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 my company supervisor, the engineering team, and all colleagues for their support during this internship. Their guidance and trust were instrumental to the success of this project. I also thank my professors at INSAT for their academic support and the knowledge they provided.

Contents

List of Figures

List of Tables

Abstract

This report presents the design and implementation of a **centralized log management system** for a microservices environment. The primary objective was to **collect**, **aggregate**, **analyze**, and **monitor** application logs reliably and at scale. 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*. The results show a **reduction in operational effort** and an **improvement in service observability**.

General Introduction

Modern distributed systems generate high volumes of heterogeneous logs across multiple services and infrastructure layers. Without centralization and structure, troubleshooting becomes slow and error-prone. This internship addresses these challenges by implementing a production-ready logging platform that enables end-to-end visibility, proactive alerting, and operational insights with minimal manual configuration.

Chapter 1

Company Context and Objectives

1.1 Host Company Overview

Industry: *to be completed.* Annual revenue: *to be completed.* Organization chart: *to be completed.* Certifications: *ISO 9001, ISO 27001 if applicable.* Engineering staff ratio: *to be completed.*

1.2 Internship Context and Problem Statement

The microservices architecture used by the company required a robust, centralized approach to log collection and analysis. Disparate logs across services were slowing incident resolution and hindering monitoring. The challenge was to implement a scalable pipeline that ingests logs asynchronously, enriches them with context, and renders them searchable and actionable in near real-time.

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.

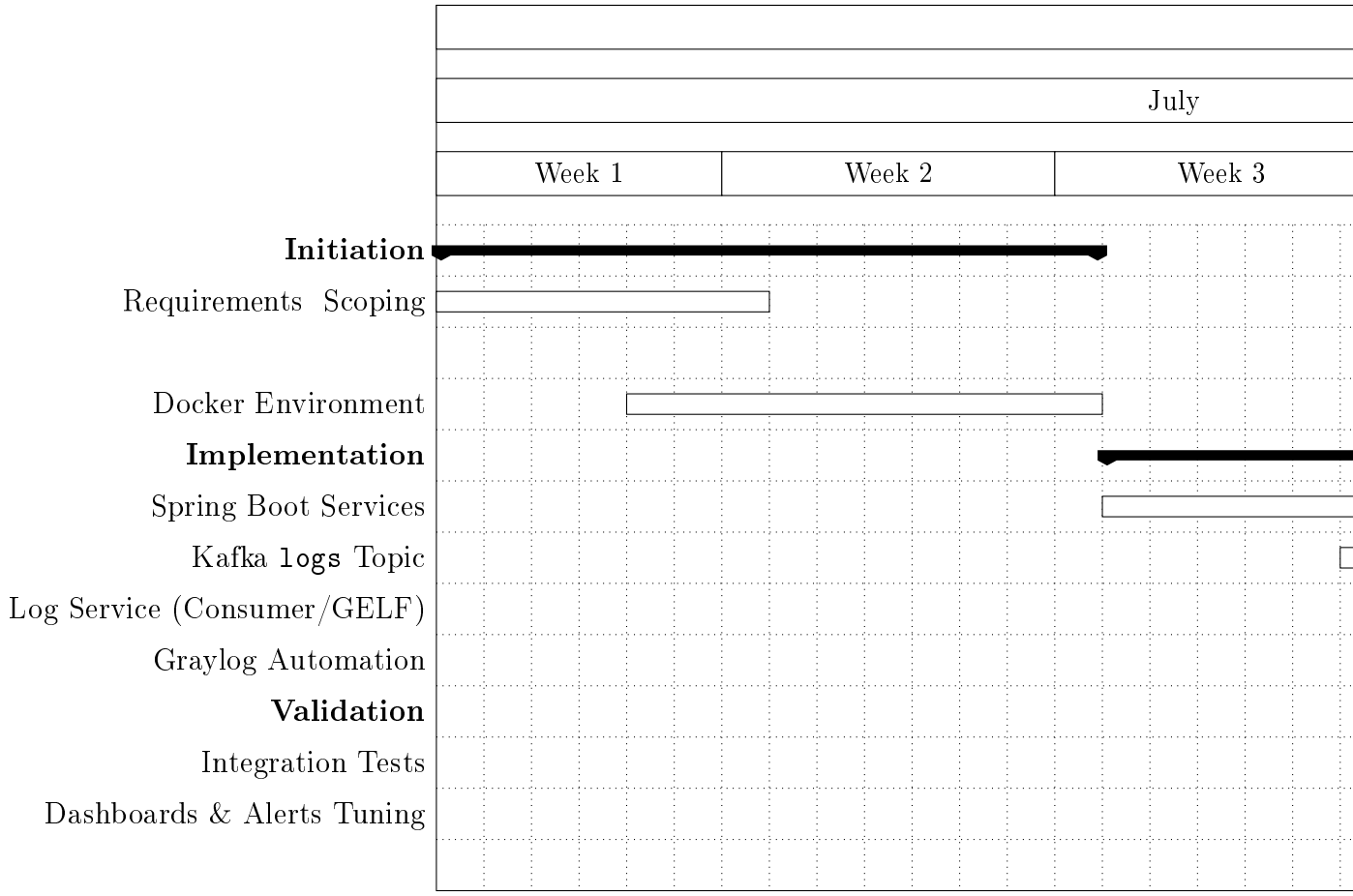

Figure 1.1: Project Gantt chart

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 (discussed in Chapter 20).
- Observability and maintainability for SRE workflows.

Chapter 3

Background and Related Work

Log management in distributed systems builds on decades of work in observability. Traditional centralized approaches relied on syslog; modern stacks include ELK/OS (Elasticsearch/OpenSearch, Logstash, Kibana) and Graylog. Kafka-based pipelines decouple producers from consumers, increasing resilience. Related work explores schema-on-write vs. schema-on-read trade-offs, log sampling, and privacy-preserving logging.

3.1 Positioning of This Work

This project emphasizes operational automation (Graylog provisioning) and reproducible environments to reduce setup costs. It demonstrates an end-to-end pattern from application logging to dashboards and alerts using open-source components.

Chapter 4

Methodology

4.1 Process Model

An incremental and iterative process was used (adapted Scrum): bi-weekly iterations with planning, implementation, review, and retrospective. Each iteration delivered a testable increment (e.g., service endpoints, Kafka ingestion, init scripts, dashboards).

4.2 Tools and Practices

Version control with Git, issue tracking via GitHub Issues, code reviews for critical changes, and containerized dev environments ensured consistency. Observability was validated with synthetic traffic and exploratory testing in Graylog.

4.3 Week 1: Technology Survey and Selection

In the first week, I conducted a focused survey of log management stacks: **ELK (Elasticsearch/OpenSearch + Logstash + Kibana)**, **Graylog**, and **Loki**.

