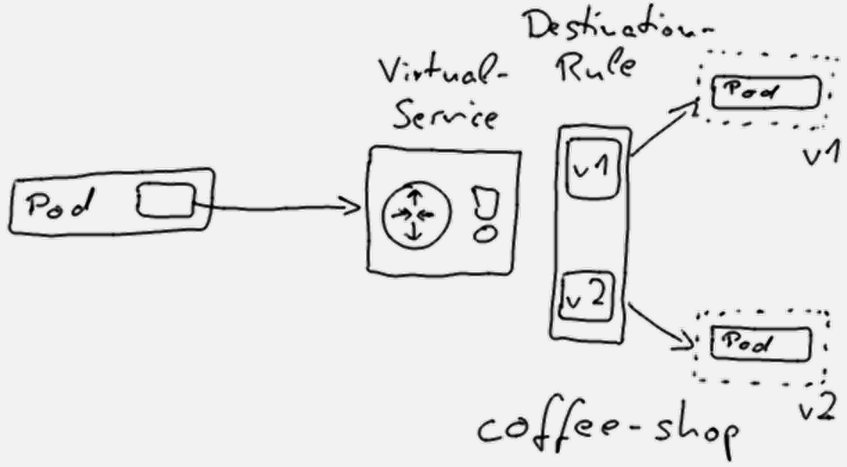
Quelle: istio.io

ISTIO

Traffic Management (Destination Rules)

Zusammen mit den virtuellen Services sind Destination Rules ein wichtiger Bestandteil der Traffic-Routing-Funktionalität von Istio. Virtuelle Services haben die Aufgabe Datenverkehr zu einem bestimmten Ziel zu leiten und dann dort Destination Rules anzuwenden, um zu konfigurieren, was mit dem Datenverkehr für dieses Ziel geschieht. Destination Rules werden angewendet, nachdem die Routing-Regeln für virtuelle Services ausgewertet wurden, sie gelten also für das "echte" Ziel des Datenverkehrs.

Insbesondere verwendet man Destination Rules, um benannte Service-Untergruppen anzugeben, z. B. die Gruppierung aller Instanzen eines bestimmten Services nach Version. Dadurch können Service-Untergruppen dann in den Routing-Regeln der virtuellen Services verwendet werden, um den Datenverkehr zu verschiedenen Instanzen von Services zu steuern.



Istio unterstützt in Destination Rules u.a. die folgenden Lastverteilungsmodelle, die für Services oder Service-Untergruppen angeben werden können:

- **ROUND ROBIN (default):** Alle Serviceinstanzen erhalten der Reihe nach Anfragen.
- **RANDOM:** Anfragen werden nach dem Zufallsprinzip an Instanzen im Pool weitergeleitet.
- **WEIGHTED:** Anfragen werden nach einem bestimmten Prozentsatz an Instanzen im Pool weitergeleitet.
- **LEASTS REQUESTS:** Anfragen werden an Instanzen mit der geringsten Anzahl von Anfragen weitergeleitet.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
  - name: v3
    labels:
      version: v3
```

Subset Definition

Beispiel: *Diese Destination Rule konfiguriert drei verschiedene Untergruppen „v1", „v2", „v3" für den Service „my-svc" mit unterschiedlichen Load-Balancing Stratgien.*

Quelle: istio.io

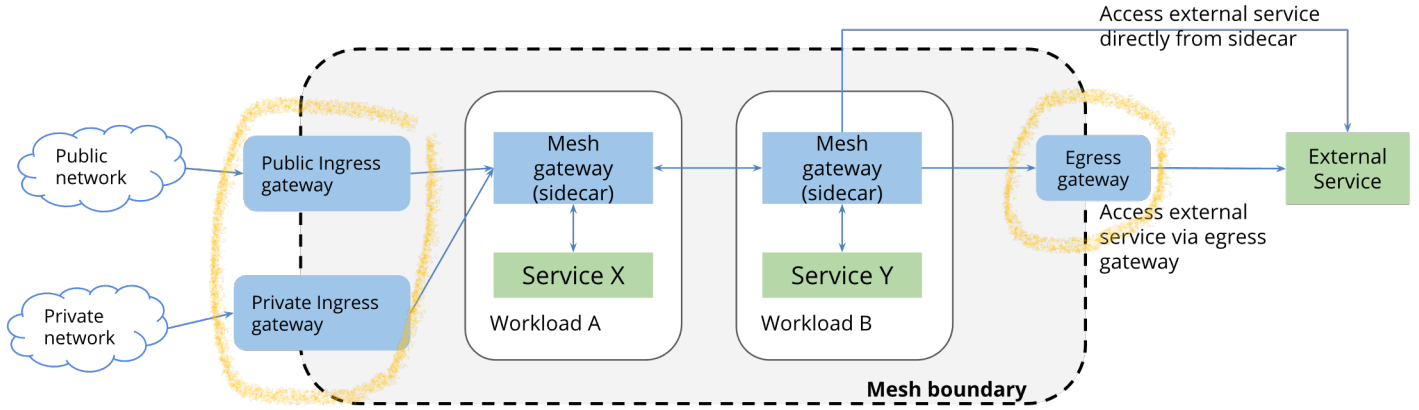
ISTIO

Traffic Management (Gateways)

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ext-host-gwy
spec:
  selector:
    app: my-gateway-controller
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - ext-host.example.com
    tls:
      mode: SIMPLE
      credentialName: ext-host-cert
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-svc
spec:
  hosts:
  - ext-host.example.com
  gateways:
  - ext-host-gwy
```

Beispiel: *Gateway-Konfiguration für externen HTTPS-Eingangsverkehr. Um das Routing festzulegen, muss das Gateway an einen virtuellen Service gebunden werden. Dies erfolgt über das Feld gateways des virtuellen Services.*

Istio verwendet Gateways, um den ein- und ausgehenden Datenverkehr für ein Mesh zu verwalten. So kann angegeben werden, welcher Datenverkehr in das Mesh eintreten oder es verlassen darf. Gateways sind eigenständige Envoy-Proxys, die am „Rand" des Mesh laufen.



Gateways sind ein auf Istio abgestimmter Ersatz für Kubernetes-Ingresses. Mit der Gateway-Ressource wird lediglich das Load Balancing auf Schicht 4-6 konfiguriert, aber auch Ports, TLS-Einstellungen usw..

Das Traffic-Routing der Anwendungsschicht (L7) kann mittels regulären virtuellen Services an das Gateway gebunden werden. Dadurch kann der Gateway-Verkehr im Grunde wie jeder anderer Data Plane-Verkehr in einem Istio-Mesh konfiguriert werden. Dadurch sind Istio Gateways wesentlich flexibler einsetzbar als Standard Kubernetes Ingresses.

Quelle: istio.io

ISTIO

Traffic Management (Service Entries)

Service Entries werden verwendet, um einen Eintrag zur Istio Service Registry hinzuzufügen. **Proxies können Datenverkehr an externe Services senden, als wäre dies ein lokaler Service im Mesh.** Auf diese Weise lässt sich auch Datenverkehr verwalten, der außerhalb des Mesh abläuft:

- Umleiten und Weiterleiten von Datenverkehr für externe Ziele, wie z. B. aus dem Web konsumierte APIs oder Datenverkehr zu Services in einer Legacy-Infrastruktur.
- Definition von Wiederholungs-, Timeout- und Fault-Injection-Richtlinien für externe Ziele.

Nicht für jeden externen Service, muss ein Service Entry hinzugefügt werden. Standardmäßig konfiguriert Istio seine Proxys so, dass Requests an unbekannte externe Services durchgeleitet werden.

In diesem Fall können jedoch keine Istio-Funktionen verwendet werden, um den Verkehr zu externen Zielen zu steuern, die nicht im Mesh registriert sind.

Quelle: istio.io

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
  - ext-svc.example.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

Beispiel:
Der folgende Service Entry fügt die Mesh-externe Abhängigkeit ext-svc.example.com zur Service-Registry von Istio hinzu.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ext-res-dr
spec:
  host: ext-svc.example.com
  trafficPolicy:
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/myclientcert.pem
      privateKey: /etc/certs/client_private_key.pem
      caCertificates: /etc/certs/rootcacerts.pem
```

Beispiel:
Virtuelle Services und Destination Rules können dann genutzt werden, um den Datenverkehr zu einem Service Entry Istio-üblich zu steuern. Diese Destination Rule legt bspw. die Verwendung von Mutual TLS fest, um die Verbindung zum externen Dienst ext-svc.example.com abzusichern.

ISTIO

Traffic Management (Sidecars)

Standardmäßig ist jeder Istio Proxy so konfiguriert, dass er Datenverkehr an allen Ports des zugehörigen Workloads akzeptiert und beim Weiterleiten von Datenverkehr jeden Workload im Mesh erreicht. Sidecar-Konfigurationen können verwendet werden, um von diesem Verhalten abzuweichen:

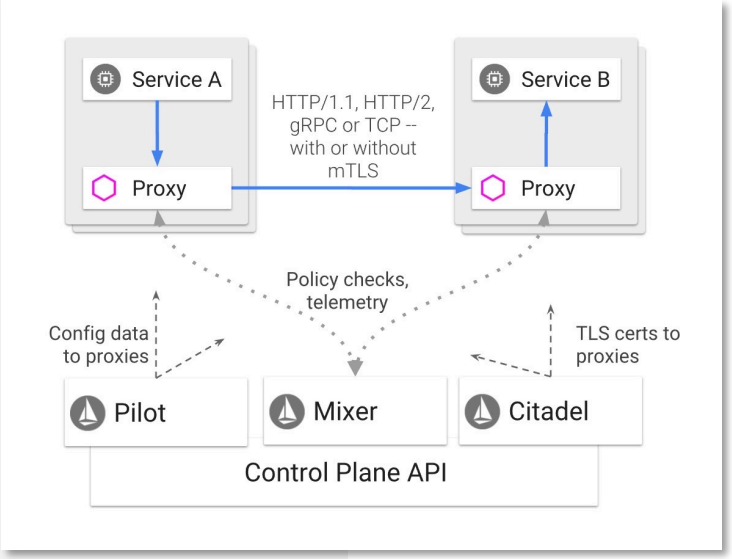
- Einschränken von Ports und Protokollen, die ein Proxy akzeptiert (eine Art Mesh-interne Firewall).
- Einschränken der Services, die der Envoy-Proxy erreichen kann.

Die Erreichbarkeit von Sidecars in größeren Anwendungen sollte auf diese Weise eingeschränkt werden, da jeder Proxy, der so konfiguriert ist, dass er jeden anderen Service im Mesh erreicht, die Leistung des Mesh aufgrund der hohen Speichernutzung beeinträchtigen kann.

Sidecar-Konfigurationen können für alle Workloads in einem bestimmten Namespace gelten, aber auch auf bestimmte Workloads eingeschränkt werden.

```
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: bookinfo
spec:
  egress:
  - hosts:
    - "./*"
    - "istio-system/*"
```

Diese Sidecar-Konfiguration konfiguriert beispielsweise alle Services im bookinfo-Namespace so, dass diese nur Dienste erreichen, die im selben Namespace und auf der Istio-Kontrollebene laufen (erforderlich, um die Richtlinien- und Telemetriefunktionen von Istio zu verwenden).



Quelle: istio.io

ISTIO

Resilienz (Timeouts)

Ein Timeout ist die Zeitspanne, die ein Proxy auf Antworten von einem bestimmten Service warten sollte, um sicherzustellen, dass Requests innerhalb eines vorhersehbaren Zeitrahmens erfolgreich sind oder fehlschlagen. Timeout für HTTP-Anfragen sind in Istio standardmäßig deaktiviert.

Für einige Anwendungen und Service ist der Standard-Timeout des Betriebssystems jedoch möglicherweise nicht geeignet. Ein zu langes Timeout könnte beispielsweise zu einer übermäßigen Latenz in Fehlersituationen führen, während ein zu kurzes Timeout dazu führen könnte, dass Requests unnötig fehlschlagen, während man auf die Rückkehr einer Operation wartet, an der mehrere Dienste beteiligt sind.

Um optimalen Timeout-Einstellungen zu finden und zu verwenden, lassen sich mit Istio Timeouts auf einfache Weise auf Ebene virtueller Services dynamisch pro Service anpassen.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    timeout: 10s
```

Für diesen virtuellen Service wird bspw. eine Timeout für Requests an die v1-Untermenge des „ratings" Service festlegt.

Quelle: istio.io

CLOUD NATIVE COMPUTING

Unter Resilienz versteht man die Widerstandsfähigkeit von Systemen gegenüber Teilsystemausfällen.

Resilienz lässt sich in Systemen unter anderem durch Wahl geeigneter Einstellungen von

- *Timeouts*
- *Wiederholungen*
- *Circuit Breaker*

erreichen.

Die gewählten Einstellungen lassen sich mittels Fault Injection testen und optimieren.

ISTIO

Resilienz (Retries)

Wiederholungen geben an, wie oft ein Proxy maximal versucht, eine Verbindung zu einem Service herzustellen, wenn der erste Aufruf fehlschlägt.

Wiederholungen können die Verfügbarkeit von Services und damit die Widerstandsfähigkeit des Gesamtsystems gegenüber Fehlerzuständen verbessern.

Wiederholungen stellen sicher, dass Requests nicht dauerhaft aufgrund von vorübergehenden Problemen wie einem überlasteten Service oder Netzwerk fehlschlagen. Das Intervall zwischen den Wiederholungsversuchen ist variabel und wird von Istio automatisch festgelegt, um zu verhindern, dass der aufgerufene Dienst mit Anfragen überlastet wird. Das Standard-Wiederholungsverhalten für Requests sind bspw. zwei Wiederholungsversuche.

Wie bei Timeouts kann auch das Wiederholungsverhalten Service-spezifisch konfiguriert werden.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    retries:
      attempts: 3
      perTryTimeout: 2s
```

Dieses Beispiel konfiguriert maximal 3 Wiederholungsversuche, um nach einem anfänglichen Anruffehler eine Verbindung zu dem Service-Subset „v1“ herzustellen, jeweils mit einem Timeout von 2 Sekunden.

Quelle: istio.io

ISTIO

Resilienz (Circuit Breaker)

Circuit Breaker sind ein weiterer Mechanismus für die Erstellung von resilienten Microservice-basierten Anwendungen.

In einem Circuit Breaker werden **Requestlimits** für Services festgelegt, z. B. die Anzahl der gleichzeitigen Verbindungen oder wie oft Aufrufe an einen Service fehlgeschlagen sind. Sobald dieses Limit erreicht ist, wird der Circuit Breaker "ausgelöst" und stoppt weitere Verbindungen. Die Verwendung eines Circuit-Breaker-Musters ermöglicht einen schnellen Ausfall von Services (fail early!!!), anstatt dass Clients versuchen, eine Verbindung zu überlasteten oder ausfallenden Services herzustellen.

Wie bei Timeouts und Wiederholungen können auch Circuit-Breaker Settings Service-spezifisch konfiguriert werden.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
```

Dieses Beispiel begrenzt die Anzahl der gleichzeitigen Verbindungen für die v1 Version des Review-Service auf 100.

Quelle: istio.io

ISTIO

Resilienz (Fault Injection)

Mittels Fault-Injection-Mechanismen kann man die Fehlerwiederherstellungskapazität von Anwendungen als Ganzes testen. Fault Injection ist eine Testmethode, die Fehler in ein System einführt, um sicherzustellen, dass es Fehlerbedingungen standhalten und sich davon erholen kann. Die Verwendung von Fault Injection kann nützlich sein, um sicherzustellen, dass Richtlinien zur Fehlerbehebung geeignet sind, die Verfügbarkeit kritischer Services sicherzustellen.

Mittels Fault Injection lassen sich somit Fehler auf der Anwendungsebene injizieren. Auf diese Weise kann man relevantere Fehler, wie z. B. HTTP-Fehlercodes, injizieren, um relevantere Ergebnisse zu erhalten.

