qaware.de

QA|WARE
SOFTWARE ENGINEERING

Service Meshes in modernen Microservice Architekturen

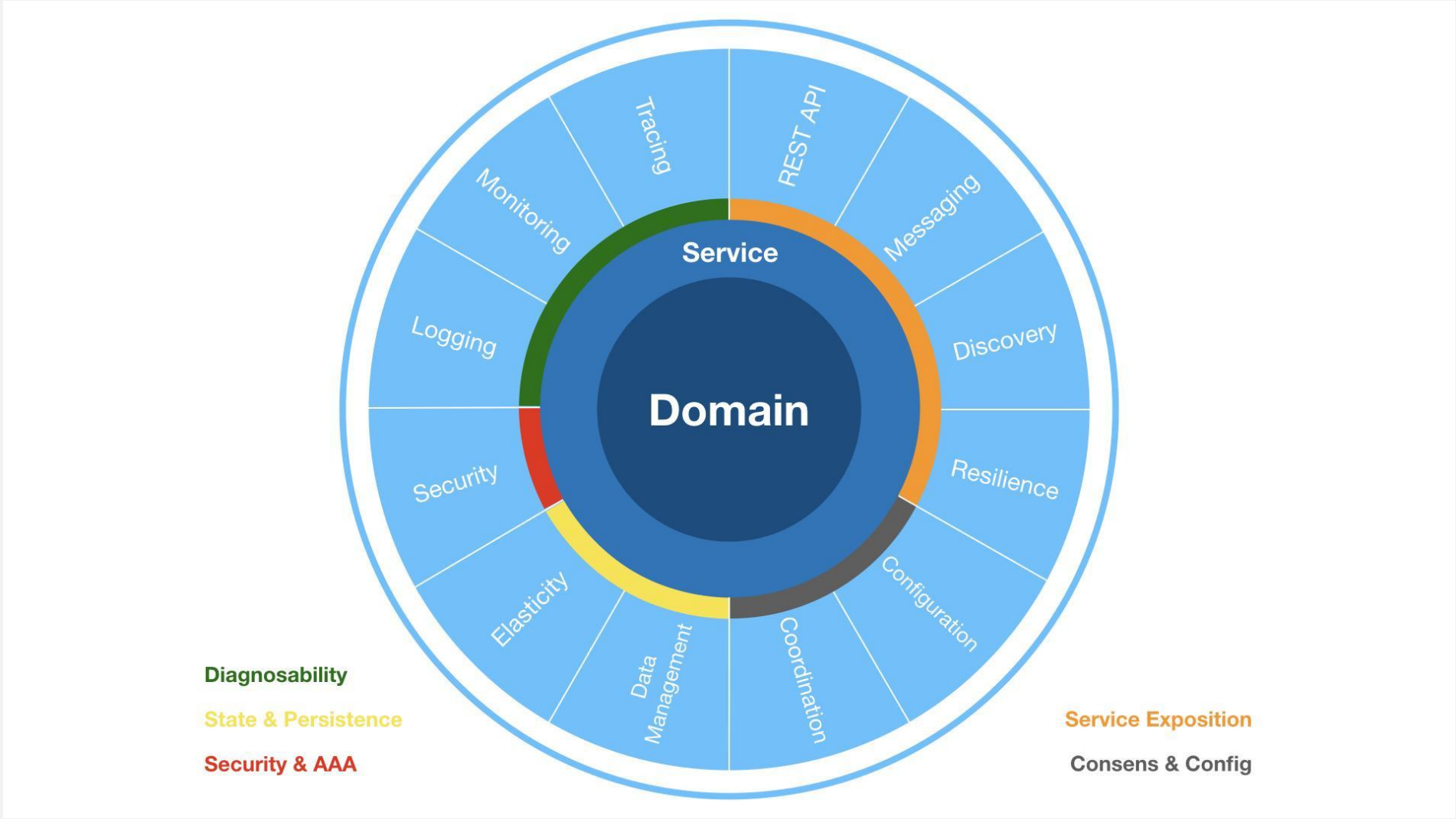
Lukas Buchner
lukas.buchner@qaware.de

Recap

Die Anwendung ist “fertig” gebaut

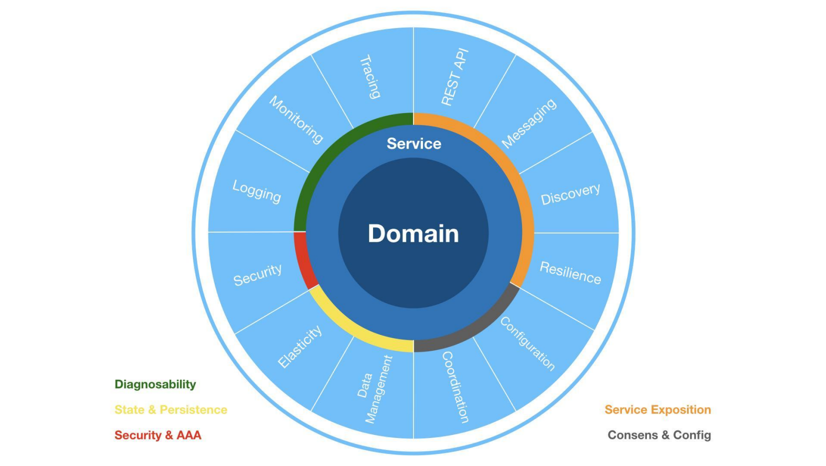
Zumindest fast…

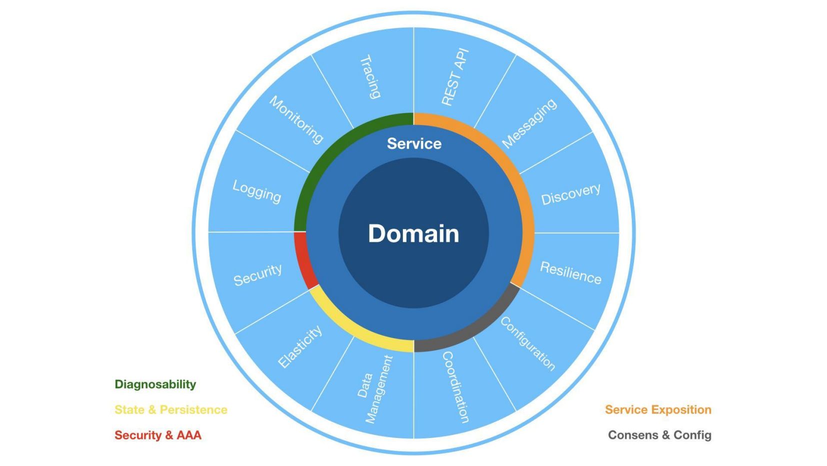
Technische Aspekte von Microservices

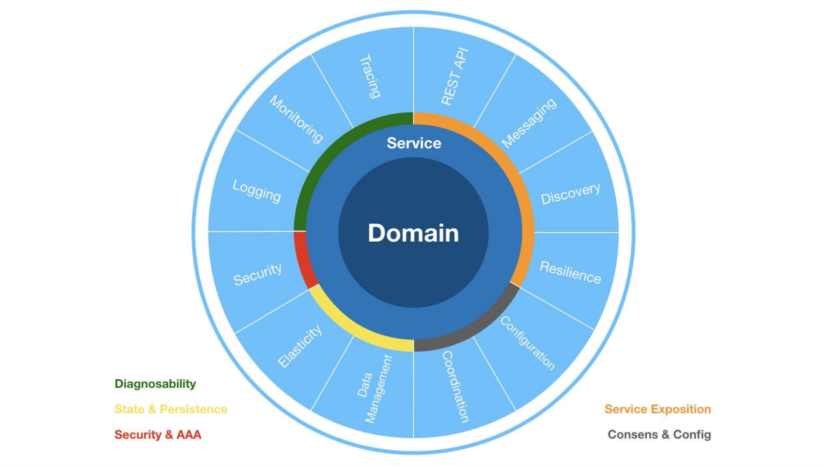


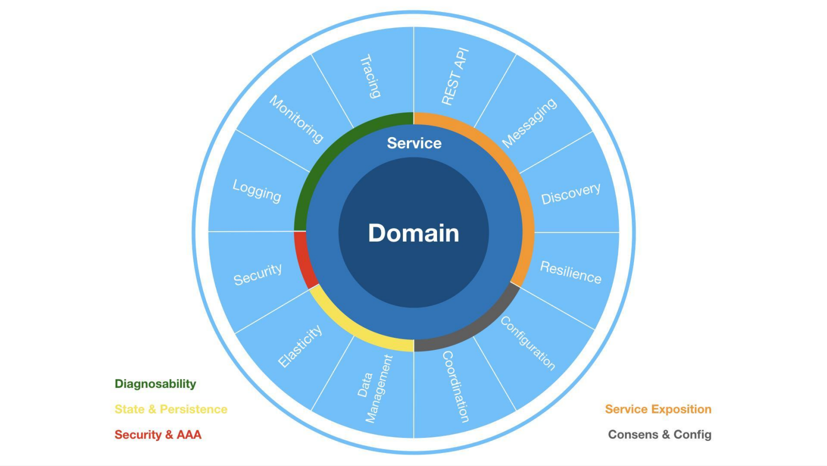
Der technische Anteil wächst bei vielen services

QA|WARE









Service Meshes

Was ist ein Service Mesh?

*In software architecture, a **service mesh** is a dedicated infrastructure layer for facilitating service-to-service communications between services or microservices using a proxy.*

A dedicated communication layer can provide numerous benefits, such as providing observability into communications, providing secure connections or automating retries and backoff for failed requests.

Ein Service Mesh bietet uns…

Traffic Management

- Load Balancing
- Service Discovery
- Routing
- Retries
- Timeouts
- Circuit Breaker

Security

- Encryption
- Authentication and Authorization
- Rate Limiting

Observability

- Metrics
- Tracing
- (Logging)

Die wichtigsten Service Meshes in Kubernetes

- Istio
- Linkerd
- Cilium
- OpenServiceMesh
- AppMesh (AWS only)

