



NATIONAL SCHOOL OF COMPUTER SCIENCE AND SYSTEMS ANALYSIS -
RABAT

DEPARTMENT OF WEB AND MOBILE ENGINEERING

Project Report

Optimization of DevOps Log Persistence in NoSQL Databases

Prepared by:

JAMAL AYMANE
OUNZAR MOHAMMED
GHANNAMI DRISS
FADLOUALLAH MOHAMMED

Supervised by:

Prof. Karima MOUMANE

Academic Year 2024/2025

List of Figures

3.1	DevOps Log Persistence Architecture - Full CI/CD + Logging Pipeline	13
3.2	Benchmarking Architecture - Logging Stack + log_comparison.py	15
4.1	Main Repository	18
4.2	logback spring file	19
4.3	Logstash Pipeline	20
4.4	Jenkins Dashboard — Pipeline status for each service	21
4.5	Docker Desktop – Container view confirming successful deployment	23
5.1	Insert Time Comparison	27
5.2	CPU Usage During Insert/Query	28
5.3	Memory Usage	28
5.4	Query Time Comparison	29
5.5	Kibana Dashboard — Log Level Distribution	30

Contents

1	Introduction	4
1.1	Context and Motivation	4
1.2	Problem Statement	4
1.3	Project objectives	4
1.4	Contributions Expected	4
2	State of the Art	6
2.1	DevOps and Log Management	6
2.2	Databases for Log Management	7
2.3	Related Work	7
3	Methodology and System Design	11
3.1	Methodological Approach	11
3.1.1	Analysis of Specific Requirements	11
3.1.2	Incremental Approach	11
3.2	Detailed Technical Architecture	11
3.2.1	Architecture Overview	11
3.2.2	Data Flow and Integration	13
3.3	Technological Choices and Justifications	14
3.3.1	The ELK Stack (Elasticsearch, Logstash, Kibana)	14
3.3.2	Complementary Databases	14
3.4	Deployment Infrastructure	14
3.4.1	Containerization	14
3.4.2	Container Organization	15
3.5	Evaluation Framework and Benchmarking	15
3.5.1	Benchmarking Tools	15
3.5.2	Evaluation Metrics	16
4	Implementation and Experimentation	17
4.1	Repository Structure and Service Distribution	17
4.2	Log Routing Configuration	19
4.2.1	Logback Configuration in Microservices	19
4.2.2	Logstash Pipeline Configuration	20
4.3	CI/CD Pipeline Implementation with Jenkins	21
4.3.1	Pipeline Architecture	21
4.3.2	Pipeline Responsibilities	21
4.3.3	Example: API Gateway Pipeline Breakdown	22

- 4.3.4 Build Monitoring and Management 22
- 4.4 Container Deployment and Verification 23
 - 4.4.1 Deployment Overview 23
 - 4.4.2 Validation Points 23
- 4.5 Benchmarking Script: Log Comparison 24
 - 4.5.1 Objectives 24
 - 4.5.2 Implementation Details 24
 - 4.5.3 Execution and Outputs 25
 - 4.5.4 Environment Preparation 25
- 5 Results and Discussion 26**
 - 5.1 Benchmarking Results 26
 - 5.1.1 Raw Benchmark Output 26
 - 5.1.2 Visual Comparison of Results 27
 - 5.2 Log Visualization in Kibana 29
 - 5.2.1 Visualization Workflow 29
 - 5.2.2 Dashboard Output 29
 - 5.3 Discussion 30
 - 5.4 Summary of Key Findings 30
 - 5.5 Limitations 31
 - 5.6 Future Work 31
- 6 Conclusion 32**

Chapter 1

Introduction

1.1 Context and Motivation

Log management plays a crucial role in DevOps pipelines, enabling teams to monitor system behavior, diagnose issues, and ensure reliability across development and production environments. As systems become more complex and distributed, efficient log handling becomes essential for maintaining performance, enabling automation, and supporting continuous integration and delivery processes.

1.2 Problem Statement

In modern DevOps pipelines, the persistence of logs is critical for ensuring long-term visibility, auditability, and troubleshooting. However, without proper log retention strategies, valuable data can be lost, making it difficult to trace issues, identify patterns, or conduct thorough audits. The challenge is how to maintain log persistence in a scalable and cost-effective manner, ensuring that logs are available over time for analysis, compliance, and continuous improvement, without overwhelming storage systems or compromising performance.

1.3 Project objectives

1. Study NoSQL solutions suitable for DevOps log persistence.
2. Implement a log management system using Elasticsearch and Kibana.
3. Compare the performance between SQL and NoSQL in this context.

1.4 Contributions Expected

Following an initial conceptual analysis of the log management system, we will concentrate on deploying its components within a reproducible environment. This includes establishing CI/CD pipelines, validating strategies for log persistence through a structured benchmarking process, and

ensuring system reliability. The solution will leverage a microservices architecture, containerization via Docker, and automation through Jenkins. Additionally, benchmarking tools and visual dashboards will be implemented to evaluate and illustrate the system's performance effectively.

Chapter 2

State of the Art

2.1 DevOps and Log Management

- The DevOps pipeline is based on a series of automated steps ranging from development to production, including testing and continuous deployment phases. In this context, logging is essential to ensure traceability of actions, quickly detect errors, and facilitate debugging. It also helps improve collaboration between development and operations teams by providing concrete data on system behavior.
- With the automation of processes, log management at each stage becomes more sophisticated. It involves real-time analysis of a large volume of data from various sources, centralizing and correlating it, and extracting meaningful insights to ensure the reliability, security, and performance of the pipeline.
- Furthermore, the persistence of logs plays a crucial role in DevOps pipelines. Keeping logs over the long term ensures a complete execution history, which is essential for audits and the continuous improvement of processes. Without this persistence, critical information may be lost, making it harder to understand past behaviors and identify the root causes of issues.
- Logs can be stored in different formats depending on the tools used and the analysis needs. The JSON format is widely used because it is structured, machine-readable, and easily exploitable by analysis or visualization tools. Some tools, such as Logstash, use their own formats or processing pipelines to parse, transform, and normalize logs before sending them to databases like Elasticsearch. The choice of format thus affects the ease of automated processing, compatibility with tools, and the readability of data for developers and operations teams.

2.2 Databases for Log Management

Database (Type)	Strengths	Limitations	Recommended Use Case
Elasticsearch (NoSQL document / search)	Very powerful full-text search, Fast indexing, Easy integration with Kibana / Logstash (ELK)	Complex maintenance at large scale, High memory consumption	Log analysis, observability, real-time dashboards
InfluxDB (NoSQL time-series)	Optimized for time series, Fast for time-based queries, Low latency	Less suitable for unstructured logs, Limited search capabilities	System logs, infrastructure monitoring, metrics
MongoDB (NoSQL document-based)	Flexible (schema-less), Good for semi-structured data, Easy horizontal scalability	Less performant on large volumes without tuning, Full-text search less powerful than ELK	Application logs, auditing, structured or JSON-like storage
PostgreSQL (SQL relational)	Solid, mature, and reliable, Good support for JSON & advanced indexes, Powerful for complex queries	Less performant at very large scale for unstructured logs	Regulatory logs, small/medium volumes, SQL-oriented projects

Synthèse finale :

- Elasticsearch est incontournable pour les recherches puissantes et dashboards dynamiques avec Kibana.
- InfluxDB et Loki se démarquent par leur légèreté et leur efficacité pour les logs temporels et infrastructure.
- MongoDB convient bien aux logs semi-structurés avec besoin de flexibilité.
- PostgreSQL est parfait pour des logs structurés avec une approche SQL classique, notamment si la conformité est importante.

