Cluster Logging mit Loki

Die kontinuierliche Überwachung eines Kubernetes-Clusters erfordert ein effizientes Logmanagement, welches das zentrale Sammeln, Speichern und Analysieren von Protokolldaten ermöglicht. In Kubernetes-Umgebungen wird hierfür traditionell ein Stack aus Logstash oder Fluentd zur Datenerfassung, Elasticsearch als Speichersystem sowie Kibana oder Graylog zur Analyse und Visualisierung eingesetzt. Neben diesem etablierten Ansatz gewinnt der Loki-Grafana-Stack zunehmend an Bedeutung, da er eine ressourcenschonendere und leichter implementierbare Alternative darstellt.

Loki ist ein Log-Aggregationssystem, das speziell für die Speicherung und Abfrage von Logdaten in Cloud-nativen Architekturen konzipiert wurde. Es verfolgt einen multidimensionalen, labelbasierten Ansatz zur Indizierung und nutzt ein binäres, von externen Abhängigkeiten weitgehend freies Datenformat. Vom Hersteller Grafana Labs wird Loki als „Prometheus-ähnlich, jedoch für Logdaten“ charakterisiert. Im Gegensatz zu Prometheus, das auf die Erhebung von Metriken fokussiert ist, verarbeitet Loki ausschließlich Protokolldaten und setzt hierbei auf ein Push-basiertes Übertragungsverfahren.

Während Kibana und Graylog ein umfangreicheres Funktionsspektrum und fortgeschrittene Analysemöglichkeiten bereitstellen, zeichnet sich Loki durch eine geringere Komplexität und eine beschleunigte Bereitstellung aus, was insbesondere für Entwicklungs- und Testumgebungen vorteilhaft ist.

Ein wesentliches Merkmal von Loki ist die ausschließliche Indizierung von Metadaten, vergleichbar mit dem Vorgehen in Prometheus. Dies reduziert den Ressourcen- und Verwaltungsaufwand im Vergleich zu einer Volltextindizierung erheblich. In Kubernetes-Umgebungen werden primär Labels indiziert, sodass sich Logdaten konsistent mit den Metadaten der jeweiligen Anwendungen verwalten lassen.

Index und Chunks - Loki Datenformat

Loki speichert seine Daten bevorzugt in einem zentralisierten Objektspeicher-Backend, wie beispielsweise Amazon Simple Storage Service (S3), Google Cloud Storage (GCS), Azure Blob Storage oder vergleichbaren Systemen. Für diesen Betriebsmodus wird der sogenannte Index Shipper (kurz: Shipper) als Adapter eingesetzt. Diese Betriebsart wurde mit der Version Loki 2.0 allgemein verfügbar und zeichnet sich durch hohe Effizienz, geringe Kosten und einfache Handhabung aus. Sie stellt den aktuellen Standard dar und bildet zugleich die Grundlage für die zukünftige Weiterentwicklung.

Falls kein Zugriff auf einen Objektspeicher gegeben ist, kann in Test- oder Entwicklungsumgebungen (beispielsweise in Trainings-Labs) ersatzweise lokaler Speicher genutzt werden.

Loki verwaltet zwei grundlegende Datentypen: **Index** und **Chunks**.

Index

Der Index dient als Inhaltsverzeichnis und enthält Verweise auf die Speicherorte der Protokolldaten für eine definierte Gruppe von Labels.

Indexformate Zur Speicherung des Index können unterschiedliche Datenbanken als Backend verwendet werden. Für lokale Setups unterstützt Loki zwei Indexformate im Single Store-Modus:

- **TSDB (Time Series Database):**
  Die TSDB wurde ursprünglich für Prometheus zur Verwaltung von Zeitreihendaten entwickelt und stellt das empfohlene Indexformat für Loki dar. Sie ist erweiterbar, leistungsfähig und bietet zahlreiche Vorteile gegenüber dem veralteten BoltDB-Index. Neue Speicherfunktionen in Loki stehen ausschließlich in Verbindung mit der TSDB zur Verfügung.
- **BoltDB:**
  Bolt ist ein in Go implementierter, transaktionaler Low-Level-Key-Value-Store. Aufgrund fehlender Replikationsunterstützung und eingeschränkter Funktionalität gilt BoltDB als veraltet (deprecated) und ist primär für Entwicklungs- und Testumgebungen geeignet.

