OpenShift Kubernetes Cluster

Monitoring & Observability

Technical Documentation & Best Practices

On-Premises Environment

Document Version	1.0
Date	February 2026
Environment	On-Premises

Table of Contents

1. Logging Infrastructure

Comprehensive logging infrastructure is essential for troubleshooting, auditing, and compliance in enterprise OpenShift Kubernetes clusters. This section covers centralized log management, retention policies, and application log configuration strategies.

1.1 Centralized Log Management (ELK Stack)

Overview

The ELK Stack (Elasticsearch, Logstash, Kibana) provides a powerful, scalable solution for centralized log aggregation, processing, and visualization. This architecture enables real-time log analysis, troubleshooting, and insights across all cluster components.

ELK Stack Architecture

- Elasticsearch: Distributed search and analytics engine that stores and indexes logs

- Logstash: Data processing pipeline that ingests, processes, and forwards logs to Elasticsearch
- Kibana: Web interface for visualization, exploration, and analysis of logs
- Beats: Lightweight data shippers (Filebeat, Metricbeat) for collecting logs and metrics

Deployment Architecture

- Elasticsearch Cluster: Deploy multi-node Elasticsearch cluster for high availability and scalability
- Node Types: Configure master, data, and ingest nodes for optimal performance
- Persistent Storage: Attach persistent volumes for Elasticsearch data persistence
- Logstash Pipelines: Deploy Logstash instances as DaemonSets or Deployments

Log Collection Setup

- Filebeat Configuration: Deploy Filebeat DaemonSet to collect logs from all nodes
- Log Sources: Collect container logs, kubelet logs, and application logs
- Parsing Filters: Configure Grok patterns and field extraction
- Enrichment: Add metadata (node names, pod names, namespaces) to logs

Elasticsearch Configuration

- Index Management: Implement index lifecycle policies (ILP) for automated index rotation
- Shard Allocation: Configure appropriate shard counts and replicas for performance
- Heap Memory: Allocate adequate JVM heap (minimum 4GB, recommended 8-16GB)
- Search Performance: Optimize queries and implement appropriate indexing strategies

Kibana Dashboards

- Cluster Health: Monitor Elasticsearch cluster status and node availability
- Node Metrics: Track resource utilization (CPU, memory, disk) across nodes
- Pod Logs: View and search logs from specific pods and containers
- Troubleshooting Views: Create dashboards for common troubleshooting scenarios

1.2 Log Rotation & Retention Policies

Overview

Log retention policies balance storage requirements with compliance needs and troubleshooting capabilities. Proper rotation and archival strategies prevent storage exhaustion and maintain system performance.

Retention Strategy

- Hot Data: Recent logs kept in Elasticsearch for fast access (7-14 days)
- Warm Data: Older logs moved to secondary storage (2-4 weeks)
- Cold Data: Archive storage for long-term retention (months to years)
- Delete: Remove logs after retention period expires

Index Lifecycle Policy (ILP)

- Rollover: Automatically create new indices based on size or time
- Phase Transitions: Define policies for moving indices between hot, warm, and cold phases
- Deletion Schedule: Configure automatic index deletion after retention period
- Force Merge: Optimize indices during warm phase for better compression

Log Rotation Mechanics

- Time-Based Rotation: Create new indices daily, weekly, or monthly
- Size-Based Rotation: Rotate when index exceeds size threshold (typically 50GB)
- Index Naming: Use consistent naming conventions (e.g., logs-2026.02.05)
- Template Management: Use index templates for consistent settings and mappings

Storage Optimization

- Compression: Enable index compression to reduce storage footprint
- Searchable Snapshots: Use Elasticsearch snapshot API for archive storage
- S3 Repository: Configure remote S3-compatible storage for snapshots
- Capacity Planning: Monitor storage usage and plan for growth

1.3 Application Log Configuration

Overview

Proper application log configuration ensures consistent, structured logging that facilitates debugging, monitoring, and compliance. Applications should emit logs in a format that integrates seamlessly with centralized logging infrastructure.