2.3 Related Work

Table 2.1: Studies and contributions related to database management and DevOps

Reference	Year	Objectives	Tools and Approach	Results / Issues	Future Work
Improving the software logging practices in DevOps	2021	Improve the quality of logging code and enrich log analysis in DevOps.	Analysis of open-source Java projects, development of the LogCoCo tool (coverage estimation via logs).	Existing tools detect few real problems. Logging becomes more complex at scale. Few guidelines/tools. Limited context in log analysis. Increasing difficulty with project size.	Create better guidelines/tools. Extend use to other systems.
A Comparative Study of NoSQL Database Performance in Big Data Analytics	2024	Compare the performance of NoSQL databases in Big Data analytics.	Tests on MongoDB, Cassandra, Couchbase, Redis with Hadoop/Spark.	Performance varies depending on use case and workload. Limited study, not fully realistic environment.	Expand the study. Add more databases and real scenarios.

Continued on next page

Table 2.1 – continued from previous page

Reference	Year	Objectives	Tools and Approach	Results / Issues	Future Work
Automating Databases for Modern DevOps Practices	2024	Automate databases in DevOps for greater efficiency.	Terraform, Ansible, Liquibase, RDS, Prometheus.	Fast deployments, fewer errors, better collaboration. Complexity, risk of over-automation.	Better integrate DBs into CI/CD. Strengthen security.
Persistence and Replication Strategies for Databases	2024	Analyze persistence and replication strategies for databases.	Sync/async replication, master-slave, MongoDB, Cassandra.	High availability, fault tolerance, lower latency. Complexity, conflicts, scalability.	Cloud-native. AI for optimization. Blockchain. Edge computing.
Persistence Strategies – Database Optimization	2024	Optimize persistence for better performance.	Deduplication, clustering, modeling, cloud.	Less redundancy, more availability and efficiency. Complexity, scalability, conflicts.	Cloud-native. AI. Blockchain. Edge computing.

Continued on next page

Table 2.1 – continued from previous page

Reference	Year	Objectives	Tools and Approach	Results / Issues	Future Work
Efficient Data Ingestion and Query Processing for LSM-Based Storage System	2019	Improve data ingestion and query processing in LSM-based storage systems.	Optimizations for batch point lookups and efficient maintenance strategies, implemented in Apache AsterixDB.	Significant improvement in LSM secondary index performance. Complexity in managing auxiliary structures in LSM systems.	Extend optimizations to other database systems. Improve support for ad hoc queries.
LSM-based Storage Techniques: A Survey	2019	Present a state of the art of storage techniques based on LSM trees.	LevelDB, RocksDB, HBase, Cassandra, AsterixDB.	Synthesis of optimizations and trade-offs. Complexity, performance/consistency balance.	Better handle writes and queries. Cloud/edge integration.

Chapter 3

Methodology and System Design

3.1 Methodological Approach

The design of our log persistence system is based on a structured DevOps approach, aimed at ensuring seamless integration between the different phases of the pipeline. This methodology revolves around the following principles:

3.1.1 Analysis of Specific Requirements

We first conducted a precise identification of logging requirements in our context. This analysis revealed several critical needs:

- The ability to process a large volume of heterogeneous logs
- The need for long-term persistence for audits and retrospective analyses
- Query performance adapted to different use cases (real-time vs. historical)
- Smooth integration with the existing microservices ecosystem

3.1.2 Incremental Approach

The system development followed an iterative approach, allowing us to validate each component before its complete integration. This method enabled us to progressively adjust the architecture based on feedback and performance test results.

3.2 Detailed Technical Architecture

The architecture designed for our log management system is based on a modular structure, ensuring both processing reliability and the flexibility required for system evolution.

3.2.1 Architecture Overview

The system is based on a modular DevOps architecture combining a complete CI/CD pipeline with a centralized logging system. The architecture can be divided into three main layers: the source code repositories, the Dockerized deployment system, and the logging materials.

1. Repositories

Several Git repositories are maintained for each service and component:

- *Logging-Materials Repo*
- *Infrastructure Repo* (Databases and Kafka)
- *Billing-Service Repo*
- *Analytics-Service Repo*
- *Patient-Service Repo*
- *Auth-Service Repo*
- *API-Gateway Repo*

Jenkins serves as the central CI/CD tool, fetching the code from these repositories, building Docker images, and deploying them into the environment.

2. Docker

The Jenkins pipeline deploys all services as Docker containers, organized into two main categories: microservices and logging materials.

2.1 Microservices

Each business domain is encapsulated in a dedicated service container:

- **API-Gateway Container:** Routes requests to respective services.
- **Auth-Service Container:** Handles authentication and authorization, connected to *Auth-DB (PostgreSQL)*.
- **Billing-Service Container:** Manages billing operations and interacts with the *Kafka-Service Container*.
- **Patient-Service Container:** Manages patient-related operations, linked to *Patient-DB (PostgreSQL)*.
- **Analytics-Service Container:** Performs analytics tasks.

2.2 Logging Materials

All microservices forward their logs to a centralized logging system based on the ELK stack and additional storage solutions:

- **Logstash Container:** Collects and parses logs from all microservices.
- **Elasticsearch Container:** Stores and indexes logs for quick querying.
- **Kibana Container:** Provides a UI for visualizing and exploring logs.

3.2. DETAILED TECHNICAL ARCHITECTURE

- **MySQL & MongoDB Containers:** Serve as additional log storage backends for persistence and analysis.

This containerized approach provides scalability, isolation, and centralized observability, allowing the system to be monitored and maintained effectively.

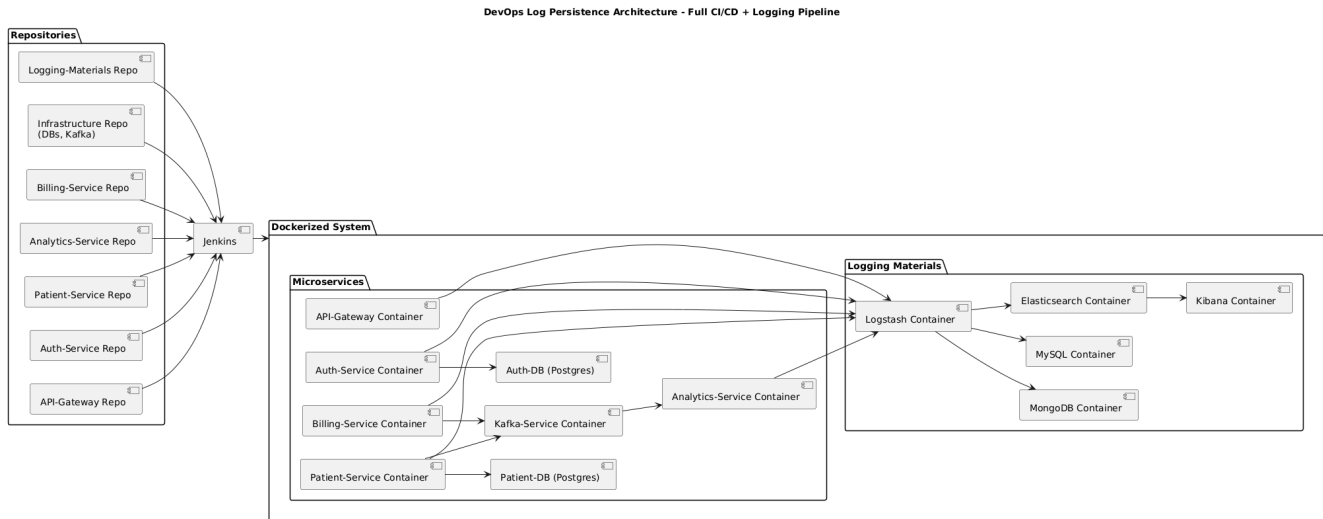


Figure 3.1: DevOps Log Persistence Architecture - Full CI/CD + Logging Pipeline

3.2.2 Data Flow and Integration

The data flow through our architecture follows a well-defined path:

- The various microservices (API-Gateway, authentication, billing services, etc.) generate logs during their execution
- These logs are collected by Logstash via HTTP POST requests
- Logstash normalizes and enriches this data before routing it to different storage systems:
 - Elasticsearch for indexing and fast searching
 - MongoDB for complex semi-structured logs
 - MySQL for data requiring well-defined relationships

This architecture allows for a clear separation of responsibilities while maintaining overall system coherence.