- **ELK/OS**: very flexible, large ecosystem; Logstash/Beats for ingestion; Kibana dashboards. Pros: mature, powerful queries and visualizations. Cons: more moving parts; provisioning and pipeline config are heavier for our needs.
- **Graylog**: opinionated platform with built-in inputs, streams, dashboards, and alerts; strong API for automation. Pros: fast to value, centralized UI, manageable provisioning via REST; native GELF. Cons: feature set aligned to logging (less general analytics than Kibana for non-log data).
- **Loki**: optimized for label-based indexing and log streaming, tight Grafana integration. Pros: cost-efficient for large volumes, great with Grafana. Cons: query

model differs (LogQL), and our pipeline required stronger message routing/alerting primitives out-of-the-box.

After evaluating the fit for *our objectives* (rapid automation, streams/routing, alerting, reproducible local stack), we chose **Graylog**. This decision enabled us to automate inputs/streams/dashboards/alerts through a single script and leverage GELF TCP with MDC for structured logs without a heavy Logstash layer.

Comparison Matrix

Feature	**ELK / OpenSearch Stack**	**Graylog**	**Loki**
Ingestion	Logstash/Beats; flexible pipelines	Built-in inputs (e.g., GELF TCP)	Promtail agents; label-based
Storage/Index	Elasticsearch/OpenSearch	OpenSearch via Graylog	Chunked object store + index
Query language	Lucene/KQL	Lucene via Graylog UI	LogQL (labels + regex)
Dashboards	Kibana	Built-in dashboards	Grafana
Alerts	Kibana/Watchers/Alerting	Native event/notification	Grafana alerting
Provisioning	APIs + Kibana objects; more moving parts	Unified REST API; streams/dashboards/alerts in one place	Grafana/Loki APIs; separate components
Operational complexity	Higher (more components)	Moderate (opinionated)	Moderate (Grafana+Loki+Promtail)
Best fit	Broad analytics, complex pipelines	Logging-centric with routing/alerts	Cost-efficient log streaming with Grafana

Chapter 5

Requirements Engineering

5.1 Elicitation and Analysis

Requirements were gathered from SRE and development stakeholders focusing on operational pain points (slow troubleshooting, inconsistent logging). The following MoSCoW prioritization was established.

Priority	**Requirement**
Must Have	Centralized log search; ERROR alerting; automated provisioning; containerized deployment
Should Have	Level distribution dashboard; MDC enrichment; response time metrics
Could Have	HTTP notifications; service-specific dashboards; top sources table
Won't Have	Distributed tracing (deferred)

Table 5.1: MoSCoW prioritization

5.2 Traceability

Each requirement mapped to concrete deliverables (scripts, configs, dashboards), captured in repository commits and README documentation.

Chapter 6

Architecture

6.1 Logical Architecture

The platform consists of two application services (`service1`, `service2`) publishing JSON logs to Kafka, a dedicated `log-service` consuming the `logs` topic and forwarding GELF messages to Graylog, and the Graylog stack backed by OpenSearch (for data) and MongoDB (for configuration).

6.2 Component Responsibilities

- **service1/2**: expose a REST endpoint `/log` to publish log events into Kafka.
- **Kafka/Zookeeper**: reliable asynchronous transport for log messages.
- **log-service**: Kafka consumer, MDC enrichment, GELF TCP forwarding.
- **Graylog**: ingestion, indexing (OpenSearch), dashboards, alerting, search.

6.3 Design Rationale

Kafka ensures asynchronous decoupling to mitigate backpressure. GELF provides structured transport with MDC support, simplifying parsing and indexing. Graylog was selected for its API-first provisioning and built-in alerts.

6.4 Sequence of Interactions

1) Client requests to service endpoint. 2) Service publishes log to Kafka (topic `logs`). 3) Log-service consumes and enriches. 4) GELF TCP to Graylog. 5) Graylog indexes into OpenSearch and manages dashboards/alerts.

6.5 Deployment Topology

6.6 End-to-End Data Flow

1. Application emits a structured JSON event using Logback JSON encoder (with a static `"service"` field derived from `spring.application.name`).
2. Logback Kafka Appender publishes the event to Kafka topic `logs` using the Spring property `spring.kafka.bootstrap-servers=kafka:9092`.
3. The `log-service` Kafka consumer (group `log-consumers`) receives the event, parses it into a `LogMessage` DTO (tolerant to extra fields), and enriches MDC with `service`, `originalTS`, and normalized level fields.
4. Logback in `log-service` forwards the enriched record via GELF TCP to Graylog on `graylog:12201`.
5. Graylog persists messages in OpenSearch and stores configuration in MongoDB. The init script creates streams, dashboards, and alert definitions per service.

6.7 Integration Points and Interfaces

- **Kafka contract**: topic name `logs`, value is JSON string (StringSerializer). No key is required by current producers.
- **Consumer DTO**: fields accepted are `@timestamp|timestamp`, `message`, `level`, `service`, optional `thread_name`, `logger_name`, `stack_trace`. Unknown fields are ignored.
- **Graylog input**: GELF TCP global input on port `12201` (no TLS in local dev). Streams route by field `service`.
- **Dashboards/alerts**: created via REST API by `graylog-init.sh`; title and widgets are deterministic per service list.

6.8 Strengths and Limitations

Strengths

- **Loose coupling via Kafka**: producers are decoupled from consumers; the pipeline tolerates bursts and partial outages.

- **Structured logging**: consistent JSON and MDC enable efficient search and aggregations in Graylog.
- **Automation**: end-to-end Graylog provisioning removes manual steps and drift.
- **Reproducible environment**: Docker Compose sets up the full stack predictably for development/testing.

Limitations

- **Single-broker dev setup**: the compose file uses a single Kafka broker and single-node OpenSearch; not fault tolerant.
- **No transport encryption in dev**: GELF TCP and Graylog UI are not TLS-enabled by default in local environment.
- **Basic schema**: the minimal event schema lacks correlation IDs and rich context (e.g., user/session) unless added by producers.
- **Backpressure visibility**: without explicit metrics, diagnosing dropped or delayed messages requires additional observability.

6.9 Scalability Considerations

- Increase Kafka partitions for `logs` and run multiple `log-service` consumers in the same group.
- Use Graylog inputs behind load balancers and scale OpenSearch with multiple data nodes and index tuning.
- Introduce schemas (Avro/Protobuf) and a schema registry for stronger contracts as the event model evolves.

6.10 Technology Stack Details

6.10.1 Runtime and Frameworks

- Java 21 (LTS) with Spring Boot 3.5.3 across all services.
- Spring Kafka for producer/consumer integration.
- Logback with `logstash-logback-encoder` (v7.3) for structured JSON events.
- Logback Kafka Appender (service1) `com.github.danielwegener:logback-kafka-appender:0.`

- GELF TCP appender (log-service) `de.siegmar:logback-gelf:4.0.2` and `org.graylog2:gelfc` available.

6.10.2 Messaging and Storage

- Kafka (`wurstmeister/kafka`) with Zookeeper (`confluentinc/cp-zookeeper:7.4.3`).
- Graylog 6.3 with OpenSearch 2.x (single-node, security disabled for local) and MongoDB 6.0.5.

6.10.3 Configuration Summary

Key properties from `application.properties` files:

- Producers (service1, service2): `spring.kafka.bootstrap-servers=kafka:9092`; String serializers.
- Consumer (log-service): group `log-consumers`, `auto-offset-reset=earliest`, String deserializers.
- log-service port: 8083; service1: 8081; service2: 8082.