Istio kann zwei Arten von Fehlern injizieren:

- **Delays:** Verzögerungen sind Timing-Fehler. Sie imitieren eine erhöhte Netzwerklatenz oder einen überlasteten Upstream-Service.
- **Aborts:** Abbrüche sind Crash-Fehler. Sie imitieren Ausfälle in Upstream-Services. Abbrüche treten normalerweise in Form von HTTP-Fehlercodes oder TCP-Verbindungsfehlern auf.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percentage:
          value: 0.1
        fixedDelay: 5s
    route:
    - destination:
        host: ratings
        subset: v1
```

Dieser virtuelle Service führt bei 1 von 1000 Requests zu einer Verzögerung von 5 Sekunden.

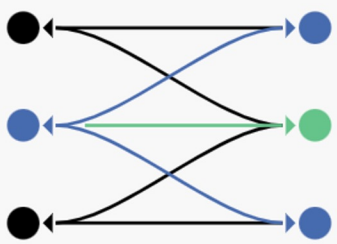
Quelle: istio.io

CLOUD NATIVE COMPUTING

Fault Injection ist quasi Chaos-Engineering auf der Data Plane.

ISTIO

Typvertreter für Service Meshs

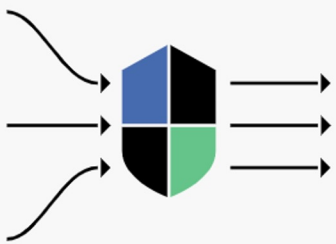


Connect

Intelligently control the flow of traffic and API calls between services, conduct a range of tests, and upgrade gradually with red/black deployments.

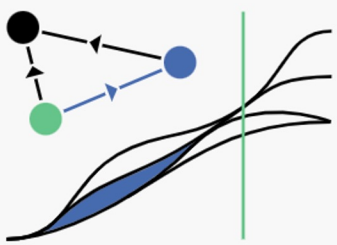
Secure

Automatically secure your services through managed authentication, authorization, and encryption of communication between services.



Control

Apply policies and ensure that they're enforced, and that resources are fairly distributed among consumers.



Observe

See what's happening with rich automatic tracing, monitoring, and logging of all your services.

Quelle: https://istio.io

CLOUD NATIVE COMPUTING

Merke:

Die Kernfunktionalitäten eines Service Mesh bestehen darin, in bestehende Service-of-Service Systeme,

- *Traffic Management-,*
- *Security-,*
- *und Observability-*

Funktionalitäten zu injizieren, ohne dafür bestehenden Service Code anfassen zu müssen.

ISTIO

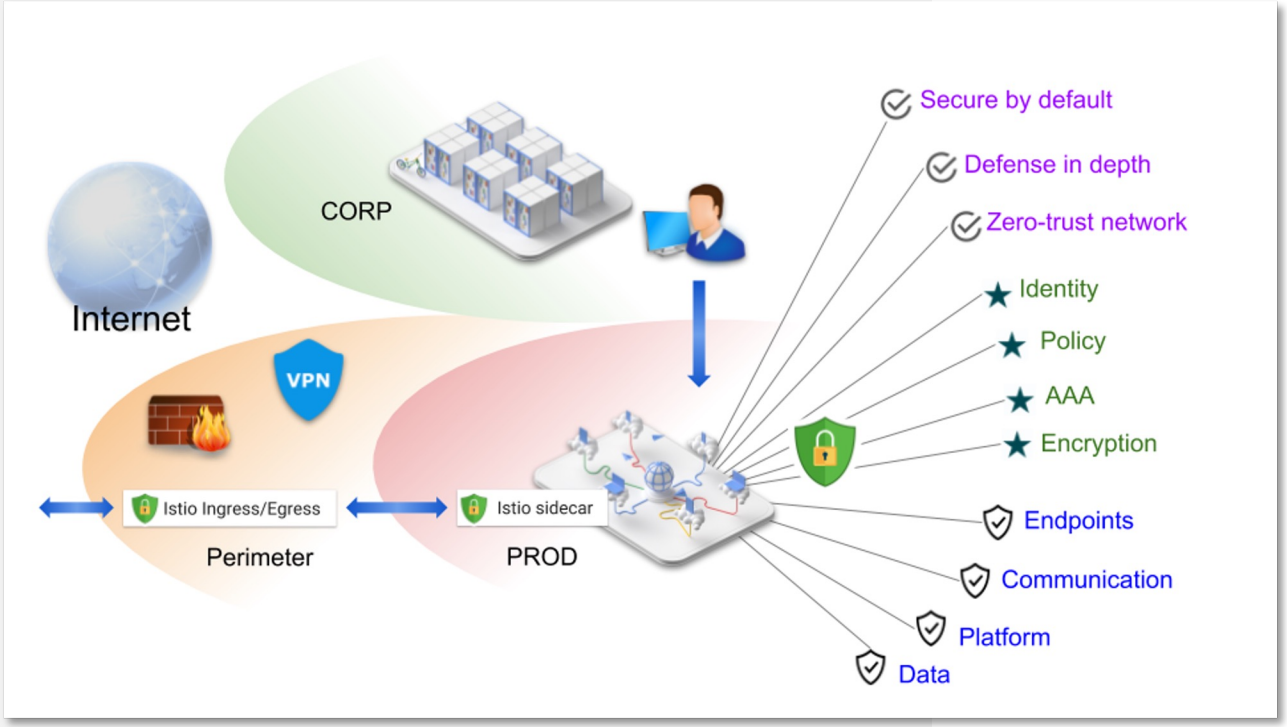
Security (Leitgedanken)

Die Zerlegung einer monolithischen Anwendung in atomare Services bietet verschiedene Vorteile, darunter eine bessere Agilität, eine bessere Skalierbarkeit und eine bessere Wiederverwendbarkeit von Services. Allerdings haben Microservices auch besondere Sicherheitsanforderungen:

- Um sich gegen **Man-in-the-Middle-Angriffe** zu schützen, benötigt man eine Verschlüsselung des Datenverkehrs.
- Um eine flexible **Service-Zugriffskontrolle** zu ermöglichen, benötigt man feinkörnige Zugriffsrichtlinien.
- Um festzustellen, wer was zu welchem Zeitpunkt getan hat, benötigt man **Auditing-Tools**.

Die Sicherheitsfunktionen von Istio bieten insbesondere eine transparente TLS-Verschlüsselung und Tools für Authentifizierung, Autorisierung und Audit (AAA). Die Leitgedanken dabei sind:

- **Security by default:** Es sollen keine Änderungen am Anwendungscode und der Infrastruktur erforderlich werden
- **Defense in Depth:** Integration mit bestehenden Sicherheitssystemen zur Bereitstellung mehrerer Verteidigungsebenen
- **Zero-Trust Networking:** Aufbau von Sicherheitslösungen auf vertrauensunwürdigen Netzwerken



Quelle: istio.io

EXKURS:

Zero Trust Networking

Zero-Trust-Netzwerke beschreiben einen Ansatz für das Design und die Implementierung von IT-Netzwerken. Das Hauptkonzept hinter Zero Trust ist, dass vernetzten Geräten, wie z. B. Laptops, standardmäßig nicht vertraut werden sollte, selbst wenn sie mit einem verwalteten Unternehmensnetzwerk wie dem Firmen-LAN verbunden sind und selbst wenn sie zuvor verifiziert wurden.

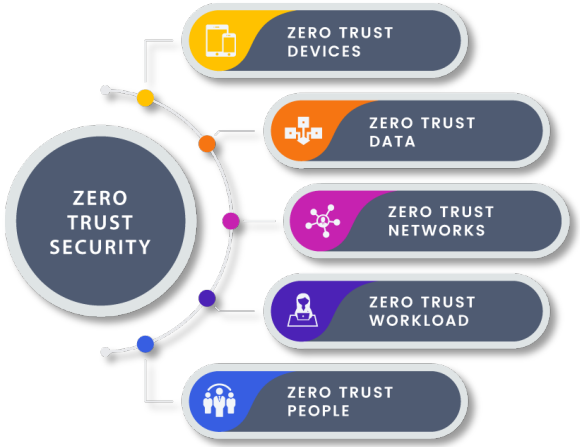
Viele Unternehmensnetzwerke bestehen aus vielen miteinander verbundenen Segmenten, Cloud-basierten Diensten und Infrastrukturen, sowie Verbindungen zu entfernten und mobilen Umgebungen und zunehmend auch Verbindungen zu nicht-konventioneller IT, wie IoT-Geräten.

Der traditionelle Ansatz, Geräten innerhalb eines fiktiven Unternehmensperimeters oder Geräten, die über ein VPN damit verbunden sind, zu vertrauen, macht in solch hochgradig diversifizierten und verteilten Umgebungen zunehmend weniger Sinn.

Zero Trust (ZT) ist eine Sammlung von Konzepten und Ideen, die darauf abzielen, die Integrität der Kommunikation in als gefährdet (also nicht 100%-ig vertrauenswürdig) angesehenen Netzwerken sicherzustellen.

Prinzipien:

- Verlässliche Quellen für die Benutzeridentität
- Benutzer-Authentifizierung
- Geräte-Authentifizierung
- Kontext-Awareness (z. B. die Einhaltung von Richtlinien und Gerätezustand)
- Autorisierungsrichtlinien für den Zugriff auf eine Anwendung
- Zugriffskontrollrichtlinien innerhalb einer Anwendung



Quelle:
Rose, Scott; Borchert, Oliver; Mitchell, Stu; Connelly, Sean. "Zero Trust Architecture". NIST. 2020
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf

EXKURS:

Mutual TLS

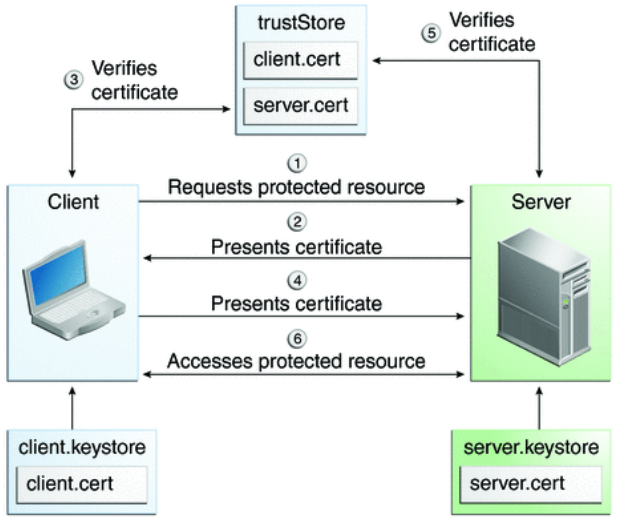
Gegenseitige (mutual) Authentifizierung oder Zwei-Wege-Authentifizierung (nicht zu verwechseln mit Zwei-Faktor-Authentifizierung) bezieht sich auf zwei Parteien, die sich in einem Authentifizierungsprotokoll gleichzeitig gegenseitig authentifizieren. Gegenseitige Authentifizierung ist in einigen Protokollen (bspw. SSH) vorgegeben und in anderen (z.B. TLS) optional.

Standardmäßig weist das TLS-Protokoll dem Client nur die Identität des Servers mit X.509-Zertifikaten nach, und die Authentifizierung des Clients gegenüber dem Server wird der Anwendungsschicht überlassen. **TLS bietet auch eine Client-zu-Server-Authentifizierung unter Verwendung der clientseitigen X.509-Authentifizierung an.** Da dies die Bereitstellung der Zertifikate für die Clients erfordert und **weniger benutzerfreundlich** ist, wird sie nur selten in Endbenutzeranwendungen eingesetzt.

Die gegenseitige TLS-Authentifizierung ("'mTLS"') ist in Business-to-Business (B2B)-Anwendungen wesentlich weiter verbreitet., da hier die Sicherheitsanforderungen im Vergleich zu Endkundenumgebungen in der Regel viel höher sind und diese Umgebungen besser automatisierbar sind.

In Cloud-nativen Systemen wird mTLS häufig durch Service Meshs transparent bereitgestellt.

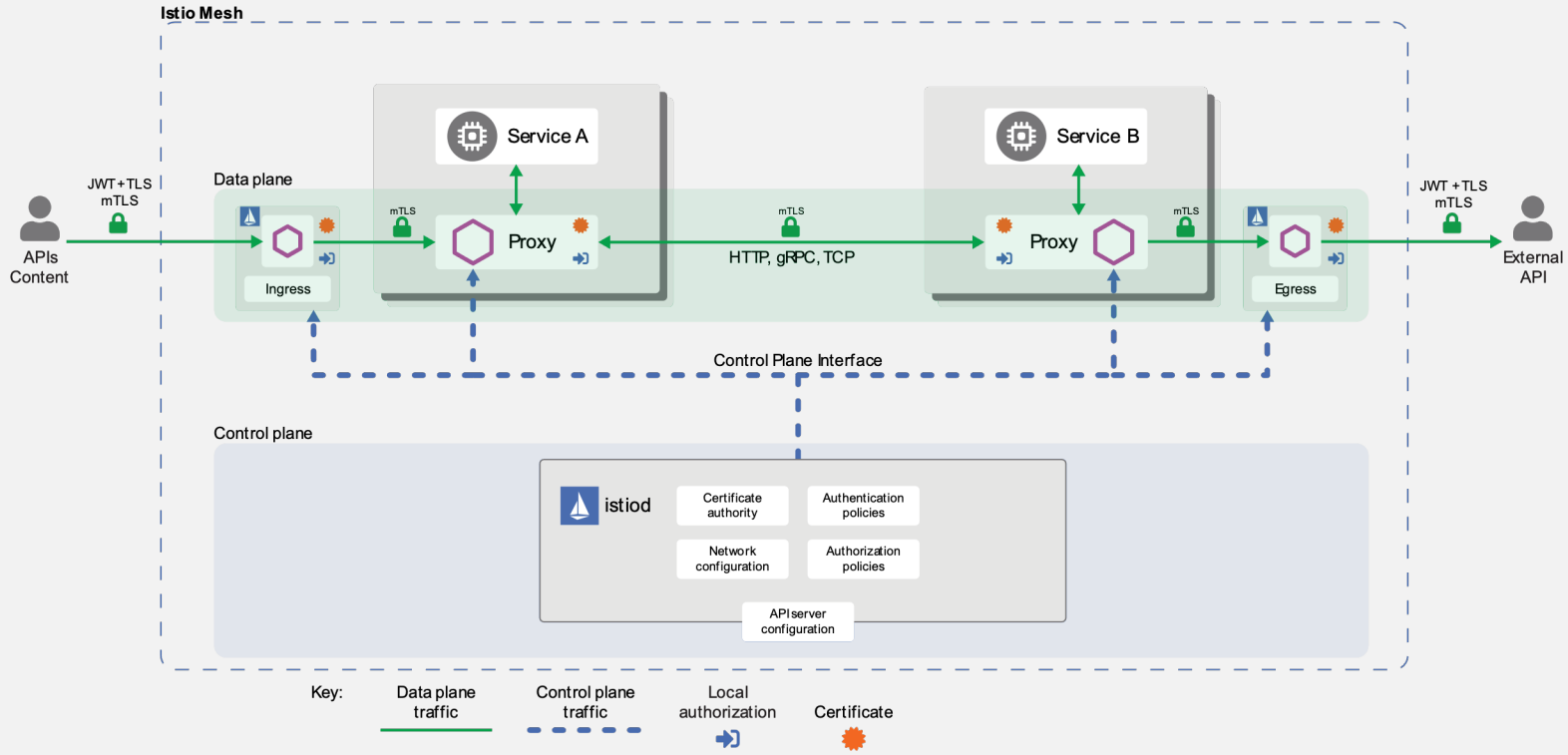
Die gegenseitige Authentifizierung ermöglicht das Zero-Trust-Networking, da sie die Kommunikations- und Informationsintegrität gewährleistet indem u.a. folgende Angriffe verhindert werden.



- Von **Man-in-the-Middle-Angriffen** (MITM) spricht man, wenn eine dritte Partei eine Nachricht abhören oder abfangen möchte und dabei manchmal die beabsichtigte Nachricht für den Empfänger verändert.
- Ein **Replay-Angriff** ähnelt einem MITM-Angriff, bei dem ältere Nachrichten aus dem Kontext gerissen wiedergegeben werden, um den Server zu täuschen.
- **Spoofing-Angriffe** beruhen auf der Verwendung falscher Daten, um sich als ein anderer Benutzer auszugeben, um Zugriff auf einen Server zu erhalten oder als jemand anderes identifiziert zu werden.

ISTIO

Security



Die Sicherheit wird in Istio durch mehrere Komponenten sichergestellt:

- **Eine Zertifizierungsstelle (CA) für die Schlüssel- und Zertifikatsverwaltung (Mutual TLS)**
- Der **Konfigurations-API-Server** zur Verteilung von **Authentication Policies**, **Authorization Policies** und **Secure Naming** Informationen an die Proxys.
- **Sidecar-** und **Perimeter-Proxys** arbeiten als **Policy Enforcement Points (PEPs),** um die Kommunikation zwischen Clients und Servern zu sichern.
- Istio-spezifische Envoy-Proxy-Erweiterungen zur Verwaltung von Telemetrie und Auditing.

Quelle: istio.io

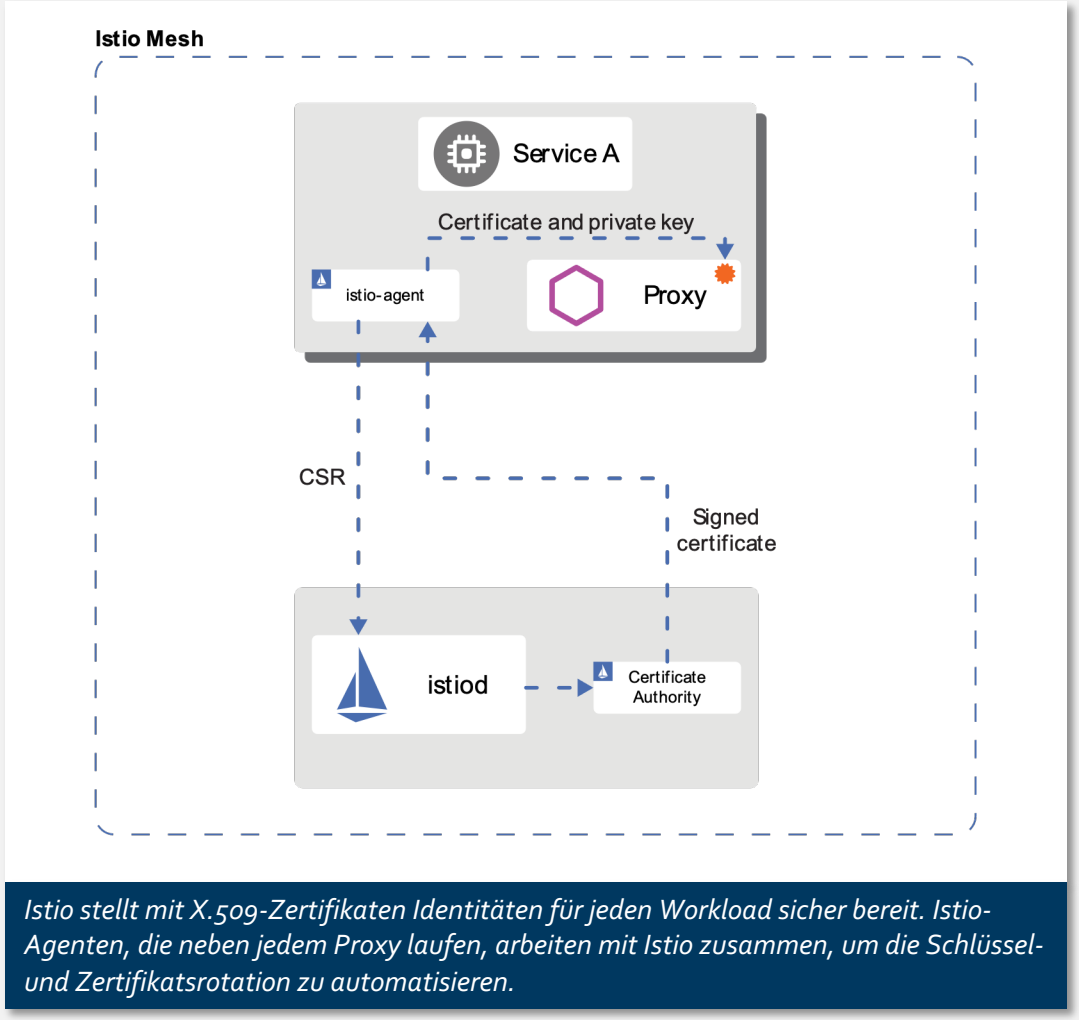
Die Control Plane übernimmt die Konfiguration vom API-Server und konfiguriert die PEPs in der Data Plane.

ISTIO

Security (Identity und Certificate Management)

Die Identität ist ein grundlegendes Konzept jeder Sicherheitsinfrastruktur. Zu Beginn einer Workload-zu-Workload-Kommunikation müssen die beiden Parteien Anmeldeinformationen mit ihren Identitätsinformationen zum Zwecke der gegenseitigen Authentifizierung austauschen.

Auf der Client-Seite wird die Identität des Services anhand der sicheren Namensinformationen überprüft, um festzustellen, ob es sich um einen autorisierten Nutzer des Services handelt. Auf der Serverseite kann basierend auf den Autorisierungsrichtlinien bestimmt werden, auf welche Informationen der Client zugreifen darf und ggf. vom Zugriff auf Services ausschließen.



Istio stellt mit X.509-Zertifikaten Identitäten für jeden Workload sicher bereit. Istio-Agenten, die neben jedem Proxy laufen, arbeiten mit Istio zusammen, um die Schlüssel- und Zertifikatsrotation zu automatisieren.

Quelle: istio.io

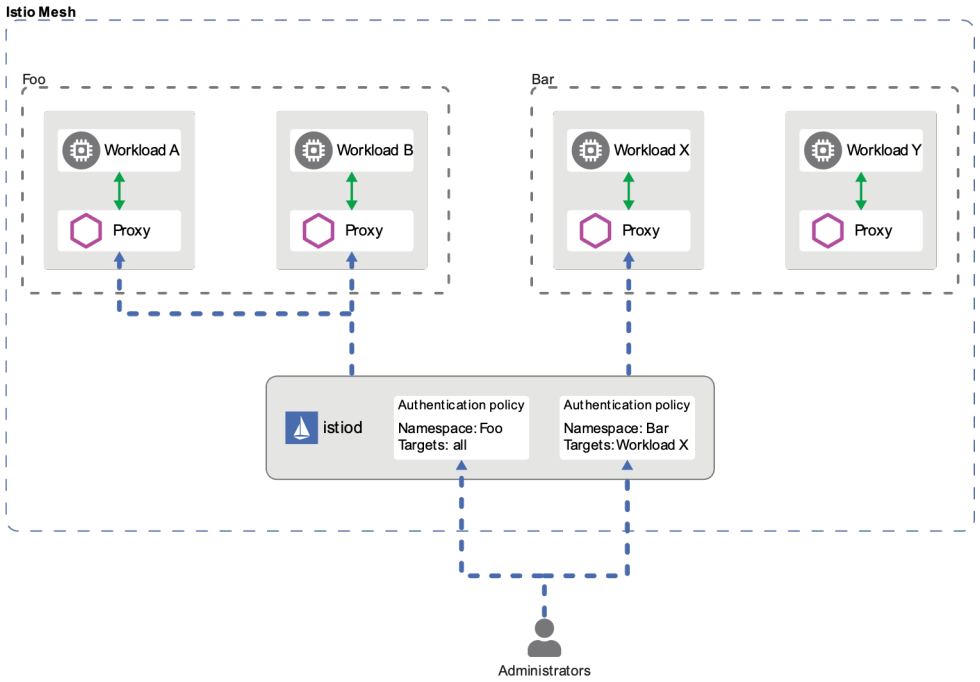
ISTIO

Security (Authentication Architektur)

Istio bietet zwei Arten der Authentifizierung:

- **Peer-Authentifizierung** wird für die Service-to-Service verwendet, um den Client zu verifizieren, der die Verbindung herstellt. Istio bietet Mutual TLS als Full-Stack-Lösung für die Transport-Authentifizierung, die ohne Änderungen am Service-Code aktiviert werden kann.

- **Request-Authentifizierung** wird für die Endbenutzer-Authentifizierung verwendet, um den an die Anfrage angehängten Berechtigungsnachweis zu verifizieren. Istio ermöglicht u.a. Authentifizierung mit JSON Web Token (JWT)-Validierung und die Verwendung benutzerdefinierter Authentifizierungsanbieters (OAUTH2) wie bspw. AuthO, Firebase Auth, Google Auth, Github, usw.



Die Authentifizierung für Services kann mithilfe von Peer- und Requets-Authentifizierungsrichtlinien festgelegt werden. Diese Authentication Policies werden im Istio-Konfigurationsspeicher gespeichert. Der Istio-Controller überwacht den Konfigurationsspeicher. Istio sendet diese Policies asynchron an die Ziel-Endpunkte. Sobald der Proxy eine Policy empfängt, wird diese sofort auf diesem Pod wirksam.

Quelle: istio.io

ISTIO

Security (Authentication Policies)

Peer-Authentifizierungsrichtlinien werden insbesondere genutzt, um Mutual TLS-Modus in einem Mesh durchzusetzen. Die folgenden Modi werden unterstützt:

- **PERMISSIVE:** Workloads akzeptieren sowohl mutual TLS als auch Klartextverkehr
- **STRICT:** Workloads akzeptieren nur gegenseitigen TLS-Verkehr.
- **DISABLE:** Wechselseitiges TLS ist deaktiviert. Nicht empfohlen, es sei denn, es wird eine eigene Sicherheitslösung bereitgestellt.

Mit Workload-spezifischen Peer-Authentifizierungsrichtlinien lassen sich auch verschiedene Mutual TLS-Modi für verschiedene Ports angeben.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

Beispiel:
Diese Peer-Authentifizierungsrichtlinie erfordert, dass alle Workloads im Namespace foo Mutual TLS verwenden.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "example-workload-policy"
  namespace: "foo"
spec:
  selector:
     matchLabels:
       app: example-app
  portLevelMtls:
    80:
      mode: DISABLE
```

