Kubernetes Observability Architecture with Grafana, Prometheus, and Loki

Author: Arman Singh Kshatri

March 31, 2025

Contents

1 Abstract

This document outlines a high-level design for implementing an observability solution using Kubernetes. The architecture consists of three primary components: Application Container, Grafana Container, and Loki Container, along with Prometheus for metrics collection. The solution aims to address the debugging challenges faced by engineering teams, including slow, repetitive, manual, and inconsistent troubleshooting processes. By implementing this observability stack, teams can benefit from automated monitoring, centralized logging, and intuitive visualization of performance metrics, ultimately reducing Mean Time to Resolution (MTTR) and decreasing dependency on experienced engineers for troubleshooting.

2 Technical Details

2.1 Kubernetes Architecture

Our observability solution leverages Kubernetes to run essential components within the same namespace:

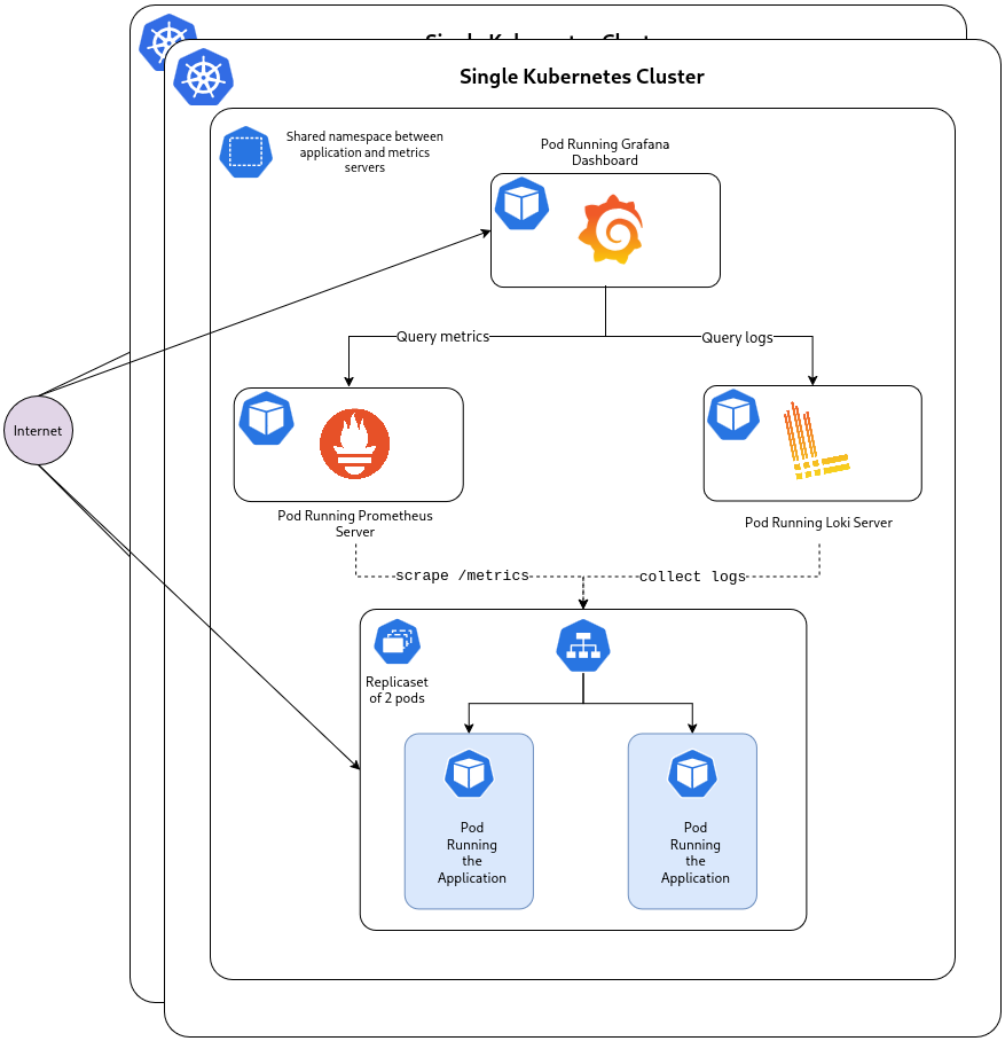

Figure 1: High-Level Design of Kubernetes Observability Architecture

- **Application Container:** Houses the core application serving API endpoints with business logic.
- **Grafana Container:** Manages visualization, dashboarding, and alerting.
- **Prometheus Container:** Handles metrics collection and storage.
- **Loki Container:** Handles log aggregation, storage, and querying.

