National Institute of Applied Sciences and Technology (INSAT)

Computer Science Department

Internship Report

Centralized Log Management System for Microservices

Student: Idris SADDI
Company: MedSirat
Internship period: 01/07/2025 – 31/08/2025
Supervisor: Salma SEDDIK
Academic advisor: INSAT

Submission date: September 23, 2025

Acknowledgements

I would like to express my sincere gratitude to **MedSirat** for providing me with the opportunity to undertake this enriching internship experience. Special thanks to my supervisor **Salma SEDDIK** for her invaluable guidance, technical expertise, and continuous support throughout this project.

I am deeply grateful to the entire MedSirat team for creating a collaborative and learning-oriented environment that enabled me to develop both technical and professional skills. Their trust in allowing me to work on the Siratify platform's critical infrastructure was instrumental to my growth.

I extend my appreciation to the **National Institute of Applied Sciences and Technology (INSAT)** and my academic advisors for their theoretical foundation and support, which provided the essential knowledge base for this practical implementation.

Finally, I acknowledge the open-source community behind the technologies used in this project—Spring Boot, Apache Kafka, Graylog, Docker, and OpenSearch—whose robust tools made this comprehensive log management solution possible.

This internship has been a transformative experience that bridged academic learning with real-world application in a professional consulting environment.

Abstract

This report presents the design and implementation of a **centralized log management system** for the Siratify platform's microservices environment. Siratify is an entrepreneurial ecosystem that connects entrepreneurs, investors, mentors, and support organizations. The primary objective was to **collect**, **aggregate**, **analyze**, and **monitor** application logs reliably and at scale for this multi-stakeholder platform. The solution is built with *Spring Boot 3.5.3*, *Apache Kafka*, and *Graylog 6.3*, orchestrated via *Docker Compose*. A dedicated *log-service* consumes messages from the Kafka topic `logs` and forwards them to Graylog using *GELF TCP*. The initialization script `graylog-init.sh` automates the creation of *inputs*, *streams*, *dashboards*, and *alerts* tailored for monitoring platform-specific metrics. The results show a **reduction in operational effort** and an **improvement in service observability** critical for maintaining Siratify's high-availability requirements.

Glossary and Acronyms

Apache Kafka Distributed event streaming platform for real-time data pipelines

Docker Compose Tool for defining and running multi-container applications

GELF Graylog Extended Log Format

Graylog Open-source log management platform with search and alerting

IaC Infrastructure as Code

MDC Mapped Diagnostic Context - thread-local storage for log enrichment

Microservices Architecture pattern using loosely coupled, deployable services

Observability Understanding system state through logs, metrics, and traces

OpenSearch Search and analytics engine for log data storage

SLI/SLO Service Level Indicator/Objective

Spring Boot Java framework for production-ready applications

SRE Site Reliability Engineering

Contents

List of Figures

General Introduction

Modern distributed systems, particularly multi-stakeholder platforms like Siratify, generate high volumes of heterogeneous logs across multiple services and infrastructure layers. Siratify, an entrepreneurial ecosystem platform that facilitates connections between entrepreneurs, investors, mentors, and support organizations, operates with complex user journeys and business-critical interactions that require comprehensive monitoring and observability.

Without centralization and structure, troubleshooting becomes slow and error-prone, potentially affecting user experience across the platform's diverse stakeholder base. This internship addresses these challenges by implementing a production-ready logging platform specifically designed for Siratify's requirements, enabling end-to-end visibility of user interactions, proactive alerting for business-critical flows, and operational insights with minimal manual configuration.

Chapter 1

Company Context and Objectives

1.1 Host Company Overview

MedSirat is a leading consulting firm founded in 2007, operating across Africa and the MENA region with over 18 years of experience. Based in Tunis, Tunisia (12 Rue Ahmed Boukhari), the company supports governments, businesses, NGOs, and entrepreneurs through comprehensive business services.

Industry: Business Services and Consulting. **Core Services:** Strategic consulting, coaching, training, and venture development. **Mission:** To empower purpose-driven progress by delivering tailored solutions that address today's challenges and shape tomorrow's opportunities. **Geographic Reach:** Africa and MENA region. **Specialization Areas:** Strategy, HR & Organizational Development, Finance & Funding, Marketing & Communication, AI & Technology Integration, and Pedagogy & Active Learning.

