

LogVista: An Intelligent Web Log Analyzer

A Full Stack Tool for Geolocation-Based,
Device-Aware, Real-Time Log Parsing and Visualization using
Spring Boot and Chart.js

A PROJECT REPORT

Prepared By:

Gaurav Maurya

MCA, Department of Computer Science

University of Delhi

Under the Guidance of:

Mr. Tarun Mishra

Project Mentor

National Informatics Centre (NIC), New Delhi

Internship Duration: Jan 2025 – June 2025

Table of Contents

Introduction.....	3
Problem Statement.....	4
Problem Analysis.....	5
Solution Approach.....	9
Technology Stack.....	11
Data Processing Flow.....	13
System Architecture.....	14
API Endpoints.....	14
Implementation.....	18
References.....	27

Introduction

In today's digital landscape, modern web applications continuously generate a **large volume of log data**. These logs are essential for maintaining and improving systems as they help in **monitoring application performance, tracking user activity, resolving technical issues, and ensuring security**. However, raw log files are often **unstructured** and lack meaningful context—such as the **geographical location** of the user or the **type of device** used—which makes them difficult to analyze directly.

To address this challenge, this project focuses on developing a **log analysis and processing tool** that converts raw log entries into **structured and enriched data**. The tool is designed to handle log entries in a specific format, including a **timestamp, email ID, IP address, and user-agent string**. Each log is:

1. **Validated**
2. **Parsed** into structured fields
3. **Enriched** with location and device-related information

All processed logs are stored in a **relational database**, making them ready for further analysis and visualization. The project also provides a set of **RESTful APIs** that allow external systems to retrieve key insights such as:

1. Requests by **city** or **country**
2. **Device** and **operating system** usage
3. **Time-based** traffic patterns

By organizing log data in a clean, structured manner, this system lays the foundation for building **dashboards, real-time monitoring tools, or business intelligence platforms**. It simplifies the log analysis process and makes the data more **actionable** and **meaningful**.

Problem Statement

Log files are a critical component of any software system. They assist developers and administrators in **tracking errors**, **monitoring user activity**, and **optimizing performance**. These logs typically contain important elements such as **timestamps**, **IP addresses**, **email IDs**, and **user-agent information**.

However, in real-world scenarios, logs are often **unstructured**, **inconsistent**, or **incomplete**. Some entries may include:

1. Invalid **email formats**
2. Missing or incorrect **IP addresses**
3. Corrupted or unreadable **user-agent strings**

These issues make **manual analysis** time-consuming, error-prone, and impractical—especially when dealing with **large volumes of data**.

In addition, modern organizations require **deeper insights** from logs, such as:

1. The **user's geographical location**
2. Their **device type** and **operating system**
3. **Peak activity hours** and **usage trends**

Unfortunately, **raw logs** do not provide this information in a readily usable format.

To overcome these challenges, there is a need for an **automated system** that can:

1. **Clean and validate** raw log data
2. **Extract and enrich** key information
3. **Store** the output in a **structured format**

Such a system would allow logs to be visualized through **charts**, **reports**, or **dashboards** to support better understanding and decision-making.

This project aims to build such a tool to simplify and automate log analysis in a **smart**, **efficient**, and **user-friendly** way.

Problem Analysis

In today's fast-paced digital world, applications and backend systems constantly generate massive volumes of log data. These logs are extremely valuable — they help engineers and analysts **monitor user activity**, **detect bugs**, **spot security threats**, and **understand system performance**.

In this project, we assume that each log entry follows a structured format like:

```
<timestamp>~<email>~<IP address>~<user-agent>
```

This consistent structure simplifies parsing and validation during processing.

However, in real-world scenarios, the log files are **rarely clean or uniform**. They often contain **missing values**, **inconsistent formats**, or **corrupted lines**, which makes them **unsuitable for direct analysis**.

To address this, it is essential to build a preprocessing layer that can filter out malformed entries, extract meaningful data from valid ones, and prepare them for further enrichment. Without this initial cleaning and validation step, any analytics or visualization derived from the data may be misleading or incomplete.

Moreover, as log data grows rapidly over time, scalability and efficiency also become key challenges. A reliable solution must be able to process large files in real-time or near real-time without compromising on accuracy.

Key Challenges with Raw Log Data

1. Malformed or Incomplete Entries

Many log lines are either partially written or don't follow the expected format. For example, some may have invalid emails or missing IP addresses. This kind of noise in the data reduces the accuracy of any analysis performed.

```
2024-05-01 10:15:23~user@example.com~192.168.1.10~Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/113.0.0.0
2024-05-01 10:16:01~invalid-email@~192.168.1.15~Mozilla/5.0 (Linux; Android 9; SM-J810G) Chrome/90.0.4430.210
2024-05-01 10:16:45~admin@example.com~INVALID_IP~Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Safari/537.36
MISSING_FIELDS_HERE
2024-05-01 10:18:12~john.doe@gmail.com~203.0.113.42~Mozilla/5.0 (iPhone; CPU iPhone OS 14_0 like Mac OS X) Safari/604.1
```

2. Lack of Automatic Validation

At large scale, it is not feasible to manually check each log for correct formatting. That's why **regex-based validation** is essential — it allows the system to automatically identify and discard malformed entries.

3. No Built-in Location Data

IP addresses alone don't tell us much. To gain meaningful insights, we need to **map IPs to real-world locations** (like city and country). This requires **external databases**, such as **GeoLite2**, to enrich the log data.

4. User-Agent Strings Are Hard to Interpret

User-agent values are long and complex strings that describe the user's device, browser, and OS. For example:

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36

- Trying to extract meaningful information from this manually is difficult. We need tools like **UserAgentUtils** to accurately detect the **device type** (mobile/desktop), **operating system**, and **browser**.

5. No Visual Summary or Real-Time Monitoring

Plain text log files offer no visual feedback. Without charts or dashboards, it's hard for teams to quickly understand trends — such as **which countries are generating the most traffic**, or **what times of day see the highest activity**.