6.11 Log Event Schema and Enrichment

6.11.1 Producer-side JSON

Services publish JSON events containing at least `timestamp`, `message`, `level`, and `service` (see Figure 6.1).

```
{
  "timestamp": "2025-07-15T10:45:12.345Z",
  "message": "UserAuthenticated",
  "level": "INFO",
  "service": "service1"
}
```

Figure 6.1: Minimal producer event structure (service1/service2)

6.11.2 Consumer DTO Mapping

The Kafka consumer maps messages to `LogMessage` with optional fields ignored (`@JsonIgnorePropertie` Supported fields:

- `@timestamp` (string), `message`, `level`, `service`,
- `thread_name`, `logger_name`, `stack_trace`.

6.11.3 MDC Enrichment and Routing

During processing, the consumer sets MDC keys: `service`, `originalTS`, `levelName`, and `level`. Logback GELF encoder includes MDC, enabling routing and dashboards in Graylog based on `service` and `level`.

6.12 Configuration Matrix

Component	Key Setting	Value / Purpose
service1/service2	Kafka bootstrap	`kafka:9092` (container DNS)
service1/service2	Kafka serializers	String/JSON via logback encoder
log-service	Consumer group	`log-consumers` (scalable)
log-service	GELF target	`graylog:12201` (TCP)
Graylog	Input	GELF TCP global input (scripted)
Graylog	Streams	Per-service stream with rule `service=...`
OpenSearch	Single-node	Dev mode, security disabled (local)

Table 6.1: Configuration matrix across components

All components are orchestrated with Docker Compose on a single bridge network ensuring service discovery via container names.

Service	Port	Purpose
Graylog UI	9000	Web interface for log management
OpenSearch	9200	Index and search backend
Service 1	8081	Example microservice with logging endpoints
Service 2	8082	Example microservice with logging endpoints
Log Service	8083	Kafka consumer and GELF forwarder
Kafka	9092	Message broker
Zookeeper	2181	Kafka coordination
GELF TCP	12201	Graylog input for log ingestion

Table 6.2: Deployed services and ports

Chapter 7

Implementation

7.1 Service Endpoints and Log Publishing

The services provide a REST endpoint that accepts a message and level, normalizes the level, serializes to JSON, and publishes to Kafka.

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

Figure 7.1: Excerpt: service1 LogController (publishing to Kafka)

7.2 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 7.2: Excerpt: log-service Kafka consumer

7.3 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 7.3: Automation script responsibilities

Chapter 8

Configuration and Deployment

8.1 Docker Compose

The environment is fully orchestrated via Docker Compose with persistent volumes and a dedicated network for internal communication.

```
# See Appendix A for the full file
```

Figure 8.1: docker-compose.yml overview

8.2 Logging Configuration

GELF TCP is configured in Logback to enrich messages with MDC and send them to Graylog.

```
<!-- See Appendix E for the full file -->
```

Figure 8.2: Logback GELF appender (log-service)

8.3 Running Locally

Prerequisites: Docker, Docker Compose, Java 21, Maven, `jq`, and `curl`.

1. Build and start: `docker compose up -d -build`
2. Wait for Graylog init container to finish
3. Test endpoints: POST/GET `/log` on services 8081 and 8082

Chapter 9

Testing Strategy and Validation

9.1 Test Plan

Test categories included unit (service-level logic), integration (Kafka publish/consume), system (end-to-end log visibility in Graylog), and acceptance (dashboards/alerts meet requirements).

9.2 Example Test Cases

- Publish INFO log → visible in service stream within SLA.
- Publish ERROR log → alert created and notification fired.
- Kafka broker disruption → logs buffered and delivered after recovery.

9.3 Validation Metrics

We tracked ingestion latency, delivery success rate, and dashboard query times. See Performance chapter for results.

Chapter 10

Monitoring and Alerting

Dashboards visualize total logs, error and warning counts, level distribution, timelines, response time metrics, and top sources. Alerts trigger on ERROR logs per service. Operations can tune the ranges and thresholds based on SLOs.

Chapter 11

DevOps and CI/CD

11.1 Branching and Environments

A typical branching strategy separates long-lived branches (`main` for production, `develop` for staging) and feature branches. Docker images are tagged per commit and environment. Configuration overlays control environment-specific settings.

11.2 Build and Test Pipeline

The CI pipeline builds Java services with Maven, runs unit tests, and publishes Docker images. A smoke test step verifies service health and log ingestion paths. Promotion to staging/production requires manual approval and change records.

11.3 Infrastructure as Code

Although this project uses Docker Compose for local orchestration, a production environment should leverage IaC (Terraform, Bicep) and GitOps practices for reproducible deployments, audit trails, and rollbacks.

Chapter 12

Data Management and Retention

12.1 Index Lifecycle and Retention Policies

OpenSearch index policies should enforce retention windows and rollover strategies to control storage cost and query performance. Templates can shard and replicate appropriately per index.

12.2 Field Mapping and Storage Efficiency

Mapping frequently queried fields (e.g., `service`, `level`, `timestamp`) enables efficient aggregations. Avoid logging sensitive or high-cardinality fields unnecessarily to reduce storage and improve performance.

12.3 Backup and Restore

Regular snapshots of OpenSearch and backups of MongoDB (Graylog metadata) are required for disaster recovery. Test restore procedures periodically.

Chapter 13

SLOs, SLIs, and Alert Strategy

13.1 Service Level Indicators

- Ingestion latency (service to searchable in Graylog).
- Log delivery success rate (messages delivered vs. produced).
- Query latency for common dashboards.

13.2 Service Level Objectives

Example SLOs: *p95 ingestion latency* $< 2s$, *delivery success rate* $> 99.9\%$, *p95 dashboard query latency* $< 3s$. Alerts should be multi-window and noise-controlled.

Chapter 14

Risk Management

Risk	Impact	Mitigation
Kafka outage	Log delivery delayed	Buffering, retries, multi-broker cluster
OpenSearch overload	Slow queries / dropped indexing	Sharding, resource scaling, index lifecycle
PII in logs	Compliance violation	Redaction at source, validation, policies
Credential leakage	Security incident	Secrets management, least privilege

Table 14.1: Key risks and mitigations

Chapter 15

Capacity and Cost Planning

Estimate log volume (events/sec), average event size, daily ingestion (GB/day), retention window, and query concurrency. Use these to size OpenSearch nodes (CPU, RAM, storage IOPS) and Kafka partitions/throughput.

Chapter 16

Incident Response and Runbooks

16.1 Operational Runbooks

- Graylog unreachable: check container, API readiness, ports 9000/12201.
- Kafka publish failures: verify broker health, topic existence, client configs.
- High query latency: review OpenSearch health, index size, slow logs.

Post-incident reviews feed improvements to dashboards, alerts, and capacity plans.

Chapter 17

Multi-Environment Configuration

Use environment variables and separate configuration files for dev/stage/prod. Consider `docker-compose.override.yml` for local overrides and distinct Graylog inputs or streams per environment to avoid cross-contamination.