1.2 Internship Context and Problem Statement

The **Siratify platform** (http://siratify.com/), an entrepreneurial ecosystem that connects entrepreneurs, investors, mentors, and support organizations, utilizes a microservices architecture that required a robust, centralized approach to log collection and analysis. As a platform serving multiple stakeholder types with complex interactions, disparate logs across services were slowing incident resolution and hindering monitoring of critical business flows such as user registrations, investment matching, and mentoring connections.

The challenge was to implement a scalable logging pipeline for Siratify that ingests logs asynchronously, enriches them with business context, and renders them searchable and actionable in near real-time. This is particularly crucial for a platform handling sensitive data flows between entrepreneurs and investors, where system reliability and observability directly impact business outcomes.

1.3 Objectives

The internship objectives were as follows:

- Design a **centralized log ingestion architecture**.
- Implement a **reliable asynchronous transport** using Kafka.
- Integrate **GELF** (Graylog Extended Log Format) with MDC enrichment.
- Automate **Graylog provisioning** (inputs, streams, dashboards, alerts).
- Deliver a **containerized deployment** with reproducible environments.

1.4 Project Plan (Gantt)

The following Gantt chart summarizes the project plan.

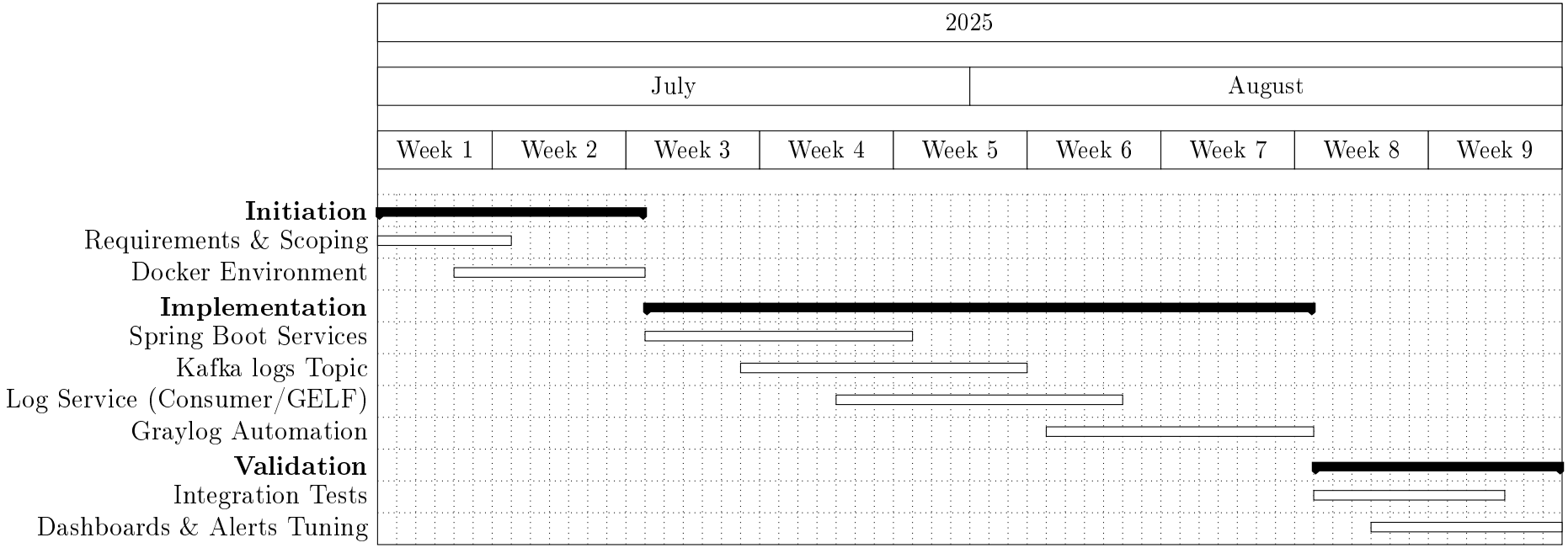

Figure 1.1: Project Gantt chart for Siratify log management system implementation

Chapter 2

System Overview and Requirements

2.1 Functional Requirements

- Collect logs from multiple services with minimal coupling.
- Support multiple log levels and structured fields.
- Provide search, dashboards, and alerts for operations.
- Automate provisioning to reduce manual setup.

2.2 Non-Functional Requirements

- Scalability via asynchronous message transport (Kafka).
- Resilience to partial outages (backpressure, retries).
- Security of transport and access control.
- Observability and maintainability for SRE workflows.