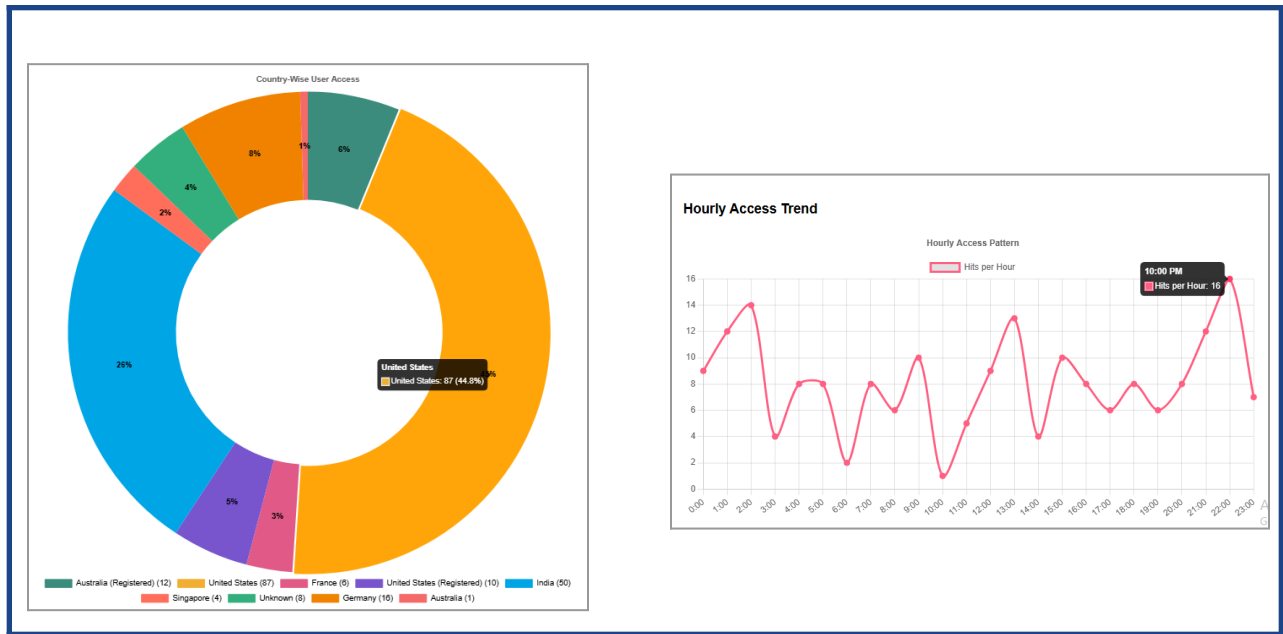


Figure: Absence of visual summaries in plain log files makes it difficult to analyze trends. The doughnut chart (left) demonstrates how country-wise traffic distribution can aid decision-making, while the line chart (right) highlights how hourly access trends improve temporal visibility.

6. Missing Admin Dashboard or Alert System

Without a centralized dashboard, valuable patterns in traffic or errors often go unnoticed. Insights like **peak hours**, **popular browsers**, or **OS usage** are buried in plain text — reducing their usability in decision-making.

Project Goals in Response to These Challenges

To solve the above problems, the project aims to build a fully automated pipeline that:

- **Reads raw log files line-by-line**, applies **regex validation**, and filters out malformed entries.
- **Enriches each log** by adding location data using **IP lookup** and extracting device/browser info from **user-agent strings**.
- **Stores the cleaned and structured logs** in a **MySQL database** for easy access and querying.
- **Provides RESTful APIs** to fetch key metrics such as **top countries**, **OS usage**, and **hourly traffic trends**.
- **Displays the results on an interactive dashboard** built with **Chart.js** and **JavaScript**, making it easier for users to visualize trends and perform real-time monitoring.

Solution Approach

To solve the challenges of malformed, inconsistent, and unstructured log data, this project follows a systematic, multi-stage processing pipeline — focusing on both data quality and actionable visualization. The approach ensures that the raw logs are not only cleaned and standardized but also enriched with external intelligence before being transformed into visual insights.

The pipeline begins with a file upload mechanism that accepts **.txt** log files via an HTTP endpoint. The uploaded file is read line by line using a custom utility that applies a **strict regex pattern** to validate each entry. This ensures that only well-formed logs in the expected format (**timestamp~email~IP~user-agent**) are allowed to proceed. Lines that are incomplete or corrupted are automatically skipped, preventing the injection of noise into the data stream.

Once validated, the clean entries are **parsed and mapped into a well-defined model class**. This model captures four primary components:

1. Timestamp
2. Email
3. IP Address
4. User-Agent

This structured model becomes the foundation for the **data enrichment phase**, which significantly enhances the analytical value of each log entry. Two key enrichment steps are performed:

1. Geolocation Enrichment:

The IP address is analyzed using the **GeoLite2** database to extract the user's **country and city**. This enables meaningful geographic segmentation in later visualizations.

2. User-Agent Parsing:

The raw user-agent string is parsed using the **UserAgentUtils** library to derive the **device type** (e.g., Mobile, Desktop), **operating system**, and **OS version**. These attributes are added to the log model to support device-based usage analysis.

After enrichment, the complete and contextualized log entry is persisted in a **MySQL relational database** using JPA. This not only supports efficient querying but also provides a solid foundation for creating analytics endpoints.

The backend exposes a set of **RESTful APIs** that aggregate this data for frontend consumption. These APIs support country-wise distribution, city-wise breakdowns, OS usage stats, and hourly traffic trends — enabling a comprehensive analytics dashboard.

On the frontend, these APIs are consumed and visualized using dynamic, interactive charts powered by **Chart.js**. This dashboard enables users to explore patterns in traffic, device usage, and geographic distribution through an intuitive UI, transforming plain log data into actionable insight.

This approach ensures:

- Clean and reliable data from the beginning
- Scalable enrichment using third-party intelligence
- Efficient storage for long-term analysis
- A clear separation of concerns between ingestion, processing, storage, and visualization
- Real-time insight generation with interactive visual feedback

Technology Stack

This project is built on a carefully selected set of technologies across backend, database, frontend, and development tooling. Each component plays a distinct role in ensuring that raw log data can be effectively processed, enriched, stored, and visualized through an interactive dashboard.

1. Backend Stack

At the core of the system is a robust Java-based backend that performs parsing, validation, enrichment, and API provisioning.

The application is built using **Java (JDK 17)**, offering strong type-safety, performance, and mature library support. To simplify development and accelerate deployment, the backend uses **Spring Boot**, a modern framework that brings built-in support for RESTful API creation, dependency injection, and auto-configuration via an embedded Tomcat server. This allows the application to be packaged and run as a standalone executable with minimal configuration overhead.

Validated and enriched log data is stored in a **MySQL** database. MySQL is chosen for its performance, reliability, and seamless compatibility with Java-based systems. It integrates smoothly with **Spring Data JPA**, which acts as an abstraction layer over JDBC, enabling developers to perform database operations using intuitive repository interfaces — without writing raw SQL queries.

To maintain the quality of incoming data, **regular expressions (Regex)** are used to validate each log line format. Only entries that match the expected structure (timestamp, email, IP, and user-agent) proceed to enrichment and storage.

Two key libraries are used for enrichment:

1. GeoLite2 (by MaxMind):

Provides precise geolocation data such as country and city by analyzing IP addresses. The integration is done using the [geoip2](#) Java library.