3.3 Technological Choices and Justifications

Our technology selection is based on a thorough analysis of the advantages and limitations of each solution, directly linked to the identified needs.

3.3.1 The ELK Stack (Elasticsearch, Logstash, Kibana)

The adoption of the ELK stack as a central component of our architecture is explained by:

- Elasticsearch's ability to efficiently index and search large volumes of textual data
- Logstash's flexibility for ingesting and transforming logs of various formats
- The advanced visualization capabilities offered by Kibana

This combination provides a proven solution for log processing in DevOps environments.

3.3.2 Complementary Databases

The integration of MongoDB and MySQL in addition to Elasticsearch addresses specific needs:

- **MongoDB:** Its schema flexibility is particularly suitable for semi-structured logs whose format may evolve. Its write performance is also an asset for continuous ingestion of large data volumes.
- **MySQL:** For certain types of logs requiring well-defined relationships and complex queries, a relational database offers advantages in terms of integrity and structured querying.

This hybrid approach optimizes storage and access to logs according to their nature and usage.

3.4 Deployment Infrastructure

3.4.1 Containerization

The entire system is deployed in a containerized environment based on Docker, as shown in the first diagram. This approach ensures:

- Appropriate isolation of components
- Facilitated horizontal scalability
- Consistency between development and production environments
- Simplified deployment via existing CI/CD pipelines

3.4.2 Container Organization

Containers are organized according to a functional logic:

- A group of containers for business microservices
- A group dedicated to logging components (Logstash, Elasticsearch, Kibana, etc.)
- Specific containers for each type of database

This organization allows for granular resource management and simplified maintenance.

3.5 Evaluation Framework and Benchmarking

To validate our architectural choices, we developed a comprehensive evaluation framework, illustrated in the second diagram.

3.5.1 Benchmarking Tools

The central tool of our evaluation framework is the `log_comparison.py` script which:

- Generates representative test data
- Performs insertion and query operations on each storage solution
- Measures performance in terms of latency, resource utilization (CPU/RAM)
- Compares different solutions under similar conditions

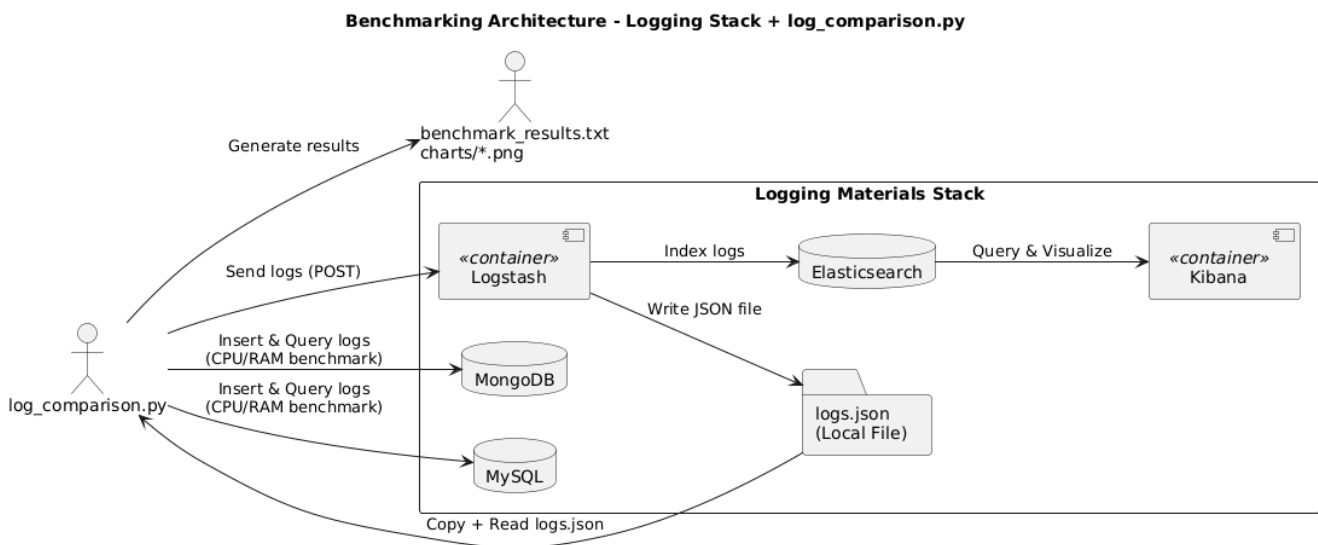


Figure 3.2: Benchmarking Architecture - Logging Stack + `log_comparison.py`

3.5.2 Evaluation Metrics

Our evaluation relies on several key metrics:

- Ingestion throughput: number of logs processed per second
- Query latency: time required to retrieve specific information
- Resource utilization: memory and CPU footprint of each component
- Scalability: system behavior under increasing load

The results of these evaluations are automatically compiled into reports (`benchmark_results.txt`) and visualized as graphs to facilitate comparative analysis.

Chapter 4

Implementation and Experimentation

In this chapter, we detail the practical realization of the proposed architecture. After a conceptual exploration of the log management system, this phase focuses on implementing each component in a reproducible environment, ensuring CI/CD automation, and validating the log persistence strategies through a benchmarking workflow. This implementation is based on a microservice approach, container orchestration via Docker, and pipeline automation with Jenkins. Benchmarking tools and dashboards are developed to assess and visualize the performance of the proposed solution.

The experimentation is conducted on an ARM-based system (MacBook M4), which ensures compatibility with lightweight and modern development environments. The goal is to validate both the technical feasibility and performance efficiency of NoSQL- and SQL-based log persistence strategies.

4.1 Repository Structure and Service Distribution

Introduction

To facilitate modular development, testing, and deployment, each service and infrastructure component is isolated into its own GitHub repository. This decision aligns with DevOps best practices for microservice architectures and allows for clean CI/CD integration using Jenkins pipelines.

Repository Organization

The project is composed of the following repositories:

- **Microservices:**
 - `api-gateway`
 - `auth-service`
 - `patient-service`
 - `billing-service`
 - `analytics-service`

- **Infrastructure:**
 - `infrastructure` — includes Dockerfiles and Docker Compose files for `auth-db`, `patient-db`, and `kafka`
- **Logging Stack:**
 - `logging-materials` — contains configurations for `logstash`, `elasticsearch`, `kibana`, `mongodb`, and `mysql`

Each repository is designed to be independently versioned and deployed. This allows services to be updated, tested, or redeployed in isolation, improving maintainability and scalability of the platform. In addition, the use of Docker guarantees environment consistency between development, test, and production phases. (Figure 4.1)

