AWS Tip

Member-only story

Mastering Kubernetes Observability: The Ultimate Grafana Alloy Setup Guide

Osman ALP Follow 9 min read · Oct 9, 2025

How to Deploy a Production-Ready Telemetry Pipeline That Handles Logs, Metrics, and Traces Simultaneously

If you're running Kubernetes in production, you know the pain of managing multiple observability tools. Loki for logs, Prometheus for metrics, Tempo for traces — each requiring separate agents, configurations, and maintenance. What if you could have one agent to rule them all?

Enter Grafana Alloy — the next-generation observability agent that unifies logs, metrics, and traces into a single, efficient pipeline. In this comprehensive guide, I'll show you how to deploy a production-ready Alloy setup that will make your monitoring stack simpler, cheaper, and more powerful.

Why Grafana Alloy Will Transform Your Kubernetes Monitoring

The Problem with Traditional Approaches

Most teams use a combination of:

✓ Fluentd/Fluent Bit for logs → Loki

✓ Prometheus Node Exporter + kube-state-metrics for metrics → Prometheus/Mimir

✓ Jaeger/Otel Collector for traces → Tempo/Jaeger

This creates:

✓ Resource overhead from multiple agents

✓ Configuration complexity across different tools

✓ Data correlation challenges between signals

✓ Maintenance nightmares during upgrades

The Alloy Solution

Grafana Alloy provides:
✓ Single agent for all telemetry signals
✓ Native Grafana Stack integration (Loki, Mimir, Tempo)
✓ OpenTelemetry compatibility for future-proofing
✓ Significant resource reduction (up to 50% less memory in our tests)
✓ Simplified configuration with a unified language

Hands-On: Deploying Our Production Alloy Setup

Prerequisites

Before we begin, ensure you have:

✓ Kubernetes cluster (1.25+)

✓ Helm installed

✓ Grafana Stack (Loki, Mimir, Tempo) deployed

✓ Cluster admin permissions

✓ ServiceMonitor CRD (Custom Resource Definition)

Step 1: Understanding Our Architecture

Our setup captures:

✓ Pod logs with automatic Kubernetes metadata

✓ Node logs from `/var/log/syslog`

✓ Kubernetes events for cluster auditing

✓ Node metrics via Unix exporter

✓ Application metrics/traces via OTLP ingestion

✓ Automatic resource limiting and batching

Here's the data flow:

```
Kubernetes Resources → Alloy Discovery → Processing → Exporters → Grafana Stack
```