2. UserAgentUtils:

Parses user-agent strings to extract user device characteristics like operating system, version, and device type (e.g., desktop, mobile).

Together, these backend technologies ensure that the system is not only scalable but also capable of generating high-quality metadata from raw inputs.

2. Frontend Stack

The frontend is responsible for visualizing enriched data in a clear and meaningful way.

While the user interface is built using standard technologies (HTML, CSS, and JavaScript), the real value lies in the use of **Chart.js**, a powerful JavaScript library for data visualization. It allows the rendering of responsive and aesthetically appealing charts such as:

- ❖ Doughnut charts showing traffic distribution by country
- ❖ Bar graphs breaking down user counts by city and operating system version
- ❖ Pie charts for operating system distribution
- ❖ Line graphs showing hourly traffic trends

These charts form the backbone of the dashboard and make it easier to identify patterns in the data.

** For visual examples of these charts, refer to the [Frontend Dashboard & Data Visualization](#) section in the Implementation chapter.*

3. Development Tools

To streamline development, testing, and deployment, the following tools are used:

- **IntelliJ IDEA / Eclipse:**

Powerful IDEs that support Java development with features like smart code completion, integrated build systems, and advanced debugging.

- **Postman / Swagger UI:**

Used extensively to test and validate REST APIs during development. These tools help verify API responses, status codes, and request payloads.

- **Git & GitHub:**

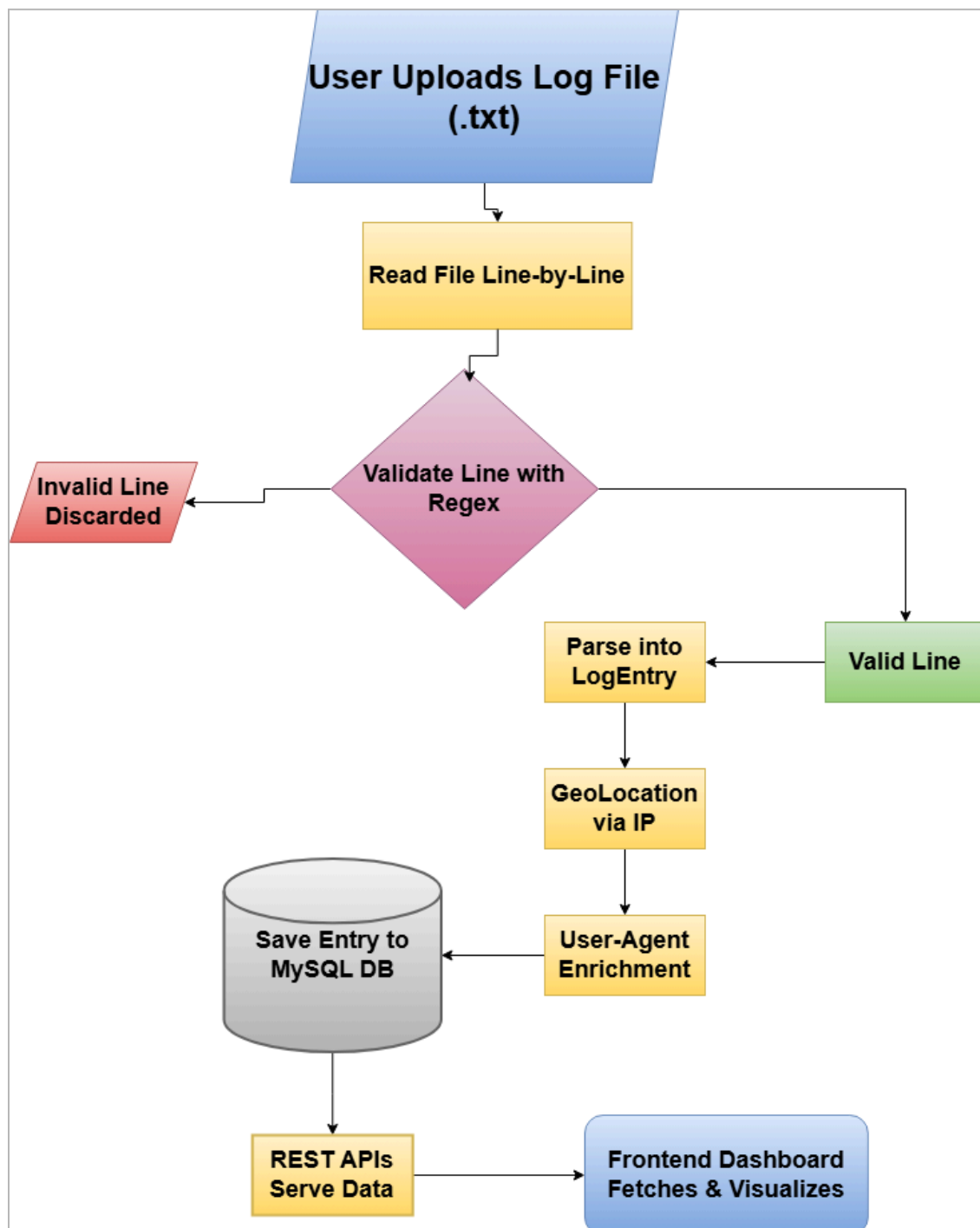
For version control, collaboration, and codebase management. GitHub also serves as a centralized repository for issue tracking, pull requests, and documentation.

** Postman screenshots showing sample API responses should be added in the [API Endpoints](#) section.*

Data Processing Flow

This section illustrates the step-by-step journey of a log file from upload to its transformation into structured, enriched data ready for visualization. The system follows a linear, rule-based pipeline to ensure only meaningful and valid data is stored and visualized. The primary stages include file intake, validation, enrichment, persistence, and data delivery.