Um Indexdateien (unabhängig vom Format: TSDB oder BoltDB) konsistent wie Chunk-Dateien im Objektspeicher zu verwalten, wird ebenfalls der Index Shipper eingesetzt. Darüber hinaus unterstützt Loki auch Cloud-Datenbanken wie BigTable und DynamoDB als Index-Backends.

Chunks

Die eigentlichen Logdaten werden in komprimierter und segmentierter Form im sogenannten Chunk Storage gespeichert. Ein Chunk stellt dabei einen Container für die Log-Einträge einer definierten Label-Gruppe dar und umfasst die Logzeilen eines spezifischen Streams innerhalb eines begrenzten Zeitraums.

Chunk-Formate Wie beim Index können auch für die Chunks verschiedene Speicher-Backends verwendet werden. Aufgrund des typischerweise größeren Datenvolumens empfiehlt sich der Einsatz eines Objektspeichers (z. B. Ceph Object Store, Amazon S3, Azure Blob Storage, Google Cloud Storage oder Swift). Der Objektspeicher übernimmt in der Regel auch die Replikation sowie die automatische Löschung abgelaufener Chunks. Für kleinere Projekte oder Entwicklungsumgebungen ist der Einsatz eines lokalen Dateisystems jedoch ebenfalls möglich.

Warum Loki

Die herkömmliche Methode für das Sammeln und Verwalten von Protokolldaten in Kubernetes-Umgebungen basiert auf einem zentralisierten Logmanagement, das in der Regel durch einen sogenannten Logging-Stack realisiert wird. Dieser besteht typischerweise aus drei Kernkomponenten:

- **Datenerfassung:** Fluentd oder Logstash
- **Datenspeicherung:** Elasticsearch
- **Datenvisualisierung:** Graylog oder Kibana

Neben dieser etablierten Architektur hat sich mit der Kombination aus Loki und Grafana Alloy ein alternativer, ressourcenschonenderer Logging-Stack etabliert.

Loki bietet im Vergleich zu traditionellen Log-Aggregationssystemen eine Reihe spezifischer Vorteile:

- Verzicht auf Volltextindizierung: Statt vollständiger Textindizes speichert Loki komprimierte, unstrukturierte Protokolldaten und indiziert ausschließlich Metadaten. Dies vereinfacht die Handhabung und reduziert die Betriebskosten erheblich. Die Indexierung beschränkt sich auf Metadaten, ähnlich dem Ansatz von Prometheus, wodurch der Verwaltungsaufwand signifikant reduziert wird.
- Integration in bestehende Label-Systeme: Log-Streams werden anhand derselben Labels gruppiert und indiziert, die auch in Prometheus verwendet werden. Dies ermöglicht einen nahtlosen Wechsel zwischen Metriken und Protokolldaten.
- Optimierte Unterstützung für Kubernetes: Loki eignet sich besonders für die Speicherung von Protokolldaten aus Kubernetes-Pods, da relevante Metadaten wie Pod-Labels automatisch erfasst und indexiert werden.
- Native Integration in Grafana: Seit Grafana Version 6.0 ist eine direkte Unterstützung von Loki gegeben.
- Einfache Architektur: Loki ist ein binäres, von externen Abhängigkeiten weitgehend freies System, das eine unkomplizierte Bereitstellung ermöglicht.
- Kosteneffiziente Speicherung: Durch die Nutzung gängiger und kostengünstiger Objektspeicher entfällt die aufwendige Administration eines Elasticsearch-Clusters.