Chapter 18

Bill of Materials (BoM)

Component	Version	Notes
Spring Boot	3.5.3	Backend framework for services
Java	21	LTS runtime
Graylog	6.3	Log management platform
OpenSearch	2.x	Search and analytics backend
MongoDB	6.0.5	Graylog metadata storage
Kafka	wurstmeister/kafka	Distributed message broker
Zookeeper	7.4.3	Kafka coordination
logstash-logback-encoder	7.3	JSON encoding for Logback
logback-kafka-appender	0.2.0-RC2	Kafka appender (service1)
logback-gelf	4.0.2	GELF TCP appender (log-service)
gelfclient	1.5.1	GELF client library
Docker Compose	v2+	Orchestration for local env

Table 18.1: Bill of Materials (as used in this project)

Chapter 19

Ethical and Legal Considerations

Avoid logging personal data (PII) and secrets. Apply data minimization, masking, and retention aligned with regulations (e.g., GDPR). Ensure access is logged and roles follow the principle of least privilege.

Chapter 20

Security and Compliance

Security considerations include access control to Graylog, secure transport in production (TLS for GELF and Graylog UI), Kafka ACLs for producers/consumers, and secret management for credentials. For compliance, log retention and data minimization policies can be enforced via index settings.

Chapter 21

Performance and Scalability

Kafka decouples producers and consumers, allowing the system to scale independently. Throughput can be increased via topic partitioning and consumer groups. GELF TCP batching and reconnect settings are tuned in Logback to handle transient failures. OpenSearch resources (heap, CPU, I/O) should be sized for indexing rates and query patterns.

21.1 Throughput Estimation

If R is the average event rate (events/s) and S the average size (bytes), the daily ingestion volume is $V = R \times S \times 86400$ bytes/day. With replication factor ρ and overhead factor α, storage needs approximate $V' = V \times \rho \times \alpha$.

21.2 Partitioning Strategy

Choose Kafka partitions P to satisfy consumer parallelism and throughput: $P \geq N_c$ (number of consumer instances). Measure processing time per message to compute required parallelism.

Chapter 22

Testing and Validation

Integration tests validated end-to-end flow: HTTP ingestion at services, Kafka delivery, consumer processing, and visibility in Graylog. Additional tests included error-level alerts and dashboard widget correctness. Synthetic load tests assessed latency and throughput across components.

22.1 Threats to Validity

Internal validity threats include environment-specific behavior in Docker Compose vs. production. External validity threats involve generalizing performance results to different workloads. Mitigations include parameterized tests and workload modeling.

Chapter 23

Troubleshooting and Operations

Common issues involve container startup order, Kafka connectivity, and Graylog API availability. The init script includes retries and readiness checks. Operators can use container logs and Graylog search to diagnose issues quickly.

Chapter 24

Results and Discussion

The platform centralizes logs across services, reduces mean time to resolution through actionable dashboards and alerts, and establishes a foundation for broader observability. Automation minimizes manual steps and configuration drift.

Chapter 25

Project Management

25.1 Organization and Roles

The project was executed by the intern (Idris SADDI) under the supervision of Salma SEDDIK. Stakeholders included SRE and backend teams. Weekly checkpoints ensured alignment with objectives and timely risk handling.

25.2 Schedule Tracking

The Gantt plan was updated as tasks progressed. Buffer periods were reserved for stabilization and tuning. Changes were documented in commit messages and issue threads.

25.3 Communication

Stand-ups and async updates (issue comments, pull request notes) maintained transparency. Decisions were captured near the code (README and scripts) for future maintainers.

Chapter 26

Budget and Resource Estimate

Item	Cost (indicative)	Notes
Compute (containers)	Internal	Developer workstation resources for local runs
Storage (OpenSearch)	Internal	Depends on retention and volume; see capacity planning
Time (engineering)	Internship duration	Design, implementation, validation, documentation

Table 26.1: Budget/resource estimation for prototype environment

Chapter 27

Knowledge Transfer and Maintainability

27.1 Documentation

The repository README details architecture, setup, testing, and troubleshooting. Scripts include inline documentation and self-checks. The report's appendices embed exact source files for traceability.

27.2 Handover

Handover sessions cover environment bootstrap, Graylog automation flows, and dashboard customization. Future contributors can extend services and alert definitions following established patterns.

Chapter 28

Evaluation and Academic Criteria

This work demonstrates: problem framing (observability challenges), method (iterative delivery), engineering rigor (automation, testing), and reflection (risks, ethics, and future work). The deliverable meets academic expectations for completeness, reproducibility, and clarity.

General Conclusion

This internship consolidated my skills in distributed architectures, messaging, observability, and containerized deployments. The delivered solution provides a robust, industrializable foundation for microservices monitoring. Future enhancements include advanced correlation, distributed tracing (OpenTelemetry), SSO for Graylog, and richer alert routing.

Future Work

Planned improvements: secure-by-default TLS across services, CI/CD integration for infrastructure changes, automated capacity planning for OpenSearch, and multi-environment (dev/stage/prod) configuration overlays.

Glossary and Acronyms

GELF Graylog Extended Log Format

MDC Mapped Diagnostic Context

SLI/SLO Service Level Indicator / Objective

SRE Site Reliability Engineering

IaC Infrastructure as Code

Appendix A

Appendix A: docker-compose.yml

```
networks:
  graynet:
    driver: bridge

volumes:
  mongo_data:
  log_data:
  graylog_data:

services:
  # MongoDB for Graylog
  mongo:
    image: mongo:6.0.5-jammy
    container_name: mongodb
    volumes:
      - mongo_data:/data/db
    networks:
      - graynet

  # OpenSearch for Graylog
  opensearch:
    image: opensearchproject/opensearch:2
    container_name: opensearch
    environment:
      - discovery.type=single-node
      - plugins.security.disabled=true
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=5jB6V9TR4JcXkNg*
      - OPENSEARCH_JAVA_OPTS=-Xms1g -Xmx1g
    volumes:
      - log_data:/usr/share/opensearch/data
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    ports:
      - 9200:9200
    networks:
      - graynet
```

```
# Graylog
graylog:
  image: graylog/graylog:6.3
  container_name: graylog
  environment:
    GRAYLOG_PASSWORD_SECRET: "somepasswordpepper"
    GRAYLOG_ROOT_PASSWORD_SHA2: "8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918" # password: admin
    GRAYLOG_HTTP_BIND_ADDRESS: "0.0.0.0:9000"
    GRAYLOG_HTTP_EXTERNAL_URI: "http://localhost:9000/"
    GRAYLOG_ELASTICSEARCH_HOSTS: "http://opensearch:9200"
    GRAYLOG_MONGODB_URI: "mongodb://mongodb:27017/graylog"
  entrypoint: /usr/bin/tini -- wait-for-it opensearch:9200 -- /docker-entrypoint.sh
  volumes:
    - .config/graylog.conf:/usr/share/graylog/config/graylog.conf
    - graylog_data:/usr/share/graylog/data
  depends_on:
    - opensearch
    - mongo
  ports:
    - 9000:9000
    - 12201:12201/tcp    # GELF TCP only
  networks:
    - graynet

graylog-init:
  container_name: graylog-init
  build:
    context: .
    dockerfile: graylog-init.Dockerfile
  depends_on:
    - graylog
  networks:
    - graynet


# Zookeeper
zookeeper:
  image: confluentinc/cp-zookeeper:7.4.3  # Compatible with ARM64
  container_name: zookeeper
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
  ports:
    - "2181:2181"
  networks:
    - graynet


# Kafka
kafka:
  image: wurstmeister/kafka
  container_name: kafka
  ports:
    - "9092:9092"
  environment:
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT
    KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
```

```
    KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  depends_on:
    - zookeeper
  networks:
    - graynet

# Log Service (consumer + forwarder to Graylog)
log-service:
  container_name: log-service
  build:
    context: ./log-service
  environment:
  - GRAYLOG_HOST=graylog
  - GRAYLOG_PORT=12201

  depends_on:
    - kafka
    - graylog
  ports:
    - "8083:8083"
  networks:
    - graynet

# Service 1
service1:
  container_name: service1
  build:
    context: ./service1
  depends_on:
    - kafka
    - log-service
  ports:
    - "8081:8081"
  networks:
    - graynet

# Service 2
service2:
  container_name: service2
  build:
    context: ./service2
  depends_on:
    - kafka
    - log-service
  ports:
    - "8082:8082"
  networks:
    - graynet
```