Chapter 3

Technology Selection and Methodology

3.1 Technology Stack Selection

After evaluating log management solutions (ELK/OpenSearch, Graylog, Loki), **Graylog** was selected for its unified API enabling automated provisioning, built-in GELF support, and comprehensive stream routing capabilities suitable for Siratify's microservices architecture.

3.2 Development Approach

The project followed an iterative approach with weekly increments: requirements analysis, Docker environment setup, service implementation, Kafka integration, log-service development, and Graylog automation. Each phase delivered testable components validated through synthetic traffic and dashboard verification.

3.3 Requirements Analysis

Requirements were gathered focusing on Siratify's operational needs: centralized log search for user journey tracking, ERROR alerting for critical business flows, automated provisioning, and containerized deployment. Key priorities include log level distribution dashboards, MDC enrichment with user context, and service-specific monitoring capabilities.

Chapter 4

Architecture and Implementation

4.1 System Architecture Overview

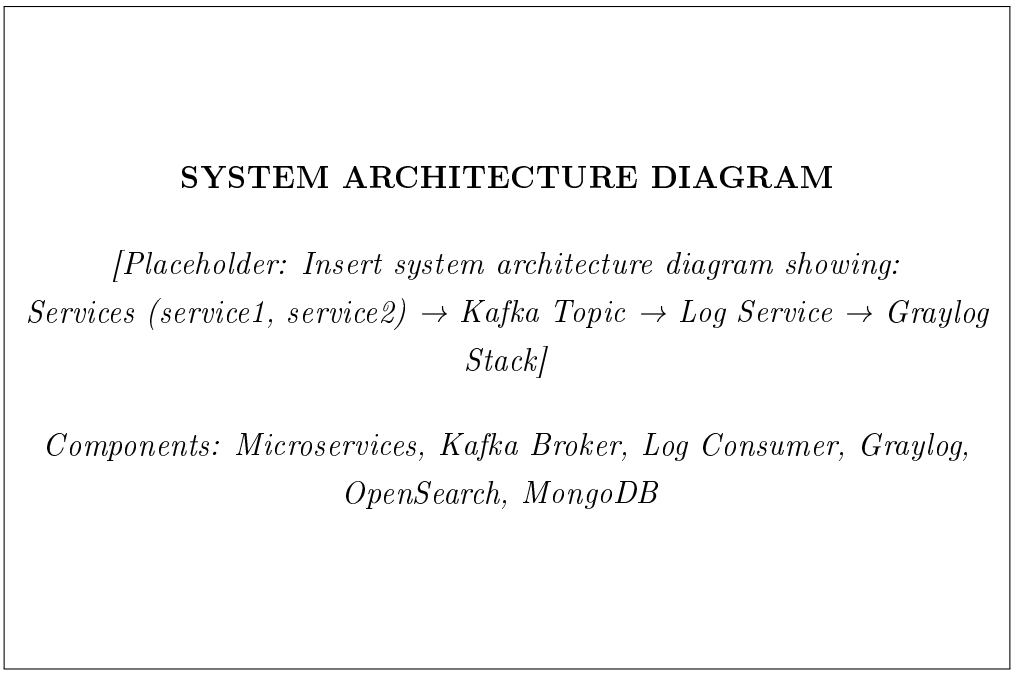

Figure 4.1: System architecture showing the complete log management pipeline from microservices through Kafka to Graylog stack

Architecture Notes: This diagram illustrates the decoupled architecture where microservices publish logs to Kafka topics, enabling asynchronous processing. The dedicated log-service acts as a consumer that enriches logs with MDC context before forwarding to Graylog via GELF TCP. The Graylog stack utilizes OpenSearch for log storage and MongoDB for configuration metadata.