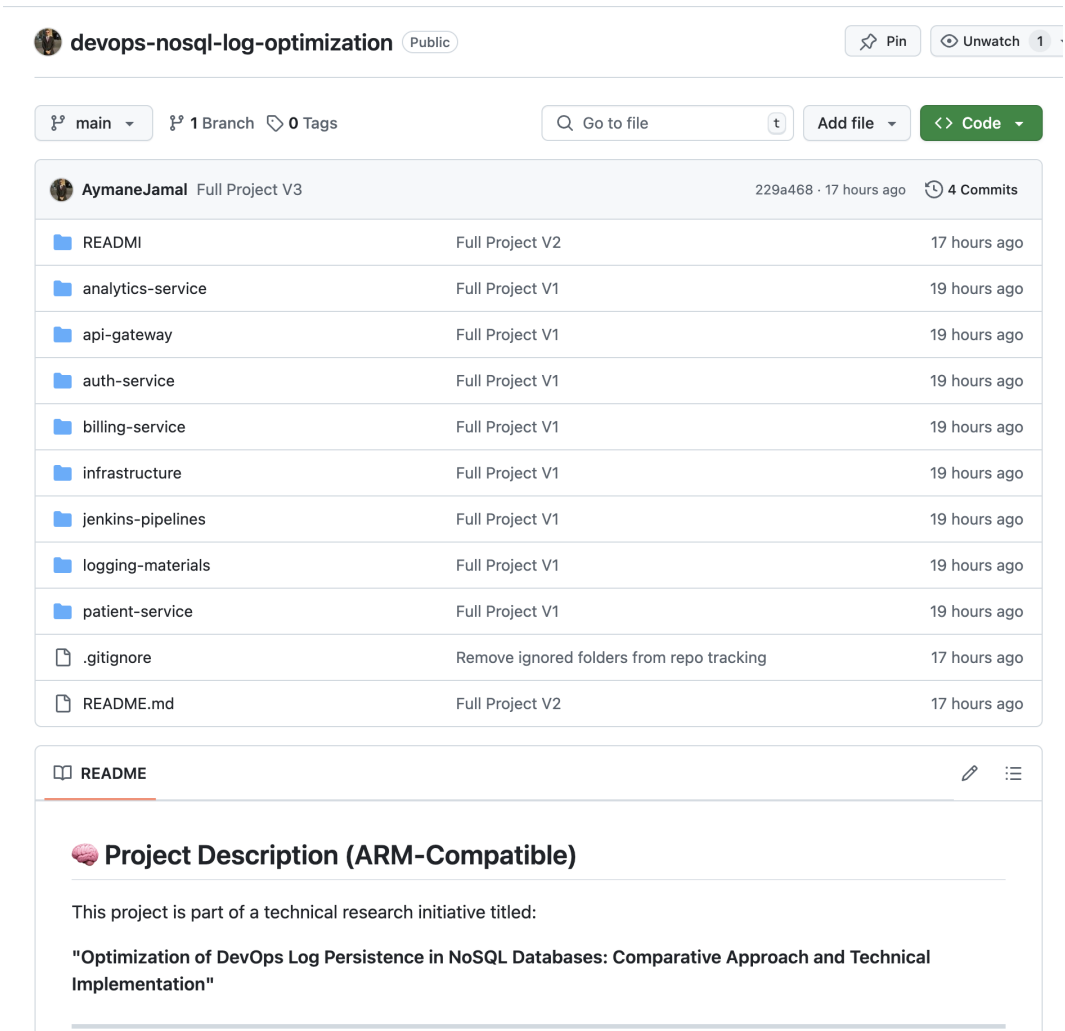


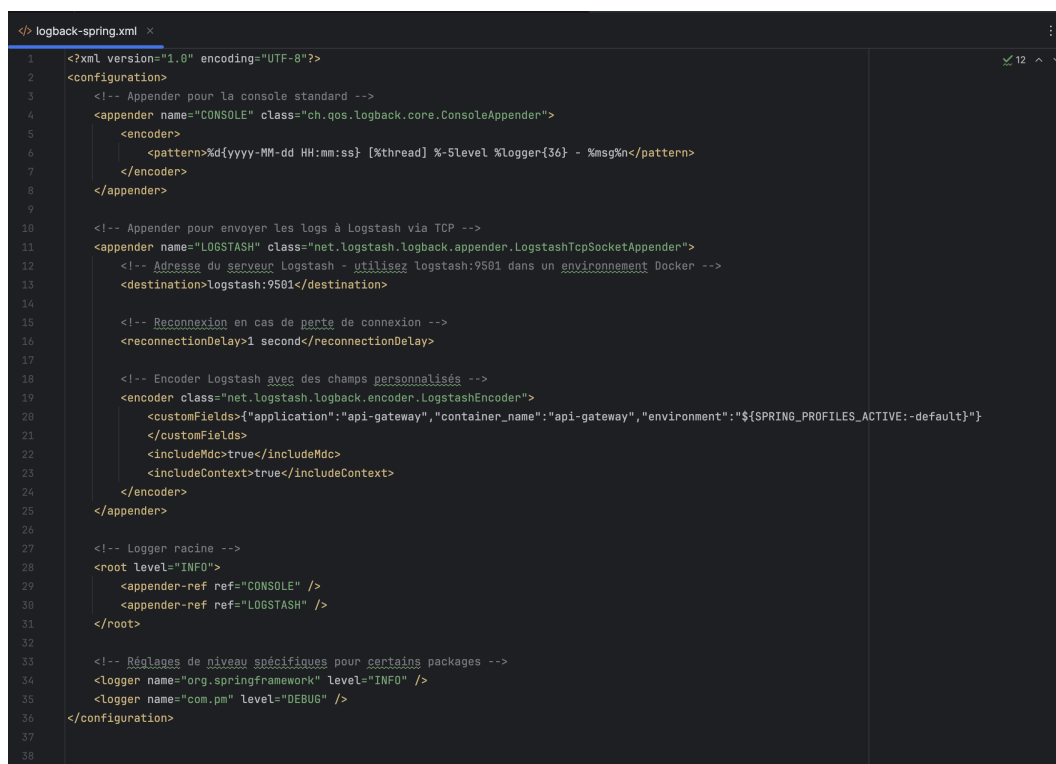
Figure 4.1: Main Repository

4.2 Log Routing Configuration

To enable centralized log collection across all microservices, the system implements a structured log forwarding mechanism. Each Spring Boot service is configured to transmit its logs in structured JSON format to a Logstash instance. Logstash then processes these logs and dispatches them to various persistence backends, including Elasticsearch, MongoDB, MySQL, and a local JSON file for benchmarking.

4.2.1 Logback Configuration in Microservices

Each microservice integrates a custom `logback-spring.xml` file to configure log output. This file leverages the `logstash-logback-encoder` library to encode logs in structured JSON and forward them over TCP to the Logstash container.



```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- Appender pour la console standard -->
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <!-- Appender pour envoyer les logs à Logstash via TCP -->
  <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <!-- Adresse du serveur Logstash - utilisez logstash:9501 dans un environnement Docker -->
    <destination>logstash:9501</destination>

    <!-- Reconnexion en cas de perte de connexion -->
    <reconnectionDelay>1 second</reconnectionDelay>

    <!-- Encoder Logstash avec des champs personnalisés -->
    <encoder class="net.logstash.logback.encoder.LogstashEncoder">
      <customFields>{"application":"api-gateway","container_name":"api-gateway","environment":"${SPRING_PROFILES_ACTIVE:-default}">
    </customFields>
    <includeMdc>true</includeMdc>
    <includeContext>true</includeContext>
    </encoder>
  </appender>

  <!-- Logger racine -->
  <root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="LOGSTASH" />
  </root>

  <!-- Réglages de niveau spécifiques pour certains packages -->
  <logger name="org.springframework" level="INFO" />
  <logger name="com.pm" level="DEBUG" />
</configuration>
```