Beispiel:
Diese Richtlinie deaktiviert bspw. Mutual TLS an Port 80 für den Workload app:example-app und verwendet die Mutual TLS-Einstellungen der Namespace-weiten Peer-Authentifizierungsrichtlinie für alle anderen Ports

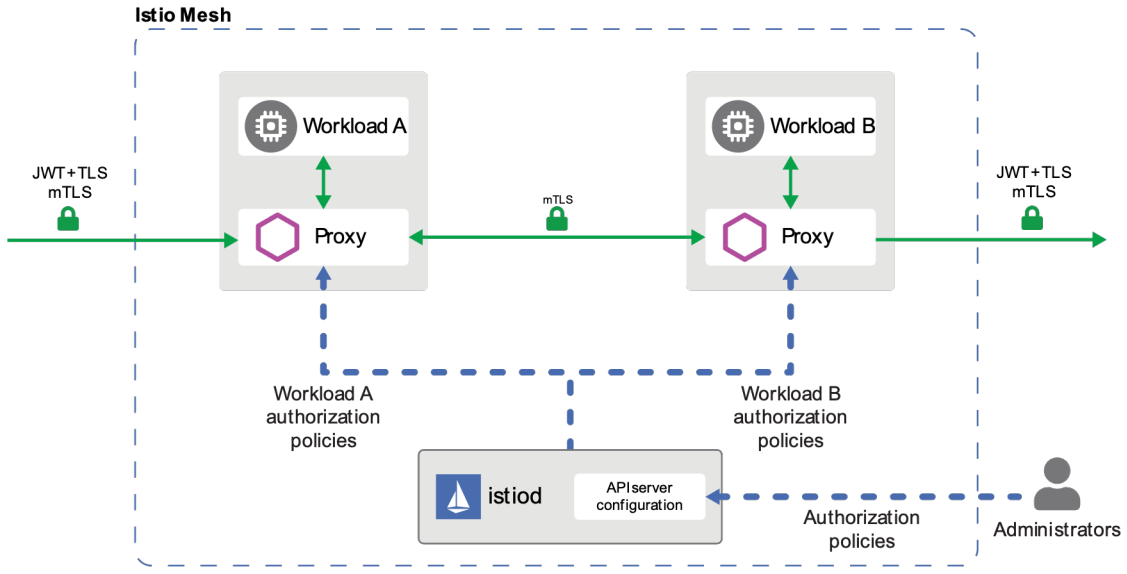
Quelle: istio.io

ISTIO

Security (Authorization Architektur)

Die Autorisierungsfunktionen von Istio bieten eine **Mesh-**, **Namespace-** und **Workload-**weite **Zugriffskontrolle** für Workloads in einem Mesh.

Jeder Proxy führt hierzu eine Autorisierungs-Engine aus, die Requests zur Laufzeit autorisiert. Wenn ein Request beim Proxy eingeht, wertet die Autorisierungs-Engine den Anfragekontext anhand der aktuellen Authorization Policy aus und gibt das Autorisierungsergebnis (ALLOW oder DENY) zurück.



Wenn eine Authorization Policy für einen Workloads mittels einer CRD formuliert wurde, wird diese anschließend von Istio durchgesetzt. Für Workloads ohne Authorization Policies lässt Istio hingegen alle Anfragen zu.

Quelle: istio.io

ISTIO

Security (Authorization Policies)

Authorization Policies können als ALLOW- oder DENY-Regeln formuliert werden.

DENY-Regeln haben Vorrang vor den ALLOW-Regeln. Beziehen sich mehrere Authorization Policies auf denselben Workload, wendet Istio diese additiv an.

Mittels Selektoren kann die Anwendung von Authorization Policies auf bestimmte Workloads eingeschränkt werden. Selektoren enthalten hierzu (Kubernetes-üblich) eine Liste von {Schlüssel: Wert}-Paaren, wobei der Schlüssel der Name des Labels ist.

Authorization Policies ohne Selektoren, gelten für alle Workloads im gleichen Namensraum der Authorization Policy.