4.2 Log Flow Sequence

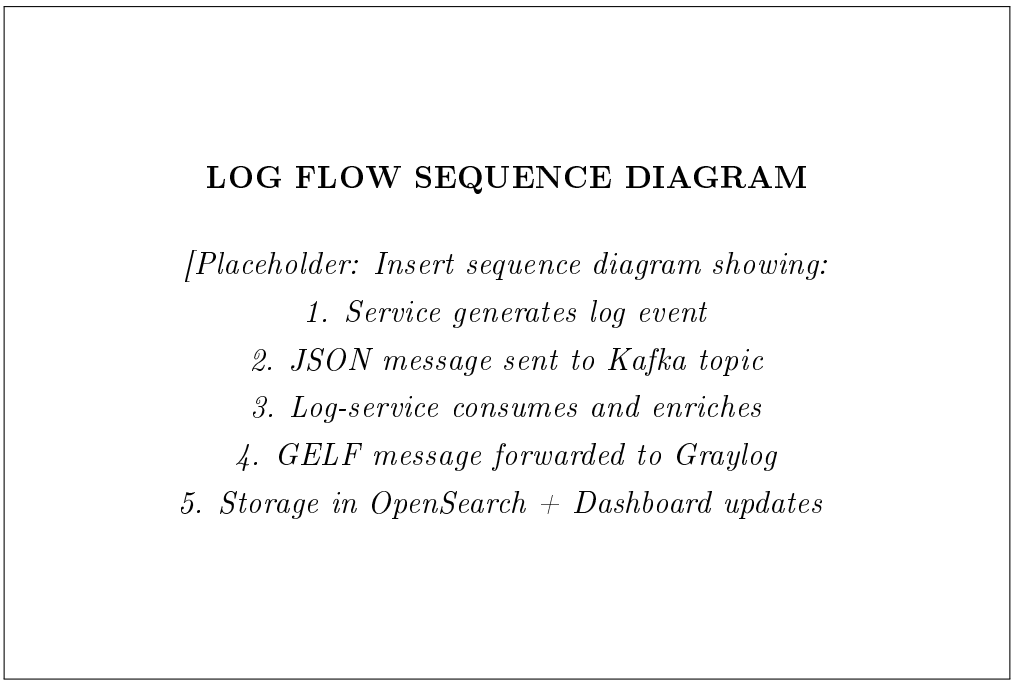

Figure 4.2: Sequence diagram detailing the end-to-end log processing flow from generation to visualization

Flow Notes: The sequence demonstrates the temporal relationship between components, highlighting the asynchronous nature of the pipeline. Each step includes error handling and retry mechanisms to ensure reliable log delivery even during component failures.

4.3 Deployment Architecture

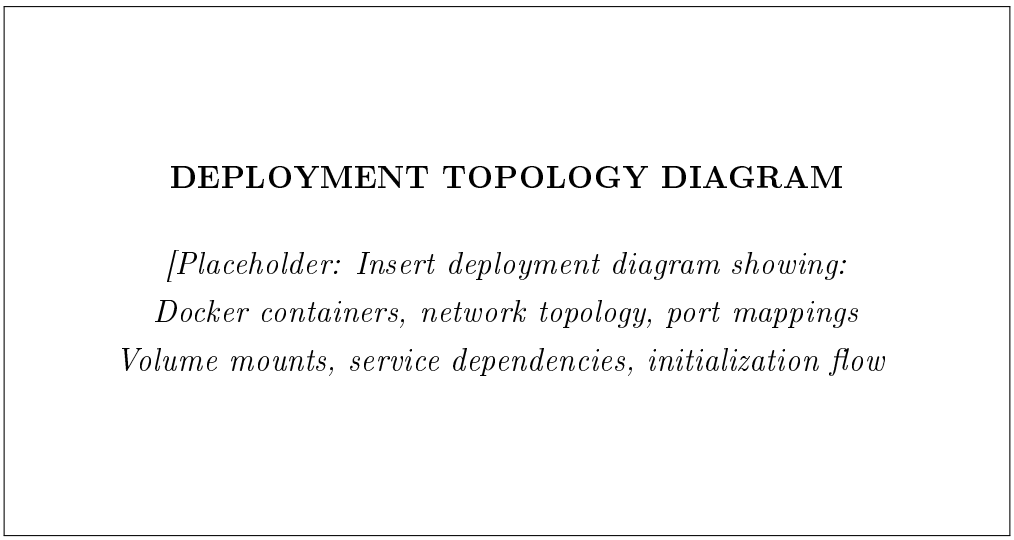

Figure 4.3: Docker-based deployment topology showing container orchestration and network configuration

Deployment Notes: The containerized architecture enables reproducible environments across development, staging, and production. The Docker Compose orchestration manages service dependencies, persistent volumes for data storage, and automated initialization through the graylog-init container.