Figure 4.2: logback spring file

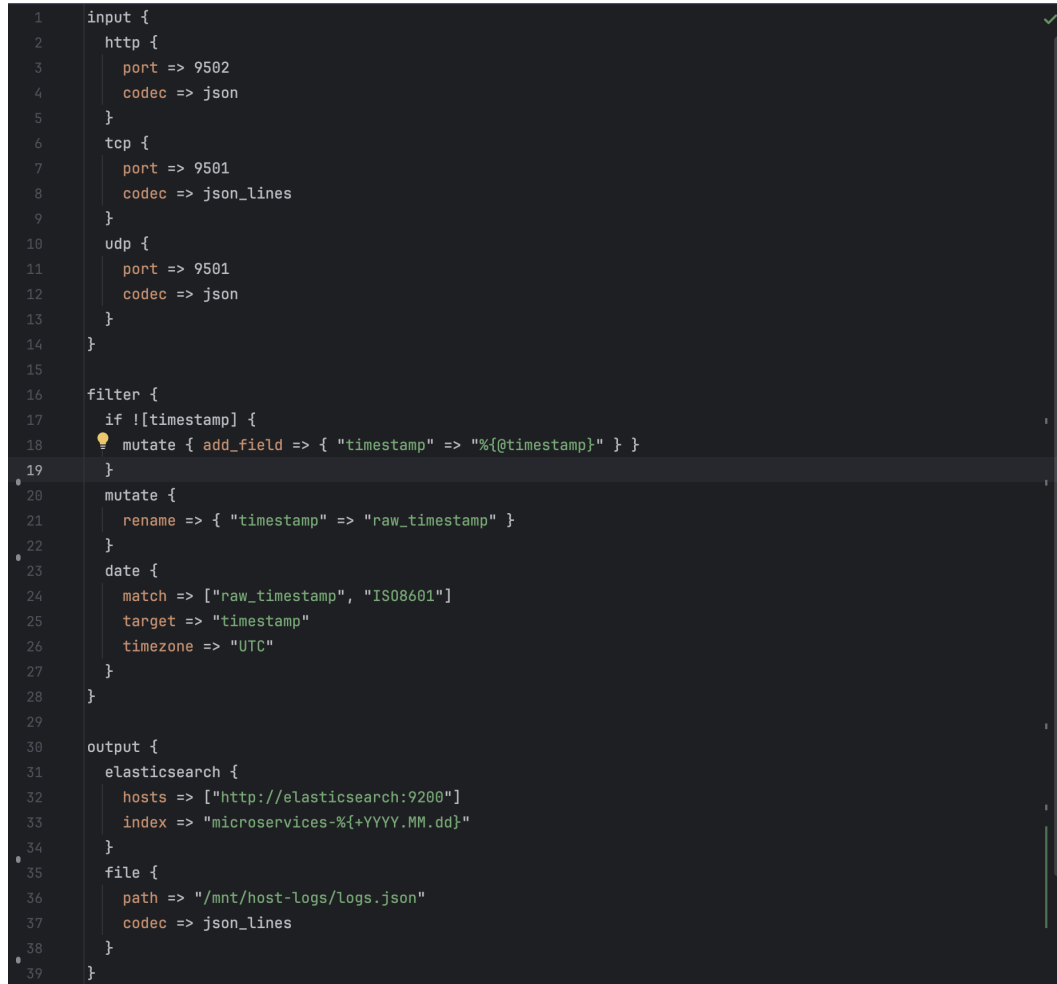
This configuration ensures that every emitted log is:

- Serialized in JSON format,
- Enriched with metadata (timestamp, level, thread, etc.),
- Sent asynchronously to the Logstash container at port 9500.

It ensures separation of logging concerns while making all logs consumable by the centralized processing pipeline. (Figure 4.2)

4.2.2 Logstash Pipeline Configuration

Logstash uses a dedicated pipeline configuration file (`logstash.conf`) to process incoming logs. This file defines inputs, filters, and outputs.

A screenshot of a code editor displaying a Logstash pipeline configuration. The configuration is written in YAML and includes three main sections: input, filter, and output. The input section defines three inputs: http (port 9502, codec json), tcp (port 9501, codec json_lines), and udp (port 9501, codec json). The filter section includes an if statement to add a timestamp field if it doesn't exist, followed by a mutate block to rename the timestamp to raw_timestamp, and a date block to parse the raw_timestamp using the ISO8601 format and set the timezone to UTC. The output section defines two outputs: elasticsearch (hosts http://elasticsearch:9200, index microservices-%{+YYYY.MM.dd}) and file (path /mnt/host-logs/logs.json, codec json_lines). The code is numbered from 1 to 39 on the left margin.

```
1 input {
2   http {
3     port => 9502
4     codec => json
5   }
6   tcp {
7     port => 9501
8     codec => json_lines
9   }
10  udp {
11    port => 9501
12    codec => json
13  }
14 }
15
16 filter {
17   if ![timestamp] {
18     mutate { add_field => { "timestamp" => "%{@timestamp}" } }
19   }
20   mutate {
21     rename => { "timestamp" => "raw_timestamp" }
22   }
23   date {
24     match => ["raw_timestamp", "ISO8601"]
25     target => "timestamp"
26     timezone => "UTC"
27   }
28 }
29
30 output {
31   elasticsearch {
32     hosts => ["http://elasticsearch:9200"]
33     index => "microservices-%{+YYYY.MM.dd}"
34   }
35   file {
36     path => "/mnt/host-logs/logs.json"
37     codec => json_lines
38   }
39 }
```

Figure 4.3: Logstash Pipeline

The configuration accomplishes the following:

- **Input:** Receives JSON-formatted logs over TCP.
- **Filter:** Ensures a consistent timestamp field using the `date` filter plugin.
- **Output:**
 - Stores logs in Elasticsearch for visualization in Kibana.
 - Writes logs to a local JSON file for offline benchmarking.

This pipeline provides a flexible, multi-target log processing solution, allowing direct comparison between SQL and NoSQL storage engines under real-time ingestion conditions. (Figure 4.3)

4.3 CI/CD Pipeline Implementation with Jenkins

To automate the build and deployment processes across services and infrastructure, the project integrates a continuous integration and continuous deployment (CI/CD) pipeline using Jenkins. Each microservice and system component is managed independently through a dedicated pipeline, enabling clean, reproducible builds and standardized deployment logic.

4.3.1 Pipeline Architecture

Each service repository (microservices, infrastructure, logging stack) includes a specific Jenkins pipeline. These pipelines are centrally managed through a local Jenkins server running on an ARM-based development machine. The Jenkins dashboard (Figure 4.4) shows the build status and history for each managed project.

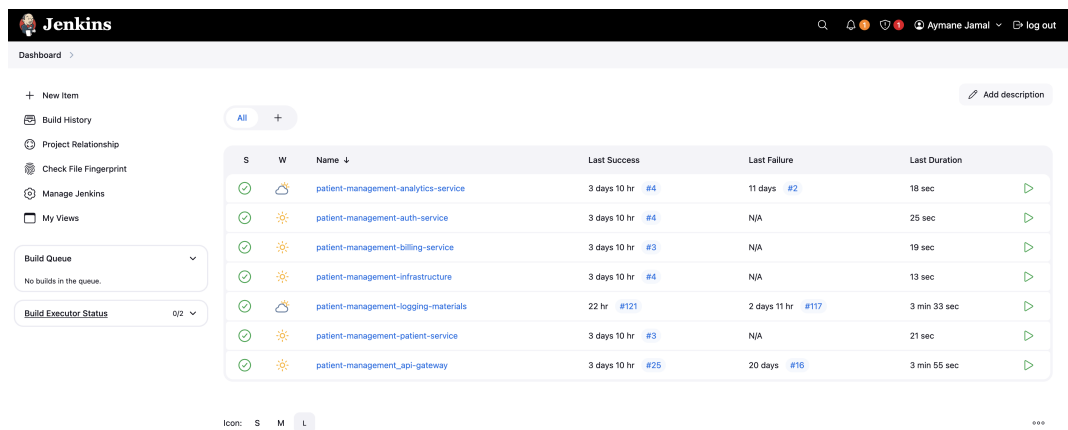


Figure 4.4: Jenkins Dashboard — Pipeline status for each service

4.3.2 Pipeline Responsibilities

Each Jenkinsfile defines the following stages:

- **Clone Repository:** Clones the appropriate GitHub repository.
- **Clean Up:** Stops any running containers and removes orphans and unused volumes.
- **Build Image:** Executes a Docker build for the corresponding service.
- **Run Container:** Starts the service in a detached mode using Docker Compose.
- **Health Checks:** Ensures the service is reachable and responsive.

4.3.3 Example: API Gateway Pipeline Breakdown

The pipeline for the `api-gateway` service follows a clear, modular structure that includes the following stages:

- **Clone Repository:** Jenkins pulls the latest version of the source code from the corresponding GitHub repository. Authentication is handled via a preconfigured credential ID to enable secure access.
- **Clean Environment:** Before rebuilding, the pipeline shuts down any existing containers related to the service and removes orphan containers. This prevents conflicts or resource leaks from previous runs.
- **Build and Deploy:** Using Docker Compose, Jenkins builds the service's Docker image and launches it in detached mode. This step ensures that the application is deployed with the latest code and dependencies.
- **Health Verification (Optional):** While not explicitly present in all Jenkinsfiles, some pipelines include logic to perform health checks or verify port bindings to confirm successful container startup.
- **Log Artifact Management (Optional):** For the logging-related pipeline, the Jenkinsfile may include stages for storing logs or output files (e.g., benchmark results) for later use or dashboard integration.