Log Levels

- DEBUG: Detailed information for diagnosing issues (disabled in production)
- INFO: General informational messages about application state
- WARNING: Warning messages for potentially problematic situations
- ERROR: Error messages indicating failures or exceptions
- CRITICAL: Critical failures requiring immediate attention

Structured Logging

- JSON Format: Output logs in JSON format for easy parsing and analysis
- Standard Fields: Include timestamp, level, message, and context fields
- Correlation IDs: Implement request ID tracing for distributed tracing
- Context Data: Include user ID, session ID, and business identifiers

Logging Best Practices

- Avoid Logging Sensitive Data: Never log passwords, tokens, or PII
- Use Appropriate Log Levels: Don't over-log to DEBUG level in production

- Include Stack Traces: Attach full stack traces for exceptions
- Performance Consideration: Minimize logging overhead through buffering

Framework Integration

- Java Applications: Use Log4j2, SLF4J with JSON layout
- Python Applications: Use Python logging with JSON formatter
- Node.js Applications: Use Winston or Bunyan for structured logging
- Environment Variables: Configure log levels via environment variables

2. Monitoring Setup

Comprehensive monitoring setup provides visibility into cluster health, application performance, and infrastructure metrics. This section covers metrics collection, health checks, and alerting systems.

2.1 Metrics Collection (Prometheus/Grafana)

Overview

Prometheus is a time-series database for collecting and storing metrics, while Grafana provides visualization and analytics capabilities. Together, they form the foundation for comprehensive infrastructure monitoring.

Prometheus Architecture

- Prometheus Server: Core component that scrapes metrics from targets
- Scrape Configuration: Define targets, scrape intervals, and authentication
- Storage: Time-series database with retention policies
- Query Engine: PromQL for querying and aggregating metrics

Exporter Configuration

- Node Exporter: Collect hardware and OS metrics from cluster nodes
- kube-state-metrics: Export Kubernetes resource metrics
- Application Exporters: Custom exporters for application-specific metrics
- Database Exporters: PostgreSQL, MongoDB, Redis exporters

Key Metrics

- Node Metrics: CPU, memory, disk I/O, network bandwidth
- Pod Metrics: CPU and memory usage, restart counts
- Application Metrics: Request rates, response times, error rates
- Database Metrics: Query performance, connection pools, replication lag

Grafana Dashboards

- Cluster Overview: High-level health and resource utilization
- Node Performance: Per-node CPU, memory, disk, and network metrics
- Application Health: Request throughput, latency, error rates

- Database Performance: Query metrics, connection pools, replication status

2.2 Health Check Endpoints

Overview

Health check endpoints allow Kubernetes and monitoring systems to determine application status and health. Properly configured health checks enable automatic recovery and alerting.

Liveness Probes

- Purpose: Determine if container is alive and should be restarted
- HTTP Probes: Implement GET endpoint that returns 200 for healthy state
- TCP Probes: Check if service port is listening
- Command Probes: Execute custom health check script

Readiness Probes

- Purpose: Determine if pod is ready to serve traffic
- Startup Checks: Verify all dependencies are initialized
- Database Connectivity: Check database connections are available
- Cache Warm-up: Ensure caches are populated before accepting traffic

Health Check Implementation

- Endpoints: Implement /health and /ready endpoints in applications
- Response Format: Return JSON with health status and component details
- Timing: Set appropriate initial delay, timeout, and period values
- Metrics Export: Include health check results in metrics for alerting

2.3 Alerting & Notification System

Overview

Alerting systems detect anomalies and notify operations teams of critical issues. Effective alerting balances sensitivity with alert fatigue prevention.