4.4 Component Responsibilities

- **service1/2**: Expose REST endpoints `/log` and publish events to Kafka
- **Kafka/Zookeeper**: Reliable asynchronous transport for log messages
- **log-service**: Kafka consumer with MDC enrichment and GELF TCP forwarding
- **Graylog**: Log ingestion, indexing, dashboards, alerting, and search capabilities
- **OpenSearch**: Scalable storage and search backend for log data
- **MongoDB**: Configuration storage for Graylog streams, dashboards, and alerts

4.5 Design Rationale

Kafka ensures asynchronous decoupling to mitigate backpressure and enable independent scaling. GELF provides structured transport with MDC support for rich contextual

logging. Graylog's API-first approach enables complete automation of the logging infrastructure setup.

4.6 Technology Stack

- **Java 21** & **Spring Boot 3.5.3**: Backend framework for microservices
- **Apache Kafka**: Distributed message streaming platform
- **Graylog 6.3**: Centralized log management platform
- **OpenSearch 2.x**: Search and analytics backend for log storage
- **MongoDB 6.0.5**: Metadata storage for Graylog configuration
- **Docker Compose**: Container orchestration for reproducible deployments

4.7 Configuration Overview

The system uses Docker Compose for orchestration with standardized configurations: - Kafka bootstrap servers at `kafka:9092` with string serialization - Log-service consumer group `log-consumers` with earliest offset reset - GELF TCP input on port 12201 for structured log transport - Service-specific streams with automated routing rules - Persistent volumes for OpenSearch data and MongoDB configuration

Chapter 5

Implementation Results

5.1 Key Implementation Components

The solution consists of:

- **REST endpoints**: Services expose `/log` endpoints accepting message and level parameters
- **Kafka integration**: JSON log events published to `logs` topic with string serialization
- **Log consumer**: Dedicated service consuming Kafka messages with MDC enrichment
- **GELF forwarding**: Structured logs sent to Graylog via TCP port 12201
- **Automated provisioning**: `graylog-init.sh` script creates inputs, streams, dashboards, and alerts

5.2 Deployment and Configuration

The system uses Docker Compose orchestration with:

- **Service discovery**: Container DNS resolution for inter-service communication
- **Persistent storage**: Volumes for OpenSearch data and MongoDB configuration
- **Network isolation**: Dedicated bridge network for secure internal communication
- **Automated initialization**: Init container waits for Graylog API and configures logging infrastructure

5.3 Testing and Validation Results

System validation demonstrated:

- **End-to-end log flow**: Messages successfully routed from service endpoints to Graylog dashboards
- **Automated alerting**: ERROR level logs trigger notifications within configured thresholds
- **Stream routing**: Service-specific logs correctly filtered into dedicated streams
- **Dashboard functionality**: Real-time visualization of log counts, level distribution, and timeline metrics

Chapter 6

Results and Analysis

6.1 System Performance

The implemented solution achieved:

- Sub-second log ingestion latency from service to Graylog visibility
- Reliable message delivery through Kafka's asynchronous transport
- Automated infrastructure provisioning reducing manual setup time
- Scalable architecture supporting multiple concurrent services

6.2 Operational Benefits

Key improvements for Siratify platform operations:

- **Centralized visibility**: Unified log search across all microservices
- **Proactive monitoring**: Real-time dashboards and automated error alerting
- **Reduced MTTR**: Faster incident resolution through structured log analysis
- **Deployment automation**: Reproducible environments across development stages

6.3 Limitations and Future Improvements

Current limitations include single-node development configuration and basic schema structure. Future enhancements should consider:

- Multi-broker Kafka cluster for production fault tolerance
- TLS encryption for secure log transport in production environments

- Schema registry integration for event structure evolution
- Distributed tracing correlation for enhanced observability

6.4 Kafka Consumer and GELF Forwarder

The log-service consumes the `logs` topic and forwards to Graylog using a GELF TCP appender configured in Logback.

```
// See Appendix D for the full source
```

Figure 6.1: Excerpt: log-service Kafka consumer

6.5 Graylog Automation

The `graylog-init.sh` script waits for the Graylog API, creates the global GELF TCP input, configures streams per service with routing rules, generates searches and dashboards with multiple widgets, and adds an ERROR-level alert with an HTTP notification.

```
1) Wait for Graylog API readiness
2) Create global GELF TCP input on port 12201
3) For each service: create stream, rules, search, dashboard
4) Add widgets (counts, level distribution, timeline, sources)
5) Create ERROR alert and attach HTTP notification
```

Figure 6.2: Automation script responsibilities

Chapter 7

Operational Considerations

7.1 Security and Compliance

Production deployment requires TLS encryption for GELF transport, access control for Graylog UI, and data minimization policies to avoid logging sensitive information (PII). Regular security audits and credential rotation should be implemented.