Obwohl Kibana und Graylog ein umfangreicheres Funktionsspektrum bereitstellen, zeichnet sich Loki durch seine geringere Komplexität und den reduzierten Ressourcenbedarf aus, was ihn insbesondere für Entwicklungs- und Testumgebungen prädestiniert.

Komponenten des Logging-Stack Loki und Grafana Alloy

Ein auf Loki basierender Logging-Stack besteht aus drei Kernkomponenten:

1. **Loki**
   Loki bildet die zentrale Komponente des Stacks und verwaltet den Log-Index. Es fungiert als Hauptserver, der eingehende Protokolldaten aggregiert und in das konfigurierte Storage-Backend schreibt. Neben der Speicherung übernimmt Loki die Verarbeitung eingehender Abfragen und dient als Datenquelle für Visualisierungstools wie Grafana. Die Log-Daten werden dabei von vorgelagerten Agenten wie Promtail empfangen.
2. **Grafana Alloy**
   Grafana Alloy übernimmt die Erfassung und Weiterleitung von Log-Daten an Loki. Seine Hauptaufgabe ist es, die Logs aus den verschiedenen Pods und Containern im Cluster zu sammeln, sie zu verarbeiten (z.B. filtern oder Labels hinzufügen) und sie anschließend im richtigen Format an Loki zu senden.
3. **Grafana**
   Grafana ist ein Open-Source-Tool zur Überwachung und Visualisierung von Daten. Es ermöglicht die Erstellung und Verwaltung von Dashboards, welche Daten aus einer Vielzahl von Quellen – darunter InfluxDB, MySQL, PostgreSQL, Prometheus und Graphite – aggregieren können. Durch ein flexibles Plug-in-System lassen sich sowohl die Datenquellen als auch die Darstellungsoptionen erweitern. Im Kontext des Loki-Stacks dient Grafana als Visualisierungskomponente für die Abfrage und Darstellung der erfassten Logs.

Installation von Loki

Das **Loki Helm-Chart** unterstützt drei verschiedene Bereitstellungsmethoden:

- **Monolithic**
- **Simple Scalable**
- **Microservice**

(vgl. https://grafana.com/docs/loki/latest/setup/install/helm/ concepts/Grafana Labs Documentation)

Standardmäßig wird das Chart im **Simple-Scalable-Modus** installiert, welcher für die meisten Anwendungsfälle die empfohlene Betriebsart darstellt.

Storage für Loki
Idealerweise nutzt Loki einen Object Storage wie Ceph, S3 oder GCS. In einfachen (Test-)Umgebungen kann lokaler Storage genutzt werden – mit einigen Einschränkungen u. a. bezüglich Geschwindigkeit und Sicherheit.

Für das Deployment im Rahmen des Trainings-Labs oder in einer Test- oder Entwicklungsumgebungen steht meistens kein Objektspeicher eines externen Storage-Providers zur Verfügung. Stattdessen wird lokaler Speicher in Kombination mit der Methode `deploymentMode: SingleBinary` verwendet.

Das Loki Helm-Chart umfasst standardmäßig **kein integriertes Monitoring**. Für eine übergreifende Überwachung des Loki-Clusters hinsichtlich Logs, Metriken und Traces empfiehlt sich der Einsatz des sogenannten *Meta-Monitoring-Stacks*. Installation des Loki-Stacks auf Kubernetes

Die Bereitstellung von Loki erfolgt üblicherweise mittels Helm, einem Package-Manager für Kubernetes. Das offizielle Helm-Chart beinhaltet die erforderlichen Komponenten für das Trainingsszenario: Loki, Grafana Alloy und Grafana.