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin-deny
 namespace: foo
spec:
 selector:
   matchLabels:
     app: httpbin
     version: v1
 action: DENY
 rules:
 - from:
   - source:
       notNamespaces: ["foo"]
```

Beispiel:
Diese DENY-Policy, verweigert Requests, wenn die Quelle nicht aus dem Namensraum foo stammt.

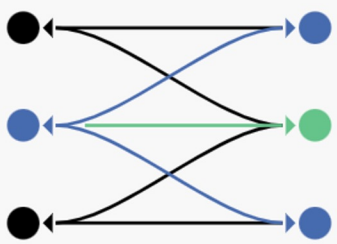
```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-read
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET", "HEAD"]
```

Beispiel:
Diese ALLOW-Policy erlaubt den "GET"- und "HEAD"-Zugriff auf den Workload mit dem Label "app: products" im Default-Namespace.

Quelle: istio.io

ISTIO

Typvertreter für Service Meshs

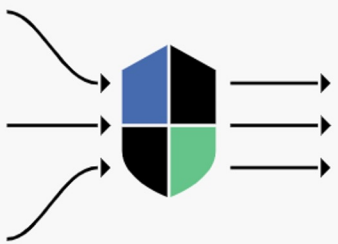


Connect

Intelligently control the flow of traffic and API calls between services, conduct a range of tests, and upgrade gradually with red/black deployments.

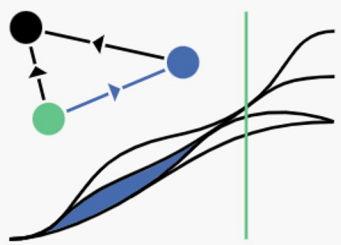
Secure

Automatically secure your services through managed authentication, authorization, and encryption of communication between services.



Control

Apply policies and ensure that they're enforced, and that resources are fairly distributed among consumers.



Observe

See what's happening with rich automatic tracing, monitoring, and logging of all your services.

Quelle: https://istio.io

CLOUD NATIVE COMPUTING

Merke:

Die Kernfunktionalitäten eines Service Mesh bestehen darin, in bestehende Service-of-Service Systeme,

- *Traffic Management-,*
- *Security-,*
- *und Observability-*

Funktionalitäten zu injizieren, ohne dafür bestehenden Service Code anfassen zu müssen.

ISTIO

Observability

Istio generiert detaillierte Telemetriedaten für die gesamte Service-Kommunikation innerhalb eines Meshes. Diese Daten ermöglichen die Beobachtung des Serviceverhaltens und ermöglicht im DevOps Feedback-Cycle Fehler zu beheben und Anwendungen zu optimieren.

Istio generiert die folgenden Arten von Telemetriedaten, um die Beobachtbarkeit eines Mesh zu ermöglichen.

- **Metriken:**
  Istio generiert eine Reihe von Service-Metriken, die auf den vier "goldenen Signalen" der Überwachung basieren (Latenz, Traffic, Fehler und Sättigung).
- **Distributed Traces:**
  Istio generiert verteilte Trace-Spans für jeden Service, die den Betreibern ein detailliertes Verständnis der Request-Flüsse und Service-Abhängigkeiten innerhalb eines Meshes ermöglichen.
- **Access Logs:**
  Wenn Datenverkehr in einen Service innerhalb eines Mesh fließt, kann Istio eine vollständige Aufzeichnung jedes Requests generieren, einschließlich Quell- und Ziel-Metadaten. Diese Informationen ermöglichen es das Serviceverhalten bis auf die Ebene der einzelnen Workload-Instanzen zu überprüfen.

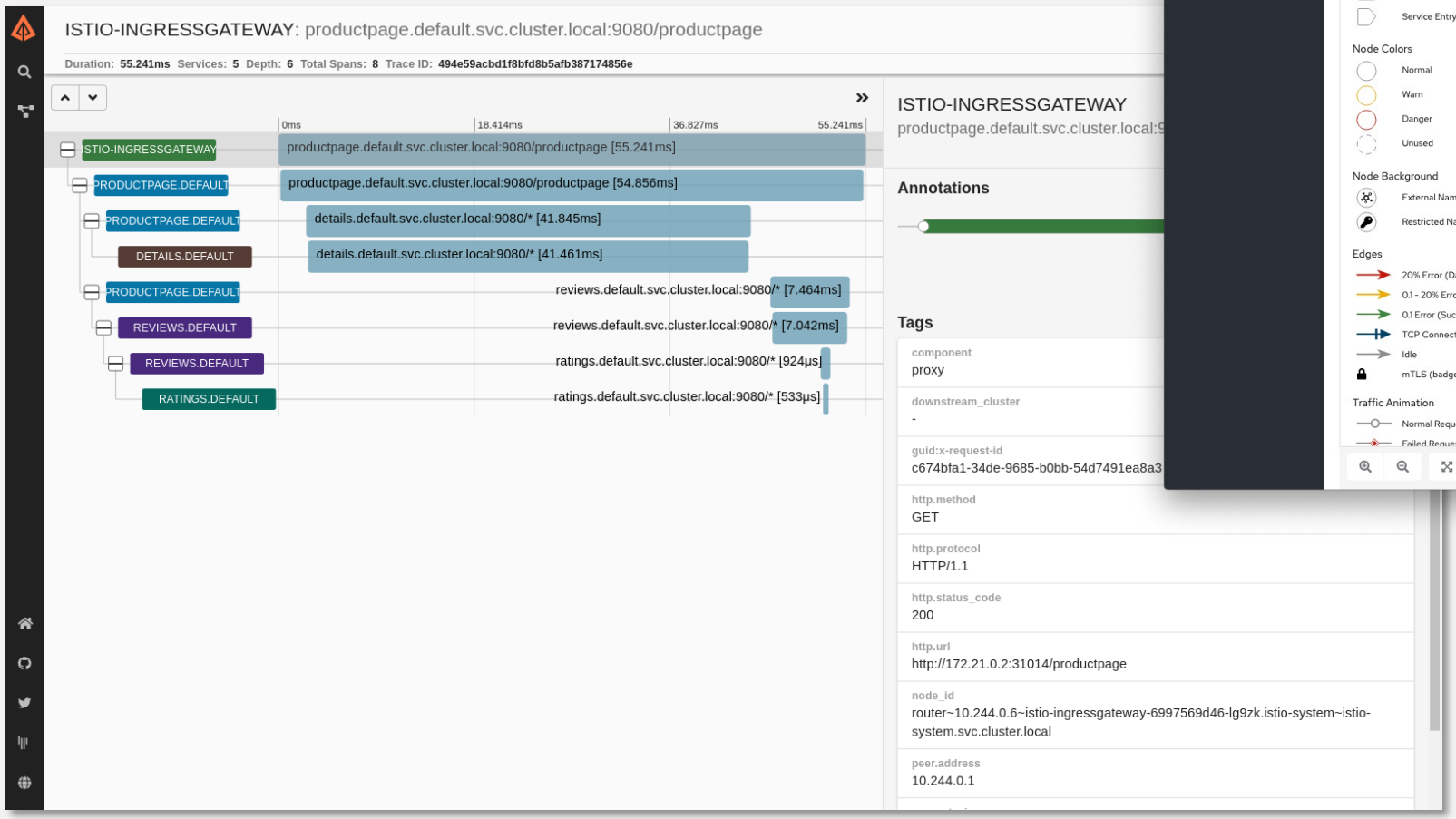
Hinweis:
Mit diesen drei Telemetriearten werden wir uns noch detaillierter befassen.

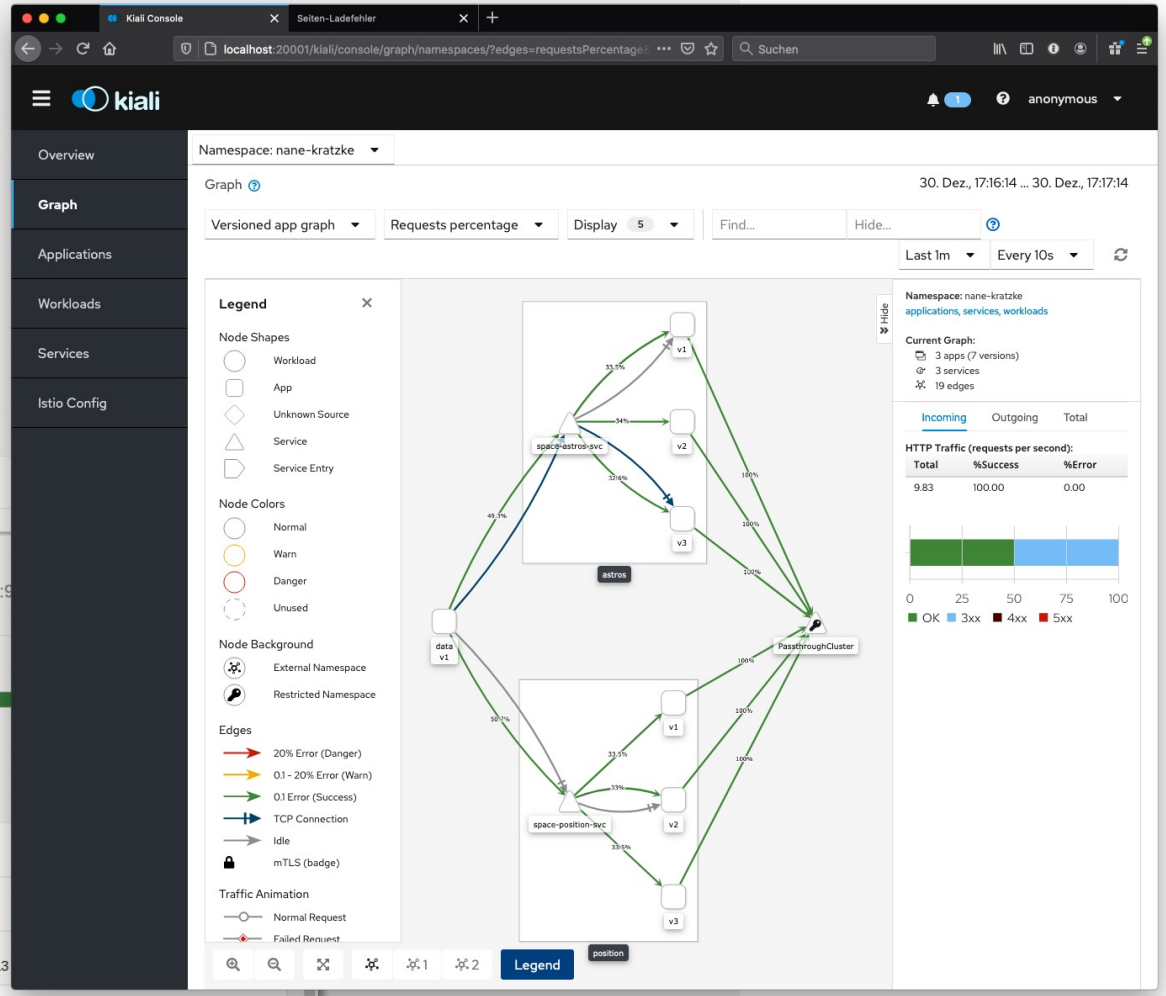
Quelle: istio.io

ISTIO

Observability

Beispiel:
Zipkin Oberfläche zum Tracen von Requests entlang von Service Aufrufen. Diese Auswertung kann durch Istio entlang von Gateways und Virtual Services sogar in einer Art Black-Box-Tracing Ansatz erhoben werden.





Beispiel:
Kiali ist eine Management-Konsole für Istio, die u.a. die Struktur Ihres Service-Meshes durch Ableitung der Verkehrstopologie anzeigt. Die hierfür erforderlichen Daten werden aus dem Netzwerkverkehr der Proxies erhoben.

Quelle: istio.io

ISTIO

Nur Versuch macht kluch ...

CLOUD NATIVE COMPUTING

Klonen Sie dieses Repository:

```
git clone https://git.mylab.th-luebeck.de/cloud-native/lab-observability.git
```


```
> Istio
> Instrumentieren von Microservices
> Inspizieren von Topologien
> Einfaches Traffic Management
```

ISTIO

Nur Versuch macht kluch ...

CLOUD NATIVE COMPUTING

Klonen Sie dieses Repository:

```
git clone https://git.mylab.th-luebeck.de/cloud-native/lab-traffic-mgmt.git
```


```
> Istio
> Quantitatives Traffic Management
> Content-based Traffic Management
```

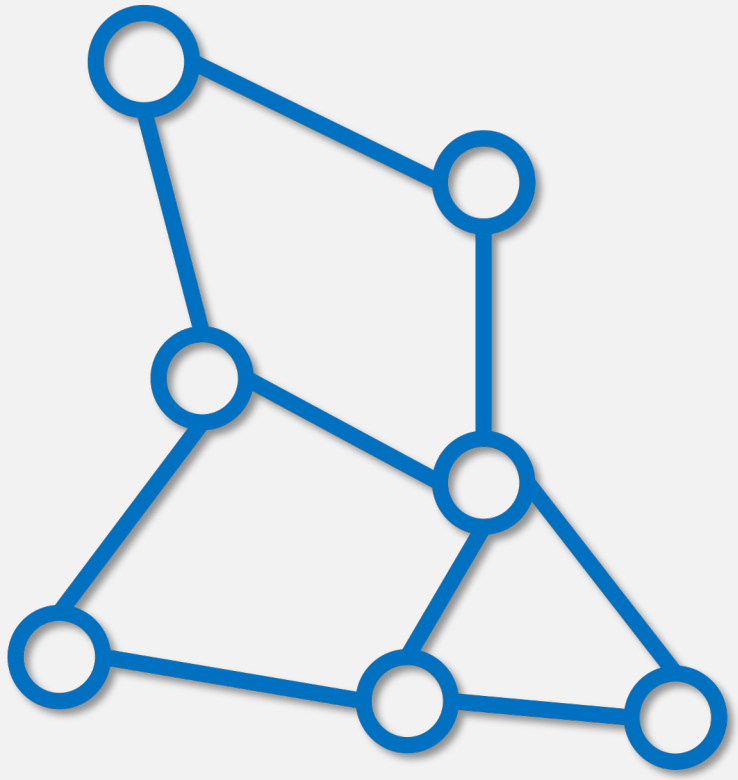
INHALTE

Warum Service Meshs?
- Was findet sich in der Literatur?
- Was sagen die Anbieter?

Service Meshs
- Traffic Management
- Resilienz
- Sicherheit
- Beobachtbarkeit

Fallstudie: Istio
- Traffic Management
- Security
- Observability

Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

Logging, Monitoring, Tracing

Bei Software bezieht sich Beobachtbarkeit typischerweise auf Telemetrie, die von Services erzeugt und häufig in drei Aspekte unterteilt wird:

- **Distributed Tracing** (verteilte Ablaufverfolgung) ermöglicht Einblick in den gesamten Lebenszyklus von Request an ein System, sodass Fehler und Latenzen bestimmbar sind.

- Metriken im Rahmen eines **Monitorings** liefern quantitative Informationen zu Prozessen, die im System ausgeführt werden.

- Mittels **Logging** (Protokollierung) lässt sich Einblick in anwendungsspezifische Nachrichten, die von Prozessen ausgegeben werden, gewinnen.

Diese Aspekte sind eng miteinander verbunden. Metriken können verwendet werden, um beispielsweise eine Teilmenge von Traces mit schlechter Performance zu lokalisieren. Mit diesen Traces verknüpfte Logs können dazu beitragen, die Hauptursache für dieses Verhalten zu ermitteln. Anschließend können basierend auf dieser Ermittlung neue Metriken konfiguriert werden, um dieses Problem beim nächsten Mal früher zu beheben.

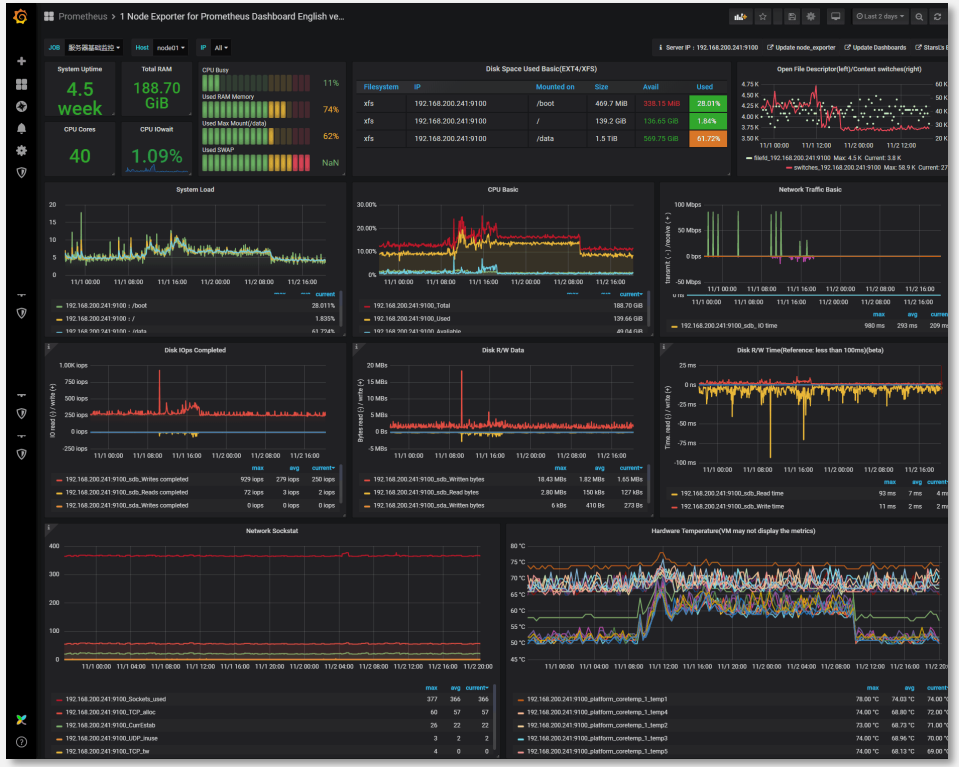
ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

Monitoring (Metrics)

Unter Monitoring versteht man allgemein die Überwachung von Vorgängen. Es ist ein Überbegriff für alle Arten von systematischen Erfassungen, Messungen oder Beobachtungen eines Vorgangs oder Prozesses mittels technischer Hilfsmittel und Beobachtungssysteme.

Eine Funktion des Monitorings besteht insbesondere darin, bei einem beobachteten Ablauf oder Prozess festzustellen, ob dieser den gewünschten Verlauf nimmt und bestimmte Schwellwerte eingehalten werden, um andernfalls steuernd eingreifen zu können.

Im Cloud-native Umfeld versteht man unter **Monitoring** insbesondere die Erfassung von Metriken in Form von Zeitreihen, um vor allem quantitative Informationen zu Prozessen und Vorgängen eines Cloud-nativen Systems erheben und analysieren zu können.



CLOUD NATIVE COMPUTING

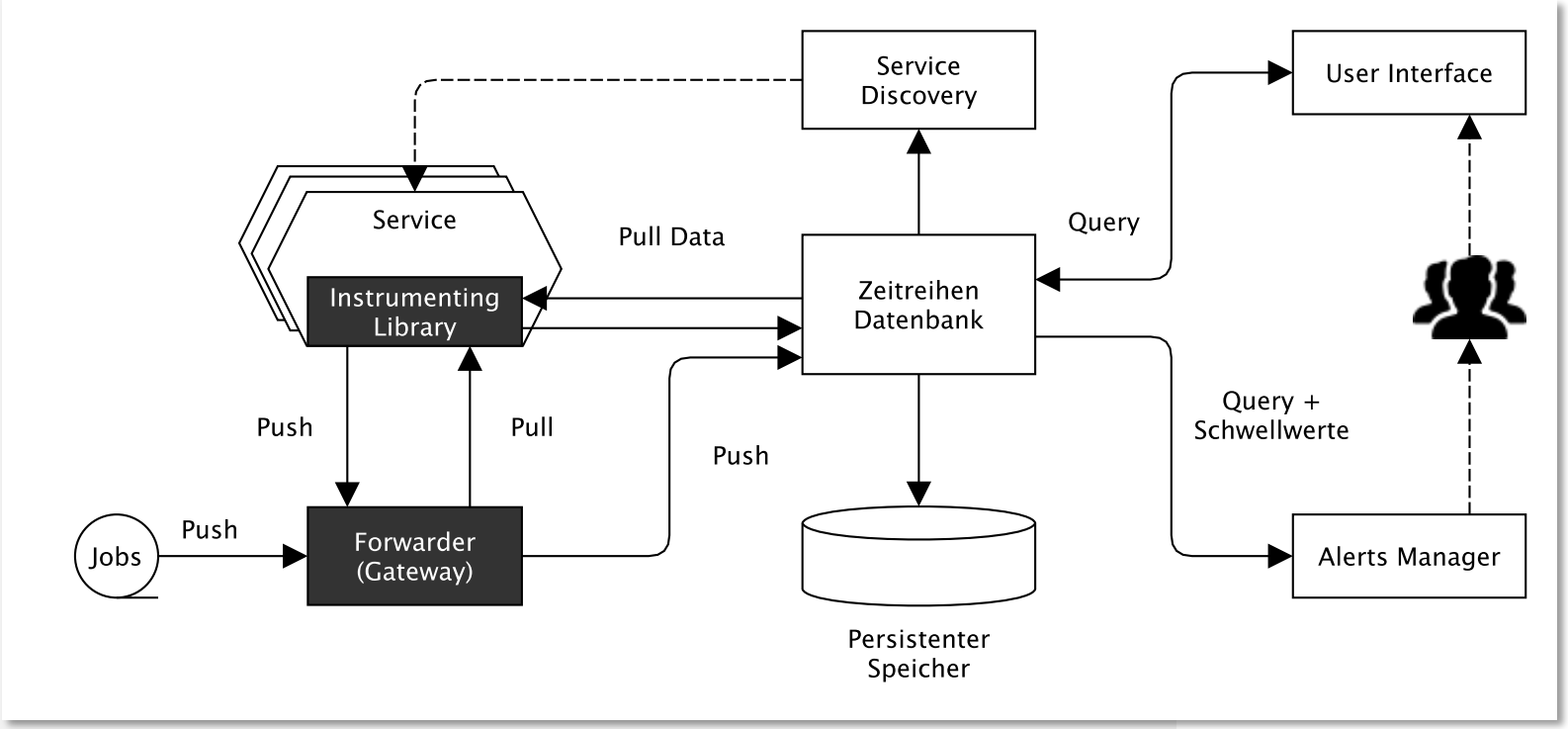
METRICS

Monitoring

Metrik-basierte Monitoring- Lösungen zeichnen Echtzeitmetriken mittels Zeitreihendatenbanken auf, die von Anwendungen und Controllern abgefragt werden und u.a. Echtzeit-Warnmeldungen ermöglichen.

Monitoring-Lösungen bestehen meist aus mehreren Tools:

- Forwarder (auch Exporter oder Gateway genannt), die normalerweise auf überwachten Hosts ausgeführt werden, um lokale Host- und Service-Metriken an einen Zentralspeicher zu exportieren.
- Zeitreihendatenbank zur zentralisierten Speicherung von Metriken.
- Alertmanager, der bei einer Schwellwertüberschreitung Benachrichtigungen verschicken kann.
- Nutzeroberfläche (UI) zum Darrstellen von Dashboards.
- Querying-System, dass zum Erstellen von Dashboards und Warnungen verwendet wird.



Architektur zur Konsolidierung von Telemetriedaten

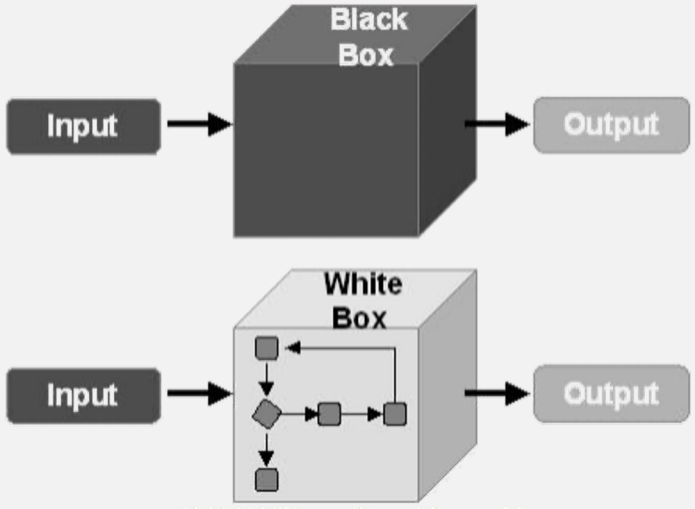
METRICS

White-Box Monitoring

Im Cloud-native Umfeld werden Services oft mittels White-Box-Monitoring überwacht. Hierzu müssen die überwachten Anwendungen (ähnlich wie beim Tracing) instrumentiert werden, um deren interne Metriken als *Exporter* selbst bereitzustellen. Hierzu stehen unterschiedliche Metrik-Exporter Programmbibliotheken für verschiedene Softwareumgebungen zur Verfügung. Es lassen sich somit sowohl Service-spezifische Metriken als auch Black-Box Standardmetriken exportieren, wie die Auslastung des Arbeitsspeicher, CPU, Festplatte oder Netzwerk.

Exporter werden regelmäßig von Monitoring-Lösungen abgefragt. Prometheus fragt diese mittels HTTP meist unter dem Standard URL-Pfad `/metrics` ab. Es werden jedoch auch etablierte Überwachungs- und Verwaltungsprotokolle, wie bspw. SNMP, JMX oder CollectD.

Jede der Datenquellen liefert die aktuellen Werte der Metriken. Eine zentrale Monitoring-Lösung aggregiert dann Daten über die Datenquellen hinweg. Einige Lösungen (wie bspw. Prometheus) verfügen hierzu über eine automatische Service-Discovery, um Ressourcen, die als Datenquellen verwendet werden sollen, automatisch ermitteln zu können. Diese gesammelten Daten werden in einer Datenbank zur Zeitreihenanalyse gespeichert.



White-Box Monitoring
Instrumentierung um Service-spezifische Metriken erfassen zu können.

Black-Box-Monitoring
Standardsystemmetriken sind hingegen ohne Instrumentierung erfassbar.

Überwachung von Maschinen mit Schwerpunkt auf Bereichen wie Speicherplatz, CPU-Auslastung, Speicherauslastung, Load averages usw. betrachten würden.

METRICS

Instrumentierung am Beispiel von Prometheus