High-Level Workflow

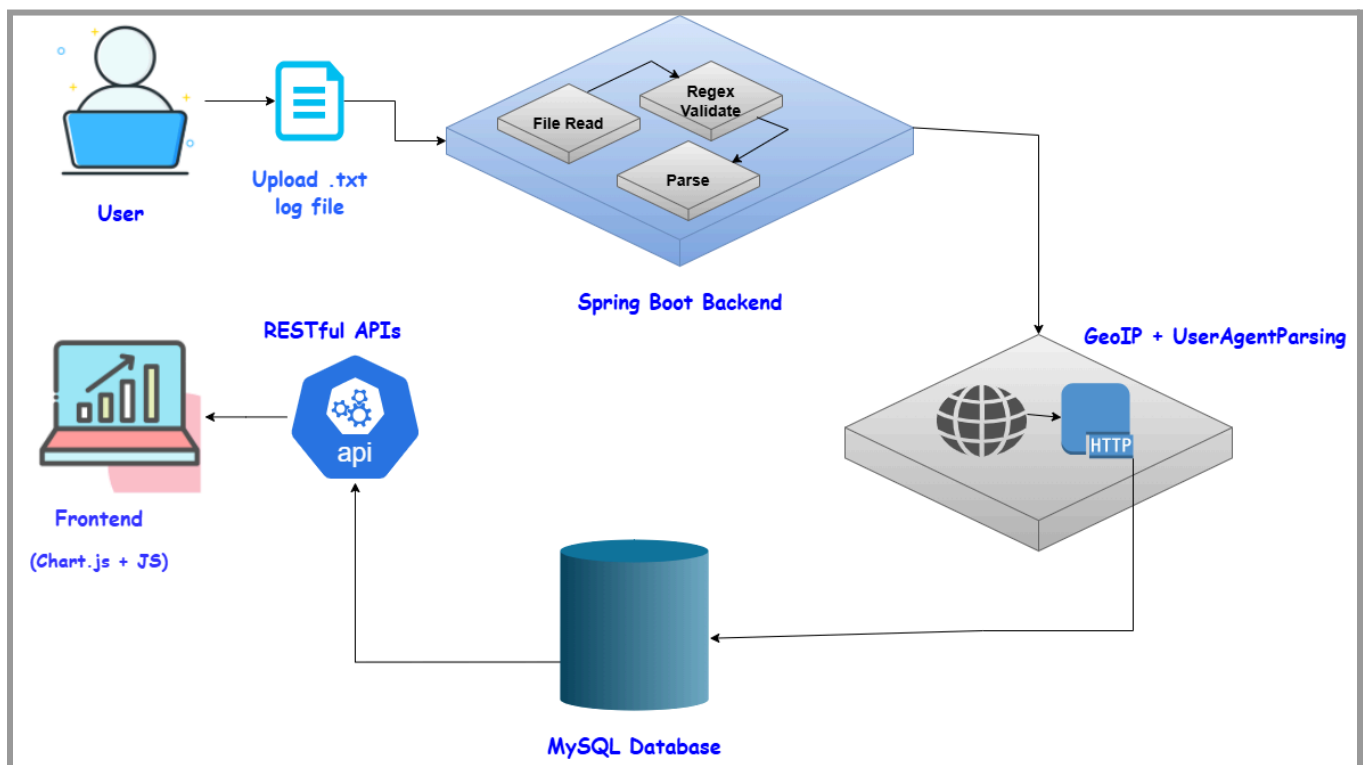


System Architecture

The **System Architecture** outlines the high-level design of all components involved in the log analytics platform. It presents a modular overview of how different layers—frontend, backend, database, and external services—interact to achieve end-to-end log processing, enrichment, storage, and visualization.

The architecture follows a layered approach:

- The **Frontend Layer** consists of a responsive web dashboard that fetches enriched log data via REST APIs and presents them through dynamic charts using Chart.js.
- The **Backend Layer** is implemented using Java and Spring Boot. It handles file ingestion, log parsing, data validation, enrichment (GeoIP + User-Agent parsing), and data storage.
- The **Database Layer** utilizes MySQL to persist structured and enriched log entries.
- **External Integration** includes the MaxMind GeoLite2 database for IP geolocation and the UserAgentUtils library for device detection.
- All components are loosely coupled and communicate via clearly defined interfaces, ensuring scalability and maintainability.



API Endpoints

RESTful API Design for Log Analytics Retrieval

The backend of the system exposes a set of **RESTful API** endpoints that serve as the communication bridge between the data processing layer and the frontend dashboard. These APIs provide access to various types of aggregated log analytics including location distribution, system usage trends, and temporal access patterns. Designed with simplicity and scalability in mind, the endpoints return **JSON-formatted responses** optimized for fast rendering in the frontend interface.

- All APIs follow RESTful principles and use the **GET** method.
- Responses are in **JSON format**, easily consumable by modern JavaScript frameworks.
- Endpoints are designed for data visualization, optimized for low-latency and high readability.
- The frontend (using Chart.js) consumes these APIs to dynamically render interactive graphs.
- Backend services are built using **Spring Boot**, and data is fetched from a **MySQL** database using efficient SQL queries.

API Endpoint Table

Endpoint	Method	Description	Response Type
/api/country-stats	GET	Aggregated log counts by country (geographical distribution)	List<LabelCountDTO>
/api/city-stats	GET	Detailed log counts by city and country	List<CityCountryDTO>
/api/os-stats	GET	Aggregated log counts by operating system	List<LabelCountDTO>
/api/os-version-stats	GET	Aggregated log counts by OS version	List<OSVersionStatsDTO>
/api/hourly-hits	GET	Log entry counts by each hour of the day (traffic trends)	List<HourlyHitsDTO>

Sample Response Schemas

LabelCountDTO

– Used in country and OS stats:

```
{
  "label": "Windows",
  "count": 1200
}
```

CityCountryDTO

– Used in city-stats:

```
{
  "city": "New York",
  "country": "United States",
  "count": 450
}
```

OSVersionStatsDTO

– Used in os-version-stats:

```
{
  "osVersion": "Android 12",
  "count": 300
}
```

HourlyHitsDTO

– Used in hourly-hits:

```
{
  "hour": 14,
  "count": 150
}
```

Log File Upload Endpoint

Apart from analytics endpoints, the backend provides an API to upload raw log files. This initiates the core data processing workflow. Once the file is uploaded, it is parsed, validated, enriched, and stored in the database, making the analytics endpoints usable.

Log File Ingestion API

Endpoint	Method	Description	Request	Response Type
/api/logs/upload	POST	Accepts a .txt log file for processing and ingestion.	multipart/form-data (file)	UploadResponseDTO

Sample Request



```
curl -X POST http://localhost:8080/api/logs/upload \  
-F "file=@sample-log.txt"
```

Sample Response (Success)



```
{  
  "message": "File processed successfully.",  
  "success": true  
}
```

Sample Response (Failure)



```
{  
  "message": "Error processing file: invalid format at line 32",  
  "success": false  
}
```


Implementation

Overview

This section outlines the end-to-end implementation of the Log Analytics System, translating the proposed solution into a working application. The system is developed using a modular approach, with a clean separation between backend processing, data enrichment, persistent storage, and frontend visualization. The implementation leverages:

- **Spring Boot** for backend REST APIs and data processing
- **MySQL** for persistent storage of enriched log data
- **Chart.js** with **HTML/CSS/JavaScript** for frontend visualizations

Each module is built for scalability, maintainability, and performance, ensuring that even large log files can be processed and visualized effectively.