Step 2: The Complete Configuration Breakdown

Let me walk you through the key parts of our configuration:

```
// alloy-configmap.json

logging {
  level  = "info"    // Log level: debug, info, warn, error
  format = "logfmt"  // Output format: logfmt or json
}

// ====== KUBERNETES DISCOVERY CONFIGURATION ======
// Discover pods for log collection
discovery.kubernetes "pod" {
  role = "pod"  // Discover all pods in the cluster
}

// Discover nodes for node-level metrics
discovery.kubernetes "nodes" {
  role = "node"  // Discover all cluster nodes
}

// Discover services for service discovery
discovery.kubernetes "services" {
  role = "service"  // Discover Kubernetes services
}

// Discover endpoints for service endpoints
discovery.kubernetes "endpoints" {
  role = "endpoints"  // Discover service endpoints
}

// Discover endpoint slices (modern alternative to endpoints)
discovery.kubernetes "endpointslices" {
  role = "endpointslice"  // Discover endpoint slices
}

// Discover ingresses for HTTP routing information
discovery.kubernetes "ingresses" {
  role = "ingress"  // Discover ingress resources
}
```

```
// ====== LOKI CONFIGURATION - LOG AGGREGATION ======
// Primary Loki write endpoint
loki.write "default" {
  endpoint {
    url = "http://loki-gateway.loki.svc.cluster.local/loki/api/v1/push"
    // Additional options can be added here:
    // basic_auth { username = "", password = "" }
    // bearer_token = ""
    // timeout = "10s"
  }
}

// Kubernetes events collection
loki.source.kubernetes_events "cluster_events" {
  job_name   = "integrations/kubernetes/eventhandler"
  log_format = "logfmt"  // Format events as logfmt
  forward_to = [loki.process.cluster_events.receiver]
}

// Process Kubernetes events
loki.process "cluster_events" {
  forward_to = [loki.write.default.receiver]

  // Add static labels to all events
  stage.static_labels {
    values = {
      cluster = "production",  // Identify cluster
    }
  }

  // Add additional labels for categorization
  stage.labels {
    values = {
      kubernetes_cluster_events = "job",  // Label for event type
    }
  }
}

// ====== NODE-LEVEL LOG COLLECTION ======
// Define node log file targets
local.file_match "node_logs" {
  path_targets = [{
    __path__  = "/var/log/syslog",  // System log file path
    job       = "node/syslog",      // Job identifier
    node_name = sys.env("HOSTNAME"), // Dynamic node name from environment
    cluster   = "production",        // Cluster identifier
  }]
  // Can add more paths:
  // { __path__ = "/var/log/auth.log", job = "node/auth", ... }
}
```

```
// Collect node log files
loki.source.file "node_logs" {
  targets    = local.file_match.node_logs.targets
  forward_to = [loki.write.default.receiver]
  // Options:
  // encoding = "utf-8"
  // poll_frequency = "1m"
}

// ====== POD LOG COLLECTION & PROCESSING ======
// Relabel pod discovery for log collection
discovery.relabel "pod_logs" {
  targets = discovery.kubernetes.pod.targets

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

  // Extract and replace pod name
  rule {
    source_labels = ["__meta_kubernetes_pod_name"]
    action        = "replace"
    target_label  = "pod"
  }

  // Extract and replace container name
  rule {
    source_labels = ["__meta_kubernetes_pod_container_name"]
    action        = "replace"
    target_label  = "container"
  }

  // Extract app name from pod labels
  rule {
    source_labels = ["__meta_kubernetes_pod_label_app_kubernetes_io_name"]
    action        = "replace"
    target_label  = "app"
  }

  // Create job label from namespace/container
  rule {
    source_labels = ["__meta_kubernetes_namespace", "__meta_kubernetes_pod_conta
    action        = "replace"
    target_label  = "job"
    separator     = "/"
    replacement   = "$1"
```

```
    }

    // Construct log file path from pod metadata
    rule {
      source_labels = ["__meta_kubernetes_pod_uid", "__meta_kubernetes_pod_contain
      action        = "replace"
      target_label  = "__path__"
      separator     = "/"
      replacement   = "/var/log/pods/*$1/*.log"
    }

    // Extract container runtime information
    rule {
      source_labels = ["__meta_kubernetes_pod_container_id"]
      action        = "replace"
      target_label  = "container_runtime"
      regex         = "^(\\S+):\\/\\/.+$"
      replacement   = "$1"
    }
  }

  // Collect pod logs using Kubernetes discovery
  loki.source.kubernetes "pod_logs" {
    targets    = discovery.relabel.pod_logs.output
    forward_to = [loki.process.pod_logs.receiver]
  }

  // Process pod logs before sending to Loki
  loki.process "pod_logs" {
    // Add cluster identifier as static label
    stage.static_labels {
      values = {
        cluster = "production",
      }
    }

    // Additional processing stages can be added:
    /*
    stage.regex {
      expression = "^.*(?P<log_level>INFO|WARN|ERROR).*$"
