

corewire

Rook-Ceph
Konfiguration

Folien-Hinweis

- `Space`, `Page down`: Nächste Folie
- `Page up`: Vorherige Folie
- `ESC`, `o`: Übersicht

[Zur Kapitelübersicht](#)

Ziele dieses Kapitels

Ich kann

- ✓: Wichtige Rook-CRDs und ihre Aufgabe benennen
- ✓: Zentrale Konfigurationsbereiche eines Clusters einordnen
- ✓: Sichere von riskanten Konfigurationsänderungen
- ✓: unterscheiden
Auswirkungen einer Änderung vor und nach der Umsetzung validieren

Custom Resource Definition (CRD)

- Kubernetes kennt Objekttypen wie `Pod`, `Deployment` oder `Service`.
- Eine CRD installiert weitere Objekttypen
- Nach der Installation einer CRD kann dieser Typ genau wie ein eingebautes Objekt verwaltet werden.
- Damit werden aus `kubectl get pods` z. B. `kubectl get cephclusters`.

Wie Rook CRDs zur Konfiguration nutzt

- Rook bringt eigene CRDs mit, z. B. `CephCluster` oder `CephBlockPool`.
- Der gewünschte Zustand des Storage-Clusters wird deklarativ als CRD-Ressource beschrieben.
- Der Rook-Operator liest diese Ressourcen und setzt sie in echte Ceph-Konfiguration um.
- Konfiguration erfolgt damit über Kubernetes-Objekte, nicht über direkte Ceph-Kommandos.

Wichtige Rook-Ressourcen und CRDs

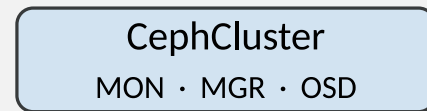
CRDs im Überblick

CRD	Definiert
<code>CephCluster</code>	Ceph-Cluster (Nodes, Devices, Netzwerk, ...)
<code>CephBlockPool</code>	RBD-Pool
<code>CephFilesystem</code>	CephFS
<code>CephObjectStore</code>	S3-kompatiblen Object Store
<code>CephObjectStoreUser</code>	Zugangsdaten Object Store

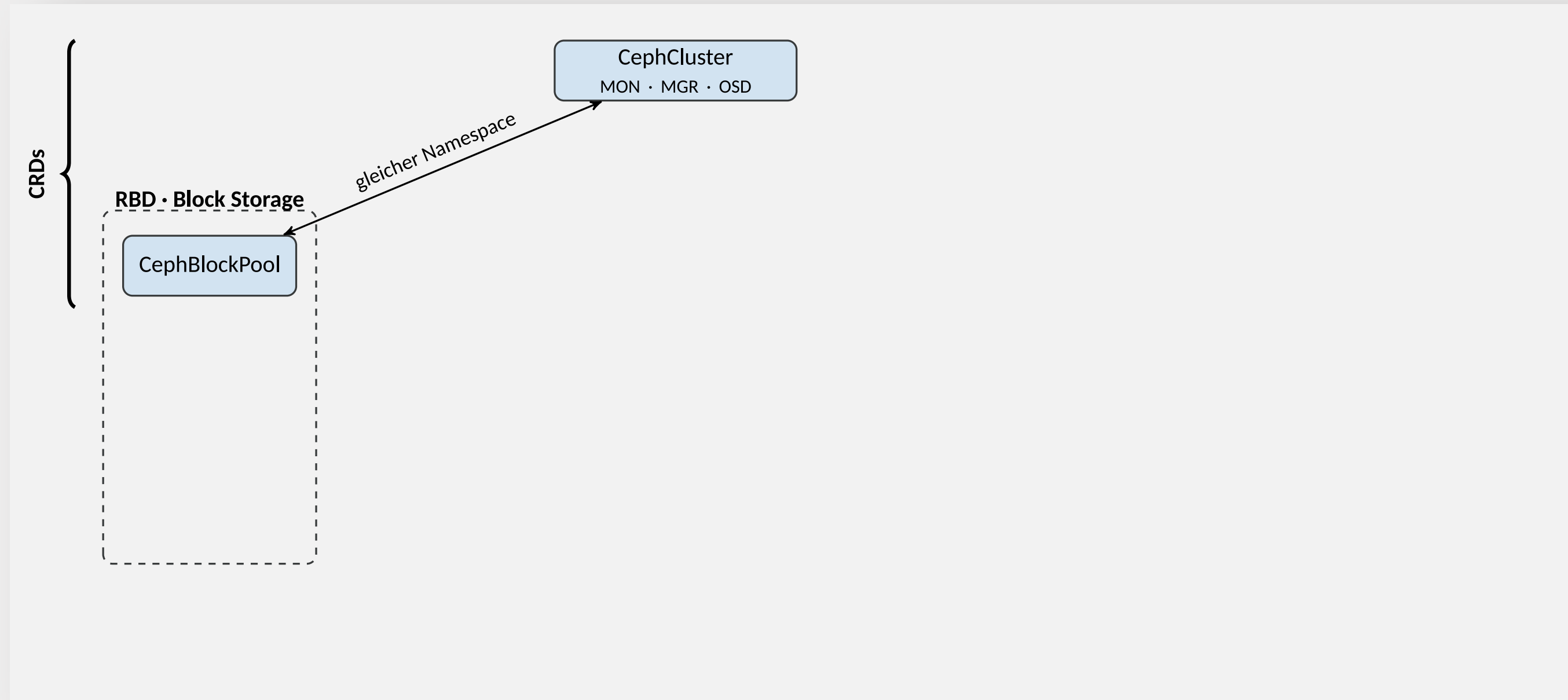
Zusammenspiel der CRDs

- Pro Namespace kann es einen `CephCluster` geben.
- Ein `CephCluster` kann mehrere `CephBlockPool`, `CephFilesystem` und `CephObjectStore` enthalten.
- StorageClasses und VolumeSnapshotClasses referenzieren die Pools, Filesysteme und Object Stores.