```python
from prometheus_client import start_http_server, Summary
import random
import time

# Create a metric to track time spent and requests made.
REQUEST_TIME = Summary('request_processing_seconds', 'Time spent processing request')

# Decorate function with metric.
@REQUEST_TIME.time()
def process_request(t):
    """A dummy function that takes some time."""
    time.sleep(t)

if __name__ == '__main__':
    # Start up the server to expose the metrics.
    start_http_server(8000)
    # Generate some requests.
    while True:
        process_request(random.random())
```

Instrumentierung einer Methode

Metrik erzeugen

Start des Exporters

```
pip3 install prometheus-client
```

Die aktualisierten Metriken werden dann auf Port 8ooo über HTTP bereitgestellt und können von Prometheus abgefragt werden.

Mittels des folgenden Kommandos können Sie das auch selber in der Konsole nachvollziehen:

```
watch curl -s http://localhost:8000
```

METRICS

Übliche Metrikarten

Zähler (Counter)

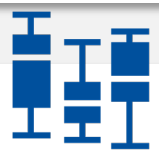
Ein Zähler ist eine kumulative Metrik, die einen einzelnen monoton ansteigenden Zähler darstellt, dessen Wert nur erhöht oder beim Neustart auf Null zurückgesetzt werden kann. Zähler sind geeignet, um die Anzahl eingegangener Requests, erledigter Aufgaben oder aufgetretenen Fehler zu überwachen.

Messung (Gauge)

Ein Messung ist eine Metrik, die einen einzelnen numerischen Wert darstellt, der willkürlich auf und ab gehen kann. Messung werden normalerweise für Messwerte wie Temperaturen oder die aktuelle Speichernutzung verwendet, aber auch für "Zählungen", die steigen und fallen können. Bspw. die Anzahl gleichzeitiger Requests.

Histogramm

Ein Histogramm erfasst Beobachtungen (oft Dinge wie Dauern oder Größen) und zählt sie in konfigurierbaren Bereichen. Histogramme stellen die Verteilung von Beobachtungen ausführlicher dar. Häufig reichen auch einfache Quantilen, um Verteilungen einschätzen zu können (=> Summary).



Zusammenfassung (Summary)

Ähnlich wie bei einem Histogramm werden in einer Zusammenfassung Beobachtungen erfasst. Zusammenfassungen, dienen allerdings zur Berechnung konfigurierbarer Quantile über ein gleitendes Zeitfenster und damit für einfache statistische Auswertungen von Beobachtungsverteilungen.

METRICS

Counter am Beispiel von Prometheus

```python
from prometheus_client import Counter

# Counters go up, and reset when the process restarts
c = Counter('my_failures', 'Description of counter')
c.inc()     # Increment by 1
c.inc(1.6)  # Increment by given value

# There are utilities to count exceptions raised

@c.count_exceptions()
def f():
  pass

with c.count_exceptions():
  pass

# Count only one type of exception
with c.count_exceptions(ValueError):
  pass
```

Instrumentierungsbeispiele, um Zählungen vorzunehmen.

METRICS

Messungen am Beispiel von Prometheus

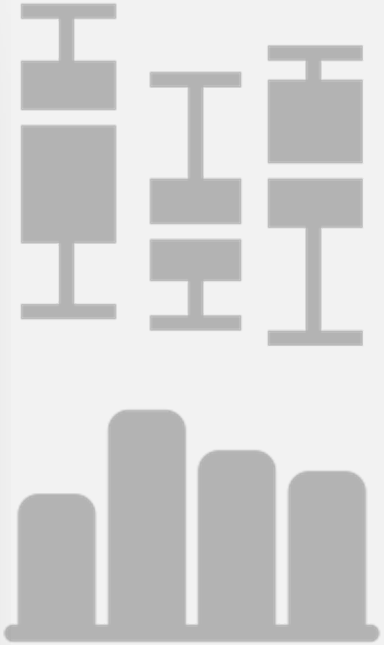
```python
# Gauges can go up and down.

from prometheus_client import Gauge
g = Gauge('my_inprogress_requests', 'Description of gauge')
g.inc()      # Increment by 1
g.dec(10)    # Decrement by given value
g.set(4.2)   # Set to a given value

# There are utilities for common use cases
g.set_to_current_time()   # Set to current unixtime

# Increment when entered, decrement when exited.
@g.track_inprogress()
def f():
  pass

with g.track_inprogress():
  pass

# A Gauge can also take its value from a callback
d = Gauge('data_objects', 'Number of objects')
my_dict = {}
d.set_function(lambda: len(my_dict))
```

Instrumentierungsbeispiele, um Messungen vorzunehmen.

METRICS

Histogramme am Beispiel von Prometheus

```python
# Histograms track the size and number of events in buckets.
# This allows for aggregatable calculation of quantiles.

from prometheus_client import Histogram

h = Histogram('request_latency_seconds', 'Description of histogram')
h.observe(4.7)    # Observe 4.7 (seconds in this case)

# There are utilities for timing code

@h.time()
def f():
  pass

with h.time():
  pass
```



Instrumentierungsbeispiele, um Verteilungen zu bestimmen.

Die Standard-Buckets decken typische Web-/RPC-Requestdauern von Millisekunden bis Sekunden ab. Sie können angepasst werden.

Da die Python Library bei Summarys keine Quantilen bestimmt, ist kein nennenswerter Unterschied zu Histogrammen gegeben. Auf die Darstellung von Summaries wird daher hier verzichtet.

METRICS

Best Practices für die Instrumentierung

- Die Instrumentierung sollte ein integraler Bestandteil Ihres Codes sein.
- In Cloud-nativen Systemen unterscheidet man dabei üblicherweise Online-Systeme und Batch-Systeme.
- Requests sollten einheitlich instrumentiert werden. Dabei ist es empfehlenswert Requests am Ende Ihrer Bearbeitung zu zählen, da dies dann mit den Fehler- und Latenzstatistiken übereinstimmt und in der Regel einfacher zu codieren ist.

Ein Online-System ist ein System, bei dem ein Nutzer oder ein anderes System eine sofortige Reaktion erwartet. Die meisten Datenbank- und Web-Services fallen in diese Kategorie.

Typische Schlüsselmetriken eines Online-Systems sind

- Anzahl der beantworteten Requests
- Anzahl aktuell laufender Requests
- Anzahl aufgetretener Fehler
- Latenz

Batch-Systeme zeichnen sich dadurch aus, dass sie nicht kontinuierlich ausgeführt werden, was für die Metrikerhebung meist ein Push Gateway erforderlich macht.

Typische Schlüsselmetriken eines Batch-Jobs sind

- der Zeitpunkt, der letzten erfolgreichen Ausführung
- Dauer dar Teilschritte eines Jobs
- Gesamtlaufzeit eines Jobs
- Zeitpunkt des letzten Abschluss eines Jobs (erfolgreich oder fehlgeschlagen)
- Gesamtzahl verarbeiteter Datensätze

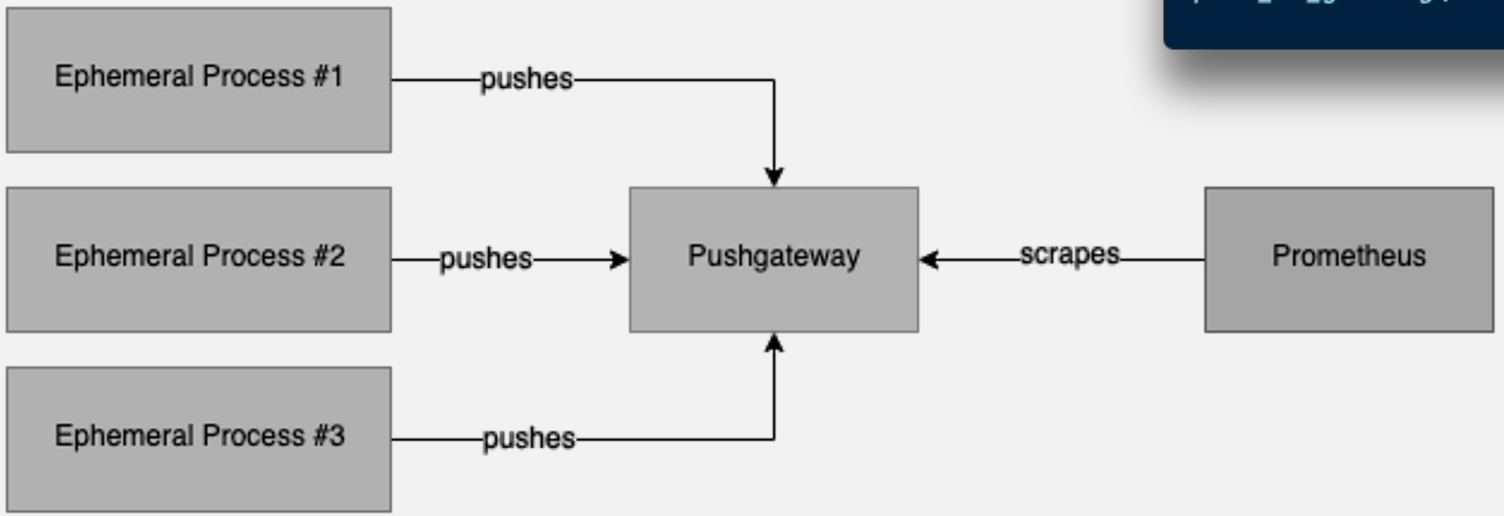
METRICS

Best Practices für die Instrumentierung von Batch-Jobs (Push Gateway)

Da Batch-Jobs keinen Always-On Charakter haben, können diese nicht periodisch von Monitoring-Systemen abgefragt werden.

Über sogenannte Push Gateways können jedoch auch kurzlebige Prozesse und Batch-Jobs, Metriken einem Monitoring-Systemen zugänglich machen.