Appendix B

Appendix B: graylog-init.sh

```
#!/bin/sh
#
# Graylog Initialization Script
# =============================
# This script automatically configures Graylog with streams, dashboards, and monitoring
# for multiple microservices in a log management system.
#
# Prerequisites:
# - Graylog server running and accessible
# - Default admin credentials (admin:admin)
# - jq command line JSON processor installed
#
# Author: Idris SADDI
# Version: 1.0
# Date: August 2025
#

# Exit on error and treat unset variables as errors
set -eu pipefail

echo "[OK] Bash is working with pipefail!"

#
# Configuration Variables
# =======================
GRAYLOG_URL="http://graylog:9000"           # Graylog server URL
AUTH="admin:admin"                          # Authentication credentials
SERVICES="service1 service2"                # List of services to configure (space-separated)

#
# Wait for Graylog API to be Ready
# ================================
echo "[CLOCK] Waiting for Graylog API..."
until curl -s -u "$AUTH" "$GRAYLOG_URL/api/system/inputs" > /dev/null; do
  sleep 3
done
echo "[OK] Graylog API is ready."

#
# Create GELF TCP Input
# ====================
# This creates a GELF (Graylog Extended Log Format) TCP input that will receive
```

```
# log messages from applications on port 12201
#
echo "[PLUG] Creating GELF TCP input..."
GELF_PAYLOAD='{
  "title": "GELF TCP",
  "type": "org.graylog2.inputs.gelf.tcp.GELFTCPInput",
  "configuration": {
    "bind_address": "0.0.0.0",
    "port": 12201,
    "recv_buffer_size": 1048576,
    "use_tls": false
  },
  "global": true,
  "node": null
}'

# Send the input creation request to Graylog API
GELF_RESPONSE=$(curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/system/inputs" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  --data "$GELF_PAYLOAD")

echo "Raw GELF input creation response:"
echo "$GELF_RESPONSE"

# Get default index set ID
DEFAULT_INDEX_SET_ID=$(curl -s -u "$AUTH" "$GRAYLOG_URL/api/system/indices/index_sets" \
  | jq -r '.index_sets[] | select(.default == true) | .id // empty')

if [ -z "$DEFAULT_INDEX_SET_ID" ]; then
  echo "[ERROR] Could not find default index set ID. Exiting..."
  exit 1
fi

echo "[PACKAGE] Default index set ID: $DEFAULT_INDEX_SET_ID"

#
# Service Configuration Loop
# ==========================
# For each service, create a complete monitoring setup including:
# 1. Stream for filtering logs
# 2. Stream rules for service identification
# 3. Search queries for log analysis
# 4. Dashboard with widgets for visualization
#
for SERVICE in $SERVICES; do
  echo "---"
  echo "[REPEAT] Setting up for $SERVICE"

  #
  # 1. Create Stream
  # ================
  # Streams allow filtering and routing of log messages based on rules
  #
  STREAM_PAYLOAD=$(jq -n \
    --arg title "$SERVICE Stream" \
    --arg description "Stream for $SERVICE" \
    --arg index_set_id "$DEFAULT_INDEX_SET_ID" \
    '{
```

```
    "title": $title,
    "description": $description,
    "rules": [],
    "index_set_id": $index_set_id,
    "remove_matches_from_default_stream": false
  }')

STREAM_RESPONSE=$(curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/streams" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  --data "$STREAM_PAYLOAD")
echo "Raw stream creation response:"
echo "$STREAM_RESPONSE"

# Extract stream ID from response
STREAM_ID=$(echo "$STREAM_RESPONSE" | jq -r '.id // .stream_id // empty')
if [ -z "$STREAM_ID" ]; then
  echo "[ERROR] Failed to create stream for $SERVICE."
  exit 1
else
  echo "[OK] Created stream for $SERVICE with ID: $STREAM_ID"
fi

#
# 2. Add Stream Rule
# ==================
# Rules define which messages are routed to this stream
# Type 1 = exact match rule
#
RULE_PAYLOAD=$(jq -n \
  --arg service "$SERVICE" \
  '{
    "field": "service",
    "value": $service,
    "type": 1,
    "inverted": false
  }')
curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/streams/$STREAM_ID/rules" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  --data "$RULE_PAYLOAD"
echo "[OK] Added rule to stream for $SERVICE"

#
# 3. Enable Stream
# ================
# Activate the stream to start processing messages
#
curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/streams/$STREAM_ID/resume" \
  -H "X-Requested-By: cli"
echo "[OK] Enabled stream for $SERVICE"

#
# 4. Create Search Query
# ======================
# Search queries enable analysis and monitoring of log data for specific services
# This creates comprehensive monitoring with multiple search types:
# - Total log count, Error count, Warning count, Log levels distribution
# - Log timeline chart, Response time analysis, Top log sources
```

```
#
SEARCH_PAYLOAD=$(jq -n \
  --arg stream_id "$STREAM_ID" \
  --arg service "$SERVICE" \
  '{
    "queries": [
      {
        "id": "main_query",
        "query": {
          "type": "elasticsearch",
          "query_string": "service:\($service)"
        },
        "timerange": {
          "type": "relative",
          "range": 300
        },
        "filter": {
          "type": "stream",
          "id": $stream_id
        },
        "search_types": [
          {
            "id": "search-type-count",
            "type": "pivot",
            "query": {
              "type": "elasticsearch",
              "query_string": "service:\($service)"
            },
            "timerange": {
              "type": "relative",
              "range": 300
            },
            "series": [
              {
                "id": "count()",
                "type": "count"
              }
            ],
            "rollup": true,
            "sort": []
          },
          {
            "id": "search-type-error-count",
            "type": "pivot",
            "query": {
              "type": "elasticsearch",
              "query_string": "service:\($service) AND level:ERROR"
            },
            "timerange": {
              "type": "relative",
              "range": 300
            },
            "series": [
              {
                "id": "count()",
                "type": "count"
              }
            ],
            "rollup": true,
```

```
    "sort": []
  },
  {
    "id": "search-type-warning-count",
    "type": "pivot",
    "query": {
      "type": "elasticsearch",
      "query_string": "service:\($service) AND level:WARN"
    },
    "timerange": {
      "type": "relative",
      "range": 300
    },
    "series": [
      {
        "id": "count()",
        "type": "count"
      }
    ],
    "rollup": true,
    "sort": []
  },
  {
    "id": "search-type-level-distribution",
    "type": "pivot",
    "query": {
      "type": "elasticsearch",
      "query_string": "service:\($service)"
    },
    "timerange": {
      "type": "relative",
      "range": 300
    },
    "row_groups": [
      {
        "field": "level",
        "type": "values",
        "limit": 10
      }
    ],
    "series": [
      {
        "id": "count()",
        "type": "count"
      }
    ],
    "rollup": true,
    "sort": []
  },
  {
    "id": "search-type-timeline",
    "type": "pivot",
    "query": {
      "type": "elasticsearch",
      "query_string": "service:\($service)"
    },
    "timerange": {
      "type": "relative",
      "range": 300
```

```
    },
    "row_groups": [
      {
        "field": "timestamp",
        "type": "time",
        "interval": {
          "type": "timeunit",
          "timeunit": "1m"
        }
      }
    ],
    "series": [
      {
        "id": "count()",
        "type": "count"
      }
    ],
    "rollup": true,
    "sort": []
  },
  {
    "id": "search-type-response-times",
    "type": "pivot",
    "query": {
      "type": "elasticsearch",
      "query_string": "service:\($service) AND response_time:*"
    },