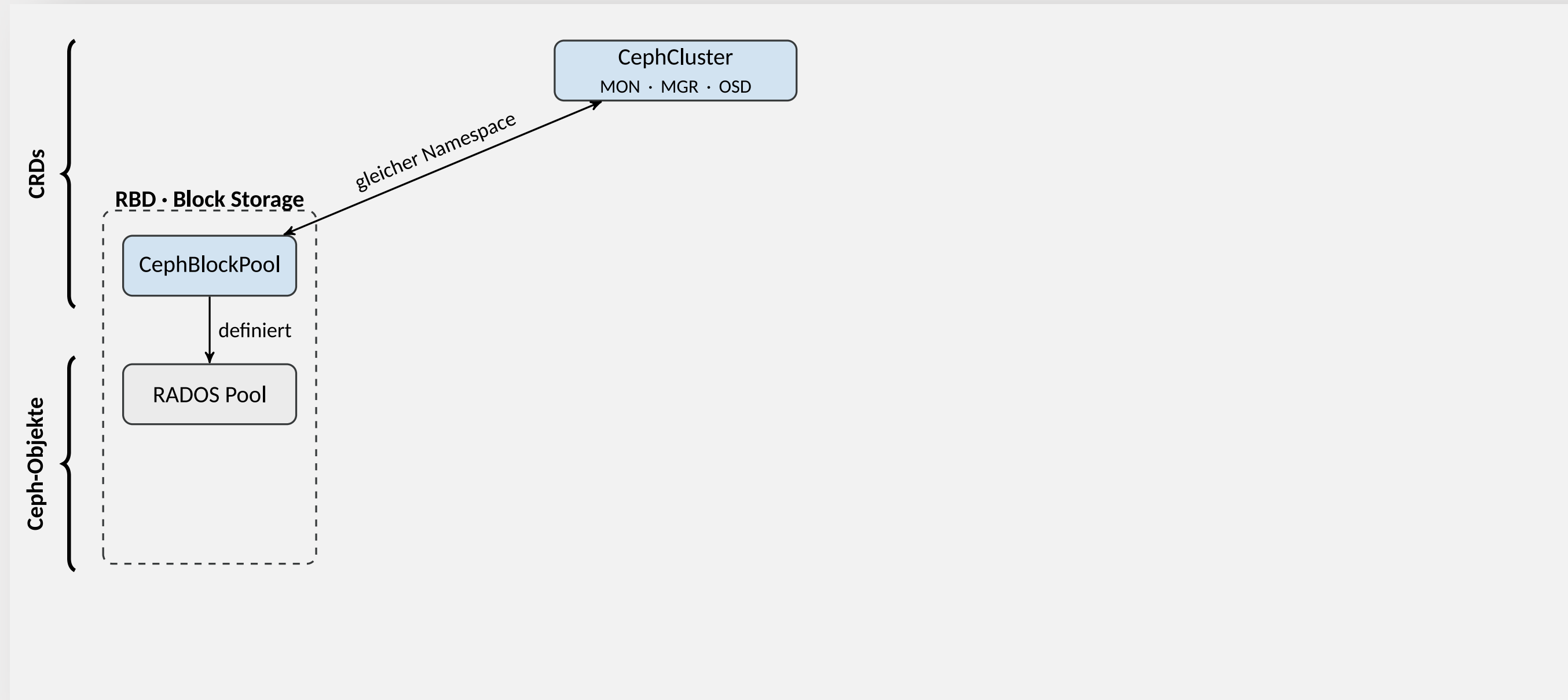
Zusammenspiel der CRDs



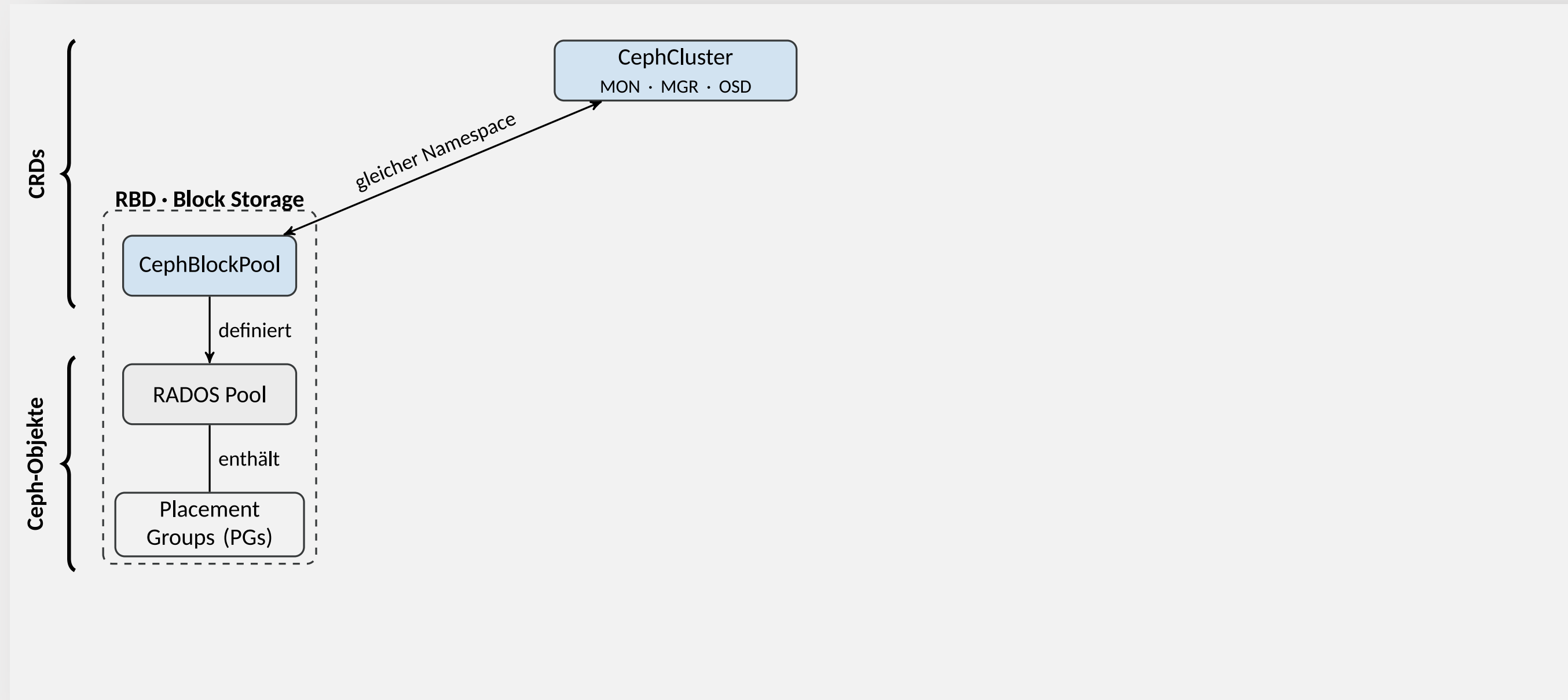
Zusammenspiel der CRDs



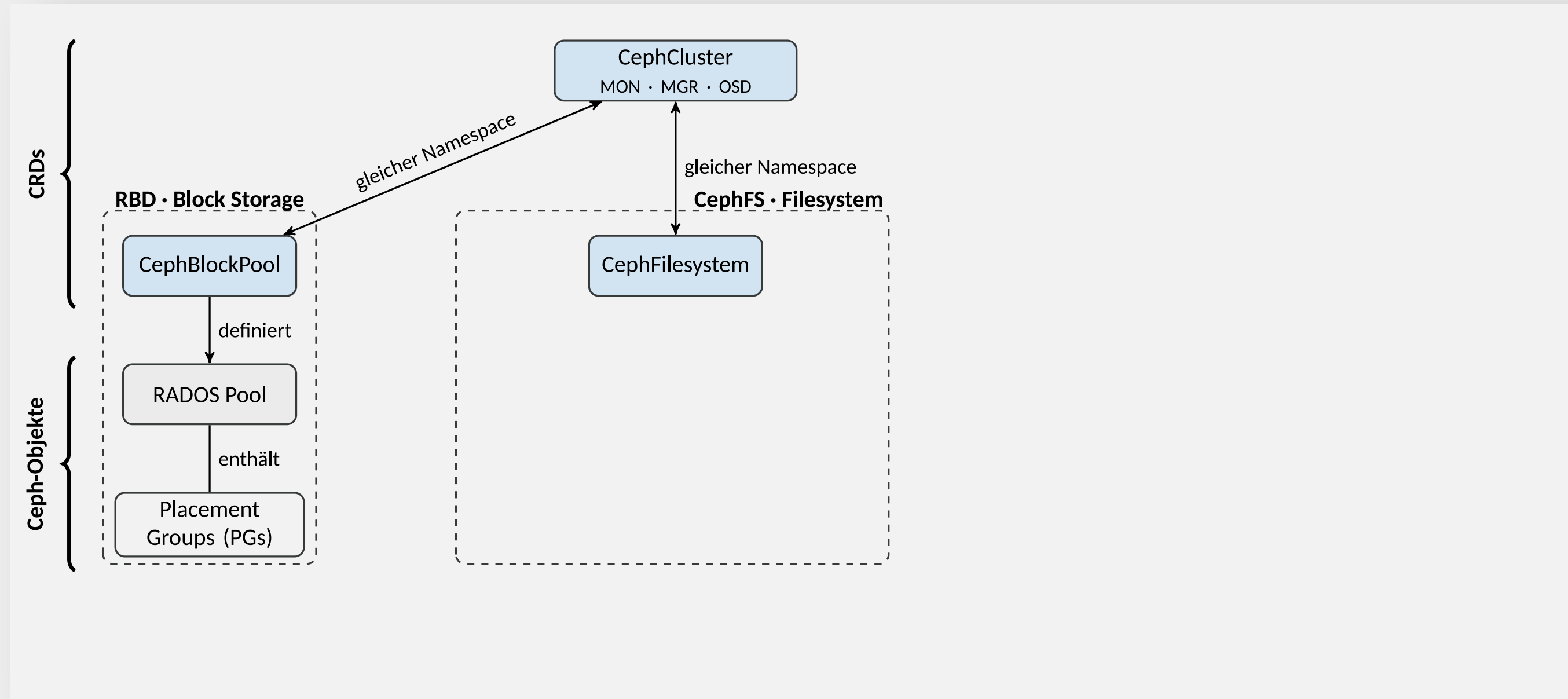
Zusammenspiel der CRDs



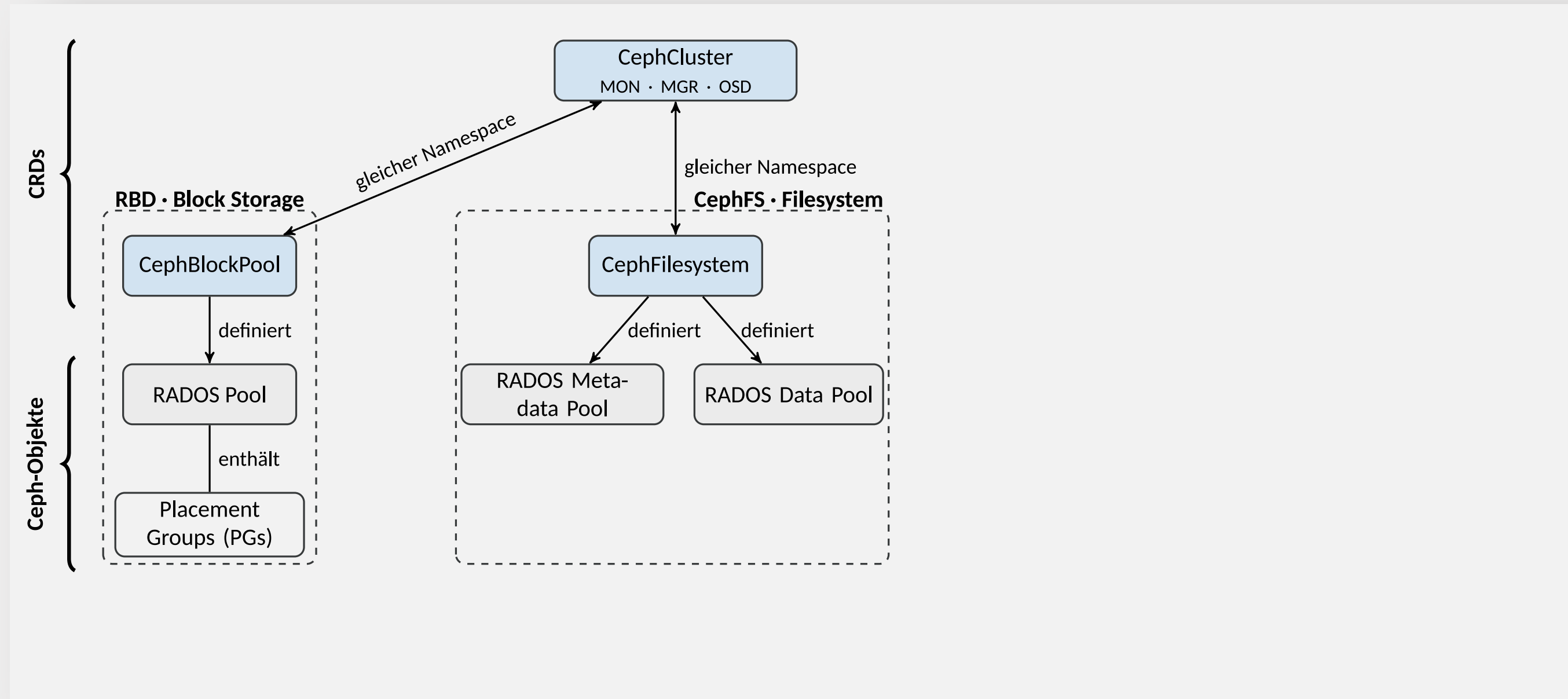
Zusammenspiel der CRDs



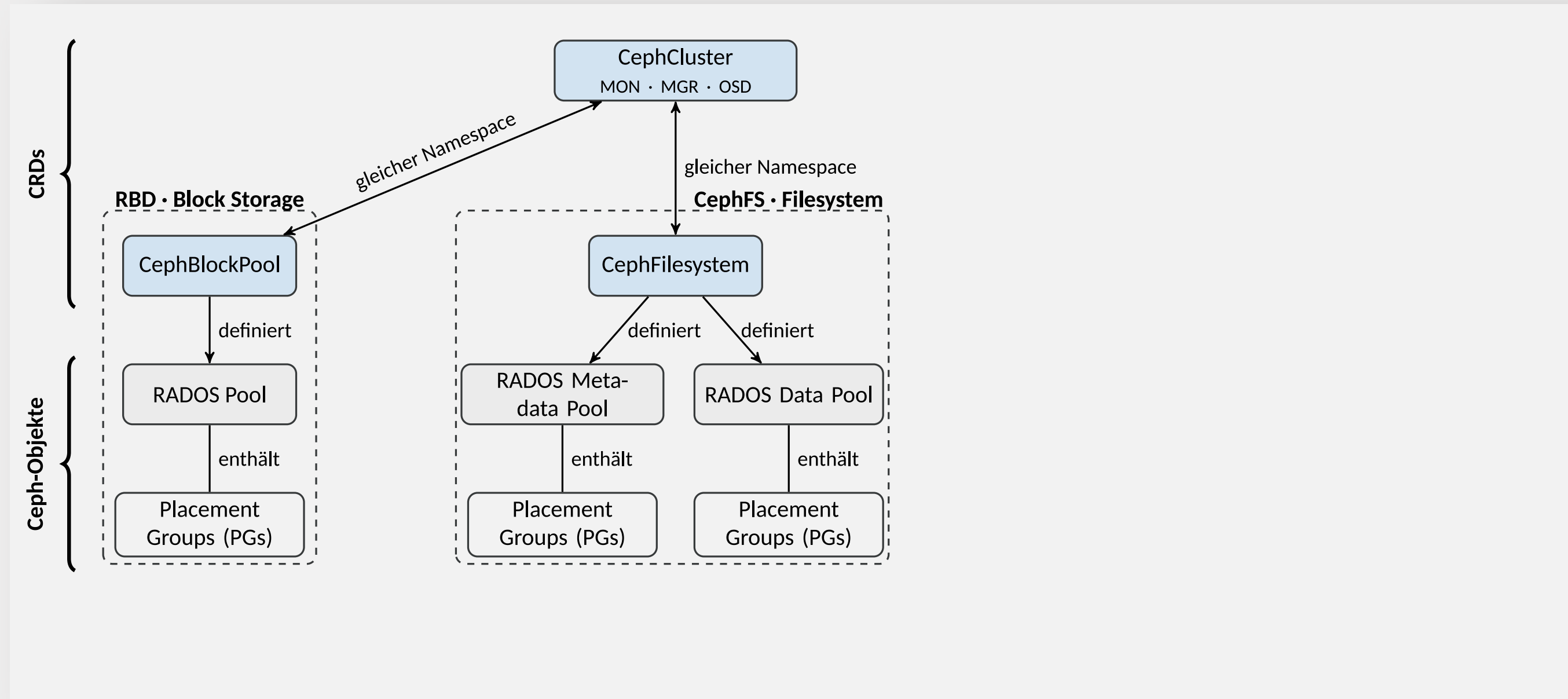
Zusammenspiel der CRDs



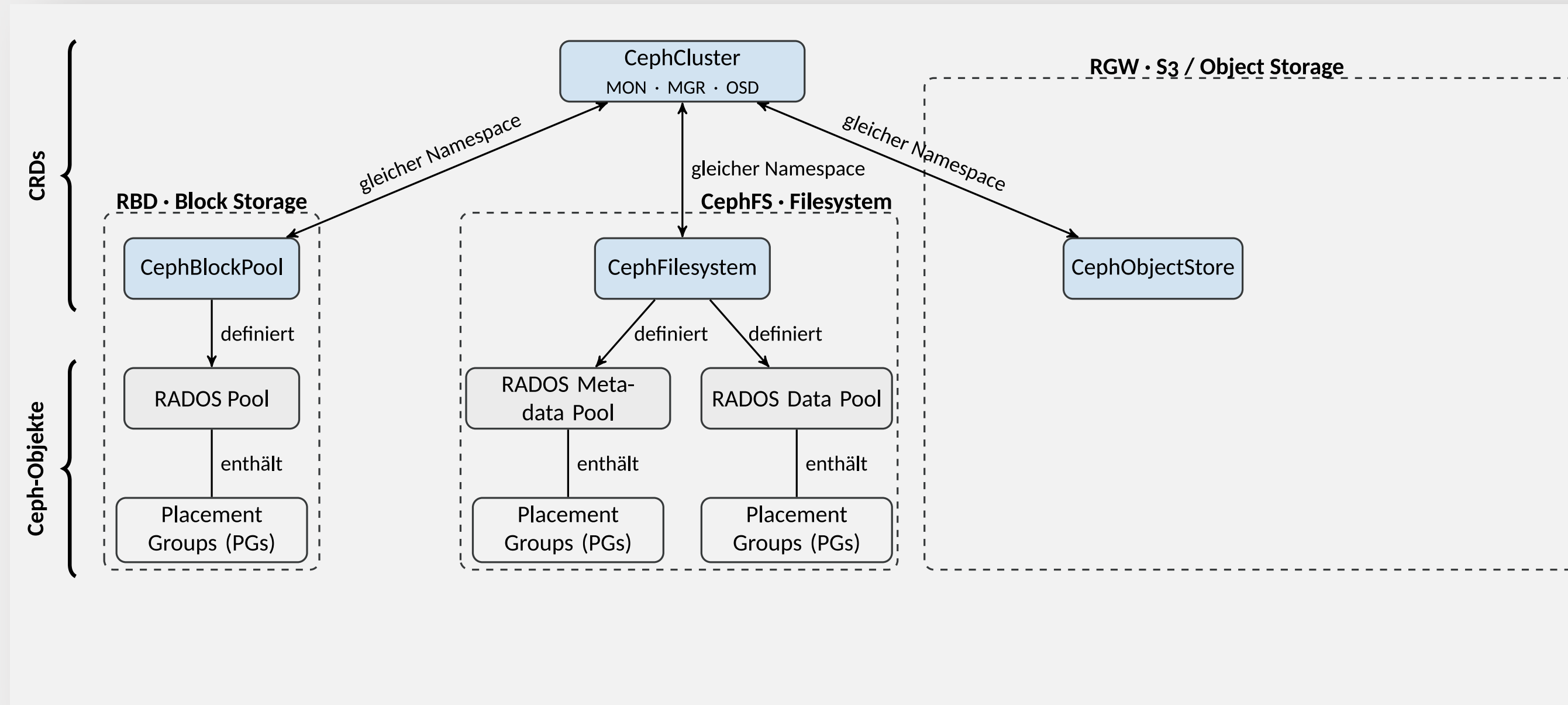
Zusammenspiel der CRDs



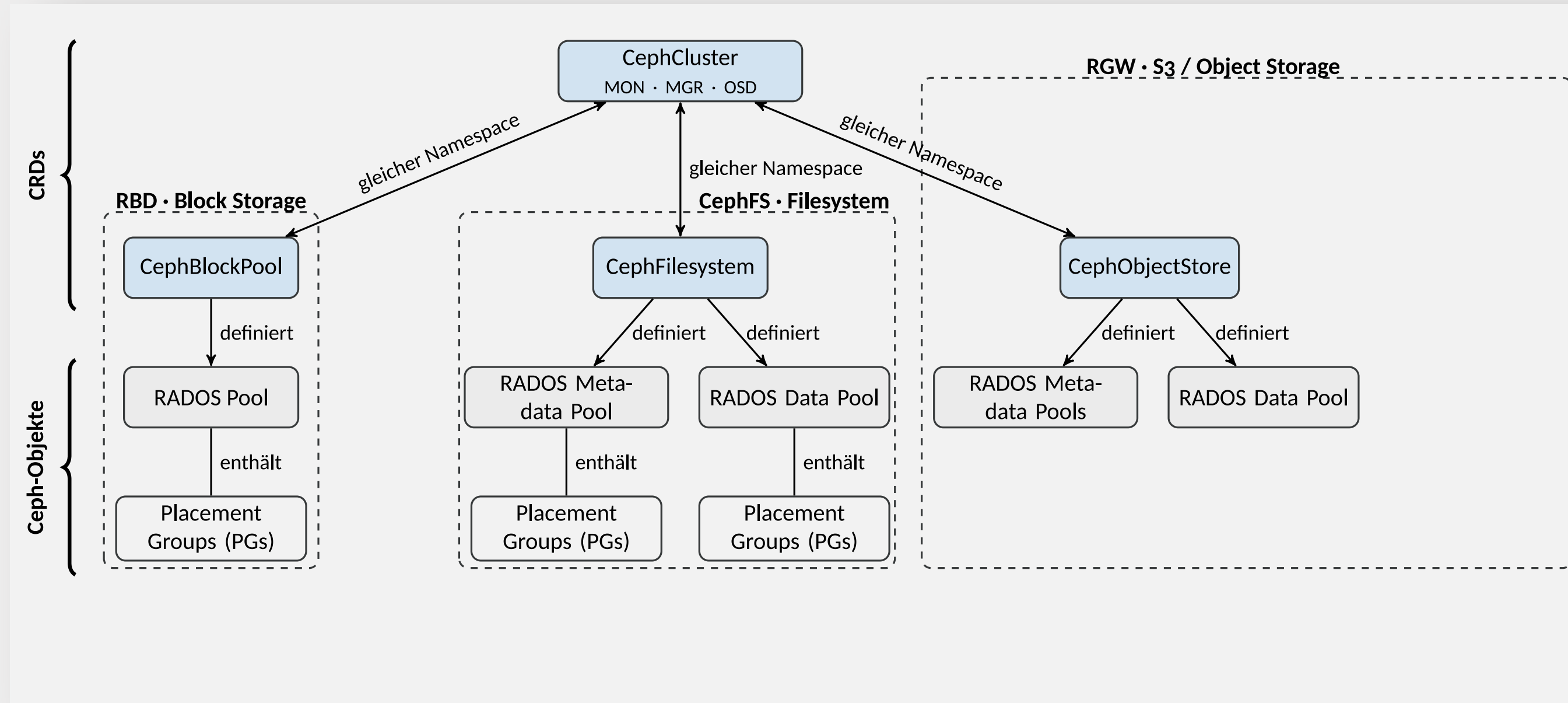
Zusammenspiel der CRDs



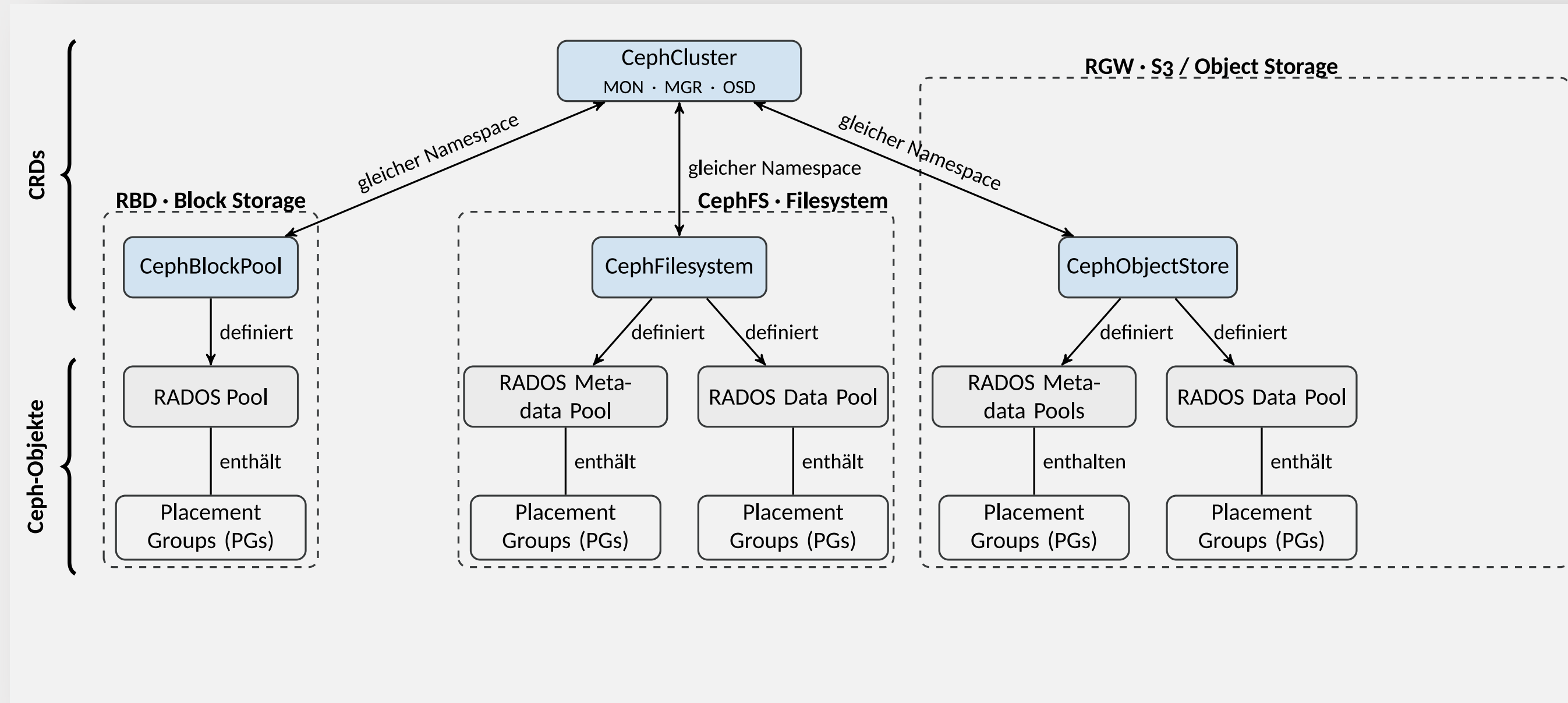
Zusammenspiel der CRDs



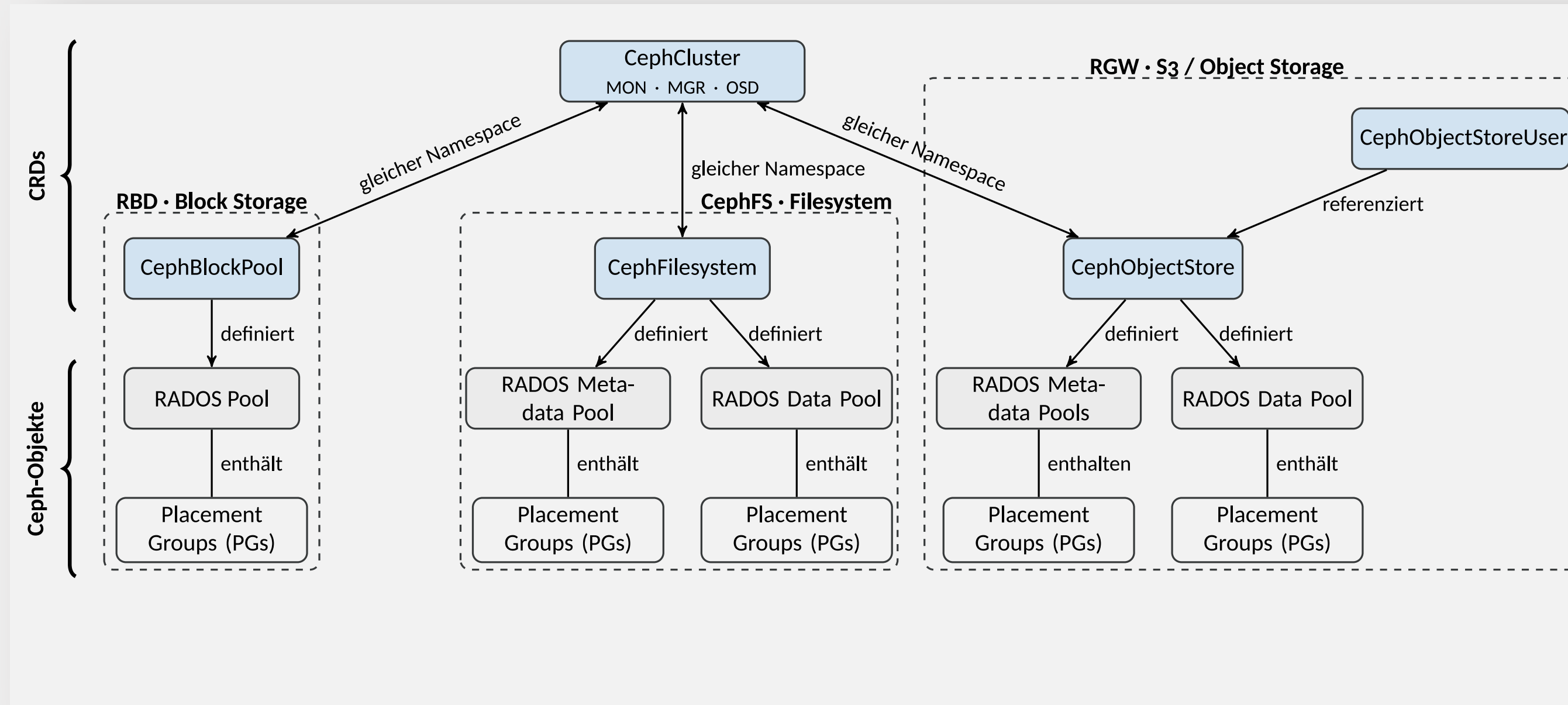
Zusammenspiel der CRDs



Zusammenspiel der CRDs



Zusammenspiel der CRDs



CephCluster

- Top-Level-Objekt für den gesamten Ceph-Cluster
- Definiert Nodes, Devices, Netzwerk, Monitore, Manager und OSDs