1. Log File Upload and Reading

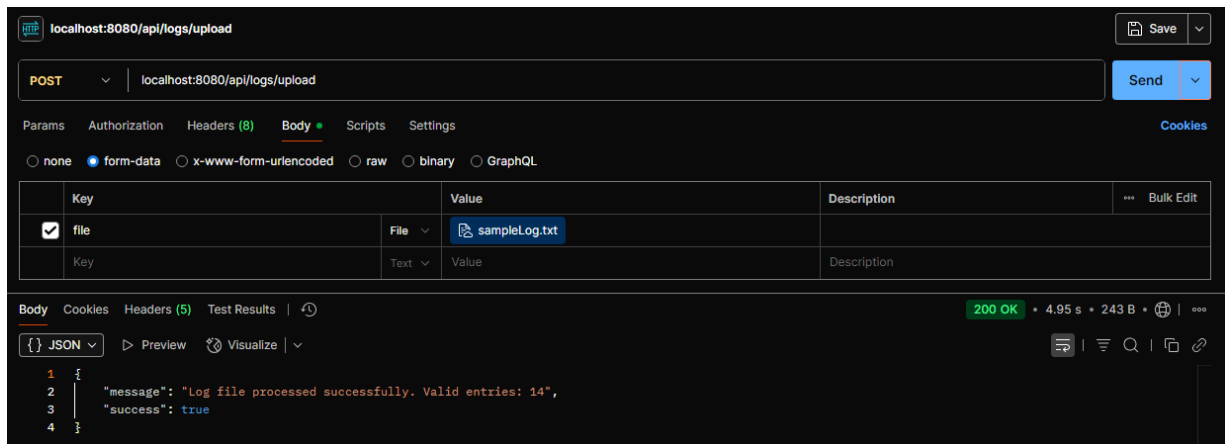
Users upload a .txt log file through a dedicated REST endpoint (/upload). The file is processed line-by-line to maintain memory efficiency. Each line is validated using a strict regular expression:

<timestamp>~<email>~<IP address>~<user-agent>

Lines that fail this pattern (due to malformed format, missing fields, or invalid entries) are automatically discarded. This ensures that only clean, processable data proceeds to the next stage.

```
@PostMapping("/upload")
public ResponseEntity<UploadResponseDTO> uploadLogFile(@RequestParam("file") MultipartFile file) {
    try {
        return ResponseEntity.ok(logProcessingService.processLogFile(file));
    } catch (Exception e) {
        return ResponseEntity.badRequest()
            .body(new UploadResponseDTO("Error: " + e.getMessage(), false));
    }
}
```

REST API endpoint to upload log file and handle response



Log file uploaded via /upload endpoint tested in Postman



Validating each log line using regex while reading the file

2. Log Line Validation and Parsing

Valid lines are split into their respective components:

- Timestamp
- Email
- IP Address
- User-Agent String

Each of these values is mapped to fields in the **LogEntry** Java model class.

```

@Entity
public class LogEntry {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Date timestamp;
    private String email;
    private String ipAddress;
    private String userAgent;

    // Enriched fields
    private String country;
    private String city;
    private String os;
    private String osVersion;
    private boolean isMobile;
    private boolean isTablet;
}

```

LogEntry model class mapping validated log components and enriched data

3. Data Enrichment

a) Geolocation Enrichment

Using the **GeoLite2** database by MaxMind, the IP address is analyzed to extract the **city** and **country**. This step adds valuable location context to each log entry.

```

InputStream databaseStream = getClass()
    .getClassLoader()
    .getResourceAsStream("geo/GeoLite2-City.mmdb");

DatabaseReader dbReader = new DatabaseReader.Builder(databaseStream).build();

InetAddress ipAddress = InetAddress.getByName(entry.getIpAddress());
CityResponse response = dbReader.city(ipAddress);

entry.setCity(response.getCity().getName());
entry.setCountry(response.getCountry().getName());

```


Enriching log entries with city and country using GeoLite2 database

b) User-Agent Parsing

The user-agent string is analyzed using the **UserAgentUtils** library to extract detailed information about the client's environment. This includes:

- **Operating System** (e.g., Windows, Linux, Android)
- **OS Version** (e.g., Windows 10, Ubuntu 20.04)
- **Device Type** (e.g., Desktop, Mobile, Tablet)

This enriched data helps classify user access patterns based on platform and device characteristics.



```
UserAgent userAgent = UserAgent.parseUserAgentString(entry.getUserAgent());  
  
entry.setIsMobile(userAgent.getOperatingSystem().getDeviceType() == DeviceType.MOBILE);  
entry.setIsTablet(userAgent.getOperatingSystem().getDeviceType() == DeviceType.TABLET);  
entry.setOs(userAgent.getOperatingSystem().getName());  
entry.setOsVersion(UserAgentUtils.getOSVersion(entry.getUserAgent(), userAgent));
```

Extracting OS, version, and device type from User-Agent string

4. Database Storage (MySQL)

Each enriched **LogEntry** is persisted in a MySQL database using Spring Data JPA. The table schema captures essential fields such as:

- **Timestamp**
- **Email**
- **IP Address**
- **Country and City** (from geolocation enrichment)
- **Operating System and OS Version**
- **Device Type** flags (mobile, tablet)

Proper indexing ensures efficient query performance, enabling fast aggregation and analytics for the application.

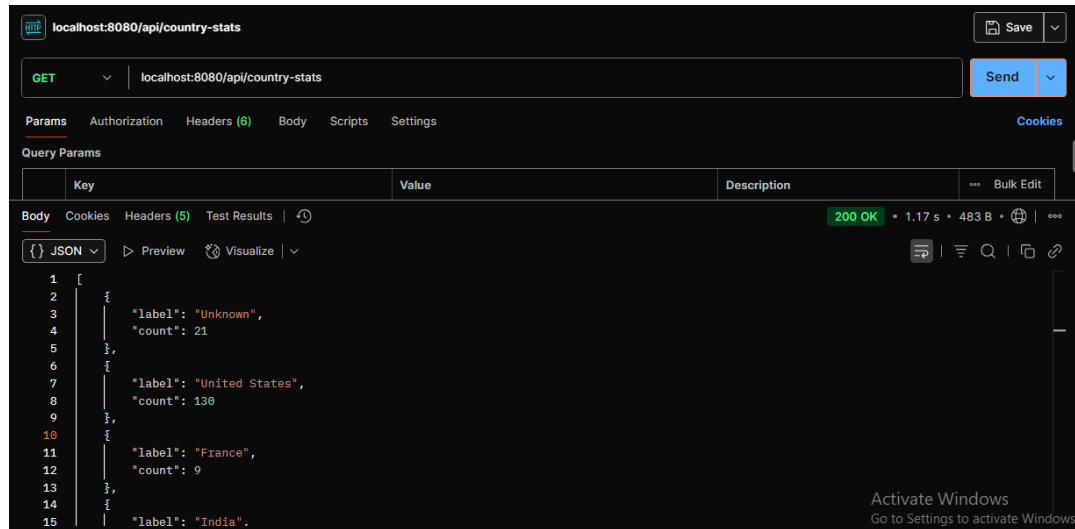
```
mysql> desc log_entry;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | bigint        | NO   | PRI | NULL    | auto_increment |
| city       | varchar(255)  | YES  |     | NULL    |                |
| country    | varchar(255)  | YES  |     | NULL    |                |
| email      | varchar(255)  | YES  |     | NULL    |                |
| ip_address | varchar(255)  | YES  |     | NULL    |                |
| is_mobile  | bit(1)        | NO   |     | NULL    |                |
| is_tablet  | bit(1)        | NO   |     | NULL    |                |
| os         | varchar(255)  | YES  |     | NULL    |                |
| os_version | varchar(255)  | YES  |     | NULL    |                |
| timestamp  | datetime(6)   | YES  |     | NULL    |                |
| user_agent | varchar(255)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
11 rows in set (0.03 sec)
```