```python
# The Pushgateway allows ephemeral and batch jobs to expose
# their metrics to Prometheus

from prometheus_client import Gauge, push_to_gateway

g = Gauge('job_last_success_unixtime',
          'Last time a batch job successfully finished',
         )
g.set_to_current_time()
push_to_gateway('localhost:9091', job='batchA')
```



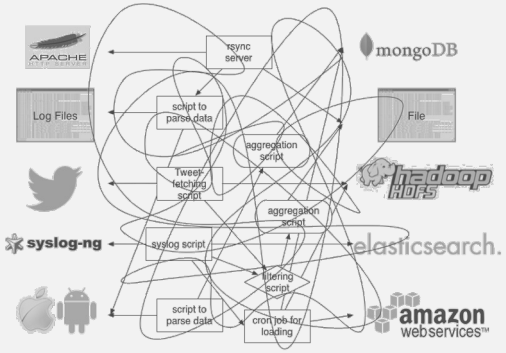
Instrumentierung eines Batch-Jobs am Beispiel von Python und Prometheus.

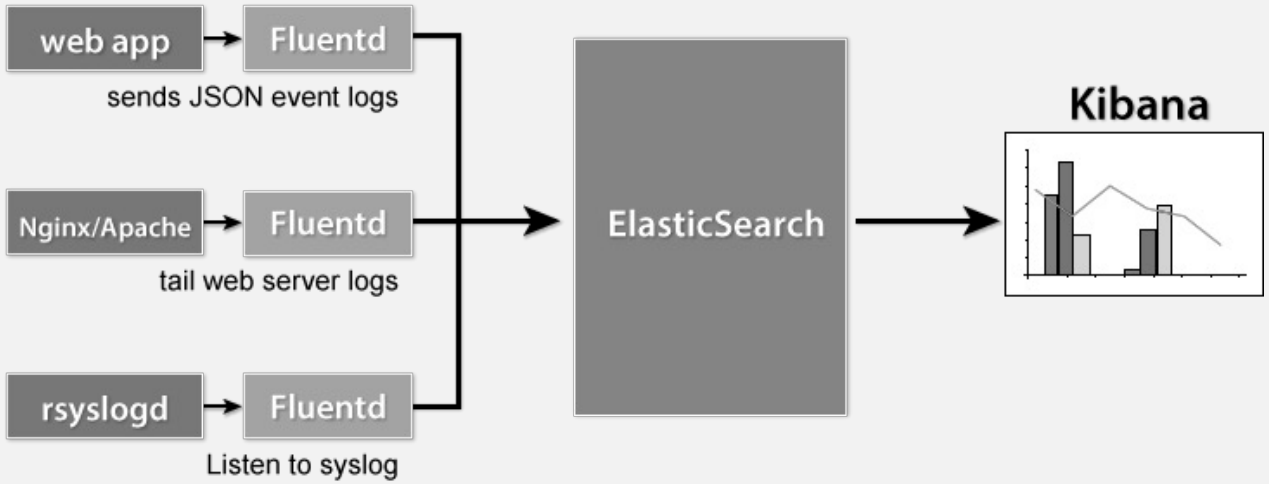
ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

Logging

Mittels **Logging** (Protokollierung) lässt sich Einblick in anwendungsspezifische Nachrichten, die von Prozessen ausgegeben werden, gewinnen.

Das Problem in Cloud-nativen Systemen ist, dass sich Logs über zahlreiche Services verteilen und Service für Service analysiert werden müssen. Bequemer wäre es, wenn diese Logs zentral in einer durchsuchbaren Datenbank für (Fehler-)analysen hinterlegt wären. Daher haben sich Tools zur Log-Aggregation und zentralisierten Log-Analyse etabliert.





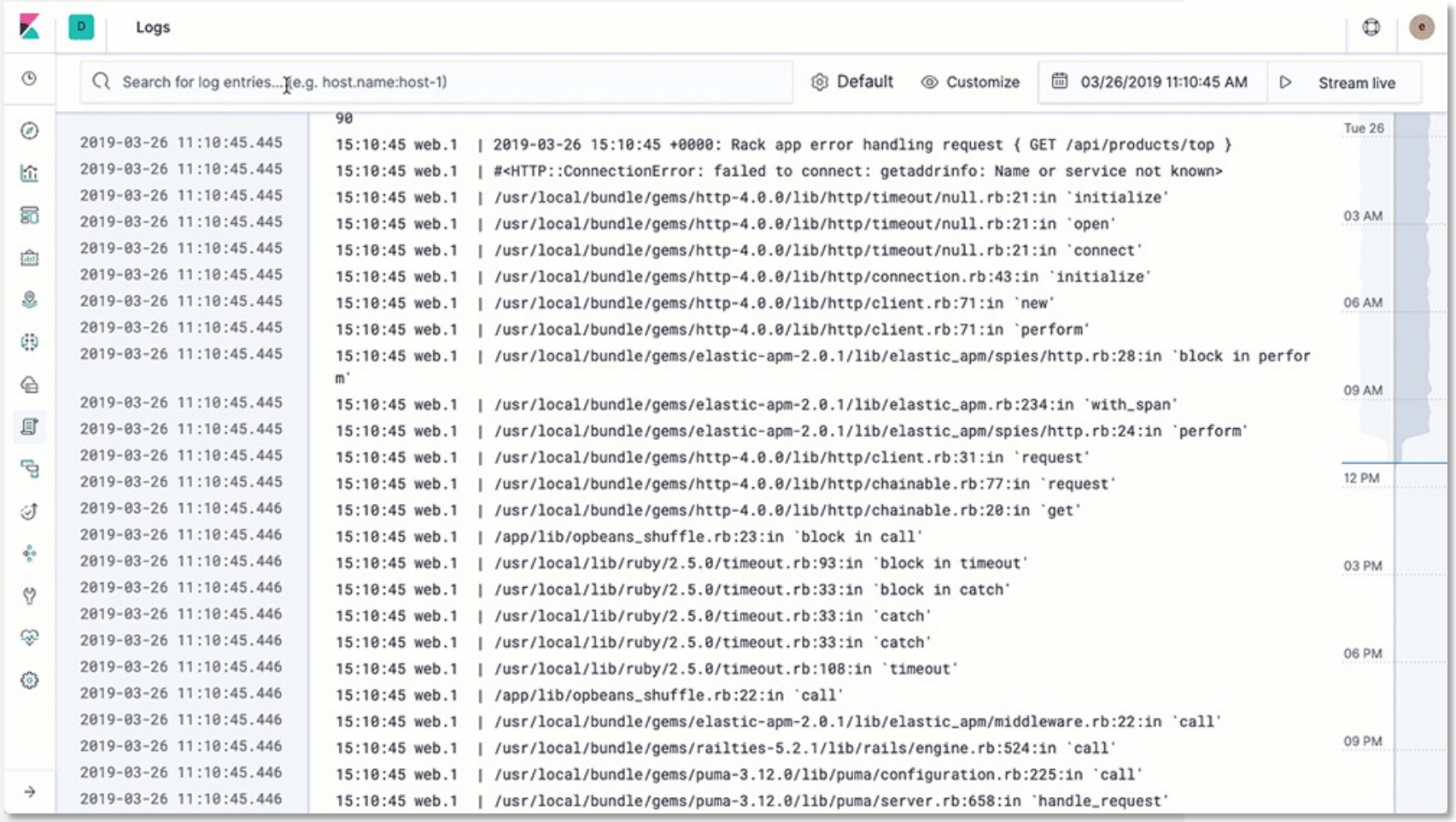
Das Prinzip des Loggens mittels print()-Statements auf der Standard-Ausgabe (stdout) ist so alt wie die Programmierung selbst und daher für viele Programmierer so verbreitet, dass diese kaum bewusst wahrnehmen, dass Code für eine bessere Observierbarkeit instrumentiert wird.

LOGGING

Konsolidierung von Logs mittels Unified Logging

Mittels eines Unified Logging Layer lassen sich alle Logs eines Cloud-nativen Systems zentralisiert durchsuchen. In entsprechenden Datenbanken, wie bspw. ElasticSearch, lassen sich Log-Daten nach Dienst, App, Host, Rechenzentrum weiteren Kriterien filtern, sodass man in den aggregierten Logs nach unerwarteten Ereignissen fahnden kann.

Die Konsolidierung von Logs kann vereinfacht werden, wenn alle Services einheitlich und einfach Ereignisse protokollieren.



Beispiel: ElasticSearch, LogStash und Kibana (ELK-Stack)

Quelle: elastic.io

LOGGING

Logs als Stream von Ereignissen betrachten (12 Faktor Prinzip, Nr. 11!!!)

Logs machen das Verhalten einer laufenden App sichtbar. Oft werden Logs in Dateien auf Platte geschrieben (eine Logdatei).

In Cloud-nativen Systemen werden Logs von Services aber auch häufig als Stream von Ereignissen ungepuffert auf stdout geschrieben. Log Router (wie LogStash, Fluentd u. ähnl.) fassen die stdout-Streams aller Prozesse eines Hosts zusammen und leiten diese an Zeitreihen-Datenbanken zur Archivierung weiter. Diese Archivierungsziele sind für die App weder sichtbar noch konfigurierbar - sie werden vollständig von der Laufzeitumgebung aus verwaltet.

Logs sind der Stream von aggregierten, nach Zeit sortierten Ereignissen und zusammengefasst aus den Output Streams aller laufenden Prozesse und Services. **Logs in ihrer rohen Form sind üblicherweise ein Textformat mit einem Ereignis pro Zeile.**

Mittels dieser Zeitreihen-Datenbanken (z.B. ElasticSearch) kann das Verhalten einer App flexibel und automatisiert beobachtet werden. Dies schließt ein:

- Bestimmte Ereignisse in der Vergangenheit zu finden
- Umfangreiche graphische Darstellungen (wie Requests pro Minute)
- Aktives Alarmieren aufgrund benutzerdefinierter Heuristiken (wie ein Alarm wenn die Anzahl von Fehlern pro Minute eine gewisse Grenze überschreitet)

Log Routing

Log Indexing + Querying

Log Analysis => Ops-Erkenntnisse

ERINNERUNG: 12 FAKTOREN METHODE

XI. Logs

```
print("Ich bin ein Log-Entry in Python")
```

Sehr einfaches Logging (log to stdout), kein Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren zu können)

```
import json
print(json.dumps({
        "event": "Log-Entry",
        "message": "Ich bin ein Log-Entry",
        "language": "Python"
}))
```

Sehr einfaches strukturiertes Logging (log JSON to stdout), ebenfalls ohne Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren und zielgerichteter auswerten zu können in einer Log-Konsolidierung)

CLOUD NATIVE COMPUTING

Merke:
Komplizierter muss es mit einer funktionierenden Log-Konsolidierung nicht unbedingt sein.

LOGGING

Übliche Log-Level

Da das Loggen aller Ereignisse die für Logging verfügbaren Ressourcen innerhalb kurzer Zeit aufbrauchen kann und die Auffindbarkeit bestimmter Ereignisse erschweren würde, werden meist die folgenden üblichen Dringlichkeitsstufen genutzt, mittels derer das Logging von Events bspw. in Production-, Staging-, Test-Environments ein- oder ausgeschaltet werden kann.

Überlicherweise werden nur Ereignisse bis zu einem Log Level dann auch geloggt.

Die Log-Ebene Info bedeutet also, das Ereignisse der Kategorien Fatal, Error, Warning und Info geloggt würden. Aber nicht die Ebenen Debug und Trace.

Log Level	Beschreibung
Fatal	Fehler, welcher zur Terminierung der Anwendung führt.
Error	Laufzeitfehler, welcher die Funktion der Anwendung behindert, oder unerwarteter Programmfehler.
Warning	Aufruf einer veralteten Schnittstelle, fehlerhafter Aufruf einer Schnittstelle, Benutzerfehler oder ungünstiger Programmzustand.
Info	Laufzeitinformationen wie der Start und Stopp der Anwendung, Benutzeranmeldungen und -abmeldungen, sowie durchgeführte Geschäftstransaktionen.
Debug	Informationen zum Programmablauf. Wird im Normalfall nur in der Entwicklung oder zur Nachvollziehung eines Fehlers verwendet.
Trace	Detaillierte Verfolgung des Programmablaufs, insbesondere zur Nachvollziehung eines Programmierfehlers.

LOGGING

Best Practices für die Logging Instrumentierung am Beispiel von Python

Die Standardbibliothek von Python enthält bereits ein eingebautes und flexibles Logging-Modul, mit dem verschiedene Logging-Konfigurationen für unterschiedliche Logging-Erfordernisse erstellt werden können. Diese ist für Unified Logging absolut ausreichend. Viele andere Sprachen wie bspw. Java bieten ähnliche Logging Bibliotheken wie bspw. log4j.

Das Logging-Modul von Python besteht aus Funktionen, die es Entwicklern ermöglichen Logs für verschiedene Zielorte (Stdout, Log-Dateien, Remote-Server, etc.) anzulegen. Hierzu kann man verschiedene Handler anlegen. Protokoll-Events werden an die entsprechenden Handler weitergeleitet und entsprechend verarbeitet.

```
import logging
logging.basicConfig(level=logging.WARNING)

logging.debug('This debug message will not be logged')
logging.info('This info message will not be logged')
logging.warning('This warning message will be logged')
logging.error('This error message will be logged')
logging.critical('This critical message will be logged')
```

LOGGING

Best Practices für die Logging Instrumentierung am Beispiel von Python (Log Level)

Das Python-Logging-Modul bietet grundsätzlich weniger (und stellenweise auch anders benannte) Stufen als andere Logging-Bibliotheken. Dies macht die Dinge allerdings auch etwas einfacher, indem so einige potenzielle Unklarheiten beseitigt werden.

Log Level	Beschreibung
DEBUG	Dieses Level sollte ausschließlich für Debugging-Zwecke in der Entwicklung verwendet werden.
INFO	Dieses Level sollte genutzt werden, wenn etwas Interessantes - aber Erwartetes - passiert (z. B. wenn ein Kunde einen Kauf in einem Online-Shop tätigt).
WARNING	Diese Ebene sollte verwendet werden, wenn etwas Unerwartetes oder Ungewöhnliches passiert. Es handelt sich nicht um einen Fehler, aber Sie sollten darauf achten. Z.B. wenn ein Nutzer sich mit falschen Credentials anzumelden versucht.
ERROR	Diese Stufe ist für Dinge, die schief laufen, aber normalerweise wiederherstellbar sind (z. B. interne Ausnahmen wie nicht vorhandene Dateien, die aber behandelt werden können, oder APIs, die Fehlerergebnisse zurückgeben).
CRITICAL	Diese Stufe sollten Sie nur in Situtationen verwenden, die Services unbrauchbar machen und zu einem Shutdown führen.

LOGGING

Best Practices der Logging Instrumentierung am Beispiel von Python