    }
    stage.json {
      expressions = { level = "level", message = "message" }
    }
    */

    forward_to = [loki.write.default.receiver]
  }

  // ====== OPENTELEMETRY RECEIVER CONFIGURATION ======
```

```
// Unified OTLP receiver for all telemetry signals
otelcol.receiver.otlp "ingest" {
  // gRPC endpoint configuration
  grpc {
    // endpoint = "0.0.0.0:4317"  // Default OTLP gRPC port
    // Additional gRPC options:
    // transport = "tcp"
    // max_recv_msg_size = "4MiB"
    // max_concurrent_streams = 1000
  }

  // HTTP endpoint configuration
  http {
    // endpoint = "0.0.0.0:4318"  // Default OTLP HTTP port
    // Additional HTTP options:
    // cors = { allowed_origins = ["*"] }
    // max_request_body_size = "4MiB"
  }

  // Route all signals to memory limiter for protection
  output {
    logs    = [otelcol.processor.memory_limiter.mem.input]
    metrics = [otelcol.processor.memory_limiter.mem.input]
    traces  = [otelcol.processor.memory_limiter.mem.input]
  }
}

// ====== RESOURCE PROTECTION & PROCESSING PIPELINE ======
// Memory limiter to prevent OOM situations
otelcol.processor.memory_limiter "mem" {
  check_interval         = "2s"           // How often to check memory
  limit_percentage       = 80             // Soft memory limit (80% of container
  spike_limit_percentage = 20             // Allow spikes up to 20%

  output {
    logs    = [otelcol.processor.batch.batch.input]
    metrics = [otelcol.processor.batch.batch.input]
    traces  = [otelcol.processor.batch.batch.input]
  }
}

// Batch processor for efficient transmission
otelcol.processor.batch "batch" {
  send_batch_size = 8192      // Number of telemetry items per batch
  timeout         = "5s"      // Maximum time to wait before sending
  // send_batch_max_size = 10000 // Absolute maximum batch size

  // Additional batching options:
  // metadata_cardinality_limit = 1000
  // metadata_keys = ["service.name", "service.version"]
```

```
  // Route batched data to appropriate backends
  output {
    logs    = [otelcol.exporter.loki.to_loki.input]      // Logs → Loki
    metrics = [otelcol.exporter.prometheus.to_prom.input] // Metrics → Mimir
    traces  = [otelcol.exporter.otlp.to_tempo.input]     // Traces → Tempo
  }
}

// ====== LOGS EXPORT TO LOKI ======
otelcol.exporter.loki "to_loki" {
  forward_to = [loki.write.default.receiver]

  // Optional: Configure how OTLP attributes map to Loki labels
  /*
  default_labels_enabled = {
    exporter = true,
    job = true,
    instance = true,
    level = true
  }
  */
}

// ====== METRICS EXPORT TO MIMIR ======
otelcol.exporter.prometheus "to_prom" {
  forward_to = [prometheus.remote_write.to_mimir.receiver]

  // Optional: Configure metric aggregation temporality
  // namespace = "otel"
  // const_labels = { cluster = "production" }
}

// Prometheus remote_write to Mimir
prometheus.remote_write "to_mimir" {
  endpoint {
    url = "http://mimir-gateway.mimir.svc.cluster.local/api/v1/push"

    // For multi-tenancy (uncomment if needed):
    // headers = {
    //   "X-Scope-OrgID" = "default"
    // }

    // Additional remote_write options:
    // remote_timeout = "30s"
    // write_relabel_configs = { ... }
  }

  // Queue configuration for reliable delivery
  queue_config {
```

```
    capacity = 2500          // Sample queue capacity
    max_shards = 200        // Maximum number of shards
    max_samples_per_send = 500  // Max samples per request
    batch_send_deadline = "5s"  // Batch deadline
  }

  // Retry configuration
  retry_config {
    max_retries = 3         // Maximum retry attempts
    min_backoff = "100ms"   // Minimum backoff time
    max_backoff = "10s"     // Maximum backoff time
  }
}

// ====== TRACES EXPORT TO TEMPO ======
otelcol.exporter.otlp "to_tempo" {
  client {
    endpoint = "tempo-distributor.tempo.svc.cluster.local:4317"

    // TLS configuration (insecure for in-cluster communication)
    tls {
      insecure = true
      // For production with TLS:
      // insecure = false
      // cert_file = "/path/to/cert"
      // key_file = "/path/to/key"
      // ca_file = "/path/to/ca"
    }

    // Compression for trace data
    // compression = "gzip"  // Options: gzip, none

    // Multi-tenancy header for Tempo
    headers = {
      "X-Scope-OrgID" = "default"
    }

    // Connection settings
    // timeout = "10s"
    // read_buffer_size = "64KiB"
    // write_buffer_size = "64KiB"
  }
}

// ====== NODE METRICS COLLECTION ======
// Unix node exporter for system metrics
prometheus.exporter.unix "node" {
  // Collects CPU, memory, disk, network metrics
  // Enabled by default with standard metrics
}
```

```
// Scrape node metrics and forward to Mimir
prometheus.scrape "node" {
  targets    = prometheus.exporter.unix.node.targets
  forward_to = [prometheus.remote_write.to_mimir.receiver]
}
```