MySQL log_entry table schema displaying fields and data types for storing enriched log data

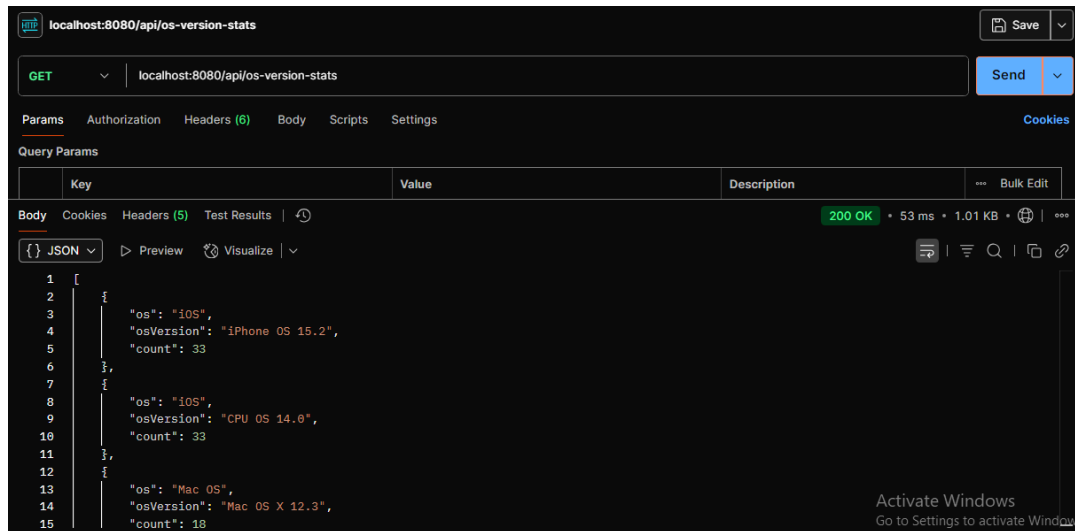
5. RESTful API Endpoints

To expose analytics to the frontend, a set of **RESTful APIs** are provided. Each endpoint returns structured JSON data, suitable for rendering visualizations:

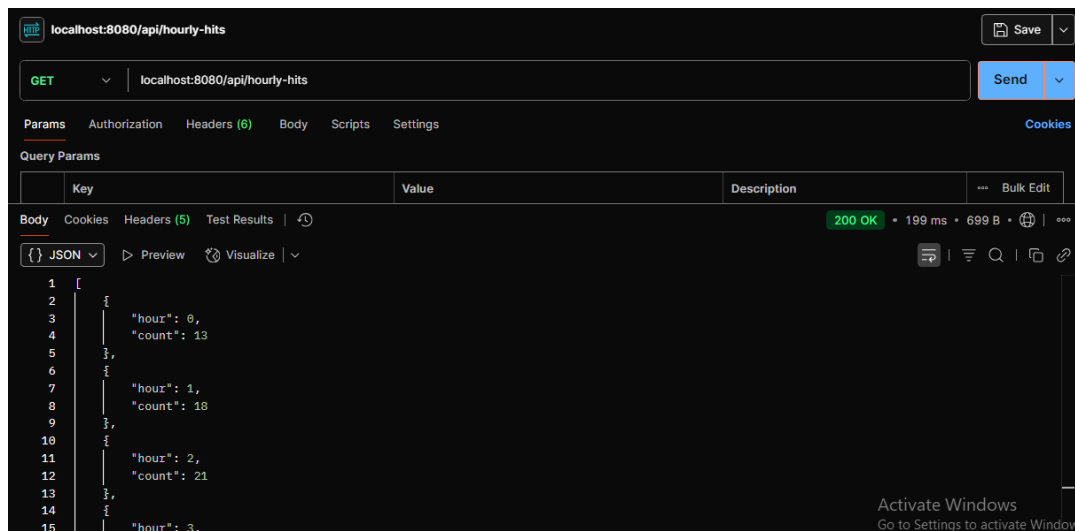
Endpoint	Description
/api/country-stats	Log counts aggregated by country
/api/city-stats	City-wise stats within each country
/api/os-stats	Operating System distribution
/api/os-version-stats	OS version distribution by OS
/api/hourly-hits	Hourly user access patterns



JSON response from /api/country-stats endpoint in Postman



JSON response from /api/os-version-stats endpoint in Postman



JSON response from /api/hourly-hits endpoint in Postman

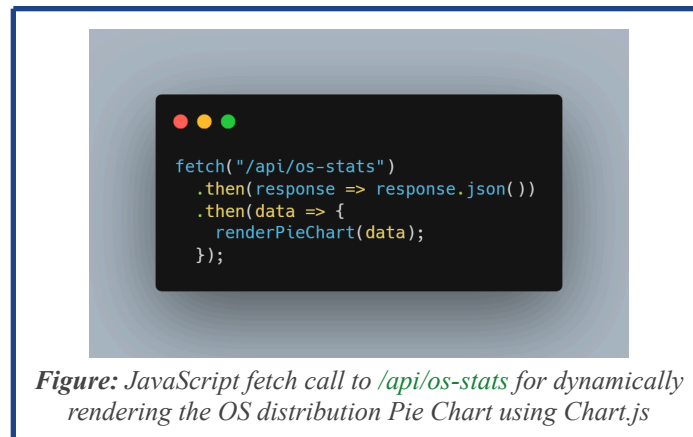
6. Frontend Dashboard & Data Visualization

The dashboard provides a responsive, real-time interface for visualizing log analytics. Interactive charts are rendered using **Chart.js**, powered by dynamic data from backend REST APIs.