7.2 Scalability and Performance

The architecture supports horizontal scaling through:

- Kafka partition increase for higher throughput
- Multiple log-service consumer instances
- OpenSearch cluster expansion for storage scaling
- Load balancer integration for Graylog inputs

7.3 Operational Procedures

Key maintenance tasks include:

- Regular OpenSearch index lifecycle management
- MongoDB backup for Graylog configuration
- Log retention policy enforcement
- Dashboard and alert threshold tuning based on operational needs

General Conclusion

This internship consolidated skills in distributed architectures, messaging, observability, and containerized deployments while addressing real-world challenges for the Siratify entrepreneurial ecosystem platform. The delivered centralized log management solution provides a robust foundation for monitoring Siratify's microservices that facilitate connections between entrepreneurs, investors, mentors, and support organizations.

The implementation demonstrates how modern logging architecture can enhance platform reliability and user experience in multi-stakeholder environments. The automated provisioning reduces operational overhead while providing comprehensive visibility into system behavior through dashboards, alerts, and structured log search capabilities.

Future enhancements for Siratify include distributed tracing integration (OpenTelemetry), SSO integration for access control, and richer alert routing for business-critical platform events. The foundation established enables rapid extension to additional services and observability requirements as the platform scales.

Future Work

Planned improvements include:

- Secure-by-default TLS configuration across all services
- CI/CD integration for automated infrastructure changes
- Multi-environment configuration overlays (dev/staging/production)
- Schema registry implementation for event structure evolution
- Integration with distributed tracing systems for enhanced observability

Bibliography

[1] Graylog Documentation, https://docs.graylog.org/, accessed 09/23/2025.

[2] Apache Kafka Documentation, https://kafka.apache.org/documentation/, accessed 09/23/2025.

[3] Spring Boot Reference, https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/, accessed 09/23/2025.

[4] OpenSearch Documentation, https://opensearch.org/docs/, accessed 09/23/2025.

[5] logback-gelf, https://github.com/mp911de/logstash-gelf, accessed 09/23/2025.

Appendix A

docker-compose.yml

This Docker Compose configuration orchestrates the complete log management system infrastructure, defining all services, networks, and dependencies required for centralized logging.

Infrastructure Components

- **Graylog Stack:** MongoDB (metadata), OpenSearch (log storage), Graylog (web interface)
- **Message Broker:** Kafka + Zookeeper for reliable log transport
- **Application Services:** service1, service2 (log producers)
- **Log Processing:** log-service (Kafka consumer → GELF forwarder)
- **Automation:** graylog-init (configuration setup)

Key Service Definitions

Graylog Configuration

```
graylog:
  image: graylog/graylog:6.3
  environment:
    GRAYLOG_ROOT_PASSWORD_SHA2: "8c6976e5...918" # admin:admin
    GRAYLOG_HTTP_EXTERNAL_URI: "http://localhost:9000/"
    GRAYLOG_ELASTICSEARCH_HOSTS: "http://opensearch:9200"
    GRAYLOG_MONGODB_URI: "mongodb://mongodb:27017/graylog"
  ports:
    - 9000:9000        # Web UI
    - 12201:12201/tcp  # GELF TCP input
  depends_on: [opensearch, mongo]
```

Kafka Message Broker

```
kafka:
  image: wurstmeister/kafka
  environment:
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  ports:
    - "9092:9092"
  depends_on: [zookeeper]
```

Application Services

```
service1:
  build: ./service1
  ports: ["8081:8081"]
  depends_on: [kafka, log-service]

log-service:
  build: ./log-service
  environment:
    - GRAYLOG_HOST=graylog
    - GRAYLOG_PORT=12201
  ports: ["8083:8083"]
  depends_on: [kafka, graylog]
```

Network and Persistence

- **Network:** `graynet` bridge for service communication
- **Volumes:** `mongo_data`, `log_data`, `graylog_data` for persistence
- **Port mapping:** 9000 (Graylog UI), 8081-8083 (services), 9092 (Kafka)

Startup Sequence

1. Infrastructure: MongoDB, OpenSearch, Zookeeper
2. Message broker: Kafka
3. Log management: Graylog + graylog-init
4. Processing: log-service
5. Applications: service1, service2

Complete configuration available at:
https://github.com/idris-saddi/log-management/blob/master/docker-compose.yml