- Steuert globale Einstellungen wie Placement, Affinity, Tolerations und Ressourcen
- Wird als Grundlage für alle weiteren Storage-Dienste verwendet

Erzeugte Kubernetes-Objekte

- Ein `Deployment` je MON-Daemon (z. B. `rook-ceph-mon-a`)
- Ein `Deployment` je MGR-Daemon (z. B. `rook-ceph-mgr-a`)
- Ein `Deployment` je OSD (z. B. `rook-ceph-osd-0`)
- `Services` für den Zugriff auf MONs und den Dashboard/MGR-Endpunkt
- `ConfigMaps` und `Secrets` mit Cluster- und Zugangsdaten

CephBlockPool

- Definiert einen logischen RBD-Pool innerhalb des Clusters
- Konfiguriert Replikation, Failure Domain, Erasure Coding und Größen
- Wird von `StorageClass`-Objekten für Kubernetes-Volumes genutzt

Erzeugte Kubernetes-Objekte

- Keine, `CephBlockPool` erzeugt nur eine Ceph-interne Ressource (den RADOS-Pool)
- Erst eine passende `StorageClass` macht den Pool für Kubernetes nutzbar

CephFilesystem

- Definiert ein CephFS-Dateisystem
- Konfiguriert Mount-Optionen, Platzierung und Performance-Einstellungen
- Bietet gemeinsam genutzten Dateizugriff für Anwendungen und Workloads
- Wird von `StorageClass`-Objekten für Kubernetes-Volumes genutzt

Erzeugte Kubernetes-Objekte

- Ein `Deployment` je MDS-Daemon (z. B. `rook-ceph-mds-myfs-a`)
- Optional ein zweites MDS-`Deployment` für Standby/Multi-Active
- Ein `Service` für den internen Zugriff auf die MDS-Daemons

CephObjectStore

- Definiert einen S3-kompatiblen Object Store im Ceph-Cluster
- Steuert Gateway-Deployment, Endpunkte, TLS und Storage-Konfiguration
- Enthält die Buckets und Objektspeicher-Logik für Anwendungen
- Repräsentiert die Objekt-Storage-Schicht neben RBD und CephFS