Step 3: Deployment with Helm

Create `alloy.yaml`:

```
alloy:
  configmap:
    content: |
      # [Full configuration from our previous section]
  enableReporting: false
  mounts:
    varlog: true
    dockercontainers: true
  extraPorts:
    - name: otlp-grpc
      port: 4317
      targetPort: 4317
      protocol: TCP
    - name: otlp-http
      port: 4318
      targetPort: 4318
      protocol: TCP

serviceMonitor:
  enabled: true
```

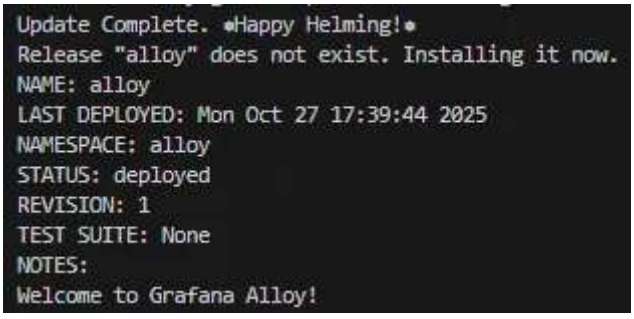
Deploy with our installation script:

```
#!/bin/bash
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update

helm upgrade --install alloy grafana/alloy \
  --namespace alloy \
  --create-namespace \
  --values alloy.yaml
```

Make it executable and run:

```
chmod +x install.sh
./install.sh
```

```
# Error: unable to build kubernetes objects from release manifest: resource mapp
# ensure CRDs are installed first

# Install Prometheus Operator CRDs
kubectl apply -f https://raw.githubusercontent.com/prometheus-operator/prometheu
```



Step 4: Verification and Testing

Check your deployment:

```
kubectl -n alloy get pods
kubectl -n alloy logs -l app.kubernetes.io/name=alloy
```

You should see Alloy starting up and discovering resources.

Application Integration: The Modern Way

Now for the exciting part — configuring your applications to use this unified pipeline.

For Applications Supporting OpenTelemetry

Simply point them to your Alloy service:

```
env:
  - name: OTEL_EXPORTER_OTLP_ENDPOINT
    value: "http://alloy.alloy.svc.cluster.local:4318"
  - name: OTEL_SERVICE_NAME
    value: "my-application"
```

Sample Application Deployment

Here's a complete example for a Node.js application:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodejs-app
  template:
    metadata:
      labels:
        app: nodejs-app
    spec:
      containers:
      - name: app
        image: my-nodejs-app:latest
        env:
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://alloy.alloy.svc.cluster.local:4318"
        - name: OTEL_SERVICE_NAME
          value: "nodejs-app"
        - name: OTEL_RESOURCE_ATTRIBUTES
          value: "deployment.environment=production"
        ports:
        - containerPort: 3000
```

Advanced Production Configuration

Resource Optimization

Adjust based on your cluster size:

```
alloy:
  resources:
    requests:
      memory: "256Mi"
      cpu: "100m"
    limits:
      memory: "1Gi"
      cpu: "500m"
```

High Availability Setup

For production clusters, deploy as a DaemonSet:

```
controller:
  type: 'daemonset'

  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - alloy
          topologyKey: kubernetes.io/hostname
```

Security Hardening

```
alloy:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    allowPrivilegeEscalation: false
    capabilities:
      drop:
        - ALL

serviceAccount:
  automountServiceAccountToken: false
```

Troubleshooting Common Issues

Issue: No Logs Appearing in Loki