Die Installation erfolgt in folgenden Schritten:

1. **Hinzufügen des Helm-Repositories:**

```
helm repo add grafana https://grafana.github.io/helm-charts
```

2. **Installation von Loki in einem separaten Namespace (z.B. `loki`):**

```
helm install loki grafana/loki --values loki-values.yaml --create-namespace -n loki
```

3. **Überprüfung der laufenden Pods und Services:**

```
kubectl get pods,svc -n loki
```

Konfiguration

Zur Anpassung der Loki-Konfiguration kann die aktuelle Standardkonfiguration des Charts mit folgendem Befehl angezeigt und in eine Vorlage exportiert werden:

```
helm show values grafana/loki > config-loki.yaml
```

Auf Basis dieser Datei können die erforderlichen Anpassungen für die eigene Umgebung vorgenommen werden.
`loki-values.yaml`

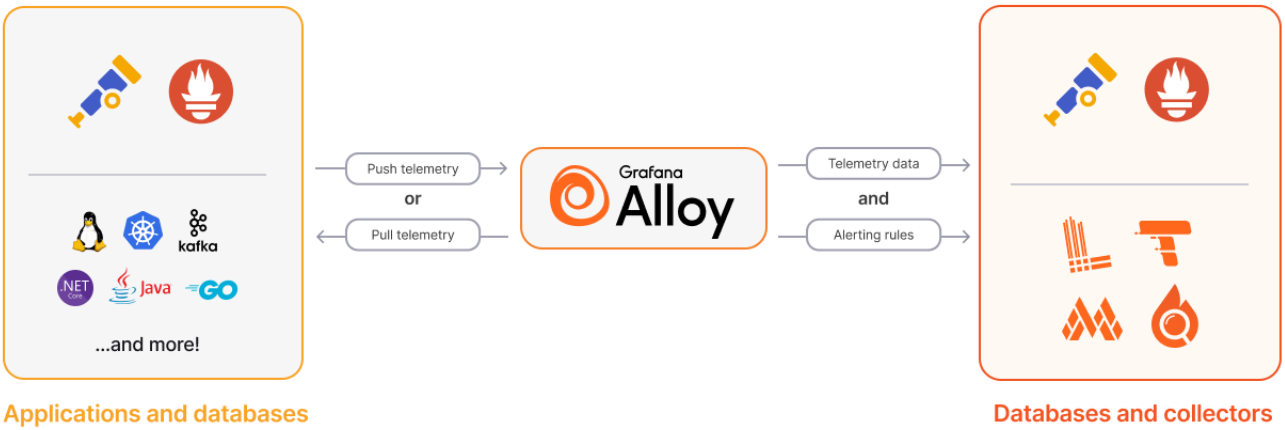
```
loki:
  commonConfig:
    replication_factor: 1
  storage:
    type: 'filesystem'
    bucketNames:
      chunks: chunks
      ruler: ruler
      admin: admin
  schemaConfig:
    configs:
      - from: "2024-04-01"
        store: tsdb
        object_store: filesystem
        schema: v13
        index:
          prefix: loki_index_
          period: 24h
  storage_config:
    filesystem:
      directory: /tmp/loki/chunks
  pattern_ingester:
    enabled: true
  limits_config:
    allow_structured_metadata: true
    volume_enabled: true
  ruler:
    enable_api: true
  auth_enabled: false

minio:
  enabled: false
deploymentMode: SingleBinary

singleBinary:
  replicas: 1
  persistence:
    storageClass: nfs-delete
    accessModes:
```

```
      - ReadWriteOnce
    size: 20Gi

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

sidecar:
  image:
    repository: kiwigrid/k8s-sidecar
    tag: 1.30.0
  resources:
    requests:
      cpu: 50m
      memory: 50Mi
    limits:
      cpu: 100m
      memory: 100Mi

backend:
  replicas: 0
read:
  replicas: 0
write:
  replicas: 0

chunksCache:
  allocatedMemory: 500
```

Weitere Konfigurationsbeispiele sind in der offiziellen Dokumentation verfügbar: Configuration Examples

Einspeisen von Logs in Loki mittels Alloy

Grafana Alloy ist ein vielseitiger Observability-Collector, der in der Lage ist, Protokolldaten in unterschiedlichen Formaten zu erfassen und an Loki weiterzuleiten. Alloy wird als bevorzugte Methode zur Log-Ingestion empfohlen, da es eine robuste und funktionsreiche Grundlage für den Aufbau hochskalierbarer und zuverlässiger Observability-Pipelines bietet.