Erzeugte Kubernetes-Objekte

- Ein `Deployment` je RGW-Gateway-Instanz (z. B. `rook-ceph-rgw-store-a`)
- Ein `Service` als S3-Endpunkt für Anwendungen
- Optional ein `Ingress` bzw. `Route` für externen Zugriff

CephObjectStoreUser

- Erstellt Benutzer für den S3-kompatiblen Object Store
- Liefert Zugangsdaten wie Access Key und Secret Key
- Wird für Anwendungen genutzt, die Bucket-Zugriff über S3 benötigen
- Enthält typischerweise Zugriffseinstellungen und Berechtigungen

Erzeugte Kubernetes-Objekte

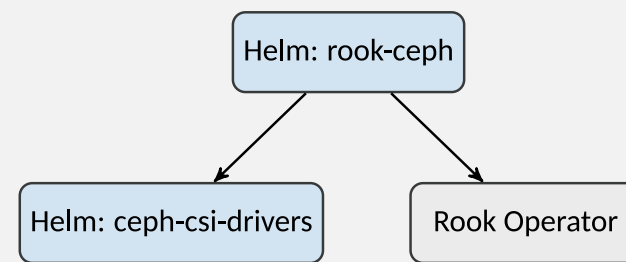
- Keine eigenen Pods, nur ein `Secret` mit Access Key und Secret Key
- Das `Secret` wird von Anwendungen als Umgebungsvariable oder Volume eingebunden

Helm

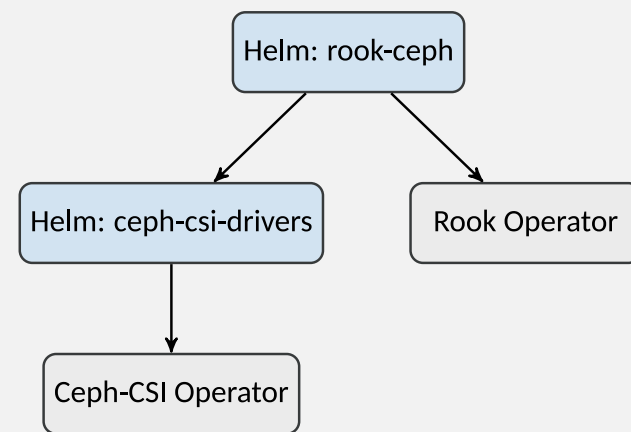
Helm Charts und die erzeugten Ressourcen