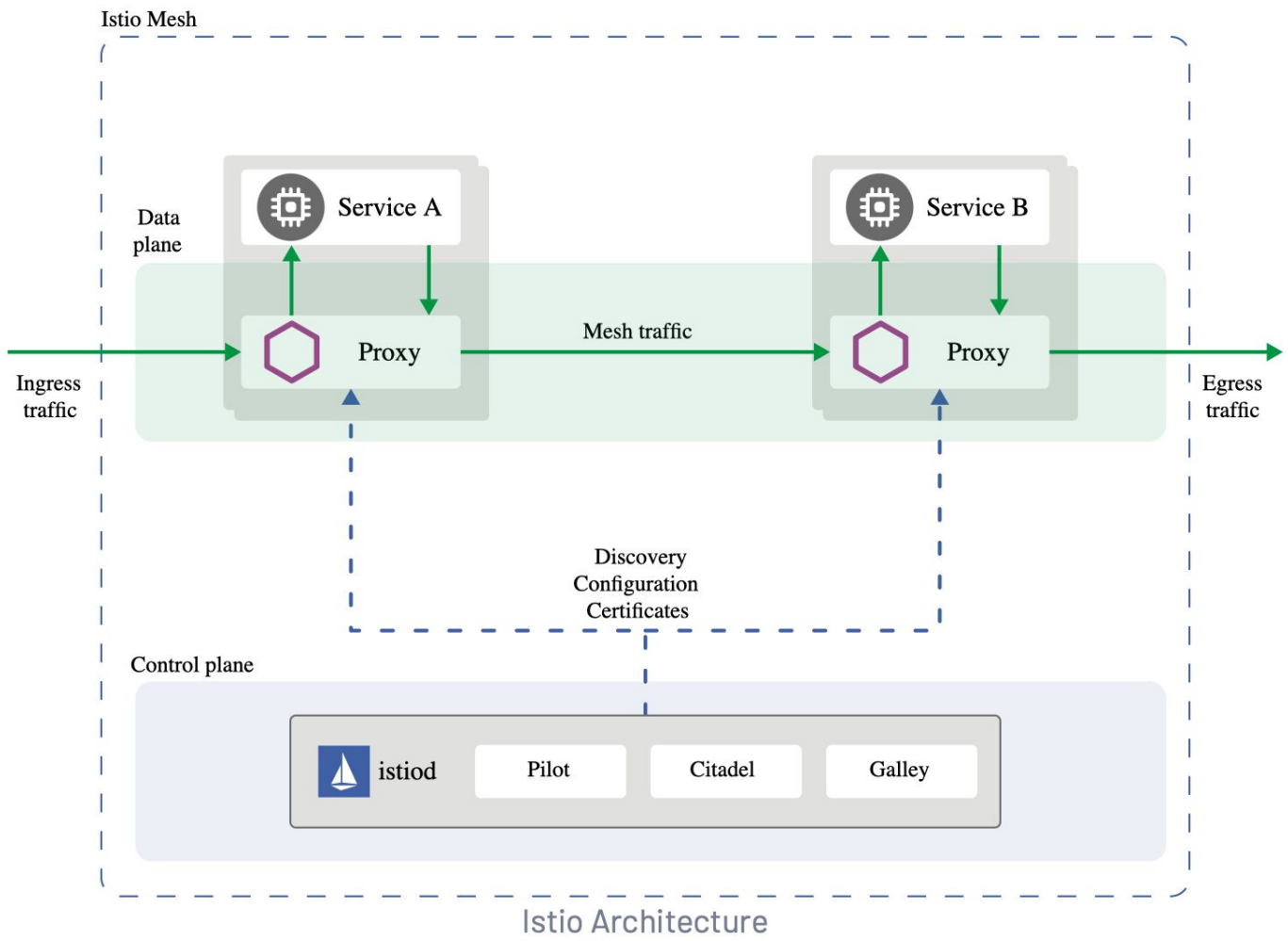
High Level Architektur von Service Meshes

Control Plane:

- umfasst Management Komponenten, sozusagen das “Hirn” des Service Meshes.
- Typische Komponenten umfassen Service Discovery & Traffic rules, Configuration Stores & APIs, Identity & Credential Management

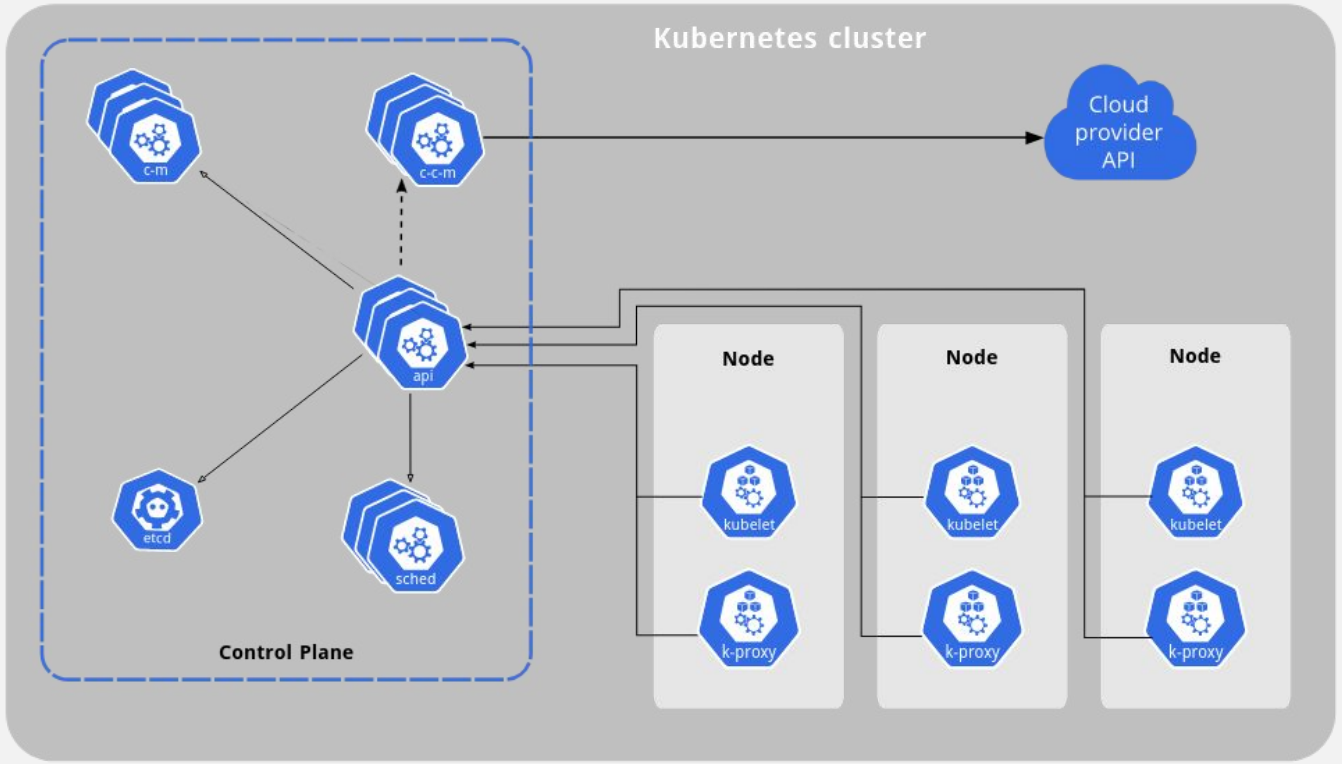
Data Plane:

- umfasst workloads
- umfasst proxy Komponenten, die features des service Meshes implementieren



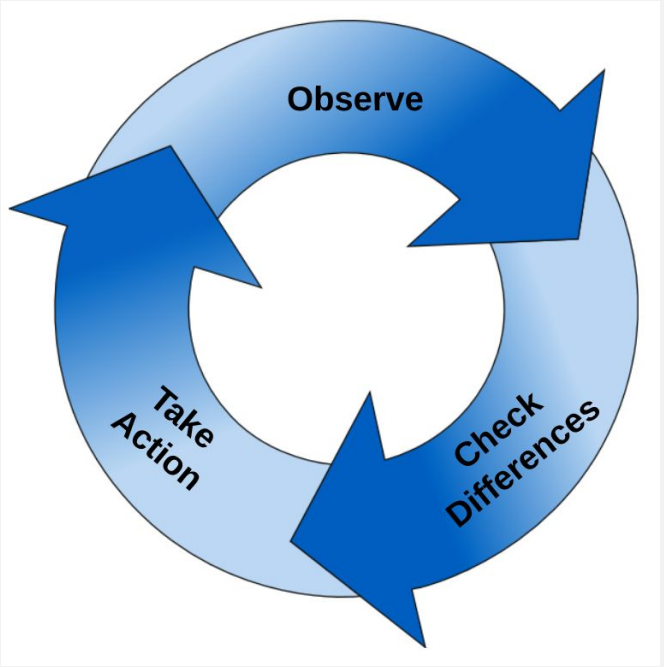
Istio Architecture

Exkurs: Wie erweitert man eigentlich Kubernetes?



Exkurs: Kubernetes APIs sind Zustandsbeschreibungen


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```



- Kubernetes APIs sind nicht imperativ.
- Für jeden **kind** gibt es einen oder mehrere zuständige **Controller** in Kubernetes.
- **Controller** arbeiten Event basiert in einer Control Loop:
  - **observe**: in welchem Zustand befindet sich mein Cluster hinsichtlich der vom Controller verwalteten Ressource
  - **diff**: gibt es Abweichungen vom deklarativ beschriebenen Zielzustand?
  - **take action**: bringe die verwalteten Ressourcen in den Zielzustand

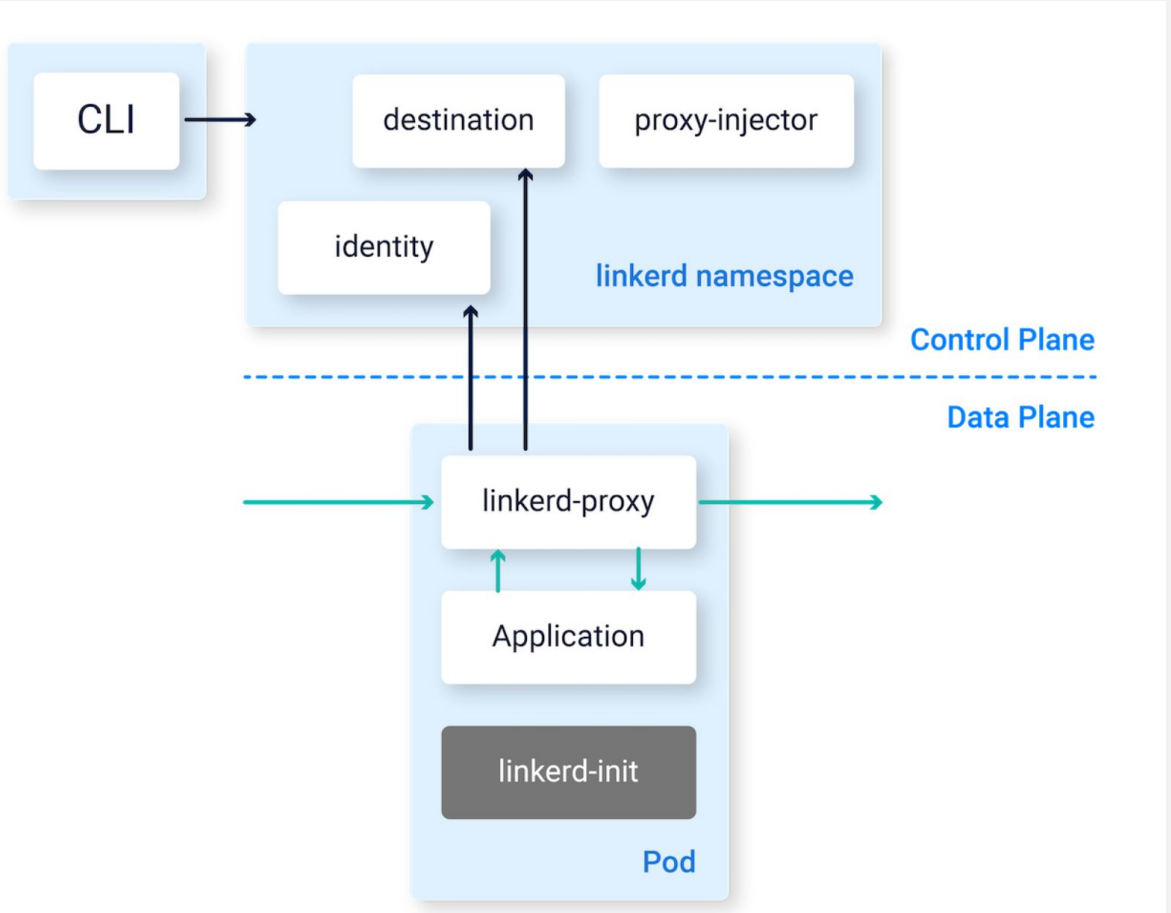
Data plane Architekturen - Sidecar

Sidecar-Pattern:

Es wird ein extra Container in jedem Applikations Pod hochgefahren

Im Falle von Linkerd:

- Jeder Pod im Mesh bekommt automatisch einen Proxy injected
- Über IP-Tables Regeln wird eingehender sowie ausgehender Traffic über den Sidecar Proxy geleitet. Die Regeln werden entweder über den linkerd-init startup container gesetzt, oder über ein CNI-Plugin implementiert.
- Der Sidecar Proxy implementiert die Mesh Funktionalität



Linkerd's architecture

Data plane Architekturen - Sidecar

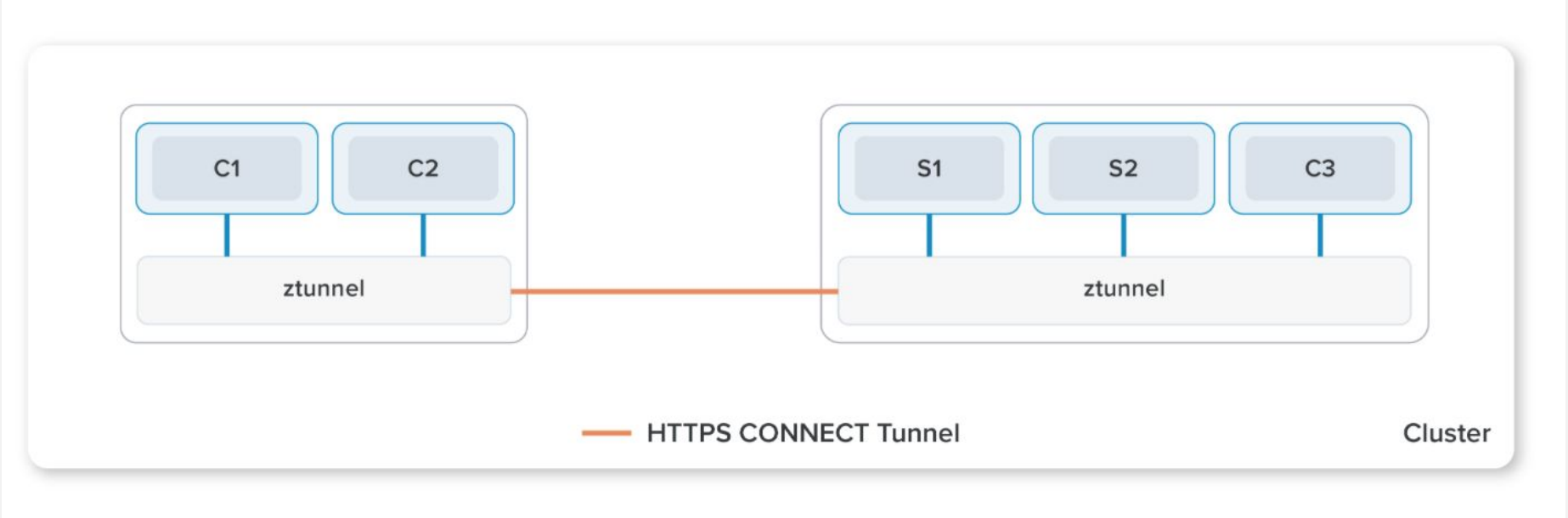
Vorteile:

- **Resilienz:** Kein single point of failure - da ein proxy pro Applikations Pod
- **Uniform:** Traffic wird uniform für alle Applikationen identisch durch den Proxy geschleift
- **Multi-Tenancy**: Es gibt in der Dataplane keine geteilte Infrastruktur zwischen verschiedenen Parteien
- **Skalierbarkeit**: Proxy skaliert horizontal mit Applikation
- **Zero code changes**: Die Anwendung kann ohne Anpassungen im Code gemeshed werden

Nachteile:

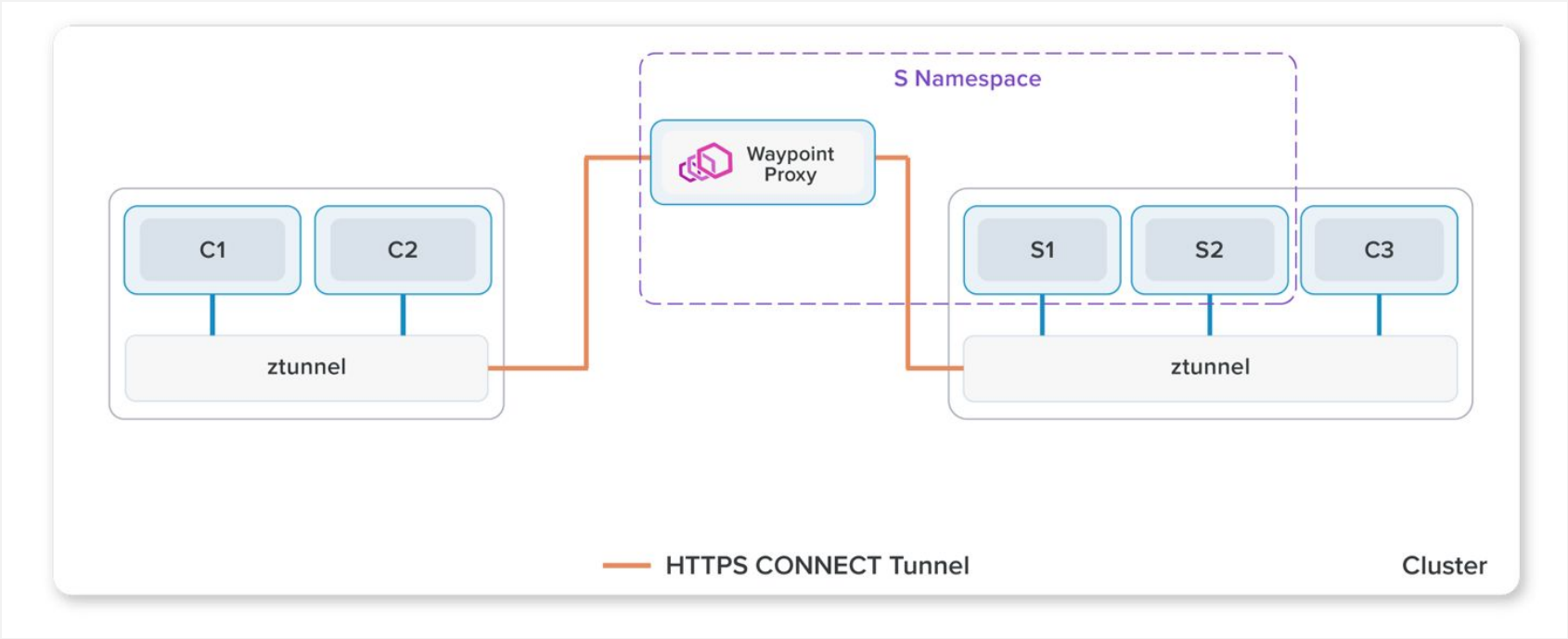
- **Resource Overhead**: Jeder Pod im Mesh bekommt automatisch einen Proxy injected, der wiederum Ressourcen wie CPU und RAM verbraucht.
- **Höhere Latenz**: Jeglicher Traffic wird durch Proxies geschleift, wodurch mehr Hops notwendig sind. Die Latenz verschlechtert sich. Abhängig von der Applikation kann das ein Problem darstellen.
- **Resource Management**: Der Sidecar Proxy läuft als Container in Kubernetes und unterliegt daher auch den typischen resource constraints. Diese müssen sinnvoll gewählt und gepflegt werden, um eine optimale Ressourcennutzung zu gewährleisten.
- **Security**: Es wird eine neue Infrastrukturkomponente deployed. Das erhöht potenziell die Angriffsoberfläche

Data plane Architekturen - Ambient Mesh



Unterstützt nur L4 features. Die Implementierung ist daher aber viel einfacher und performanter.

Data plane Architekturen - Ambient Mesh



Wenn L7 features benötigt werden, werden zusätzlich envoy proxies deployed.
Aller Traffic wird dann von den zTunneln durch den Proxy geschleift.

Data plane Architekturen - Ambient Mesh

Vorteile:

- **Skalierbarkeit**: Waypoint Proxy skaliert horizontal unabhängig von Applikationen
- **Zero code changes**: Die Anwendung kann ohne Anpassungen im Code gemeshed werden
- **Resource efficiency:** Teure envoy proxies werden erst dann verwendet, wenn L7 features gebraucht werden. Es werden in Summe nun auch weniger Proxies in Gesamtheit deployed, da ein waypoint proxy pro Namespace verwendet werden kann.
- **Flexibilität**: Kann theoretisch neben dem Sidecar Model deployed werden

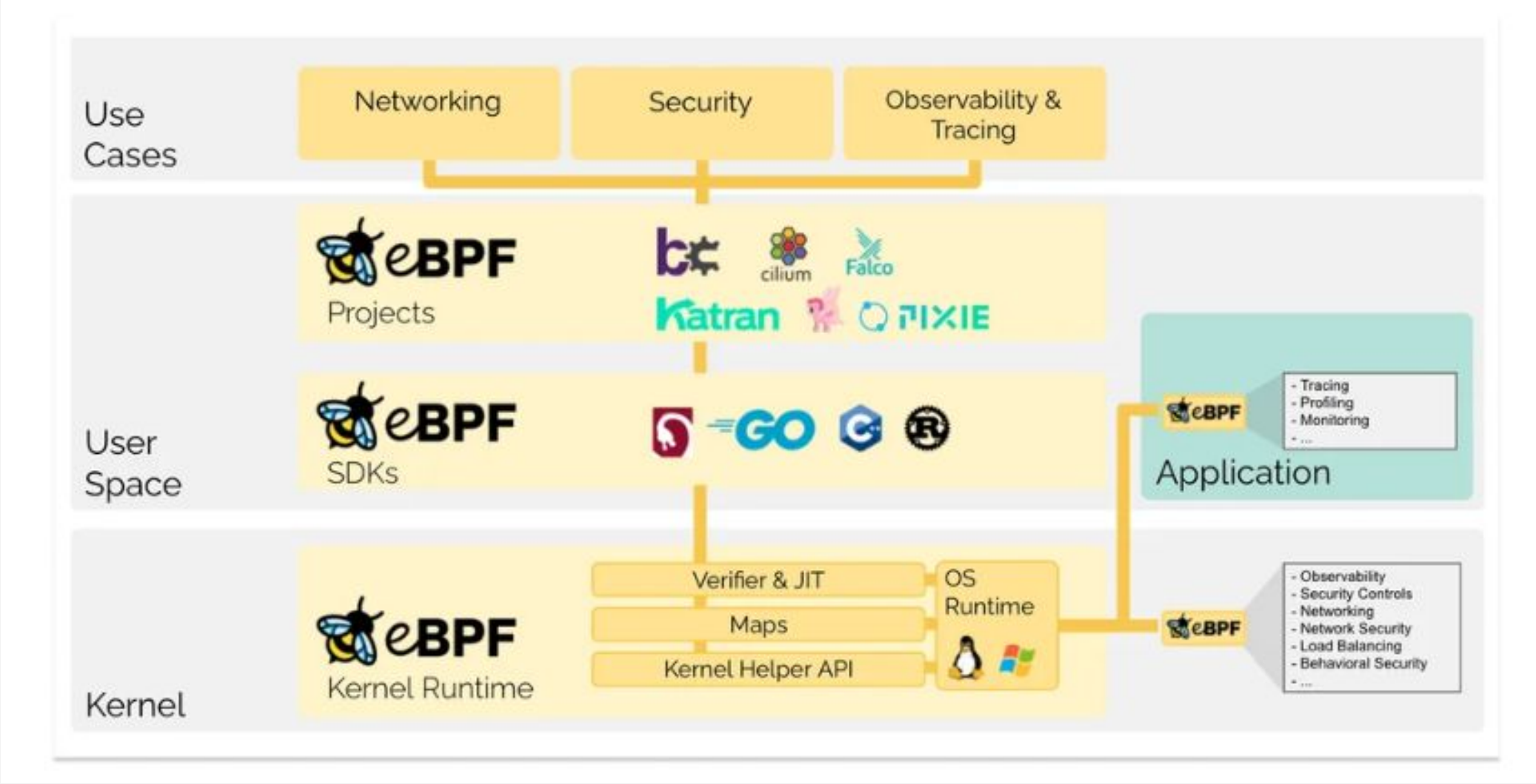
Nachteile:

- **Potenziell noch höhere Latenz**: Wenn L7 features gebraucht werden, wird der Traffic durch einen waypoint Proxy geschleift. Der Proxy ist jetzt aber möglicherweise nicht mehr auf demselben Knoten wie die Applikation.

Exkurs: extended berkeley packet filter (eBPF)

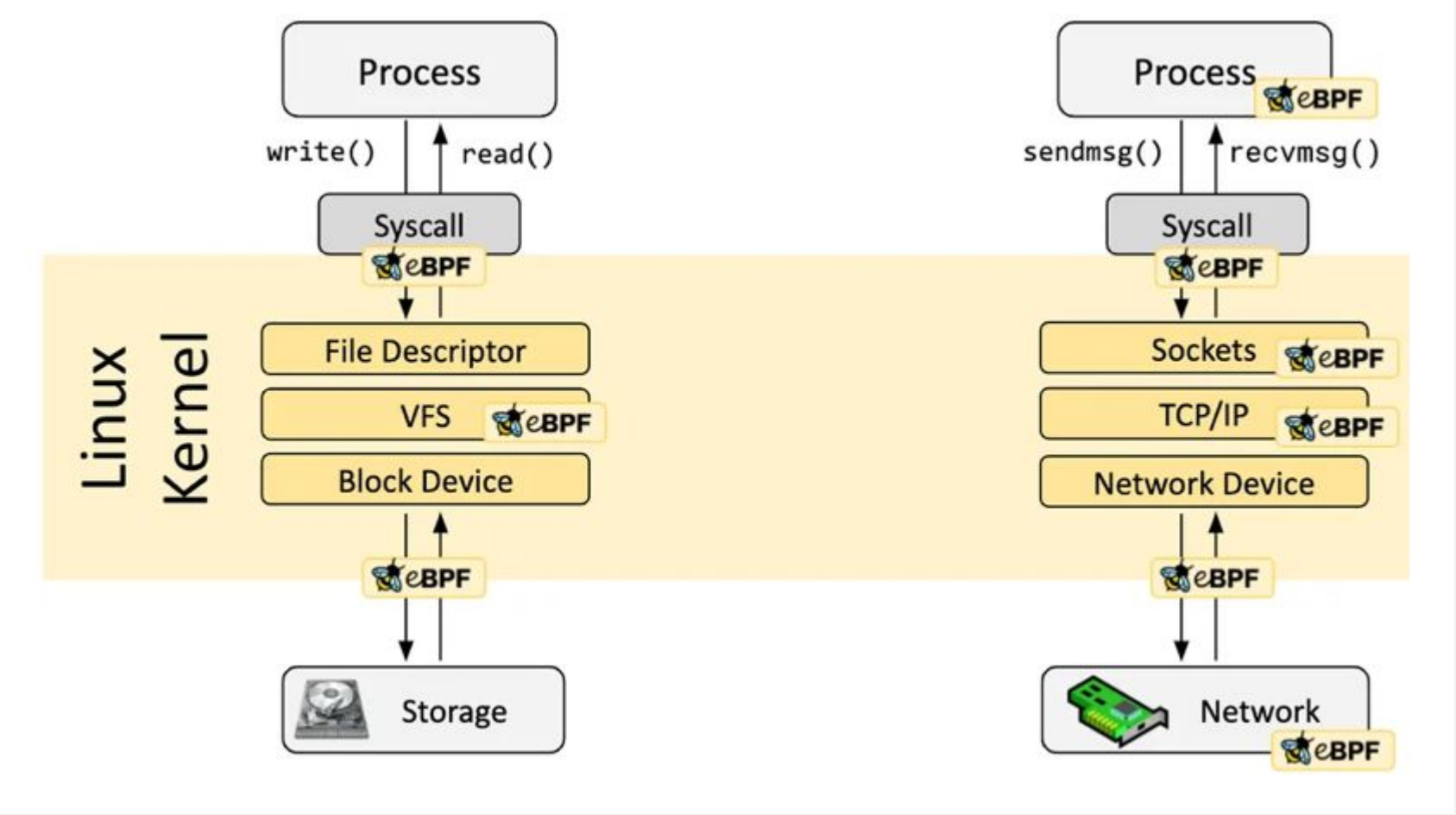
- Historisch aus dem Berkeley Packet Filter (BPF) entstanden.
- BPF war vor allem ein Netzwerk Tool
- eBPF hat den BPF umfassend erweitert, sodass das Akronym im Grunde keinen Sinn mehr macht
- eBPF erlaubt es dem Anwender den **Kernel dynamisch** mit Programmen zu erweitern
  - Diese Programme laufen in einer Sandbox im Kernel
  - Diese Programme werden JIT compiled & müssen einer Verification engine im Kernel genügen
  - Diese Programme laufen event driven und subscriben auf bestehende hooks im Kernel z.B. system calls
  - Falls es keinen bestehenden Hook gibt, kann mit einer kProbe/uProbe das Programm dennoch an relativ frei wählbare Punkte attached werden