Key Visualizations:

- **Doughnut Chart** – Displays **country-wise user distribution**
→ Clicking reveals a Bar Chart for city-level breakdown
- **Pie Chart** – Visualizes **operating system usage share**
→ Clicking reveals a Bar Chart for OS-version stats
- **Line Chart** – Shows **hourly access trends**

Data is fetched asynchronously using the `fetch()` API in JavaScript, ensuring seamless updates without reloading the page.



Dashboard UI

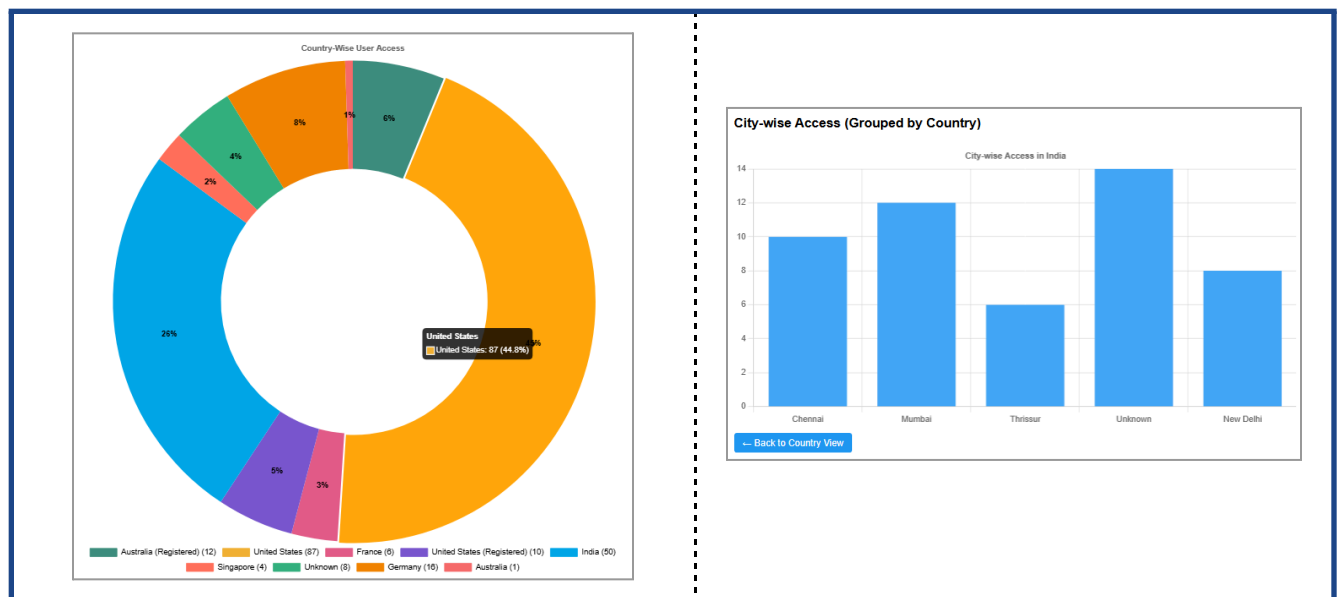


Figure: Doughnut Chart showing users by country with drill-down to city-level bar chart