    "timerange": {
      "type": "relative",
      "range": 300
    },
    "series": [
      {
        "id": "avg(response_time)",
        "type": "avg",
        "field": "response_time"
      },
      {
        "id": "max(response_time)",
        "type": "max",
        "field": "response_time"
      }
    ],
    "rollup": true,
    "sort": []
  },
  {
    "id": "search-type-top-sources",
    "type": "pivot",
    "query": {
      "type": "elasticsearch",
      "query_string": "service:\($service)"
    },
    "timerange": {
      "type": "relative",
      "range": 300
    },
    "row_groups": [
      {
```

```
                "field": "source",
                "type": "values",
                "limit": 5
              }
            ],
            "series": [
              {
                "id": "count()",
                "type": "count"
              }
            ],
            "rollup": true,
            "sort": [
              {
                "type": "pivot",
                "field": "count()",
                "direction": "Descending"
              }
            ]
          }
        ]
      }
    ]
  }')

SEARCH_RESPONSE=$(curl -s -u "$AUTH" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  -X POST "$GRAYLOG_URL/api/views/search" \
  --data-raw "$SEARCH_PAYLOAD")

# Check if search creation was successful
if echo "$SEARCH_RESPONSE" | grep -q '"id"'; then
  echo "[OK] Search queries created successfully"
else
  echo "[ERROR] Search creation failed:"
  echo "$SEARCH_RESPONSE"
  exit 1
fi

# Extract search and query IDs from response
SEARCH_ID=$(echo "$SEARCH_RESPONSE" | jq -r '.id // empty')
QUERY_ID=$(echo "$SEARCH_RESPONSE" | jq -r '.queries[0].id // empty')
if [ -z "$SEARCH_ID" ] || [ -z "$QUERY_ID" ]; then
  echo "[ERROR] Failed to create search for $SERVICE."
  exit 1
fi
echo "[OK] Search created with ID: $SEARCH_ID (query ID: $QUERY_ID)"

#
# 5. Create Dashboard
# ==================
# Dashboards provide visual interface for monitoring service logs
# Each dashboard contains widgets displaying metrics and log counts
#
DASHBOARD_PAYLOAD=$(jq -n \
  --arg service "$SERVICE" \
  --arg search_id "$SEARCH_ID" \
  '{
```

```
    "type": "DASHBOARD",
    "title": "Dashboard for \($service)",
    "summary": "Auto-created dashboard for \($service)",
    "description": "Visualizations for \($service)",
    "search_id": $search_id,
    "state": {},
    "share_request": null,
    "favorite": false
  }')

CREATE_DASHBOARD_RESPONSE=$(curl -s -u "$AUTH" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  -X POST "$GRAYLOG_URL/api/views" \
  --data-raw "$DASHBOARD_PAYLOAD")

if echo "$CREATE_DASHBOARD_RESPONSE" | grep -q '"id"'; then
  DASHBOARD_ID=$(echo "$CREATE_DASHBOARD_RESPONSE" | jq -r '.id // empty')
  echo "[OK] Dashboard created successfully"
else
  echo "[ERROR] Dashboard creation failed:"
  echo "$CREATE_DASHBOARD_RESPONSE"
  exit 1
fi

#
# 6. Add Widgets to Dashboard
# ===========================
# Configure dashboard widgets with comprehensive monitoring visualization:
# - Total log count, Error count, Warning count (numeric widgets)
# - Log levels distribution (pie chart)
# - Log timeline (line chart)
# - Response time metrics (bar chart)
# - Top log sources (table)
# Using a larger 4x4 grid layout for better visibility
#
NEW_STATE=$(jq -n \
  --arg qid "$QUERY_ID" \
  --arg service "$SERVICE" \
  '{
    ($qid): {
      "selected_fields": [],
      "static_message_list_id": null,
      "titles": {
        "widgets": {
          "widget-1": "[CHART] Total Logs (5min)",
          "widget-2": "[ALARM] Error Logs (5min)",
          "widget-3": "[WARNING] Warning Logs (5min)",
          "widget-4": "[TREND] Log Levels Distribution",
          "widget-5": "[TIMER] Log Timeline",
          "widget-6": "[ROCKET] Response Time Metrics",
          "widget-7": "[SEARCH] Top Log Sources"
        }
      },
      "widgets": [
        {
          "id": "widget-1",
          "type": "aggregation",
          "filter": null,
```

```
      "filters": [],
      "config": {
        "visualization": "numeric",
        "row_pivots": [],
        "column_pivots": [],
        "series": [
          {
            "function": "count()"
          }
        ],
        "sort": [],
        "rollup": true
      }
    },
    {
      "id": "widget-2",
      "type": "aggregation",
      "filter": null,
      "filters": [],
      "config": {
        "visualization": "numeric",
        "row_pivots": [],
        "column_pivots": [],
        "series": [
          {
            "function": "count()"
          }
        ],
        "sort": [],
        "rollup": true
      }
    },
    {
      "id": "widget-3",
      "type": "aggregation",
      "filter": null,
      "filters": [],
      "config": {
        "visualization": "numeric",
        "row_pivots": [],
        "column_pivots": [],
        "series": [
          {
            "function": "count()"
          }
        ],
        "sort": [],
        "rollup": true
      }
    },
    {
      "id": "widget-4",
      "type": "aggregation",
      "filter": null,
      "filters": [],
      "config": {
        "visualization": "pie",
        "row_pivots": [
          {
```

```
        "field": "level",
        "type": "values",
        "config": {
          "limit": 10
        }
      }
    ],
    "column_pivots": [],
    "series": [
      {
        "function": "count()"
      }
    ],
    "sort": [],
    "rollup": true
  }
},
{
  "id": "widget-5",
  "type": "aggregation",
  "filter": null,
  "filters": [],
  "config": {
    "visualization": "line",
    "row_pivots": [
      {
        "field": "timestamp",
        "type": "time",
        "config": {
          "interval": {
            "type": "timeunit",
            "unit": "minutes",
            "value": 1
          }
        }
      }
    ],
    "column_pivots": [],
    "series": [
      {
        "function": "count()"
      }
    ],
    "sort": [],
    "rollup": true
  }
},
{
  "id": "widget-6",
  "type": "aggregation",
  "filter": null,
  "filters": [],
  "config": {
    "visualization": "bar",
    "row_pivots": [],
    "column_pivots": [],
    "series": [
      {
        "function": "avg(response_time)"
```

```
            },
            {
              "function": "max(response_time)"
            }
          ],
          "sort": [],
          "rollup": true
        }
      },
      {
        "id": "widget-7",
        "type": "aggregation",
        "filter": null,
        "filters": [],
        "config": {
          "visualization": "table",
          "row_pivots": [
            {
              "field": "source",
              "type": "values",
              "config": {
                "limit": 5
              }
            }
          ],
          "column_pivots": [],
          "series": [
            {
              "function": "count()"
            }
          ],
          "sort": [
            {
              "type": "pivot",
              "field": "count()",
              "direction": "Descending"
            }
          ],
          "rollup": true
        }
      }
    ],
    "widget_mapping": {
      "widget-1": ["search-type-count"],
      "widget-2": ["search-type-error-count"],
      "widget-3": ["search-type-warning-count"],
      "widget-4": ["search-type-level-distribution"],
      "widget-5": ["search-type-timeline"],
      "widget-6": ["search-type-response-times"],
      "widget-7": ["search-type-top-sources"]
    },
    "positions": {
      "widget-1": {"col": 1, "row": 1, "height": 2, "width": 2},
      "widget-2": {"col": 3, "row": 1, "height": 2, "width": 2},
      "widget-3": {"col": 5, "row": 1, "height": 2, "width": 2},
      "widget-4": {"col": 1, "row": 3, "height": 3, "width": 3},