2.2 Container Communication

The containers within the Kubernetes cluster communicate using the following mechanisms:

- Kubernetes service discovery enables direct communication between pods.
- Each service is exposed via Kubernetes Service objects.
- ConfigMaps and Secrets manage configuration and sensitive data.

2.3 Data Flow Process

The observability data flows through the system through several key pathways:

1. **Log Collection Flow:**
   - Application container generates logs during operation.
   - Fluent Bit sidecar collects logs from the application container.
   - Logs are sent to Loki for storage and indexing.
2. **Metrics Collection Flow:**
   - Application exposes Prometheus metrics endpoint.
   - Prometheus scrapes metrics from the application.
   - Key metrics include response times, error rates, and resource utilization.
3. **Visualization and Alerting Flow:**
   - Grafana connects to both Loki (for logs) and Prometheus (for metrics).
   - Dashboards in Grafana provide visualizations of both logs and metrics.
   - Alerts can be configured based on thresholds for critical metrics.

2.4 Security Considerations

Security is an essential aspect of the observability architecture:

- Kubernetes RBAC controls access to resources.
- Network policies restrict pod-to-pod communication.
- Authentication is implemented for Grafana access.
- Secrets management for sensitive data.

2.5 Storage Requirements

Storage is a critical consideration for the observability stack:

- Loki requires persistent storage for log retention.
- Prometheus needs storage for metrics data.
- Kubernetes PersistentVolumes provide storage for both components.

3 Implementation

3.1 Kubernetes Deployment Configuration

The core of the implementation is the Kubernetes deployment configuration. Here's an example of the key components:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: slow-server
  namespace: slow-server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: slow-server
  template:
    metadata:
      labels:
```

```
        app: slow-server
    spec:
      containers:
      - name: slow-server
        image: slow-server:latest
        ports:
        - containerPort: 8080
        env:
        - name: SERVER_PORT
          value: "8080"
        - name: SIMULATE_ERRORS
          value: "true"
        - name: ERROR_RATE
          value: "0.15"
      - name: fluent-bit
        image: fluent/fluent-bit:2.2.0
        volumeMounts:
        - name: fluent-bit-config
          mountPath: /fluent-bit/etc/
        - name: varlog
          mountPath: /var/log
```

3.2 Application Instrumentation

The application is instrumented using Prometheus client libraries to emit metrics:

```
// Metrics definition
var (
    RequestsTotal = promauto.NewCounterVec(
        prometheus.CounterOpts{
            Name: "http_requests_total",
            Help: "Total number of HTTP requests",
        },
        []string{"path", "method", "status"},
    )

    DBQueriesTotal = promauto.NewCounter(
        prometheus.CounterOpts{
            Name: "db_queries_total",
            Help: "Total number of database queries",
        },
    )

    ExternalAPICallsTotal = promauto.NewCounter(
        prometheus.CounterOpts{
            Name: "external_api_calls_total",
            Help: "Total number of external API calls",
        },
    )
)
```

3.3 Grafana Configuration

Grafana is configured with Prometheus and Loki as data sources:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: slow-server
data:
  prometheus.yaml: |
```

```
    apiVersion: 1
    datasources:
    - name: Prometheus
      type: prometheus
      url: http://prometheus:9090
      access: proxy
      isDefault: true
    - name: Loki
      type: loki
      url: http://loki:3100
      access: proxy
```

3.4 Loki Configuration

Loki is configured to efficiently collect and store logs:

```
auth_enabled: false

server:
  http_listen_port: 3100

schema_config:
  configs:
    - from: 2020-10-24
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      index:
        prefix: index_
        period: 24h

storage_config:
  boltdb_shipper:
    active_index_directory: /loki/boltdb-shipper-active
    cache_location: /loki/boltdb-shipper-cache
    cache_ttl: 24h
    shared_store: filesystem
  filesystem:
    directory: /loki/chunks
```

4 Monitoring Metrics

4.1 Key API Performance Metrics

The following metrics are monitored for API performance:

1. **Request/Response Metrics:**
   - **Response Time:** Measured using Prometheus histograms.
   - **Error Rates:** Tracked via Prometheus counters.
   - **Throughput:** Requests per second by endpoint.
2. **Resource Utilization Metrics:**
   - **CPU Usage:** Container CPU utilization.
   - **Memory Usage:** Container memory consumption.
   - **Network I/O:** Bytes transmitted and received.
3. **Dependency Metrics:**
   - **Database Query Performance:** Query execution times and error rates.
   - **External API Call Performance:** Response times and failure rates.

4.2 Log Management Strategy

Effective logging is implemented through:

- **Structured Logging:** Using JSON format for all logs.
- **Request ID Tracking:** Each request has a unique ID.
- **Context-Rich Logs:** Including relevant business context.
- **Appropriate Log Levels:** Using debug, info, warning, and error appropriately.