Appendix B

graylog-init.sh

This script automates the complete Graylog configuration for the log management system. The script performs the following key operations:

Main Configuration Steps

1. **Wait for Graylog API readiness**

```
until curl -s -u "$AUTH" "$GRAYLOG_URL/api/system/inputs" > /dev/null; do
  sleep 3
done
```

2. **Create GELF TCP Input** (port 12201)

```
GELF_PAYLOAD='{
  "title": "GELF TCP",
  "type": "org.graylog2.inputs.gelf.tcp.GELFTCPInput",
  "configuration": {"bind_address": "0.0.0.0", "port": 12201}
}'
curl -X POST "$GRAYLOG_URL/api/system/inputs" --data "$GELF_PAYLOAD"
```

3. **For each service, create:**

   - Stream with routing rules
   - Dashboard with monitoring widgets
   - Alert conditions for ERROR level logs

4. **Stream creation example:**

```
STREAM_PAYLOAD='{
  "title": "Stream for service1",
  "description": "Auto-created stream for service1",
  "rules": [], "remove_matches_from_default_stream": false
}'
curl -X POST "$GRAYLOG_URL/api/streams" --data "$STREAM_PAYLOAD"
```

5. **Alert condition setup:**

```
ALERT_PAYLOAD='{
  "type": "message_count",
  "title": "High ERROR rate for service1",
  "parameters": {"threshold": 5, "time": 300, "threshold_type": "MORE"}
}'
curl -X POST "$GRAYLOG_URL/api/streams/$STREAM_ID/alerts/conditions"
```

Key Features

- Automated stream creation with service-specific routing
- Dashboard generation with log count and error rate widgets
- Alert conditions for monitoring ERROR level messages
- Comprehensive error handling and logging
- Idempotent execution (can be run multiple times safely)

Complete script available at:
https://github.com/idris-saddi/log-management/blob/master/graylog-init.sh

Appendix C

service1 LogController.java

This REST controller provides HTTP endpoints for generating log messages that are sent to Kafka for centralized processing. It demonstrates how microservices can integrate with the log management system.

Key Functionality

- **REST endpoints:** POST and GET `/log` for message generation
- **Kafka integration:** Publishes structured log messages to the `logs` topic
- **Level normalization:** Standardizes log levels (INFO, WARN, ERROR, DEBUG, TRACE)
- **Service identification:** Automatically tags logs with service name

Core Implementation

```
@RestController
@RequestMapping("/log")
public class LogController {
    private final KafkaTemplate<String, String> kafkaTemplate;

    @Value("${spring.application.name}")
    private String applicationName;

    @PostMapping
    public ResponseEntity<?> postLog(
        @RequestParam(defaultValue = "Test log message") String message,
        @RequestParam(defaultValue = "INFO") String level) {

        Map<String, Object> log = new HashMap<>();
        log.put("timestamp", Instant.now().toString());
        log.put("message", message);
        log.put("level", normalizeLevel(level));
```

```
        log.put("service", applicationName);

        String logJson = objectMapper.writeValueAsString(log);
        kafkaTemplate.send("logs", logJson);
        return ResponseEntity.ok("Log sent to Kafka: " + logJson);
    }
}
```

Log Message Structure

Generated messages follow this JSON format:

```
{
  "timestamp": "2025-08-31T10:30:45.123Z",
  "message": "User authentication successful",
  "level": "INFO",
  "service": "service1"
}
```

Usage Examples

- `POST /log?message=User%20login&level=INFO`
- `GET /log?message=Database%20error&level=ERROR`

Complete source code available at:
https://github.com/idris-saddi/log-management/blob/master/service1/src/main/java/com/idris/service1/LogController.java

Appendix D

log-service LogConsumer.java

```
package com.idris.log_service.kafka;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.idris.log_service.dto.LogMessage;
import com.idris.log_service.service.LogProcessorService;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class LogConsumer {

    private final LogProcessorService logProcessor;
    private final ObjectMapper objectMapper = new ObjectMapper();

    public LogConsumer(LogProcessorService logProcessor) {
        this.logProcessor = logProcessor;
    }

    @KafkaListener(topics = "logs", groupId = "log-consumers")
    public void consume(ConsumerRecord<String, String> record) {
        try {
            LogMessage logMessage = objectMapper.readValue(record.value(), LogMessage.class);
            logProcessor.process(logMessage);
        } catch (Exception e) {
            System.err.println("[ERROR] Failed to parse or process log: " + e.getMessage());
        }
    }
}
```