Prometheus Alerting Rules

- Alert Definition: Write alert rules in YAML with threshold conditions
- For Clause: Specify evaluation duration to prevent flaky alerts
- Labels: Attach severity, team, and service labels for routing
- Annotations: Include descriptions and runbook links

AlertManager Configuration

- Alert Routing: Route alerts to different channels by severity/team
- Grouping: Group related alerts to reduce notification volume
- Inhibition: Suppress low-severity alerts when critical alerts fire
- Silencing: Temporarily suppress alerts during maintenance

Notification Channels

- Email: For non-urgent alerts and incident reports
- Slack: Real-time alerts to team channels
- PagerDuty: On-call integration for critical incidents
- SMS/Phone: Critical alerts requiring immediate attention

Alert Definition Examples

- High CPU Usage: Alert when pod CPU exceeds 80% for 5 minutes
- Memory Pressure: Alert when memory usage exceeds 90%
- Pod Restarts: Alert on excessive pod restart frequency
- Database Connection Exhaustion: Alert when connections approach limits

3. Tracing & Performance

Distributed tracing and performance monitoring provide deep insights into application behavior across service boundaries. This section covers tracing infrastructure, APM, and business metrics tracking.

3.1 Distributed Tracing (Jaeger/Zipkin)

Overview

Distributed tracing systems like Jaeger and Zipkin track requests across multiple services in microservice architectures. They enable visualization of request flows and identification of performance bottlenecks.

Jaeger Architecture

- Jaeger Client: Library for instrumenting application code
- Jaeger Agent: Local agent receiving traces from clients
- Jaeger Collector: Receives spans and writes to storage backend
- Query Service: Retrieves and displays traces in web UI

Instrumentation

- OpenTelemetry: Use standardized instrumentation API
- Service Spans: Create spans for each service interaction
- Database Spans: Track database operations and query timing
- Context Propagation: Pass trace context through service calls

Span Attributes

- HTTP Methods: Track GET, POST, PUT, DELETE operations
- Status Codes: Record HTTP response codes
- Error Tags: Mark spans with error information
- Custom Tags: Add business-relevant metadata

Trace Analysis

- Latency Breakdown: Identify slowest service components
- Error Detection: Trace failed requests to origin service
- Critical Path: Visualize request flow through services
- Performance Trends: Analyze trace metrics over time

3.2 Application Performance Monitoring

Overview

APM platforms provide comprehensive visibility into application behavior, capturing metrics about code execution, resource usage, and performance characteristics.

APM Components

- Transaction Tracing: Track individual requests through application
- Code Profiling: Analyze CPU usage at code level
- Memory Analysis: Track memory allocation and leaks
- Error Tracking: Capture and analyze application exceptions

Performance Metrics

- Response Time: Measure end-to-end request latency
- Throughput: Count requests processed per second
- Error Rate: Track percentage of failed requests
- Resource Usage: Monitor CPU, memory, and I/O consumption

Implementation Tools

- Datadog: Full-stack monitoring with APM, infrastructure, and logs
- New Relic: Application performance monitoring and analytics
- Dynatrace: AI-powered application monitoring
- Open Source: Elastic APM, Zipkin, or Jaeger

3.3 Business Metrics Tracking

Overview

Business metrics provide insight into application value delivery and user experience. These metrics connect infrastructure and application health to business outcomes.

Key Business Metrics

- User Engagement: Track active users and session counts
- Conversion Rates: Monitor purchase and sign-up completion
- Revenue Metrics: Track transaction volume and value
- User Experience: Measure page load time and feature adoption

Implementation Approach

- Event Tracking: Instrument application to emit business events
- Data Pipeline: Aggregate events to analytics platform
- Dashboards: Create business intelligence dashboards
- Alerts: Set up alerts for anomalies in business metrics

Correlation Analysis

- Incident Impact: Measure business impact of infrastructure issues
- Performance Correlation: Link application latency to user metrics
- Feature Performance: Analyze impact of new deployments
- Cost Analysis: Track infrastructure costs against revenue

Conclusion

Comprehensive monitoring and observability capabilities are essential for operating reliable OpenShift Kubernetes clusters. A well-designed monitoring infrastructure enables rapid incident detection, efficient troubleshooting, and continuous performance optimization.

Organizations should implement monitoring at multiple levels: infrastructure metrics, application performance, and business outcomes. Integration of logging, metrics, and tracing provides complete visibility into system health and enables data-driven operational decisions.

Regular review of monitoring strategies, alert tuning, and dashboard refinement ensures the monitoring infrastructure remains effective as applications and infrastructure evolve.