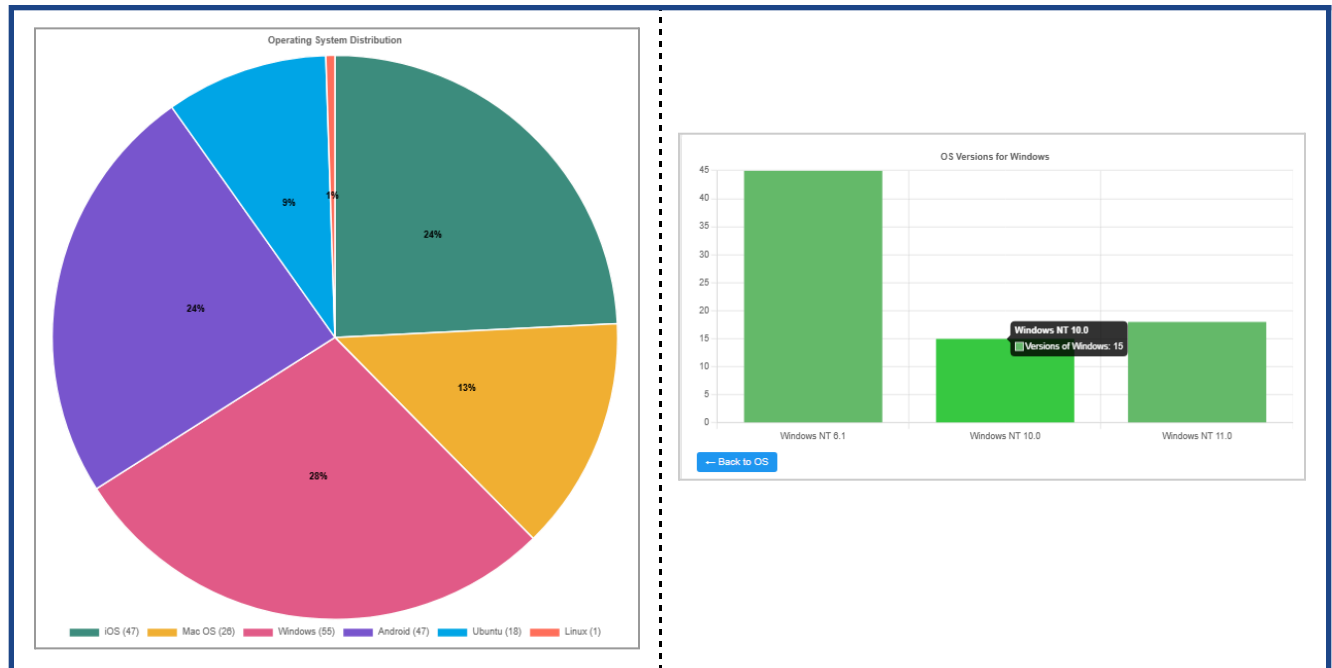


Figure: Pie Chart displaying OS usage and corresponding version breakdown

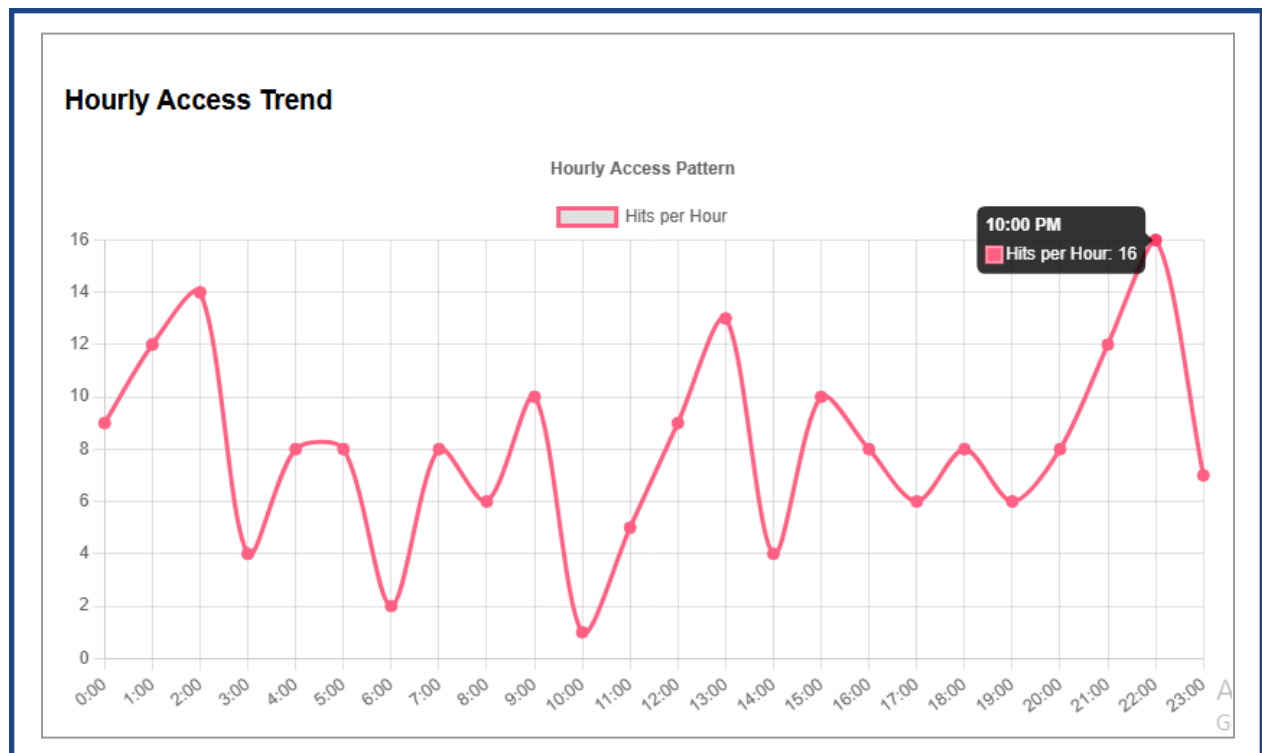


Figure: Line Chart showing number of requests per hour

7. Workflow Summary

The following summarizes the overall flow of data through the system:

1. **User uploads** the `.txt` log file.
2. Backend **reads, validates, and parses** each line.
3. **Geo-location and device info** are enriched via external libraries.
4. Enriched data is **stored in MySQL** for persistence.
5. Aggregated analytics are exposed through **REST APIs**.
6. Frontend **fetches and visualizes** data dynamically using Chart.js.

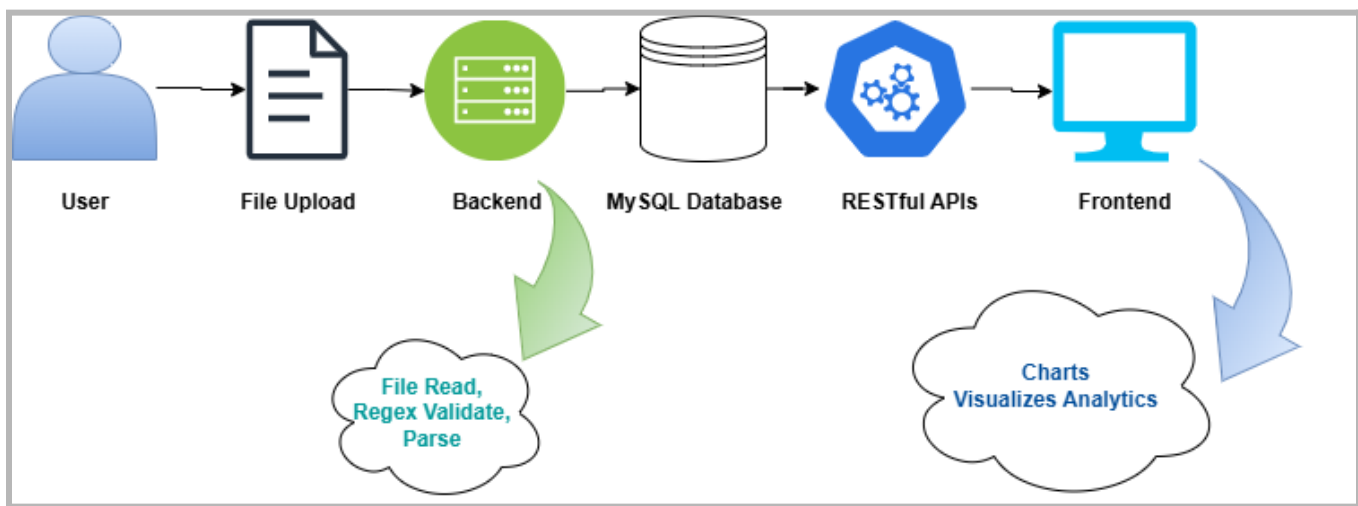


Figure: End-to-end workflow of the log analytics system from file upload to data visualization.

Technologies and Tools Used

Technology	Purpose
Spring Boot	Backend framework and REST API development
MySQL	Data persistence and querying
GeoLite2 (MaxMind)	Geolocation based on IP
UserAgentUtils	Parsing browser/device info from user-agent
HTML/CSS/JS	Frontend structure and interactivity
Chart.js	Data visualization through interactive charts

With the successful implementation of all major components, the system is now capable of transforming raw log data into actionable insights through a seamless backend-frontend integration.

References

1. Spring Boot Documentation

<https://spring.io/projects/spring-boot>

Official documentation and guides for building the backend using the Spring Boot framework.

2. MySQL Documentation

<https://dev.mysql.com/doc/>

Comprehensive resource for database setup, schema design, and query optimization in MySQL.

3. MaxMind GeoIP2 Java API

<https://github.com/maxmind/GeoIP2-java>

Java library used for enriching IP addresses with geolocation data such as country and city.

4. User-Agent Utils

<https://github.com/HaraldWalker/user-agent-utils>

Open-source utility for parsing user-agent strings to extract OS, device type, and browser information.

5. Chart.js

<https://www.chartjs.org/>

Lightweight JavaScript library used for building dynamic and responsive charts in the frontend.

6. Java Regular Expressions Tutorial – Oracle Docs

<https://docs.oracle.com/javase/tutorial/essential/regex/>

Official tutorial explaining how to construct and apply regular expressions in Java.

7. REST API Design Best Practices

<https://restfulapi.net/>

Industry-standard principles and guidelines for designing scalable and clean RESTful APIs.