```python
import logging
import os

logging.basicConfig(
    level=os.getenv('LOGLEVEL', logging.WARNING),
    format='%(asctime)s | %(name)s | %(levelname)s | %(message)s',
    datefmt='%Y-%m-%dT%H:%M:%S%z'
)

log = logging.getLogger("service-name")

log.debug('This debug message will not be logged')
log.info('This info message will not be logged')
log.warning('This warning message will be logged')
log.error('This error message will be logged')
log.critical('This critical message will be logged')
```

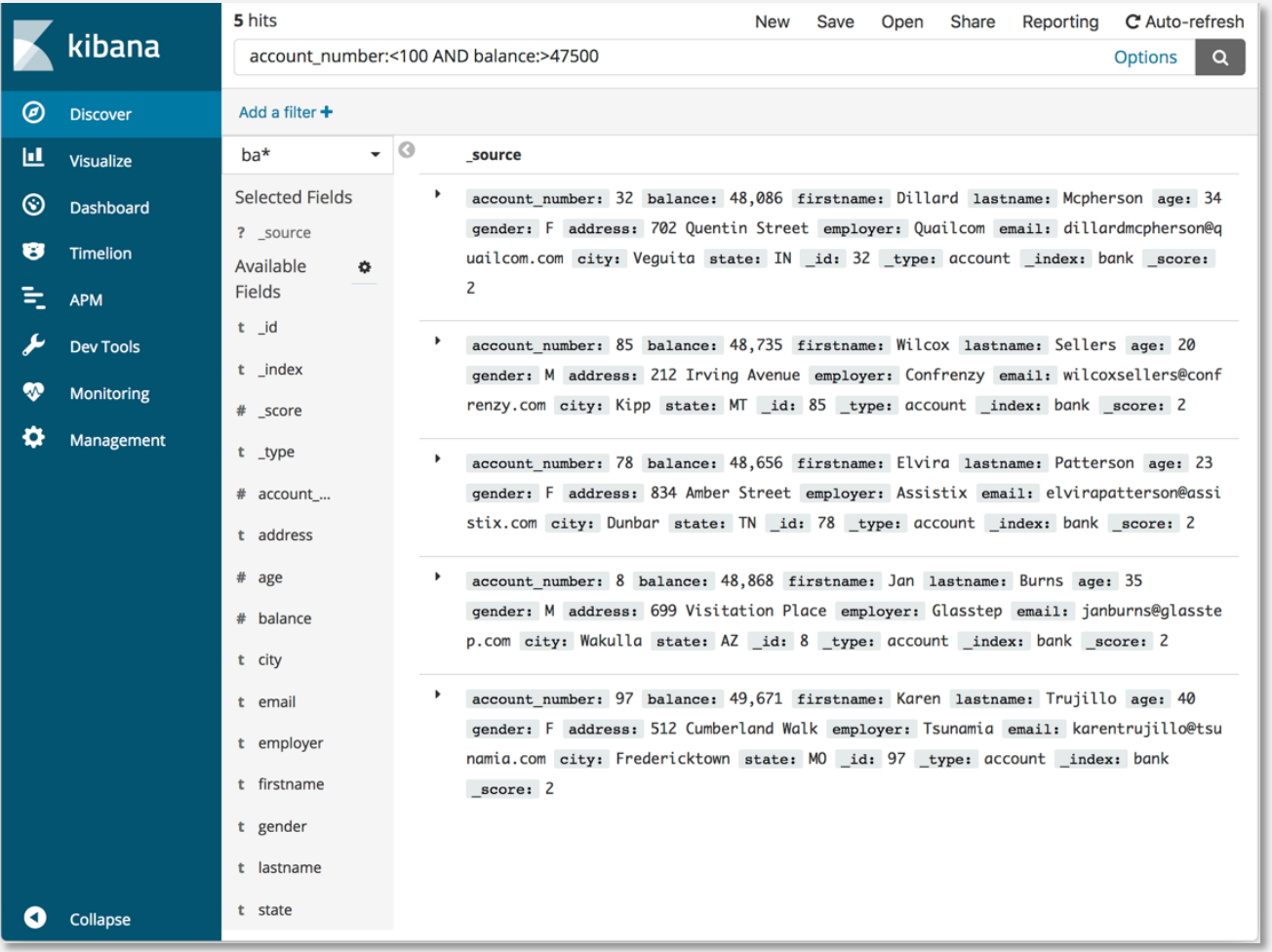
(1) (d.h. nicht in eine Datei!)
(2)
(3)
(4)
(5)

1. Logge auf stdout (das Unified Logging System + die Plattform macht den Rest)
2. Definiere den Loglevel über eine Umgebungsvariable
3. Logge immer auch den Service-Namen in dem Ereignis ausgelöst wurde
4. Logge den Zeitpunkt, den Level und das Event
5. Logge Zeitpunkte im ISO8601 Format inkl. Zeitzone

```
2021-02-20T12:33:46+0100 | service-name | WARNING | This warning message will be logged
2021-02-20T12:33:46+0100 | service-name | ERROR | This error message will be logged
2021-02-20T12:33:46+0100 | service-name | CRITICAL | This critical message will be logged
```

LOGGING

Abfragen in zentralisierten Logs (Discover Data mittels Queries)



Logs werden oft als JSON-Dokumente in zentralen Zeitreihen-/Dokumenten-Datenbanken gespeichert und können dann dort mittels Queries durchsucht werden.

Nachfolgendes Beispiel sucht bspw. nach Log-Einträgen des Services „my-service" in denen die Begriffe „failed" und „login" auftauchen (weil man vielleicht nach verdächtigen Login-Aktivitäten sucht).

```
{
  "query": {
    "match": {
      "phrase": {
        "query": "WARNING my-service failed login",
        "operator" : "AND"
      }
    }
  }
}
```

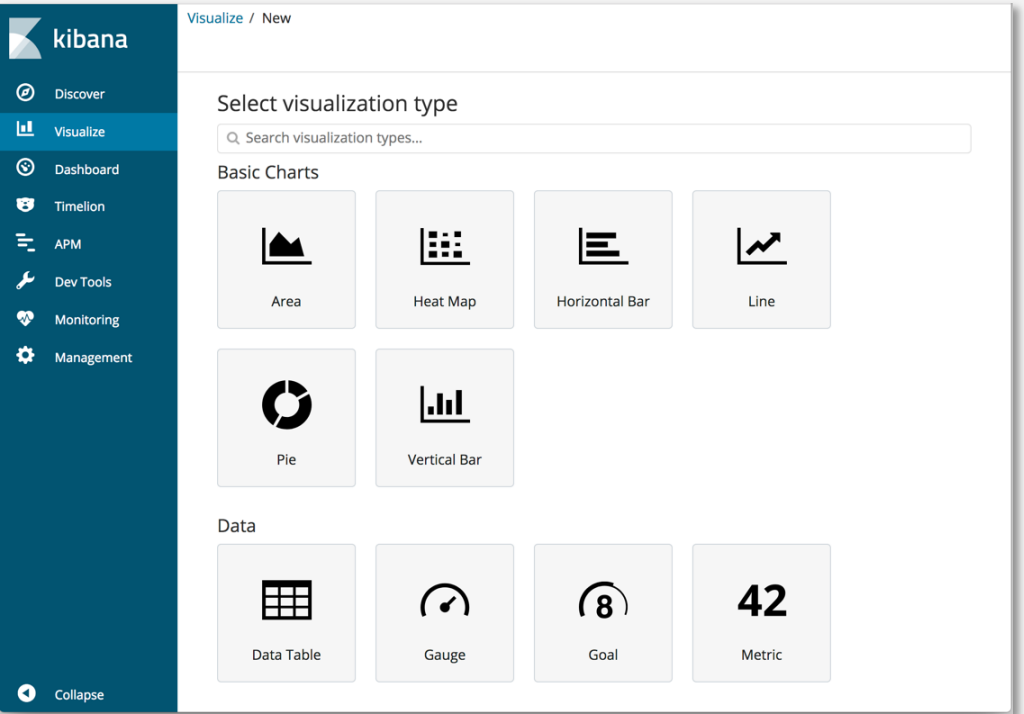
Quelle: elastic.io

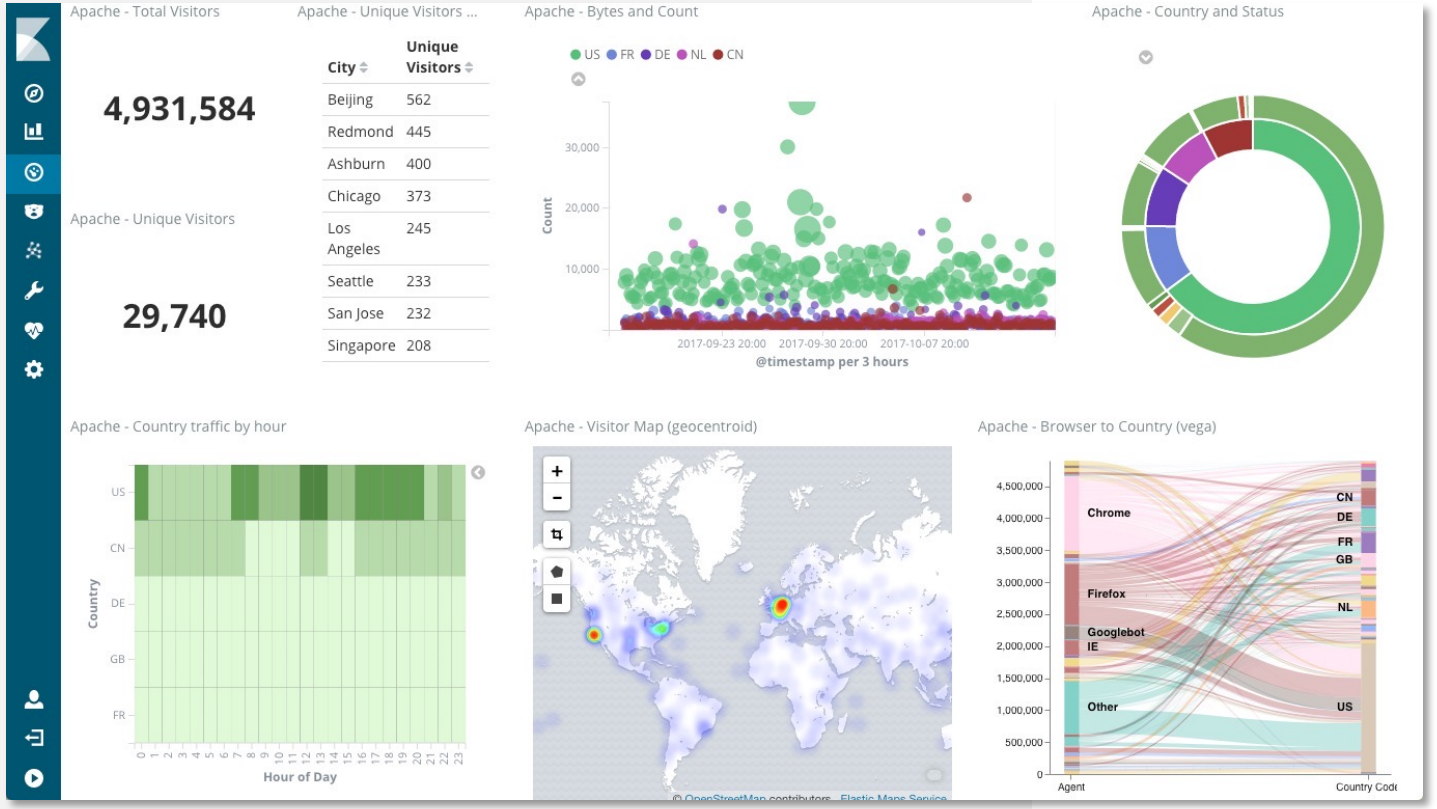
LOGGING

Visualisieren von Logs

Mittels Queries lassen sich auch Häufigkeiten von Treffern in Form von Diagrammen, Karten und andere Arten von Visualisierungen unter Verwendung der zentralisierten Log-Daten erstellen.

Um Karten zu erstellen, sollten Logs Informationen wie geografische Koordinaten enthalten. Für quantitative Darstellungen sollten Logs numerische Daten beinhalten.





Quelle: elastic.io

Derartige Query-basierte Visualisierungen werden häufig in Dashboards zusammengefasst, um ein ganzheitliches Bild von Log- und Monitoring-Daten aufzubereiten, die es im Betrieb (Ops) ermöglichen Echtzeit-Darstellungen von Systemzuständen und -verhalten darzustellen (z.B. eine Häufung von fehlerhaften Login-Versuchen).

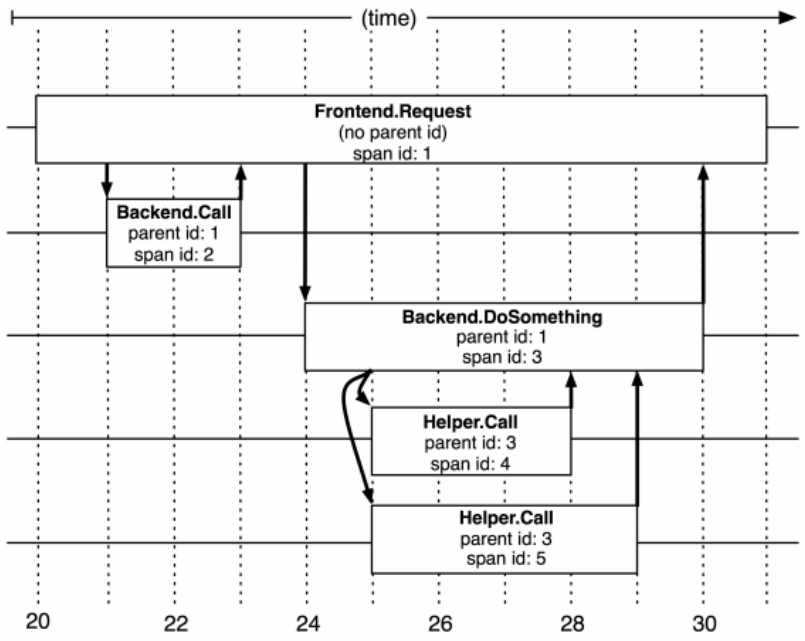
ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

Tracing

Verteilte Tracing Systeme ermöglichen die Rückverfolgung von Service Interaktionen im laufenden Betrieb und helfen beim Erfassen von Zeitreihendaten (Traces), die u.a. zur Behebung von Latenzproblemen in Servicearchitekturen hilfreich sein können. Zu den Funktionen gehören üblicherweise das Erfassen, die Auswertung von Traces und die Suche nach Traces in größeren Traces-Datensätzen.

Mit der Kenntnis einer Trace-IDs kann man einzelne Vorgänge nachvollziehen. Traces lassen sich aber auch meist mittels Abfragen anhand von Attributen wie bspw. Service-Name, Operationsname, Tags und Labels oder Dauern bestimmen.

Oberflächen von Tracing-Systemen zeigen meist Abhängigkeitsdiagramme an, um das Gesamtverhalten einschließlich Fehlerpfaden oder Aufrufabfolgen von Services einfacher in Analysen erfassen zu können.



Quelle: Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag; **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**, Google Technical Report, 2010
https://research.google/pubs/pub36356.pdf

KLASSEN VON TRACING-SYSTEMEN

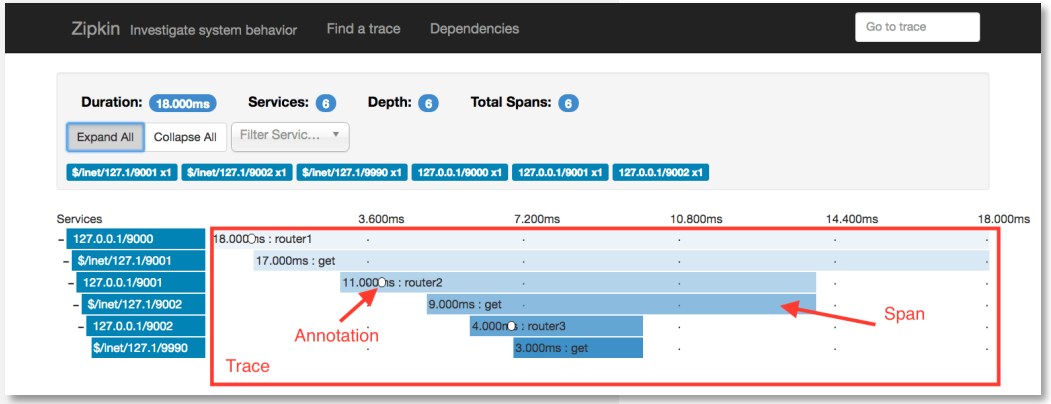
Black-Box vs. Annotation-basierte Tracing Propagation

Jedes Tracing-System benötigt eine Möglichkeit, den Kausalzusammenhang zwischen Aktivitäten in vielen unterschiedlichen Services (Prozessen) zu ermitteln. Das Problem hierbei ist, das die Interaktion extrem heterogen ist. Services können über RPC-Frameworks, Pub-Sub-Systeme, generische Nachrichtenwarteschlangen, direkte HTTP-Aufrufe, UDP-Pakete oder gänzlich anders mit einander interagieren.

Man kann dabei Tracing-Systeme hinsichtlich ihrer Strategien unterscheiden, wie Kausalzusammenhänge hergestellt werden:

- **Black-Box Tracing Systeme** (wie bspw. Project5, Wap5, **The Mystery Machine**) stellen Zusammenhänge zwischen Services her, indem deren Netzwerkverkehr beobachtet wird und daraus Kausalzusammenhänge ermittelt werden. Der Vorteil ist, das Services nicht instrumentiert werden müssen. Allerdings muss man einen Kompromiss zwischen der Beobachtungsdauer und der Qualität abgeleiteter Traces eingehen. Resultierende Traces sind ferner mit Unsicherheit behaftet.

- **Annotations-basierte Tracing Systeme** sind aufwändiger im Betrieb, da zu tracende Services instrumentiert werden müssen. Hierzu müssen die Codierung der Services „angefasst" werden, um Aufrufe zwischen Systemen für das Tracing-System zu kennzeichnen. Allerdings lassen sich so die Kausalzusammenhänge zweifelsfrei und unmittelbar ermitteln. Im Cloud-native Umfeld hat sich daher dieser Ansatz durchgesetzt, der insbesondere durch die **X-Trace** und **Dapper**-Paper 2010 terminologisch geprägt wurde.



Michael Chow and David Meisner and Jason Flinn and Daniel Peek and Thomas F. Wenisch; **The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services**, 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014
https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

Rodrigo Fonseca and George Porter and Randy H. Katz and Scott Shenker; **X-Trace: A Pervasive Network Tracing Frameworks**, 4th USENIX Symposium on Networked Systems Design \& Implementation (NSDI 07), 2007
https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag; **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**, Google Technical Report, 2010
https://research.google/pubs/pub36356.pdf

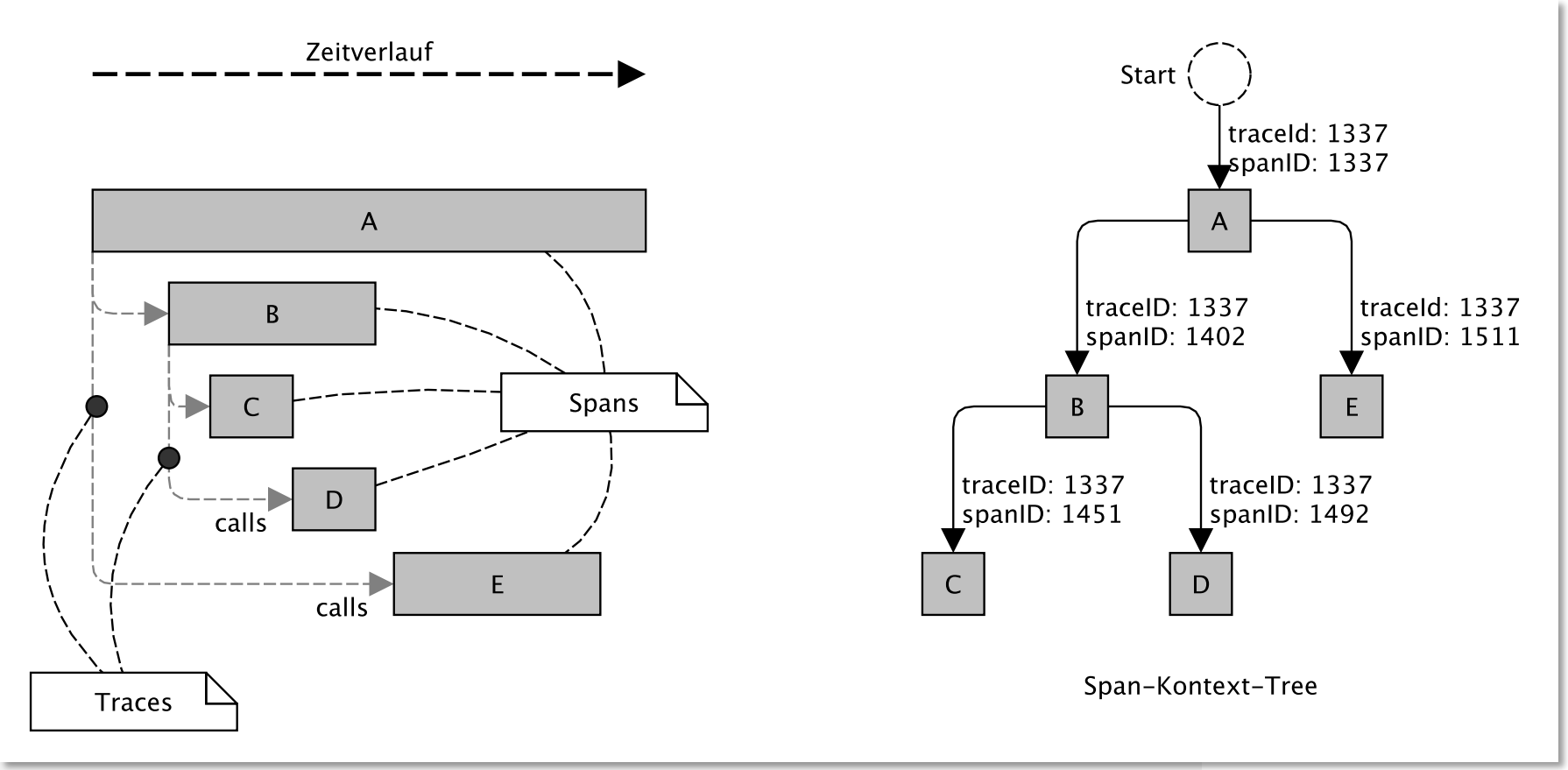
TRACING

Begrifflichkeiten: Traces + Spans

Trace: Die Beschreibung einer Transaktion, die sich durch ein verteiltes System bewegt.

Span: Eine benannte, zeitgesteuerte Operation, die einen Teil des Workflows darstellt. Spans können mit Daten annotiert werden (Log), die zur Verfolgung und Auswertung von Transaktionen erforderlich sind.

Span-Kontext: Trace-Informationen, die die verteilte Transaktion zwischen Services begleiten. Der Span-Kontext enthält die Trace-ID, die Span-ID und alle anderen Daten, die das Tracing-System zur Verfolgung von Transaktionen in nachgeschalteten Services benötigt.



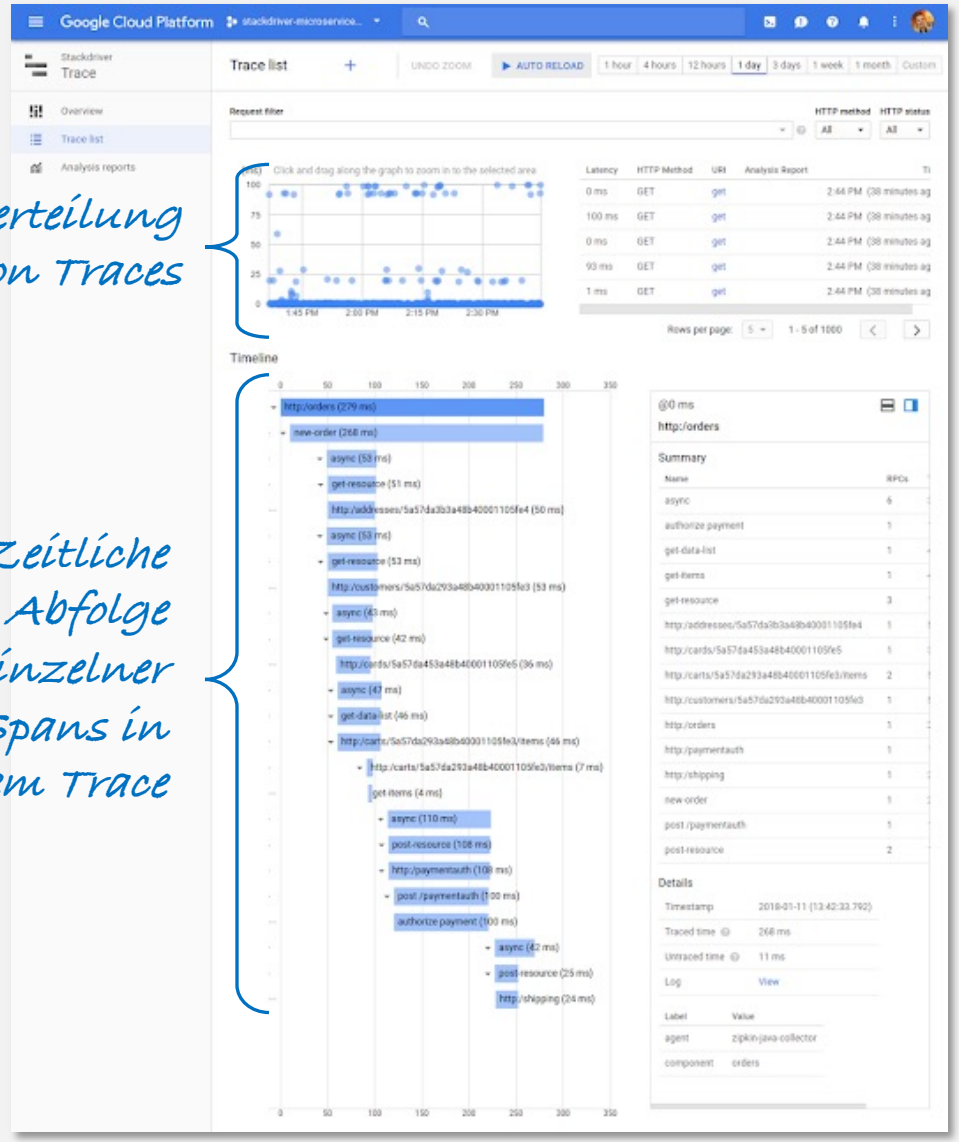
Quelle: OpenTracing.io

TRACING

Begrifflichkeiten: Traces + Spans

Beispiel eines Managed Services
Google Trace:

„Cloud Trace ist ein System für verteiltes Tracing, mit dem die Latenzdaten von Anwendungen erfasst und in der Google Cloud Console angezeigt werden. So kann man den Ablauf von Anfragen in Anwendungen verfolgen und detaillierte Leistungsdaten echtzeitnah erhalten. Mit Cloud Trace werden automatisch alle Traces eine Anwendung analysiert, um detaillierte Latenzberichte zu generieren, um bspw. Leistungsverschlechterungen zu erkennen. Außerdem können Sie damit Traces von allen Ihren VMs, Containern oder App Engine-Projekten erfassen.“



Beispiel eines Managed Tracing Services (Google Trace)

Quelle: https://cloud.google.com

TRACING

Instrumentierung