Komponenten von Alloy für die Log-Verarbeitung

Alloy-Pipelines bestehen aus einzelnen Komponenten, die jeweils spezifische Funktionen erfüllen. Für die Verarbeitung von Log-Daten lassen sich diese in drei Hauptkategorien unterteilen:

- **Collector:**
  Diese Komponenten sind verantwortlich für die Erfassung bzw. den Empfang von Logs aus unterschiedlichen Quellen. Dies kann beispielsweise durch das Auslesen von Log-Dateien, den Empfang von Logs über HTTP oder gRPC oder das Einlesen aus einer Message Queue erfolgen.
- **Transformer:**
  Transformationskomponenten dienen der Manipulation von Logs vor deren Weiterleitung an eine Zielkomponente. Typische Anwendungsfälle sind das Anreichern mit zusätzlichen Metadaten, das Filtern unerwünschter Logs oder das Bündeln (Batching) von Logs zur Optimierung des Transports.
- **Writer:**
  Diese Komponenten sind für die Weiterleitung der verarbeiteten Logs an ein definiertes Zielsystem verantwortlich. Im Rahmen dieser Dokumentation liegt der Fokus auf der Anbindung an Loki, jedoch unterstützt Alloy prinzipiell auch andere Zielsysteme.

Log-Komponenten in Alloy

Nachfolgend findet sich eine nicht abschließende Übersicht von Komponenten, die für den Aufbau einer Log-Pipeline in Alloy verwendet werden können. Für eine vollständige Auflistung aller verfügbaren Komponenten wird auf die offizielle Komponentenübersicht verwiesen.

Type	Component
Collector	loki.source.api
Collector	loki.source.awsfirehose
Collector	loki.source.azure_event_hubs
Collector	loki.source.cloudflare
Collector	loki.source.docker
Collector	loki.source.file
Collector	loki.source.gcplog
Collector	loki.source.gelf

Type	Component
Collector	loki.source.heroku
Collector	loki.source.journal
Collector	loki.source.kafka
Collector	loki.source.kubernetes
Collector	loki.source.kubernetes_events
Collector	loki.source.podlogs
Collector	loki.source.syslog
Collector	loki.source.windowsevent
Collector	otelcol.receiver.loki
Transformer	loki.relabel
Transformer	loki.process
Writer	loki.write
Writer	otelcol.exporter.loki

Installation von Grafana Alloy

Der Alloy lässt sich über Helm Chart installieren. Diesen finden wir im selben Helm-Repository wie unseren Loki Chart. Daher ist kein weiteres Helm Repository hinzufügen nötig.

Die Installation erfolgt in folgenden Schritten:

1. **Installation von Alloy im selben Namespace wie loki:**

```
helm install grafana-alloy grafana/alloy --values alloy-values.yaml -n loki
```

2. **Überprüfung der laufenden Pods und Services:**

```
kubectl get pods,svc -n loki
```

Für die Installation und Konfiguration von Alloy kann eine Values-Datei verwendet werden, in der sämtliche Parameter des Helm-Charts angepasst werden. Die verfügbaren Standardoptionen lassen sich mit folgendem Befehl anzeigen und gleichzeitig in eine Vorlage exportieren:

```
helm show values grafana/loki > config-loki.yaml
```