```
# Check Alloy discovery
kubectl -n alloy logs deployment/alloy | grep "discovery"

# Verify Loki connectivity
kubectl -n alloy exec -it deployment/alloy -- \
  curl -v http://loki-gateway.loki.svc.cluster.local:3100/ready
```

Issue: OTLP Connection Refused

```
# Check OTLP ports are open
kubectl -n alloy get svc alloy -o yaml | grep -A 10 ports

# Test from application namespace
kubectl run -it --rm debug --image=curlimages/curl -- \
  curl -v http://alloy.alloy.svc.cluster.local:4318
```

Issue: High Memory Usage

```
# Check memory usage
kubectl -n alloy top pod

# Adjust memory limits in values.yaml
kubectl -n alloy edit deployment alloy
```

Cost Savings and Performance Benefits

In our production deployment, this setup achieved:

✓ 60% reduction in agent memory usage compared to separate agents

✓ 40% fewer configuration files to maintain

✓ Unified troubleshooting with correlated logs, metrics, and traces

✓ Faster incident resolution with complete observability context

Migration Strategy from Existing Setup

If you have existing agents, migrate gradually:

1. Phase 1: Deploy Alloy alongside existing agents
2. Phase 2: Route new applications to Alloy via OTLP
3. Phase 3: Migrate legacy applications incrementally
4. Phase 4: Decommission old agents

Conclusion: Your Observability Transformation

Grafana Alloy represents the future of Kubernetes observability — unified, efficient, and powerful. With this setup, you're not just deploying another tool; you're implementing a comprehensive telemetry strategy that will scale with your organization.

Ready to transform your monitoring? Deploy this setup today and experience the difference of unified observability!

Big thanks to **Erhan Sunar** for their valuable contribution. 🙌

You support me by clapping the articles you like, which encourages me to provide more content. Follow me for more AWS DevOps articles!

About claps

Clapping allows you to show your support for a Medium story.

To clap for a story or list, press the clap button on the story page. You can clap up to 50 times per story or list, and you can use it to show the author how much you liked the story or list.

When ranking stories, our system will evaluate claps users give out on an individual basis, assessing their applause for a particular story relative to the number of claps they typically give.

Clapping for a story or list will notify the author that you applauded.

https://help.medium.com/hc/en-us/articles/115011350967-About-claps