Each Jenkinsfile across the services follows a similar pattern, tailored to the specific service context and dependencies. This modular design promotes maintainability and reuse, while ensuring consistent build and deployment logic across the entire microservice ecosystem.

4.3.4 Build Monitoring and Management

Each pipeline run is recorded and timestamped in Jenkins, allowing developers to monitor:

- **Build success/failure history**
- **Build duration**
- **Error traceability**

The automation ensures that every push or manual trigger initiates a fresh build and deploy cycle, ensuring reproducibility and minimal manual intervention. This CI/CD strategy enhances team productivity, reduces integration errors, and enforces consistency across environments.

4.4 Container Deployment and Verification

Once each Jenkins pipeline successfully completes the build and run stages, all services and infrastructure components are deployed as Docker containers. To validate the health and availability of the system, Docker Desktop is used to visually inspect the status of running containers, resource consumption, and port mappings.

4.4.1 Deployment Overview

Figure 4.5 shows a snapshot of Docker Desktop listing all running containers for this project. Each microservice (e.g., `api-gateway`, `auth-service`, `patient-service`) is deployed as an isolated container. Similarly, infrastructure services such as PostgreSQL databases and Kafka, as well as logging stack services (Logstash, Elasticsearch, MongoDB, MySQL, Kibana), are also operational in their respective containers.

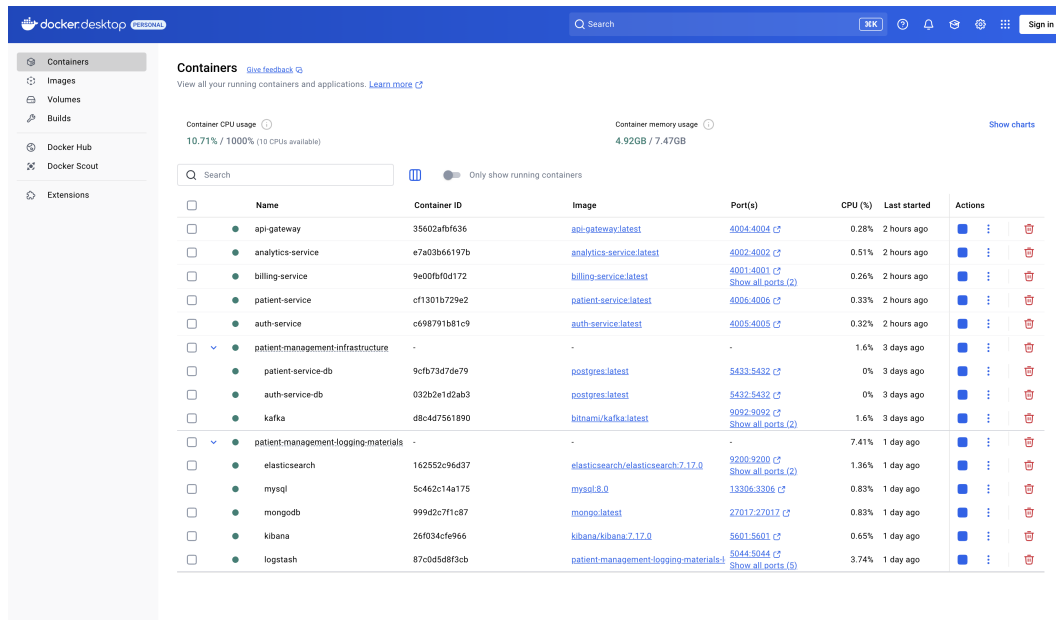


Figure 4.5: Docker Desktop – Container view confirming successful deployment

4.4.2 Validation Points

The Docker interface provides multiple indicators of successful deployment:

- **Container Health:** All containers are shown as running with active status dots.
- **Ports and Bindings:** Each service exposes ports used for HTTP requests or inter-service communication (e.g., 4004:4004 for `api-gateway`).
- **CPU and Memory Usage:** Real-time metrics indicate resource efficiency. For instance, `logstash` and `elasticsearch` show higher CPU usage, which is expected due to their indexing and transformation workloads.

- **Startup Timestamp:** Services show recent start times, confirming automatic restarts triggered by the Jenkins pipelines.

This visual inspection serves as a secondary validation layer alongside Jenkins build logs, confirming that the system is deployed and ready for interaction, benchmarking, and log visualization.

4.5 Benchmarking Script: Log Comparison

To assess the performance and efficiency of different storage backends, a custom Python script named `log_comparison.py` is used. This script is executed locally and benchmarks the log ingestion and query performance of both MongoDB and MySQL databases. It is designed to operate outside the containerized system, interacting with the log data produced and stored by the logging stack.

4.5.1 Objectives

The primary functions of the benchmarking script are:

- Simulate log generation by sending structured log entries to Logstash via HTTP or TCP.
- Measure and record the time required for:
 - Log insertion into MongoDB and MySQL.
 - Querying recently inserted logs from both databases.
- Record CPU and memory usage during insertion and querying for each database.
- Export the results in both human-readable text and graphical formats (PNG).

4.5.2 Implementation Details

The script is implemented using the following Python libraries:

- **pymongo:** To interface with MongoDB.
- **mysql-connector-python:** To interact with the MySQL database.
- **requests:** For sending log entries to Logstash over HTTP.
- **docker:** To collect container-specific resource usage metrics.
- **matplotlib:** To visualize benchmark results (e.g., CPU usage, memory usage, insertion time, and query time).
- **tabulate:** To generate well-structured benchmark tables in the terminal.

4.5.3 Execution and Outputs

The script starts by sending a batch of JSON-formatted logs to Logstash. Once Logstash routes these logs to both MongoDB and MySQL, the script measures and logs:

- The average time to insert all logs into each database.
- The time to retrieve logs using basic query conditions.
- CPU and memory usage by containers during the benchmark process.

The output is stored in two forms:

- A timestamped text file containing detailed numerical results.
- Four PNG charts:
 - CPU usage by operation and database.
 - Memory usage by operation and database.
 - Time required for data insertion.
 - Time required for query execution.

These visual and textual outputs provide a comprehensive overview of system performance, enabling a detailed comparison of SQL versus NoSQL performance under the same log load conditions.

4.5.4 Environment Preparation

Before executing the script, users must:

- Create a Python virtual environment compatible with macOS ARM64 architecture.
- Install dependencies using:

```
pip install -r requirements.txt
```

- Start the local directories `host-logs` and `local-log-comparison`, as they store outputs and contain the script logic.

This benchmark script plays a key role in validating the system's ability to handle log ingestion workloads and quantifies trade-offs between the chosen backends.

Chapter 5

Results and Discussion

This chapter presents the experimental results obtained through the benchmarking script and the real-time log visualization conducted via Kibana. The aim is to evaluate the performance of MongoDB and MySQL for log persistence in terms of ingestion speed, query efficiency, and resource usage, and to reflect on the system's observability using visual dashboards.

5.1 Benchmarking Results

After launching the script `log_comparison.py`, a synthetic batch of 12,745 log entries was generated and routed to Logstash. These logs were then simultaneously forwarded to MongoDB and MySQL. The script collected metrics during both the insertion and query phases.

The benchmarking procedure produced:

- A timestamped textual report.
- Graphs visualizing time, memory, and CPU usage.