Auf Basis dieser Konfigurationsvorlage können spezifische Anpassungen vorgenommen werden, um den Betrieb in der eigenen Umgebung sicherzustellen. Ein Beispiel für eine entsprechende Konfiguration wird nachfolgend dargestellt. `alloy-values.yaml`

```
alloy:
  configMap:
    content: |-
      logging {
        level = "debug"
        format = "logfmt"
```

```
      }
      discovery.kubernetes "pods" {
        role = "pod"
      }
      discovery.relabel "pods" {
        targets = discovery.kubernetes.pods.targets

        rule {
          source_labels = ["__meta_kubernetes_namespace"]
          target_label = "namespace"
          action = "replace"
        }

        rule {
          source_labels =
["__meta_kubernetes_pod_label_app_kubernetes_io_name"]
          target_label = "app"
          action = "replace"
        }

        rule {
          source_labels = ["__meta_kubernetes_pod_container_name"]
          target_label = "container"
          action = "replace"
        }

        rule {
          source_labels = ["__meta_kubernetes_pod_name"]
          target_label = "pod"
          action = "replace"
        }
      }
      loki.source.kubernetes "pods" {
        targets    = discovery.relabel.pods.output
        forward_to = [loki.process.process.receiver]
      }
      loki.process "process" {
        forward_to = [loki.write.loki.receiver]

        stage.drop {
          older_than          = "1h"
          drop_counter_reason = "too old"
        }
        stage.match {
          selector = "{instance=~\".*\"}"
          stage.json {
            expressions = {
              level = "\"level\"",
            }
          }
          stage.labels {
```

```
            values = {
              level = "level",
            }
          }
        }
        stage.label_drop {
          values = [ "service_name" ]
        }
      }
      loki.write "loki" {
        endpoint {
          url = "http://loki-gateway/loki/api/v1/push"
        }
      }
  mounts:
    varlog: true
    dockercontainers: true

  resources:
    limits:
      cpu: 200m
      memory: 128Mi
    requests:
      cpu: 100m
      memory: 128Mi
```

Installation von Grafana mittels Helm-Chart

Die Bereitstellung von Grafana erfolgt analog zu Loki über ein offizielles Helm-Chart. Auch hier kann wieder das selbe Repository von Grafana verwendet werden.

```
helm install grafana grafana/grafana -f grafana-values.yaml --create-
namespace -n grafana
```

Für die Installation empfiehlt sich die Verwendung einer vorbereiteten Konfigurationsdatei, welche die notwendigen Parameter für eine einfache Grafana-Instanz enthält. Diese kann unter folgender Adresse abgerufen werden: Grafana Cloud Helm-Konfiguration