Helm: rook-ceph

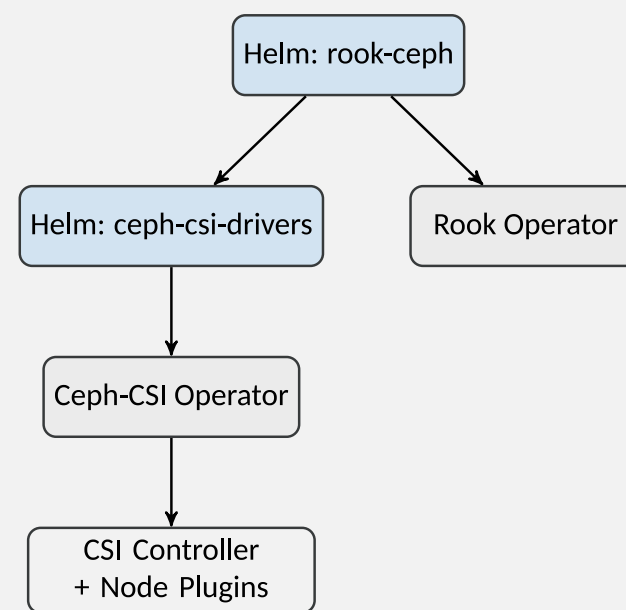
Helm Charts und die erzeugten Ressourcen



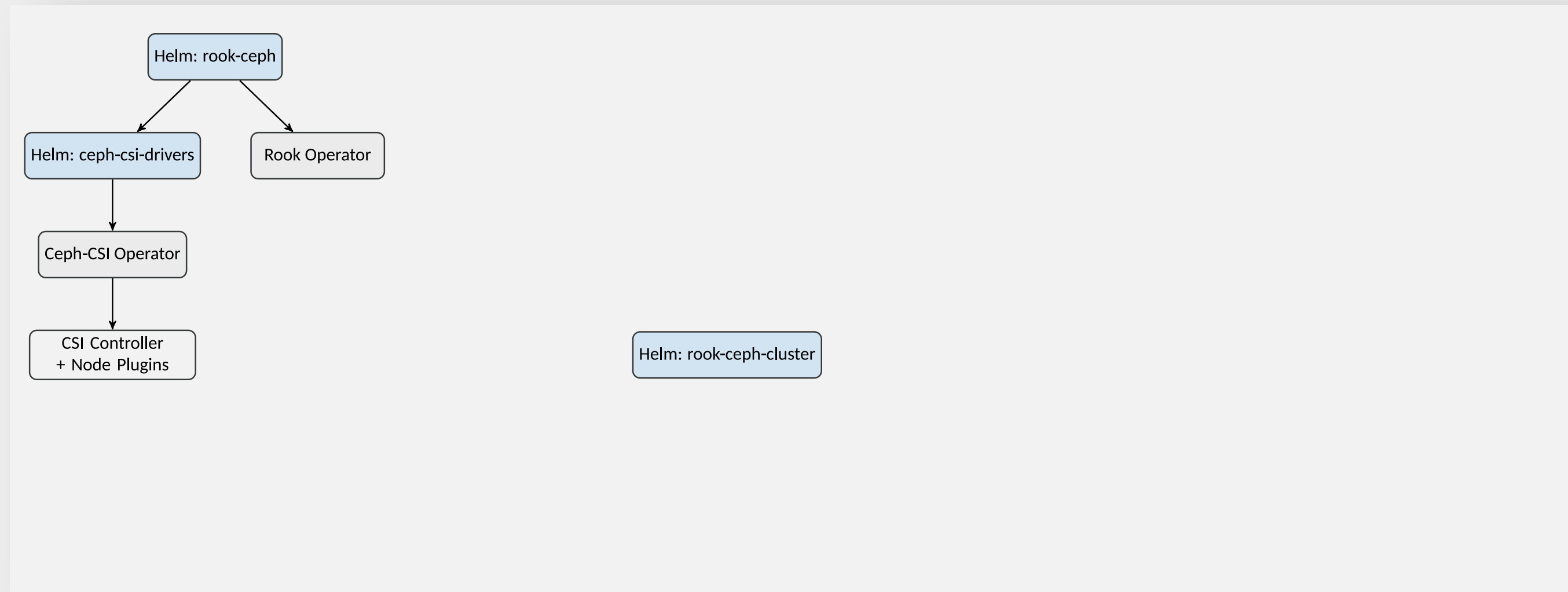
Helm Charts und die erzeugten Ressourcen



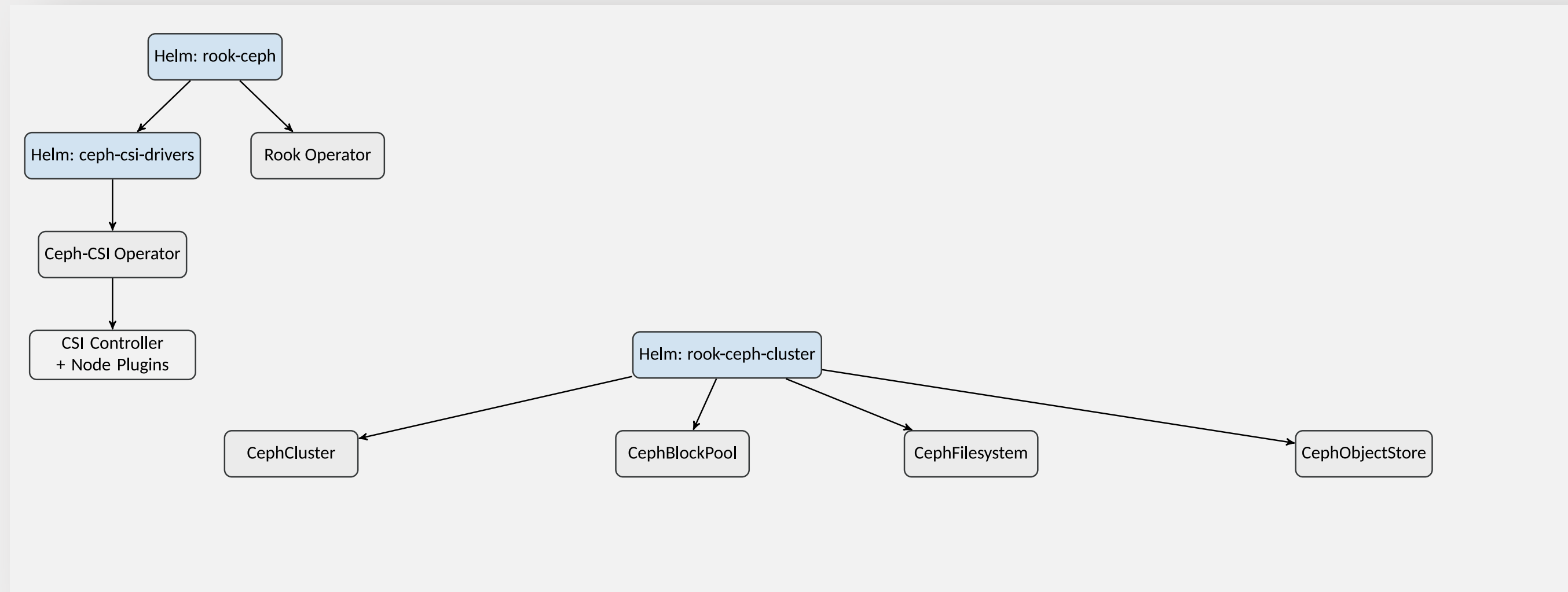
Helm Charts und die erzeugten Ressourcen