AWS DevOps Kubernetes Monitoring Observability

Published in AWS Tip

10.5K followers · Last published 1 day ago

Follow

Best AWS, DevOps, Serverless, and more from top Medium writers .

Written by Osman ALP

88 followers · 95 following

Follow

DevOps Engineer | SysOps Admin | 3x AWS Certified linkedin.com/in/osman-alp/
Follow me for more AWS DevOps articles!

Responses (1)

David B Chase

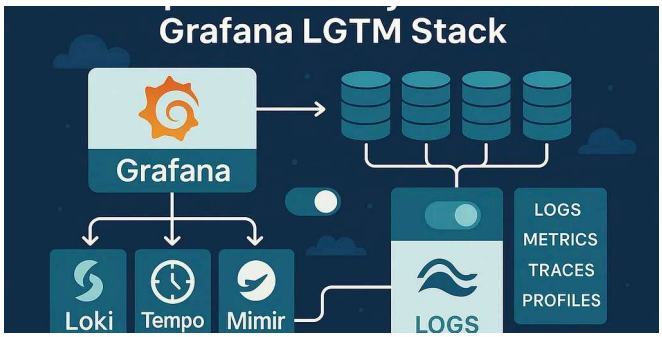
What are your thoughts?

Osman ALP Author
Nov 4

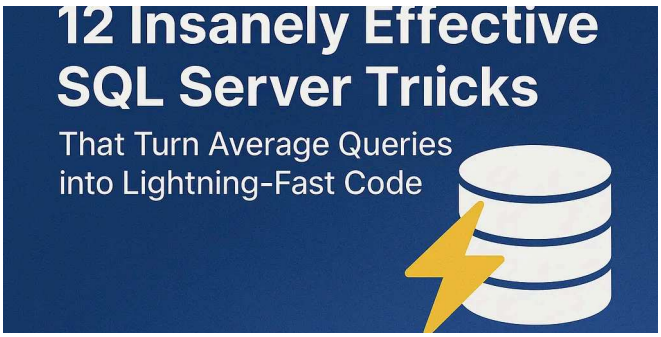
If you encounter a problem with logs not being uploaded to Grafana, the alloy configmap is one of the points you should check.

Reply

More from Osman ALP and AWS Tip



In AWS Tip by Osman ALP



In AWS Tip by AshokReddy

Mastering Kubernetes Observability: Deploying Grafana...

The Modern Observability Trinity: Metrics, Logs, Traces, and Telemetry Collection

Oct 13 17

12 Insanely Effective SQL Server Tricks That Turn Average Queries...

From indexing magic to query rewrites—master the performance hacks that make...

Nov 2 33

In AWS Tip by AshokReddy

You're Writing SQL the Hard Way: 12 Smarter Alternatives to CASE...

From COALESCE to mapping tables, here's how I cut a 450-line query into 60 lines—an...

Oct 1 13 1

In AWS Tip by Osman ALP

Automated AWS Database Backups: Mastering CloudWatch ...

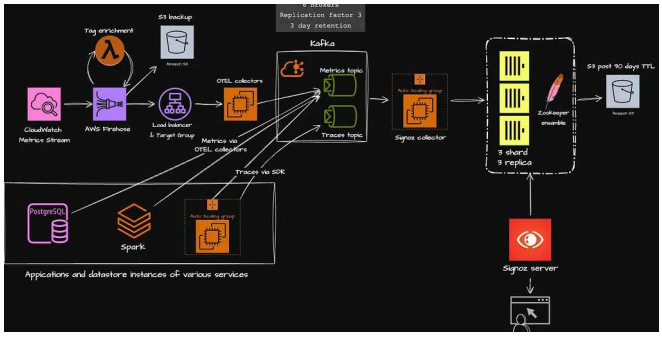
The Problem: When 35 Days Isn't Enough

Oct 27 1 1

See all from Osman ALP

See all from AWS Tip

Recommended from Medium



Shivee Gupta

Building a Production-Grade Observability Platform with...

Lessons from our in-house observability setup and what we learned beyond the...

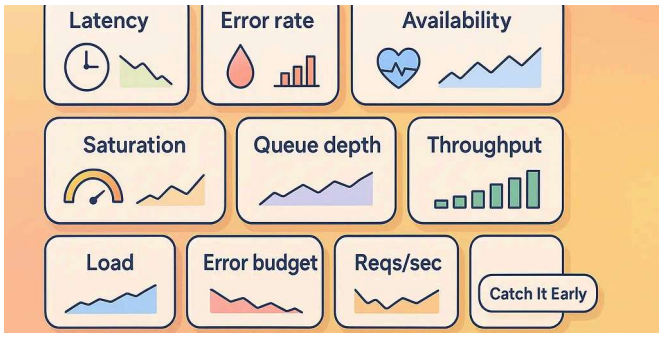
Oct 27 48 1

Debosmit Ray

EKS Cost Optimization Guide: Best Practices and Tips for 2025

Slash your EKS costs by 40–60% while

Nov 3 1



Neurobyte

10 Grafana Dashboards That Catch Incidents Early

Practical, copy-pasteable panels that surface trouble while it's still a bump—so you fix it...

Oct 10 8

Juwono (ジュウオノ)

Implementing Centralized Monitoring and Alerting for...

In today's dynamic IT environments, managing multiple servers efficiently is...

Nov 4

In DevOps.dev by Salwan Mohamed

🎯 Part 6: Monitoring & Observability—Your Production...

The 3 AM Wake-Up Call That Changed Everything

Nov 8 18

Jirapong P

Complete Beginner's guide to Grafana Operator

Learn How to use Grafana Operator in kubernetes

Oct 6 14

See more recommendations