Für unsere kleine Übung verwenden wir am besten folgende Konfiguration. Diese beinhaltet bereits ein vorkonfiguriertes Dashboard. `grafana-values.yaml`

```
# grafana-values.yaml - Erweiterte Version mit Dashboards
replicas: 1

resources:
  limits:
    cpu: 300m
```

```
    memory: 512Mi
  requests:
    cpu: 100m
    memory: 256Mi

persistence:
  enabled: true
  size: 5Gi
  storageClassName: null

adminUser: admin
adminPassword: admin123

# Service configuration
service:
  type: ClusterIP
  port: 80

# Datasources
datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
    - name: Loki
      type: loki
      url: http://loki-gateway.loki.svc.cluster.local
      access: proxy
      isDefault: true
      jsonData:
        maxLines: 1000
        derivedFields:
          - datasourceUid: loki
            matcherRegex: "traceID=(\\w+)"
            name: TraceID
            url: "$${__value.raw}"

# Dashboard providers
dashboardProviders:
  dashboardproviders.yaml:
    apiVersion: 1
    providers:
    - name: 'kubernetes'
      orgId: 1
      folder: 'Kubernetes'
      type: file
      disableDeletion: false
      editable: true
      updateIntervalSeconds: 10
      allowUiUpdates: true
      options:
        path: /var/lib/grafana/dashboards/kubernetes
```

```
    - name: 'loki'
      orgId: 1
      folder: 'Loki'
      type: file
      disableDeletion: false
      editable: true
      updateIntervalSeconds: 10
      allowUiUpdates: true
      options:
        path: /var/lib/grafana/dashboards/loki

# Custom Dashboards
dashboards:
  kubernetes:
    kubernetes-cluster-overview:
      json: |
        {
          "annotations": {
            "list": [
              {
                "builtIn": 1,
                "datasource": "-- Grafana --",
                "enable": true,
                "hide": true,
                "iconColor": "rgba(0, 211, 255, 1)",
                "name": "Annotations & Alerts",
                "type": "dashboard"
              }
            ]
          },
          "description": "Kubernetes Cluster Overview with Loki Logs",
          "editable": true,
          "gnetId": null,
          "graphTooltip": 0,
          "id": null,
          "links": [],
          "panels": [
            {
              "datasource": "Loki",
              "fieldConfig": {
                "defaults": {},
                "overrides": []
              },
              "gridPos": {
                "h": 8,
                "w": 24,
                "x": 0,
                "y": 0
              },
```

```
      "id": 1,
      "options": {
        "showLabels": false,
        "showTime": false,
        "sortOrder": "Descending",
        "wrapLogMessage": false,
        "prettifyLogMessage": false,
        "enableLogDetails": true,
        "dedupStrategy": "none"
      },
      "targets": [
        {
          "expr": "{namespace=~\".+\"} |= \"\"",
          "refId": "A"
        }
      ],
      "title": "All Kubernetes Logs",
      "type": "logs"
    },
    {
      "datasource": "Loki",
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "palette-classic"
          },
          "custom": {
            "axisLabel": "",
            "axisPlacement": "auto",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 10,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "vis": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "never",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
```

```
                    "mode": "off"
                  }
                },
                "mappings": [],
                "thresholds": {
                  "mode": "absolute",
                  "steps": [
                    {
                      "color": "green",
                      "value": null
                    },
                    {
                      "color": "red",
                      "value": 80
                    }
                  ]
                },
                "unit": "short"
              },
              "overrides": []
            },
            "gridPos": {
              "h": 8,
              "w": 12,
              "x": 0,
              "y": 8
            },
            "id": 2,
            "options": {
              "legend": {
                "calcs": [],
                "displayMode": "list",
                "placement": "bottom"
              },
              "tooltip": {
                "mode": "single"
              }
            },
            "targets": [
              {
                "expr": "sum(rate({namespace=~\".+\"}[5m])) by
(namespace)",
                "refId": "A"
              }
            ],
            "title": "Log Rate by Namespace",
            "type": "timeseries"
          },
          {
```

```
            "datasource": "Loki",
            "fieldConfig": {
              "defaults": {
                "color": {
                  "mode": "thresholds"
                },
                "custom": {
                  "align": "auto",
                  "displayMode": "auto"
                },
                "mappings": [],
                "thresholds": {
                  "mode": "absolute",
                  "steps": [
                    {
                      "color": "green",
                      "value": null
                    },
                    {
                      "color": "red",
                      "value": 80
                    }
                  ]
                }
              },
              "overrides": []
            },
            "gridPos": {
              "h": 8,
              "w": 12,
              "x": 12,
              "y": 8
            },
            "id": 3,
            "options": {
              "showHeader": true
            },
            "targets": [
              {
                "expr": "topk(10,
sum(count_over_time({namespace=~\".+\"}[5m])) by (namespace, pod_name))",
                "refId": "A"
              }
            ],
            "title": "Top 10 Pods by Log Volume",
            "type": "table"
          }
        ],
        "refresh": "30s",
        "schemaVersion": 30,
        "style": "dark",
```

```
        "tags": ["kubernetes", "logs"],
        "templating": {
          "list": []
        },
        "time": {
          "from": "now-1h",
          "to": "now"
        },
        "timepicker": {},
        "timezone": "",
        "title": "Kubernetes Cluster Overview",
        "uid": "kubernetes-cluster-overview",
        "version": 1
      }

  kubernetes-namespace-logs:
    json: |
      {
        "annotations": {
          "list": []
        },
        "description": "Kubernetes Namespace Specific Logs",
        "editable": true,
        "
```

Testen des Cluster Loggings

Um sich nun das Gesamtergebnis schnell anschauen zu können richtet man am besten ein Port-Forwarding zum Grafana WebUI auf Port 3000 ein.

```
kubectl port-forward service/grafana 3000:80 -n grafana
```