Exkurs: extended berkeley packet filter (eBPF)



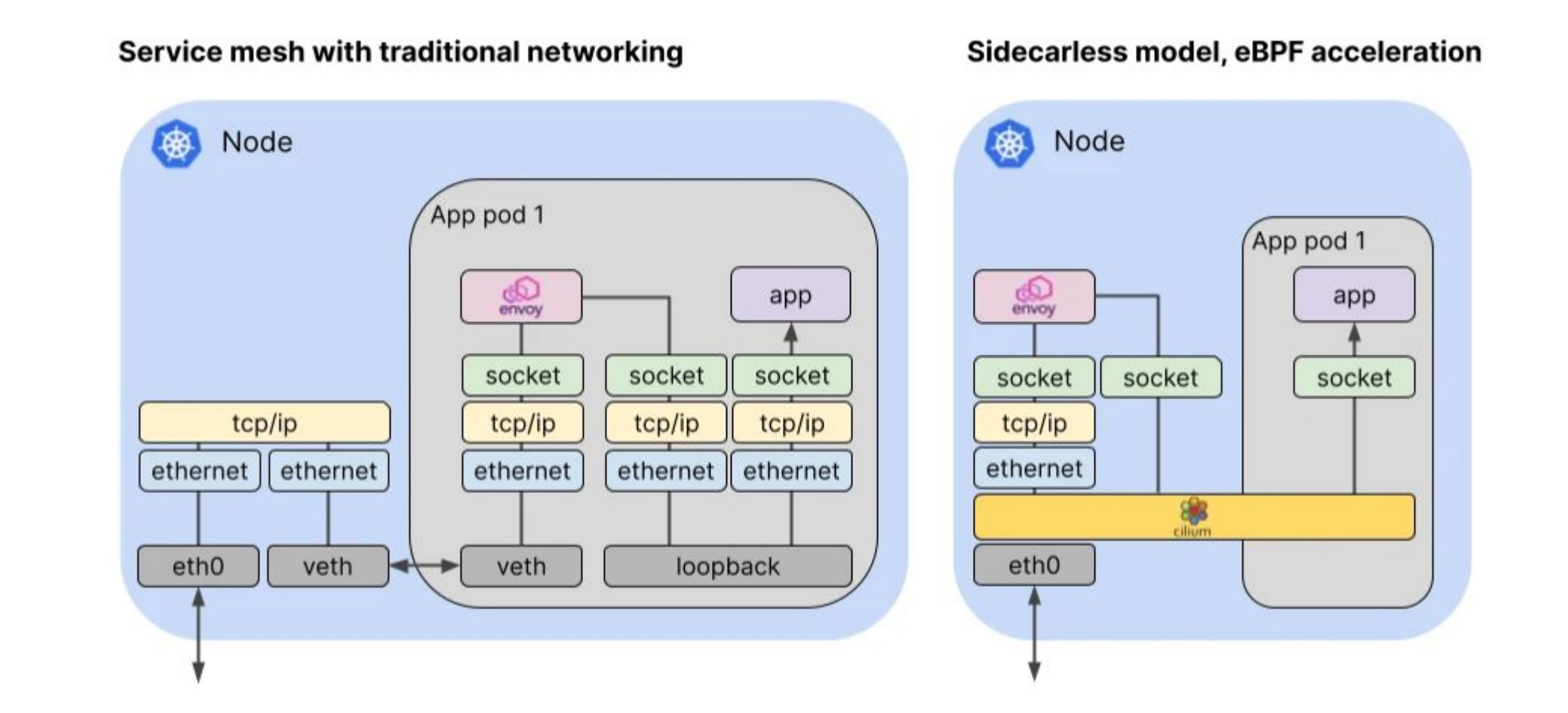
Quelle: https://ebpf.io/what-is-ebpf/#what-is-ebpf

Exkurs: extended berkeley packet filter (eBPF)



Quelle: https://ebpf.io/what-is-ebpf/#hook-overview

Data plane Architekturen - eBPF based



Quelle: https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta/

Data plane Architekturen - eBPF based

Vorteile:

- **Zero code changes**: Die Anwendung kann ohne Anpassungen im Code gemeshed werden
- **Resource efficiency:** Die Performance ist dem bisherigen Modell überlegen.
- **Flexibilität**: Es können viele Features in eBPF umgesetzt werden. Nur manche L7 Features erfordern nach wie vor einen L7 Proxy.

Nachteile:

- **Komplexität:** Neue, schwierige Technologie. Wie debugge ich das als Anwender?

LinkerD in action!