5.1.1 Raw Benchmark Output

The textual output of the benchmark was saved in a file named `benchmark_results_2025-04-23_16-03-00.t` with the summarized content:

=== INSERTION RESULTS ===

Storage	Log Count	Insert Time	CPU	Memory
MongoDB	12745	0.60s	4.55%	271.25MB
MySQL	12745	1.60s	5.9%	523.03MB

=== QUERY RESULTS ===

Storage	Query Time	CPU	Memory
MongoDB	0.11s	0.61%	271.16MB
MySQL	0.04s	0.23%	527.38MB

5.1.2 Visual Comparison of Results

- **Insert Time:** MongoDB performed inserts more than twice as fast as MySQL.
- **Query Time:** MySQL executed queries slightly faster, though the difference is minimal.
- **CPU Usage:** MongoDB was more efficient under write loads, while MySQL used less CPU during reads.
- **Memory Consumption:** MySQL consistently used nearly twice as much memory as MongoDB.

Figures 5.1, 5.2, 5.3, and 5.4 illustrate these results.

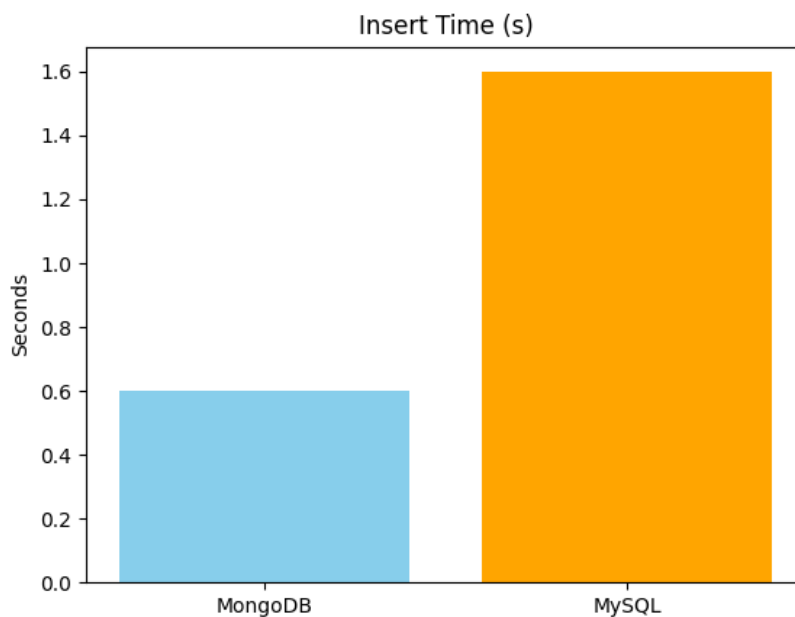


Figure 5.1: Insert Time Comparison

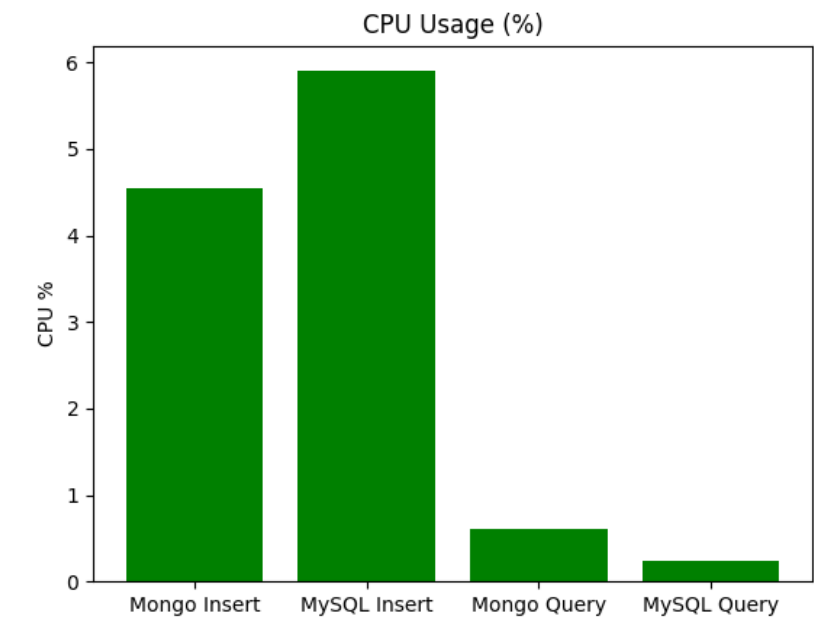


Figure 5.2: CPU Usage During Insert/Query

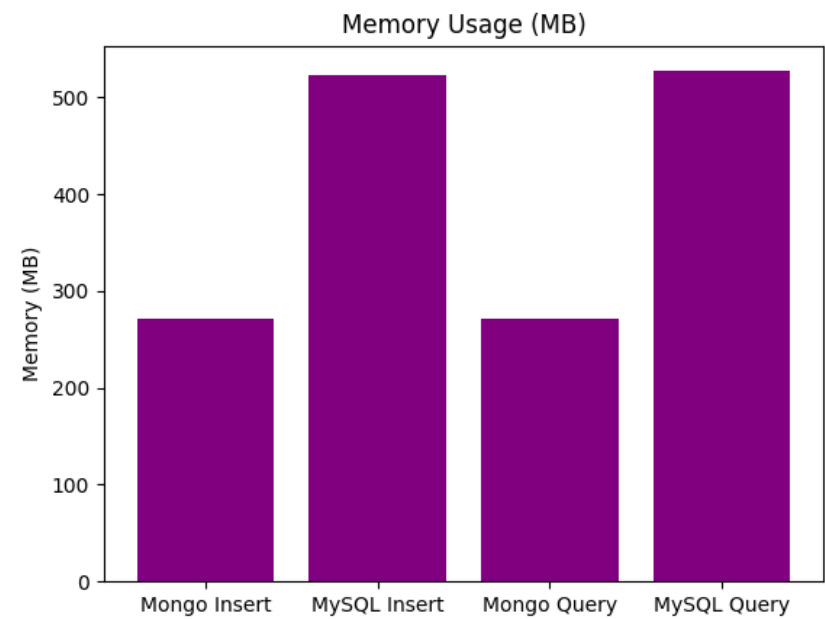


Figure 5.3: Memory Usage

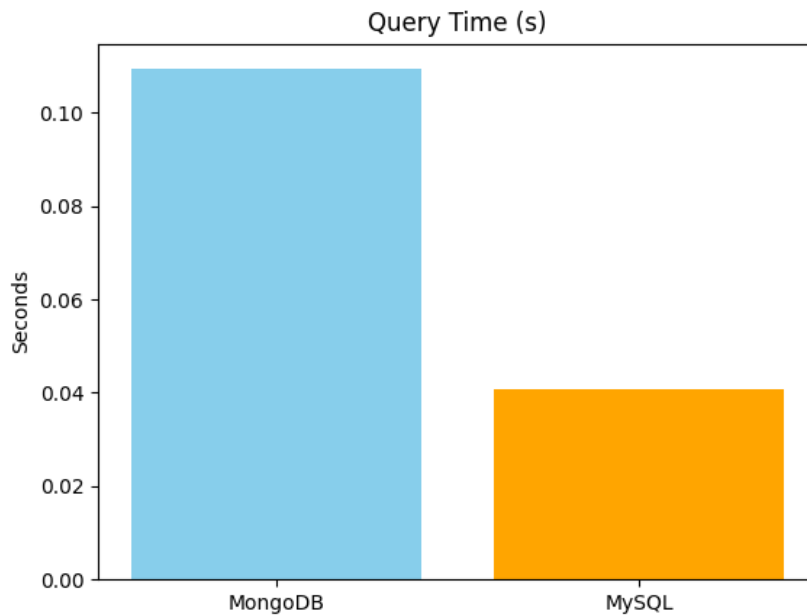


Figure 5.4: Query Time Comparison

5.2 Log Visualization in Kibana

To evaluate the system's real-time observability, logs forwarded to Elasticsearch were explored using the Kibana dashboard hosted at <http://localhost:5601>.

5.2.1 Visualization Workflow

1. Create a new index pattern: `microservices-*`
2. Select the time field: `@timestamp`
3. Navigate to **Visualize Library** → **Lens**
4. Set up a horizontal bar chart:
 - X-axis: top values of `level.keyword`
 - Y-axis: count of records
5. Save and add the visualization to a custom dashboard.