      "widget-5": {"col": 4, "row": 3, "height": 3, "width": 4},
      "widget-6": {"col": 1, "row": 6, "height": 2, "width": 3},
      "widget-7": {"col": 4, "row": 6, "height": 2, "width": 4}
```

```
      },
      "formatting": {"highlighting": []},
      "display_mode_settings": {"positions": {}}
    }
  }')

# Merge new widget state with existing dashboard
EXISTING_VIEW=$(curl -s -u "$AUTH" "$GRAYLOG_URL/api/views/$DASHBOARD_ID")
MERGED_VIEW=$(echo "$EXISTING_VIEW" | jq --argjson frag "$NEW_STATE" '.state = (.state // {}) * $frag')

# Update dashboard with widget configuration
UPDATE_RESPONSE=$(curl -s -u "$AUTH" -X PUT "$GRAYLOG_URL/api/views/$DASHBOARD_ID" \
  -H "Content-Type: application/json" -H "X-Requested-By: cli" \
  --data-raw "$MERGED_VIEW")

# Check if dashboard update was successful
if echo "$UPDATE_RESPONSE" | grep -q '"id"'; then
  echo "[OK] Dashboard widgets configured successfully"
else
  echo "[ERROR] Dashboard update failed:"
  echo "$UPDATE_RESPONSE"
  exit 1
fi

#
# 7. Create Event Definition for Error Monitoring
# ================================================
# Event definitions trigger alerts when specific conditions are met
# This creates an alert that fires when ERROR logs are detected for the service
#
echo "[ALARM] Creating error alert event definition for $SERVICE..."
EVENT_DEF_PAYLOAD=$(jq -n \
  --arg service "$SERVICE" \
--arg stream_id "$STREAM_ID" \
  '{
    "title": "\($service) - ERROR Alert",
    "description": "Alert triggered when ERROR logs are detected for \($service)",
    "priority": 2,
    "alert": true,
    "config": {
      "type": "aggregation-v1",
  "query": "service:\($service) AND (level:ERROR OR level:3)",
  "streams": ["\($stream_id)"],
      "group_by": [],
      "series": [
        {
          "id": "count()",
          "type": "count"
        }
      ],
      "conditions": {
        "expression": {
          "expr": ">",
          "left": {
            "expr": "number-ref",
            "ref": "count()"
          },
          "right": {
            "expr": "number",
```

```
            "value": 0
          }
        }
      },
      "search_within_ms": 60000,
      "execute_every_ms": 60000
    },
    "field_spec": {},
    "key_spec": [],
    "notification_settings": {
      "grace_period_ms": 60000,
      "backlog_size": 5
    },
    "notifications": []
  }')

EVENT_DEF_RESPONSE=$(curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/events/definitions" \
  -H "Content-Type: application/json" \
  -H "X-Requested-By: cli" \
  --data-raw "$EVENT_DEF_PAYLOAD")
if [ -z "$EVENT_DEF_RESPONSE" ]; then
  echo "[ERROR] Failed to create event definition for $SERVICE."
  exit 1
fi


EVENT_ID=$(echo "$EVENT_DEF_RESPONSE" | jq -r '.id // empty')
if [ -z "$EVENT_ID" ]; then
  echo "[ERROR] Failed to create event definition for $SERVICE."
  echo "Error details: $EVENT_DEF_RESPONSE"
  exit 1
else
  echo "[OK] Created event definition for $SERVICE with ID: $EVENT_ID"
SERIES_TYPE_RETURNED=$(echo "$EVENT_DEF_RESPONSE" | jq -r '.config.series[0].type // empty') || true
  if [ -z "$SERIES_TYPE_RETURNED" ] || [ "$SERIES_TYPE_RETURNED" = "null" ]; then
    echo "[TOOL] Patching event definition to add missing series type..."
    EVENT_DEF_CURRENT=$(curl -s -u "$AUTH" "$GRAYLOG_URL/api/events/definitions/$EVENT_ID")
    if echo "$EVENT_DEF_CURRENT" | grep -q '"id"'; then
EVENT_DEF_PATCHED=$(echo "$EVENT_DEF_CURRENT" | jq '.config.series = [ { "id":"count()", "type":"count" } ]')
      PATCH_RESP=$(curl -s -u "$AUTH" -X PUT "$GRAYLOG_URL/api/events/definitions/$EVENT_ID" \
        -H "Content-Type: application/json" -H "X-Requested-By: cli" \
        --data-raw "$EVENT_DEF_PATCHED")
      if echo "$PATCH_RESP" | grep -q '"series"'; then
        echo "[OK] Series type patched"
      else
        echo "[WARNING] Failed to patch series type: $PATCH_RESP"
      fi
    else
      echo "[WARNING] Could not retrieve event definition for series patch"
    fi
  fi

  # Enable the event definition (set state = ENABLED)
  EVENT_DEF_GET=$(curl -s -u "$AUTH" "$GRAYLOG_URL/api/events/definitions/$EVENT_ID")
  if echo "$EVENT_DEF_GET" | grep -q '"id"'; then
    EVENT_DEF_ENABLED=$(echo "$EVENT_DEF_GET" | jq '.state = "ENABLED"')
    ENABLE_EVENT_RESPONSE=$(curl -s -u "$AUTH" -X PUT "$GRAYLOG_URL/api/events/definitions/$EVENT_ID" \
      -H "Content-Type: application/json" -H "X-Requested-By: cli" \
      --data-raw "$EVENT_DEF_ENABLED")
```

```
    if echo "$ENABLE_EVENT_RESPONSE" | grep -qi 'RequestError'; then
      echo "[WARNING] Failed to enable event definition (may already be enabled): $ENABLE_EVENT_RESPONSE"
    else
      echo "[OK] Event definition enabled"
echo "[INFO] Event scoped to stream $STREAM_ID for $SERVICE"
    fi
  else
    echo "[WARNING] Could not retrieve event definition for enabling: $EVENT_DEF_GET"
  fi

  #
  # 8. Create UI Notification
  # =========================
  # Notifications define how alerts are delivered to users
  # This creates an in-app notification for the error alerts
  #
  echo "[ANNOUNCE] Creating HTTP notification for $SERVICE (fires webhook)..."

  # Webhook endpoint (override by setting GRAYLOG_WEBHOOK_URL env before running script)
  WEBHOOK_URL=${GRAYLOG_WEBHOOK_URL:-"http://example.com/graylog-webhook"}
  BODY_TEMPLATE='{"service":"${event_definition_title}","event_id":"${event.id}","message":"${event.message}"}'

  NOTIF_PAYLOAD=$(jq -n \
    --arg service "$SERVICE" \
    --arg url "$WEBHOOK_URL" \
    --arg body "$BODY_TEMPLATE" \
    '{
      title: ($service + " - HTTP Notification"),
      description: ("Webhook notification for " + $service + " error alerts"),
      config: {
        type: "http-notification-v1",
        url: $url,
        api_key_as_header: false,
        skip_tls_verification: true
      }
    }')

  NOTIF_RESPONSE=$(curl -s -u "$AUTH" -X POST "$GRAYLOG_URL/api/events/notifications" \
    -H "Content-Type: application/json" \
    -H "X-Requested-By: cli" \
    --data-raw "$NOTIF_PAYLOAD")
  if echo "$NOTIF_RESPONSE" | grep -q '"id"'; then
    echo "[OK] Notification created"
  else
    echo "[ERROR] Notification creation failed: $NOTIF_RESPONSE"
    exit 1
  fi

  NOTIF_ID=$(echo "$NOTIF_RESPONSE" | jq -r '.id // empty')
  if [ -z "$NOTIF_ID" ]; then
    echo "[ERROR] Skipping linking due to notification creation failure."
  else
    echo "[OK] Created notification for $SERVICE with ID: $NOTIF_ID"

    #
    # 9. Link Notification to Event Definition
    # ========================================
    # Connect the notification to the event definition so alerts are sent
    # when the event conditions are triggered
```

```
      #
      echo "[LINK] Attaching notification to event definition for $SERVICE (updating event definition)..."
      CURRENT_EVENT_DEF=$(curl -s -u "$AUTH" "$GRAYLOG_URL/api/events/definitions/$EVENT_ID")
      if echo "$CURRENT_EVENT_DEF" | grep -q '"id"'; then
        UPDATED_EVENT_DEF=$(echo "$CURRENT_EVENT_DEF" | jq --arg nid "$NOTIF_ID" '.notifications = [ { "notification_id"
        UPDATE_NOTIF_RESPONSE=$(curl -s -u "$AUTH" -X PUT "$GRAYLOG_URL/api/events/definitions/$EVENT_ID" \
          -H "Content-Type: application/json" -H "X-Requested-By: cli" \
          --data-raw "$UPDATED_EVENT_DEF")
        if echo "$UPDATE_NOTIF_RESPONSE" | grep -q '"notifications"'; then
          echo "[OK] Notification attached to event definition"
        else
          echo "[WARNING] Failed to attach notification: $UPDATE_NOTIF_RESPONSE"
        fi
      else
        echo "[ERROR] Could not retrieve event definition for notification attachment: $CURRENT_EVENT_DEF"
      fi
    fi

  fi

  echo "[PARTY] $SERVICE setup complete!"
done

echo "[ROCKET] All services successfully configured in Graylog."
```