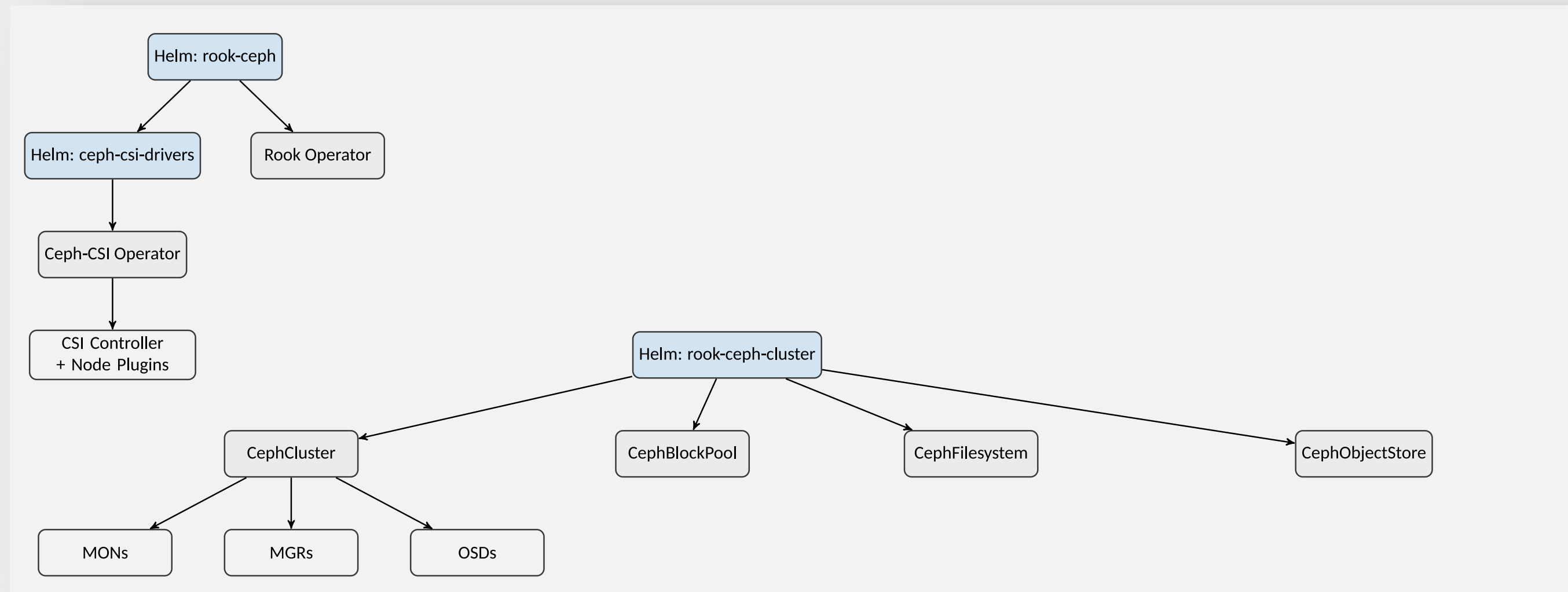
Helm Charts und die erzeugten Ressourcen



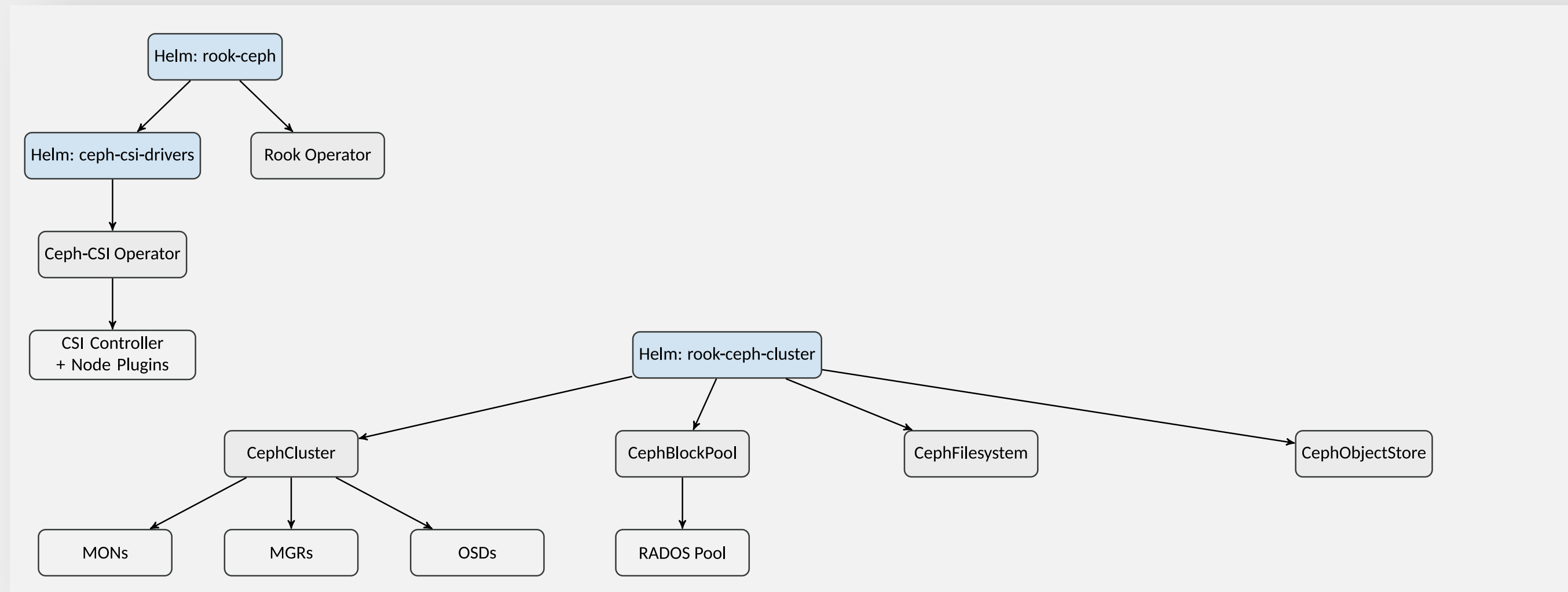
Helm Charts und die erzeugten Ressourcen



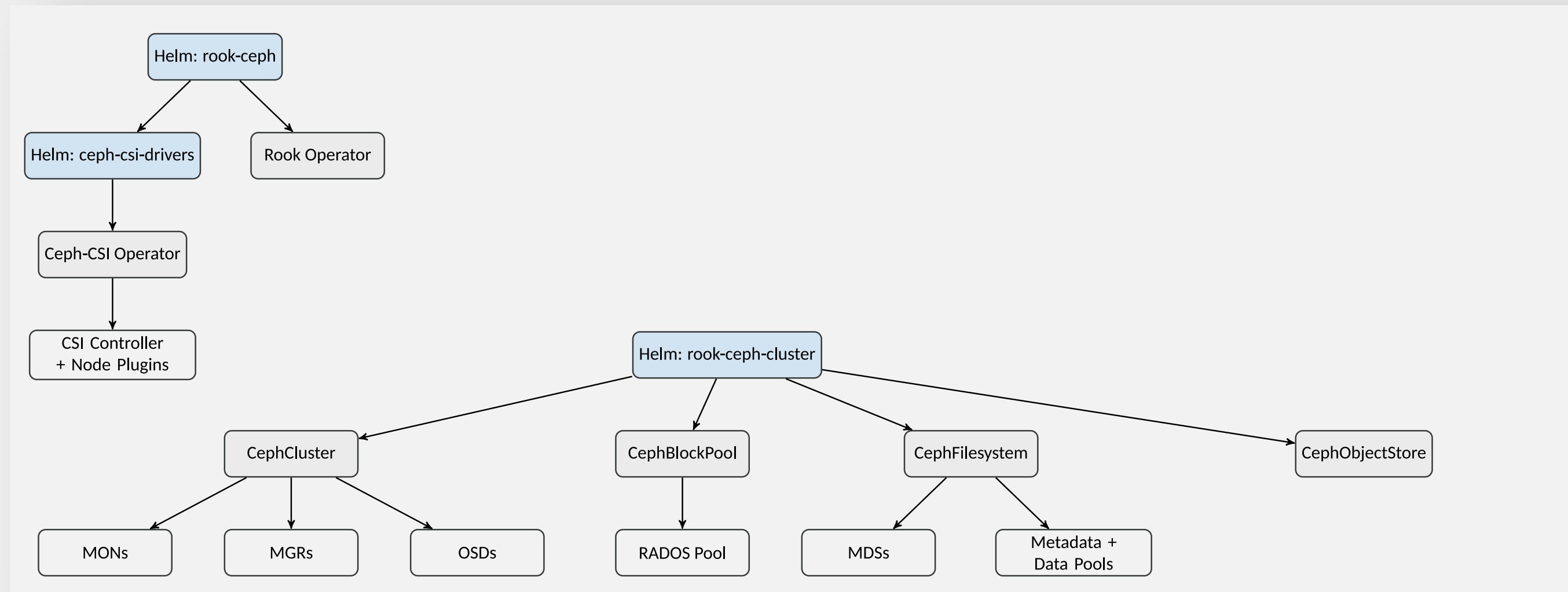
Helm Charts und die erzeugten Ressourcen