4.2.1 Criteria for Evaluating Logs

When designing our logging strategy, We should evaluate logs based on the following criteria:

- **Actionability:** Each log should provide information that can be used to take action.
- **Uniqueness:** Logs should provide information not available in metrics.
- **Debugging Value:** Logs must help in troubleshooting issues.
- **Context:** Logs should provide important contextual information.
- **Signal-to-Noise Ratio:** Logs should maintain high utility among many other logs.

4.2.2 Logs to Keep

I would prioritize the following types of logs:

- **Request Entry/Exit Points:** Basic information about incoming requests and responses.
- **Error Details:** Specific error information with stack traces and context.
- **State Changes:** Important application state transitions.
- **Critical Business Operations:** Success/failure of key business operations.
- **External System Interactions:** Communication with external dependencies.

4.2.3 Logs to Remove

I will systematically eliminate logs that:

- **Debug Statements:** Excessive debug logs used during development.
- **Redundant Information:** Logs that duplicate information available in metrics.
- **Routine Success Messages:** Logs that merely confirm normal operation.
- **Low-Level Details:** Technical details irrelevant to troubleshooting.
- **Periodic Status Updates:** Repeating logs that don't indicate changes.

4.2.4 Logs to Add

We enhance our logging system by adding:

- **Request Context:** Adding request IDs and user context to all logs.
- **Performance Boundaries:** Logging when operations exceed expected duration thresholds.
- **External Dependency Details:** Enhanced logs for third-party service interactions.
- **Business Context:** Adding business operation identifiers to technical logs.
- **Structured Data:** Converting text logs to structured JSON with searchable fields.

4.3 Dashboard Design

The Grafana dashboard provides:

- **Overview Panel:** High-level health metrics.
- **Endpoint Performance:** API endpoint metrics.
- **Resource Utilization:** Container resource metrics.
- **Error Analysis:** Error types and frequencies.
- **Application Logs:** Real-time log viewing.

5 Design Decisions and Tradeoffs

5.1 Major Design Decisions

The observability solution architecture is built on several key design decisions.

- **Separate Specialized Tools:** Using Prometheus for metrics, Loki for logs, and Grafana for visualization ensures that that each component excels at its specific role rather than using a monolithic solution.
- **Kubernetes-Native Integration:** Leveraging Kubernetes for deployment, service discovery, and scaling provides a consistent operational model across all observability components.
- **Structured Logging:** Standardizing on JSON-formatted logs enables more powerful querying, filtering, and correlation capabilities.

5.2 Tradeoffs

Key tradeoffs that I acknowledge:

- **Complexity vs. Capability:** The multi-component architecture increases operational complexity but provides greater observability capabilities than simpler solutions.
- **Storage vs. Retention:** This solution is optimized for storing high value logs rather than retaining all logs, which reduces storage requirements but **potentially limits some historical investigations.**

5.3 Known Gaps

Some limitations of this solution:

- **Deep Infrastructure Visibility:** Limited visibility into underlying Kubernetes infrastructure and node-level metrics, which is acceptable as our focus is application-level observability.
- **Historical Trend Analysis:** Limited capacity for multi-month historical trend analysis due to storage constraints, which is acceptable as our primary use case is operational troubleshooting.
- **Business Metrics:** The solution focuses on technical rather than business metrics, which is appropriate as business metrics are handled by separate analytics systems.

6 Improvements

6.1 Addressing Current Problems

The solution addresses debugging challenges through:

1. **Automated Monitoring:**
    - Real-time metrics collection.
    - Automated log aggregation.

- Centralized visualization.

2. **Standardized Logging:**
   - Consistent log format.
   - Request tracing.
   - Error tracking.
3. **Performance Tracking:**
   - Response time monitoring.
   - Resource utilization tracking.
   - Dependency performance metrics.

6.2 Future Enhancements

Planned improvements include:

- **Alert Management:** Configurable alerting rules.
- **Log Retention:** Configurable log retention policies.
- **Dashboard Templates:** Reusable dashboard components.
- **Performance Optimization:** Enhanced query performance.

7 Conclusion

The Kubernetes observability architecture with Grafana, Prometheus, and Loki provides a comprehensive solution for monitoring and debugging applications. By implementing this design, teams can benefit from:

- Reduced Mean Time to Resolution (MTTR) for incidents
- Improved visibility into system performance
- Centralized log management
- Real-time metrics visualization

This solution establishes a foundation for continuous improvement in observability practices, leading to more reliable systems and more productive engineering teams.

8 References

1. Kubernetes Documentation
2. Grafana Documentation
3. Loki Documentation
4. Prometheus Documentation
5. Fluent Bit Documentation