```python
from opentracing_instrumentation.request_context import get_current_span, span_in_context

def init_tracer(service):
    logging.getLogger('').handlers = []
    logging.basicConfig(format='%(message)s', level=logging.DEBUG)
    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name=service,
    )
    return config.initialize_tracer()

def check_cinema(movie):
    with tracer.start_span('CheckCinema', child_of=get_current_span()) as span:
        with span_in_context(span):
            num = random.randint(1,30)
            time.sleep(num)
            cinema_details = "Cinema Details"
            flags = ['false', 'true', 'false']
            random_flag = random.choice(flags)
            span.set_tag('error', random_flag)
            span.log_kv({'event': 'CheckCinema' , 'value': cinema_details })
            return cinema_details
```

Tracing Initialisierung

Tracing Instrumentierung

Code muss „instrumentiert" werden, um Trace-Daten an Verteilte Tracing-Systeme zu melden. Dies bedeutet normalerweise die Konfiguration und Einbettung von Tracing-Code unter Nutzung von Instrumentenbibliotheken für spezifische Tracing-Systeme.

Da die Instrumentierung dann leicht Tracing-System-spezifisch werden kann, entwickeln sich auch Tracing-Standards wie bspw. Open-Tracing API um zu vermeiden, das eine Instrumentierung zwar mit ZIPKIN aber nicht mit Jaeger funktioniert.

CLOUD NATIVE COMPUTING

Näheres dazu im Lab

OPENTRACING

TRACING

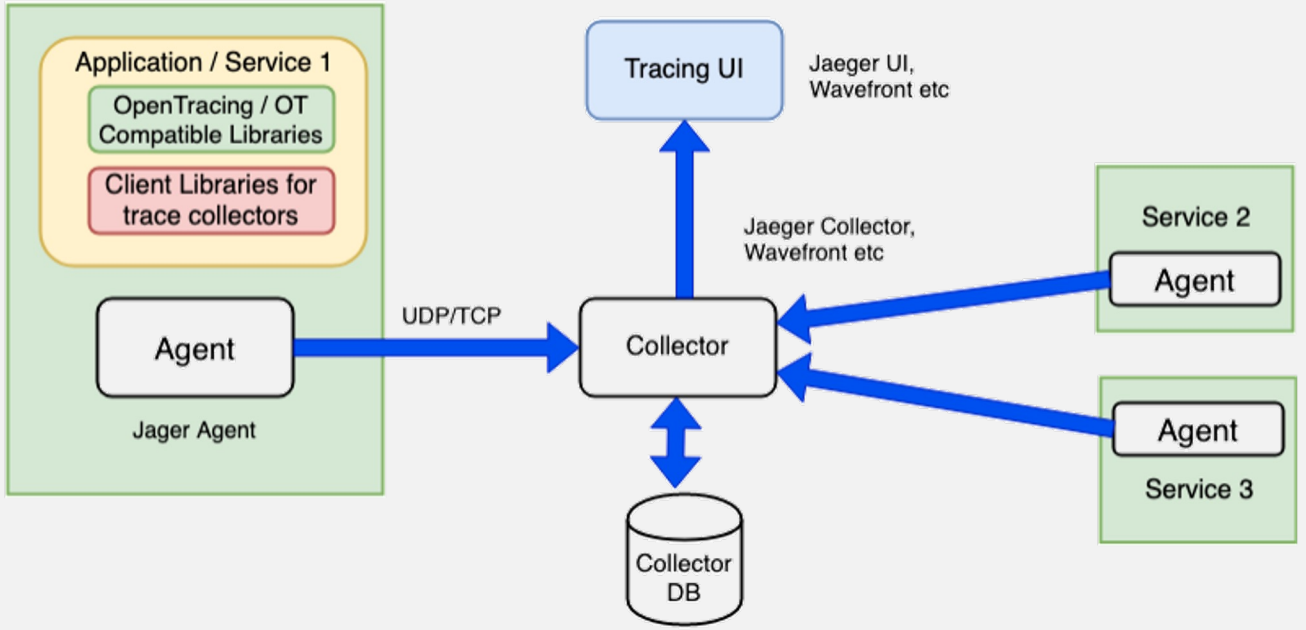
Wie Service Meshs Tracing unterstützen können?

Wie bei Service Meshs beruhen auch viele Tracing-Systeme auf Agenten (im Service Mesh Sprachgebrauch Proxy).

Hier am Beispiel des Jaeger Tracing Systems dargestellt.

Das Prinzip ist also wie bei Service Meshs alle Kommunikation mit dem Tracing Collector erst einmal durch ein Sidecar zu geben. Der Agent kann dann die Trace-Daten an einen Collector weitergeben, ohne dass der Service weiß, dass er gerade getraced wird bzw. die Adresse des Tracing-Service kennen muss.

Die Injection solcher Agents erfolgt wie bei Service Meshs auch.



High Level Tracing Architektur am Beispiel von Jaeger

Grundsätzlich ist also denkbar, das Service Meshs auch Tracing Funktionalität perspektivisch mit abdecken könnten. Die Deployment Prinzipien sind sehr ähnlich. Aktuell wird Tracing meist nicht als Feature von Service Meshs gesehen.

TRACING

Überblick über Verteilte Open Source Tracing-Systeme

A distributed tracing system to gather timing data needed to troubleshoot latency problems in service architectures.

Open source, end-to-end distributed tracing system to monitor and troubleshoot transactions in complex distributed systems

CNCF (graduated)

Vendor-neutral APIs and instrumentation for distributed tracing

CNCF (Incubating)

TRACING

Überblick über Managed Distributed Tracing-Systeme

Google Cloud Trace

und viele mehr ....

ISTIO

Nur Versuch macht kluch ...

GitLab

CLOUD NATIVE COMPUTING

Klonen Sie dieses Repository:

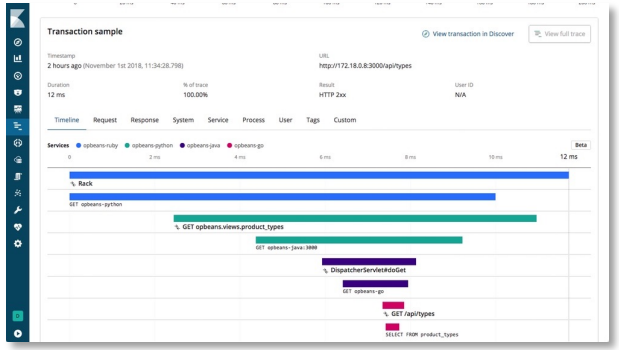
```
git clone https://git.mylab.th-luebeck.de/cloud-native/lab-tracing.git
```

Elasticsearch

Kibana

Elastic APM

> Tracing von Service Calls
> Instrumentierung
> Propagieren von Traces



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de