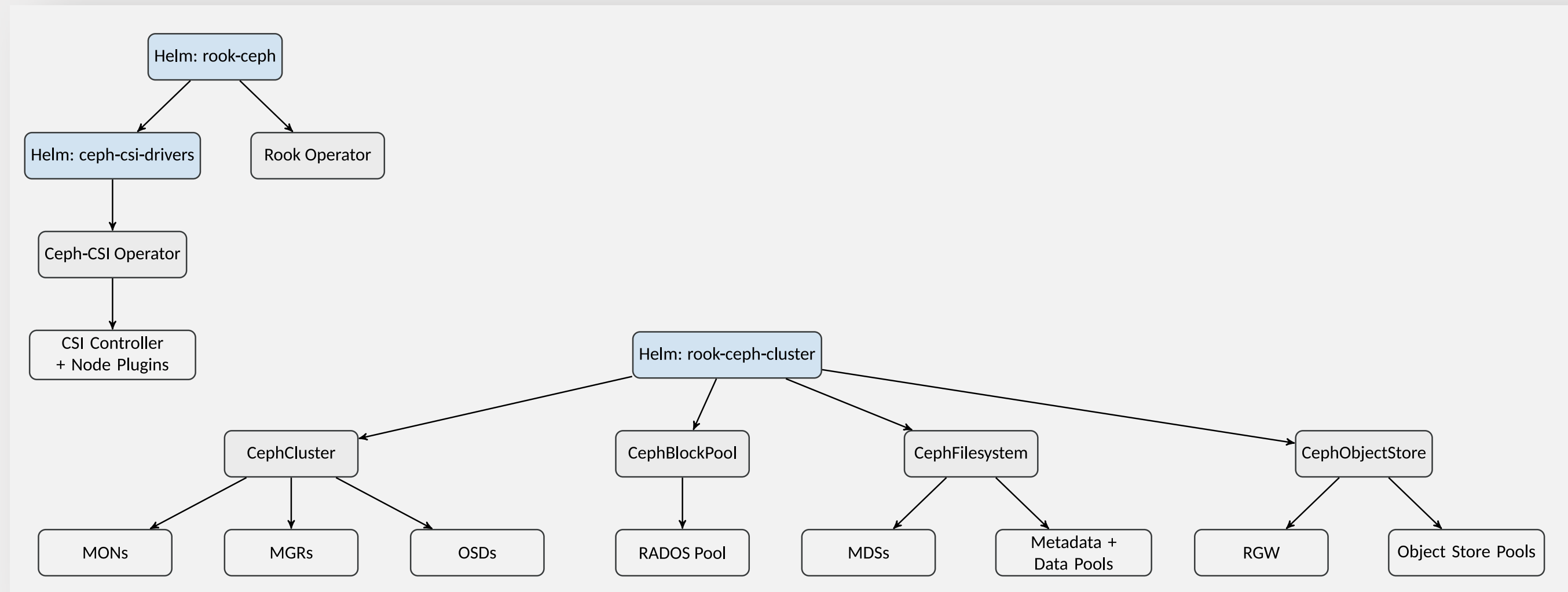
Helm Charts und die erzeugten Ressourcen



Helm Charts und die erzeugten Ressourcen



Helm Charts und die erzeugten Ressourcen



CRDs vs. Helm Values

- Rook und Ceph werden in Produktion meist über zwei offizielle Helm Charts betrieben:
 - `rook-ceph`: installiert den Operator
 - `rook-ceph-cluster`: erzeugt CRDs
- Änderungen werden dann in der `values.yaml` des `rook-ceph-cluster`-Charts durchgeführt

Beispiel: `values.yaml`

```
cephClusterSpec:  
  mgr:  
    count: 2  
  
cephBlockPools:  
  - name: replicapool  
    spec:  
      replicated:  
        size: 3  
      failureDomain: host
```

Wird über `helm upgrade` ausgerollt

Zentrale Konfigurationsbereiche

Was wird typischerweise konfiguriert?

- Auswahl und Verwaltung von Storage-Geräten
- Placement, Affinity und Tolerations
- Ressourcenanforderungen, Limits und Verteilung
- Pool-Replikation und Failure Domains
- Recovery- und Backfill-Strategien

Storage-Geräte

- Legt fest, welche Devices/Nodes Rook als OSDs nutzen darf
- Möglich über explizite Device-Listen oder Auswahlfilter
- Steuert, ob neue Devices automatisch aufgenommen werden

Beispiel: Device-Auswahl

```
storage:
  useAllNodes: false
  useAllDevices: false
  nodes:
    - name: "worker-1"
      devices:
        - name: "sdb"
    - name: "worker-2"
      deviceFilter: "^sd[b-d]"
```

Placement, Affinity und Tolerations

- Legt fest, auf welchen Nodes Ceph-Daemons laufen dürfen
- Node-Affinity steuert die Zuordnung zu bestimmten Nodes
- Tolerations erlauben den Betrieb auf tainted Nodes
- Wichtig für dedizierte Storage-Nodes

Beispiel: Placement für OSDs

```
placement:
  osd:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "storage-node"
                operator: In
                values: ["true"]
    tolerations:
      - key: "storage-only"
        operator: "Exists"
        effect: "NoSchedule"
```

Ressourcenanforderungen, Limits und Verteilung

- Requests/Limits für CPU und Memory je Daemon-Typ
- Zu knappe Limits können zu OOMKills und instabilen Daemons führen
- Zu großzügige Requests verschwenden Cluster-Kapazität
- Beeinflusst auch, wie viele Daemons pro Node gleichzeitig laufen können

Beispiel: Ressourcenlimits für OSDs

```
resources:  
  osd:  
    requests:  
      cpu: "1"  
      memory: "2Gi"  
    limits:  
      cpu: "2"  
      memory: "4Gi"
```

Pool-Replikation und Failure Domains

- Replikationsfaktor legt fest, wie viele Kopien eines Objekts existieren
- Failure Domain legt fest, über welche Ebene Kopien verteilt werden (z. B. Host, Rack)
- Größere Failure Domain erhöht die Ausfallsicherheit
- Änderungen hier lösen typischerweise Recovery/Backfill aus

Beispiel: Replikation und Failure Domain

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  replicated:
    size: 3
  failureDomain: host
```

Exkurs: Recovery vs. Backfill

- **Recovery:** fehlende/veraltete Kopien von Objekten in bestehenden PGs werden wiederhergestellt
 - Typischer Auslöser: OSD fällt kurzzeitig aus und kommt wieder zurück
- **Backfill:** Daten werden komplett neu über PGs/OSDs verteilt
 - Typischer Auslöser: OSD dauerhaft entfernt/hinzugefügt, Replikationsfaktor geändert
- Backfill verschiebt deutlich mehr Daten und belastet den Cluster stärker als Recovery
- Beide Prozesse laufen im Hintergrund, während der Cluster weiter Daten zur Verfügung stellt

Recovery- und Backfill-Strategien

- Steuern, wie aggressiv Ceph nach einem Ausfall wiederherstellt
- Parameter wie Anzahl paralleler Recovery-Operationen beeinflussen die Geschwindigkeit
- Schnellere Recovery bedeutet mehr Last auf Cluster und Netzwerk
- Wichtig, um Wiederherstellung und Produktionslast gegeneinander abzuwägen

Beispiel: Recovery-Parameter setzen

```
spec:
  cephConfig:
    "osd.*":
      osd_mclock_profile: "high_client_ops"
```

```
spec:
  storage:
    nearFullRatio: 0.85
    backfillFullRatio: 0.90
    fullRatio: 0.95
```

`osd_mclock_profile`

- Steuert, wie OSDs Client-I/O und Recovery/Backfill gegeneinander priorisieren
- QoS-Scheduler (mClock) statt einfacher fester Recovery-Limits
- Profile wie `high_client_ops` bevorzugen Anwendungs-I/O gegenüber Recovery
- Andere Profile (z. B. `high_recovery_ops`) beschleunigen die Wiederherstellung auf Kosten der Client-Performance

Kapazitäts-Schwellwerte

- `nearFullRatio`: Cluster warnt (`HEALTH_WARN`), wenn Auslastung diesen Wert überschreitet
- `backfillFullRatio`: OSDs oberhalb dieses Werts nehmen kein Backfill mehr an
- `fullRatio`: Cluster blockiert Schreibzugriffe, sobald dieser Wert erreicht wird
- Reihenfolge ist immer `nearFullRatio` < `backfillFullRatio` < `fullRatio`

Konfigurationsänderungen sicher durchführen

Fragen vor jeder Änderung

- Welche Einstellungen sind im laufenden Betrieb sicher?
- Welche Änderungen lösen Reconciliation oder Rebalancing aus?
- Wie bewertet man Risiken vor einer Konfigurationsänderung?
- Konfiguration vor und nach einer Änderung validieren

Zwei Beispiele im Vergleich

Sicher im laufenden Betrieb

`CephCluster:`

`spec.mgr.count`

erhöhen

Startet zusätzlichen Standby-MGR-Pod

Daten bleiben unberührt

Löst Backfill aus

`CephBlockPool:`

`spec.replicated.size`

erhöhen

Ceph passt Placement Groups an

Datenbewegung zwischen OSDs

Lab

- Gehen Sie auf: <https://learn.corewire.de>
- Navigieren Sie nach:
 - Rook
 - Aufgaben
 - Konfiguration

Zur Kapitelübersicht

- Vorheriges Kapitel: [Architektur](#)
- Nächstes Kapitel: [Administration](#)