5.2.2 Dashboard Output

Figure 5.5 shows a completed dashboard displaying the frequency distribution of log levels (INFO, ERROR, WARN) across services.

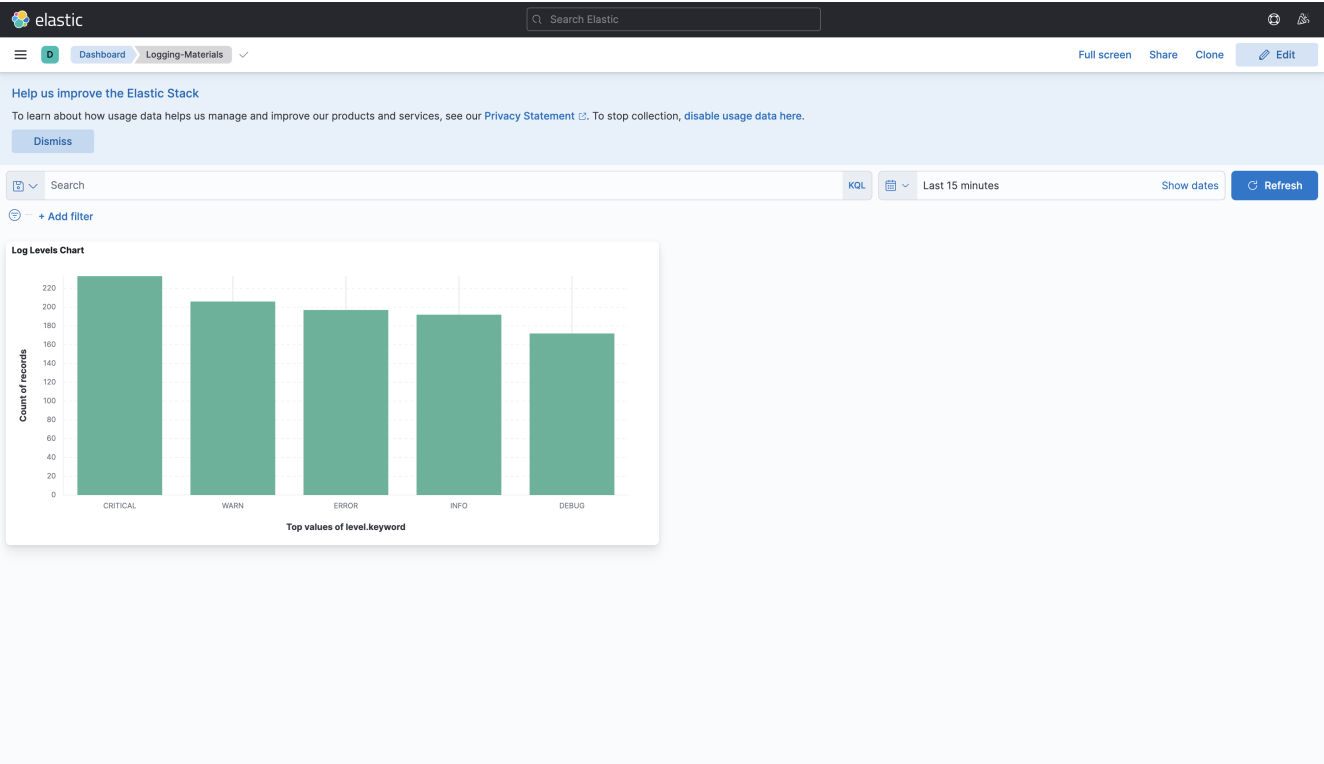


Figure 5.5: Kibana Dashboard — Log Level Distribution

5.3 Discussion

The results confirm that MongoDB provides a better trade-off in log persistence scenarios where write performance and memory efficiency are key. MySQL remains a viable option for faster reads, but comes at the cost of higher resource consumption.

Kibana dashboards complement this benchmark by offering real-time insights into system behavior, validating the relevance of structured logging and centralized visualization in production-grade DevOps workflows.

5.4 Summary of Key Findings

The experiments demonstrate the technical viability and efficiency of centralized log persistence in a microservices environment, especially when deployed on ARM-based infrastructure.

- MongoDB outperformed MySQL in insertion speed and memory usage, making it a strong candidate for high-throughput log ingestion scenarios.
- MySQL demonstrated slightly faster query performance, but with significantly higher memory consumption.
- Kibana effectively visualized log-level distributions, supporting real-time system monitoring.
- The system architecture proved modular, reproducible, and adaptable to real-world CI/CD practices using Docker and Jenkins.

These results confirm that NoSQL databases are well-suited for DevOps log persistence pipelines when write efficiency and lightweight footprint are priorities.

5.5 Limitations

Despite promising results, the system and experimental setup have some limitations:

- Benchmarks were conducted on a single local machine (MacBook M4), which may not fully reflect performance in distributed or production environments.
- Elasticsearch performance was not directly benchmarked due to its concurrent use in the Kibana dashboard.
- Only two storage engines (MongoDB and MySQL) were compared; further alternatives like PostgreSQL, InfluxDB, or Cassandra remain unexplored.
- The simulated logs are generated synthetically and may not perfectly replicate the variety and unpredictability of production traffic.

5.6 Future Work

Several extensions can be considered to enrich this study:

- Benchmarking Elasticsearch's indexing and query capabilities under different workloads.
- Incorporating horizontal scaling to observe behavior in a distributed multi-node environment.
- Testing other NoSQL systems (e.g., Cassandra, DynamoDB) and modern SQL databases (e.g., TimescaleDB, PostgreSQL JSONB).
- Automating end-to-end deployment using Kubernetes or Terraform for cloud-native infrastructure.
- Integrating alerting and anomaly detection into the Kibana dashboards for proactive monitoring.

These directions can extend the relevance and applicability of the current system to enterprise-scale DevOps infrastructures.

Chapter 6

Conclusion

This project explored the implementation, deployment, and evaluation of a microservice-based log persistence pipeline designed to operate in modern DevOps environments. With a focus on containerization, observability, and benchmarking, the solution was deployed on ARM-compatible infrastructure (MacBook M4) using Docker, Jenkins, and lightweight NoSQL/SQL databases.

The architecture separated concerns across multiple Git repositories and Docker containers, ensuring modularity and independent scalability. Log forwarding was achieved using a unified logging layer built with Logstash, while Elasticsearch and Kibana provided real-time visualization capabilities.

A major contribution of this project was the development of a custom Python benchmarking script, designed to evaluate the performance of MongoDB and MySQL under a simulated logging workload. By analyzing insertion time, query response time, CPU usage, and memory consumption, the system provided empirical insight into the trade-offs between SQL and NoSQL solutions in log-heavy applications.

Furthermore, the integration of Jenkins pipelines enabled full CI/CD automation. Each microservice was independently built, deployed, and monitored via automated pipelines that ensured reproducibility and reduced manual intervention during testing.

From a monitoring perspective, Kibana dashboards were successfully used to visualize log levels and distribution patterns. These dashboards enhanced the observability of the system and simulated a production-ready logging environment.

Key Achievements

- Implemented and deployed a multi-repository, containerized architecture for DevOps log persistence.
- Benchmarked MongoDB and MySQL for ingestion and query performance using real metrics (CPU, memory, time).
- Enabled real-time observability via Kibana integrated with Logstash and Elasticsearch.
- Applied DevOps best practices by automating service deployment with Jenkins and Docker.

Broader Impact

This project offers a practical blueprint for DevOps engineers and researchers seeking to evaluate log storage engines under realistic microservice workloads. The modular and reproducible design enables future integration of alternative storage backends, visualization tools, and cloud-native deployment stacks (e.g., Kubernetes, AWS).

Final Thoughts

The combination of real-time log routing, distributed service architecture, and structured benchmarking provides a powerful foundation for future work in scalable log management. This system is both lightweight and extensible, capable of supporting operational observability in production-grade environments with minimal overhead.