Appendix C

Appendix C: service1 LogController.java

```
package com.idris.service1;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/log")
public class LogController {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public LogController(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

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

    @PostMapping
    public ResponseEntity<?> postLog(
        @RequestParam(defaultValue = "Test log message") String message,
        @RequestParam(required = false, defaultValue = "INFO") String level
    ) {
        try {
            Map<String, Object> log = new HashMap<>();
            log.put("timestamp", java.time.Instant.now().toString());
            log.put("message", message);
            log.put("level", normalizeLevel(level));
            log.put("service", applicationName);

            String logJson = new com.fasterxml.jackson.databind.ObjectMapper().writeValueAsString(log);
            kafkaTemplate.send("logs", logJson);
            return ResponseEntity.ok("Log sent to Kafka: " + logJson);
        } catch (Exception e) {
```

```
            return ResponseEntity.internalServerError().body("Failed to send log: " + e.getMessage());
        }
    }

    @GetMapping
    public ResponseEntity<?> getLog(
        @RequestParam(defaultValue = "Test log message") String message,
        @RequestParam(required = false, defaultValue = "INFO") String level
    ) {
        try {
            Map<String, Object> log = new HashMap<>();
            log.put("timestamp", java.time.Instant.now().toString());
            log.put("message", message);
            log.put("level", normalizeLevel(level));
            log.put("service", applicationName);

            String logJson = new com.fasterxml.jackson.databind.ObjectMapper().writeValueAsString(log);
            kafkaTemplate.send("logs", logJson);
            return ResponseEntity.ok("Log sent to Kafka: " + logJson);
        } catch (Exception e) {
            return ResponseEntity.internalServerError().body("Failed to send log: " + e.getMessage());
        }
    }

    private String normalizeLevel(String level) {
        if (level == null)
            return "INFO";
        String upper = level.trim().toUpperCase();
        switch (upper) {
            case "ERROR":
                return "ERROR";
            case "WARN":
            case "WARNING":
                return "WARN";
            case "DEBUG":
            case "TRACE":
            case "INFO":
                return upper;
            default:
                return "INFO"; // fallback
        }
    }
}
```

Appendix D

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

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

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

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

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

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

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>
```

References

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.