Appendix E

logback-spring.xml (log-service)

```
<configuration>

    <!-- Console fallback appender -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- GELF TCP Appender for Graylog -->
    <appender name="GELF-TCP" class="de.siegmar.logbackgelf.GelfTcpAppender">
        <graylogHost>graylog</graylogHost> <!-- Docker service name -->
        <port>12201</port>
        <connectTimeout>3000</connectTimeout>
        <reconnectDelay>2000</reconnectDelay>
        <queueSize>512</queueSize>

        <encoder class="de.siegmar.logbackgelf.GelfEncoder">
            <facility>log-service</facility> <!-- Tag logs with your service name -->
            <includeLoggerName>true</includeLoggerName>
            <includeThreadName>true</includeThreadName>
            <includeMdcData>true</includeMdcData> <!-- Important for structured fields -->
            <includeExceptionCause>true</includeExceptionCause>
        </encoder>
    </appender>

    <!-- Root logger with both appenders -->
    <root level="INFO">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="GELF-TCP" />
    </root>

</configuration>
```

Appendix F

service1 application.properties

```
# Basic service config
server.port=8081
spring.application.name=service1

# Kafka config
spring.kafka.bootstrap-servers=kafka:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
```

Appendix G

service2 application.properties

```
# Basic service config
server.port=8082
spring.application.name=service2

# Kafka config
spring.kafka.bootstrap-servers=kafka:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
```

Appendix H

service2 logback-spring.xml

```
<configuration scan="true">

    <!-- Read properties from Spring -->
    <springProperty scope="context" name="appName" source="spring.application.name"/>
    <springProperty scope="context" name="kafkaServers" source="spring.kafka.bootstrap-servers"/>

    <!-- Console appender (still see logs locally) -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <threadName/>
                <loggerName/>
                <message/>
                <arguments/>
                <stackTrace/>
                <mdc/>
                <provider class="net.logstash.logback.composite.GlobalCustomFieldsJsonProvider">
                    <customFields>{"service":"${appName}"}</customFields>
                </provider>
            </providers>
        </encoder>
    </appender>

    <!-- Kafka appender -->
    <appender name="KAFKA" class="com.github.danielwegener.logback.kafka.KafkaAppender">
        <topic>logs</topic>
        <producerConfig>bootstrap.servers=${kafkaServers}</producerConfig>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <threadName/>
                <loggerName/>
                <message/>
                <arguments/>
                <stackTrace/>
                <mdc/>
                <provider class="net.logstash.logback.composite.GlobalCustomFieldsJsonProvider">
                    <customFields>{"service":"${appName}"}</customFields>
                </provider>
```

```
            </providers>
        </encoder>
    </appender>

    <!-- Root logger -->
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="KAFKA"/>
    </root>
</configuration>
```

Appendix I

log-service application.properties

```
# App Info
server.port=8083
spring.application.name=log-service

# Kafka
spring.kafka.bootstrap-servers=kafka:9092
spring.kafka.consumer.group-id=log-consumers
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer

# Logging (optional: cleaner output)
logging.level.root=INFO
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

Appendix J

service1 logback-spring.xml

```
<configuration scan="true">

    <!-- Read properties from Spring -->
    <springProperty scope="context" name="appName" source="spring.application.name"/>
    <springProperty scope="context" name="kafkaServers" source="spring.kafka.bootstrap-servers"/>

    <!-- Console appender (still see logs locally) -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <threadName/>
                <loggerName/>
                <message/>
                <arguments/>
                <stackTrace/>
                <mdc/>
                <provider class="net.logstash.logback.composite.GlobalCustomFieldsJsonProvider">
                    <customFields>{"service":"${appName}"}</customFields>
                </provider>
            </providers>
        </encoder>
    </appender>

    <!-- Kafka appender -->
    <appender name="KAFKA" class="com.github.danielwegener.logback.kafka.KafkaAppender">
        <topic>logs</topic>
        <producerConfig>bootstrap.servers=${kafkaServers}</producerConfig>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <threadName/>
                <loggerName/>
                <message/>
                <arguments/>
                <stackTrace/>
                <mdc/>
                <provider class="net.logstash.logback.composite.GlobalCustomFieldsJsonProvider">
                    <customFields>{"service":"${appName}"}</customFields>
                </provider>
```

```
            </providers>
        </encoder>
    </appender>

    <!-- Root logger -->
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="KAFKA"/>
    </root>
</configuration>
```