SPE Report

Project Title:

CodeCollab: A collaborative coding platform

Submitted By:

IMT2022014 Shashank Tippanavar

IMT2022057 Kushal Suvan Jenamani

Contents

1 What is our work about:

CodeCollab is a modern, feature-rich platform designed for developers to practice problem-solving, collaborate in real-time, and leverage the power of AI to enhance their coding skills. It combines a robust code execution engine with a suite of AI-powered tools, making it the ultimate training ground for technical interviews and competitive programming. Built with a scalable microservices architecture, CodeCollab provides a seamless and interactive experience—from solving problems to getting intelligent feedback on your solutions.

Originally this was supposed to be cloud native (But i just recently exhausted my GCP free credits so i had to redeploy everything from a local perspective).

1.1 Tech Stack

- **Frontend:** React, Vite, Recoil (State Management), Tailwind CSS, Framer Motion
- **Backend:** Node.js, Express.js
- **API Gateway:** Node.js with `http-proxy-middleware`
- **Database:** PostgreSQL (NeonDB), MongoDB, MinIO
- **Caching and Background Jobs:** Redis, BullMQ
- **AI Integration:** Google Gemini API (Gemini-2,5-flash)
- **Code Execution Engine:** Judge0
- **Authentication and Authorization:** JSON Web Tokens (JWT)

We discuss about the below throughout this document:

- Overall system architecture
- Backend microservices and their responsibilities
- Database design and persistence layer
- Inter-service communication
- Object storage using MinIO (S3-compatible) (For local object store deployments)

2 High-Level Architecture Overview

At a high level, CodeCollab follows a distributed microservices architecture with clear separation of concerns:

- **Frontend (Client Layer):** Web-based user interface for end users.
- **API Gateway:** Central entry point for all client requests, responsible for request routing and aggregation.
- **Core Backend Microservices:** Domain-driven microservices implementing the core business logic.
- **Execution and Evaluation Layer:** Secure code execution and evaluation using Judge0.

- **Persistence Layer:** Data storage layer consisting of a relational database and object storage.
- **Infrastructure Layer:** Kubernetes-based container orchestration and deployment infrastructure.

Key architectural features:

- **Loose Coupling Between Services:** Each microservice operates independently with minimal inter-service dependencies.
- **Single Responsibility Principle:** Every microservice is designed to handle a single, well-defined domain concern.
- **Stateless Services:** Services are implemented to be stateless wherever possible, enabling easier scaling and fault tolerance.
- **Horizontal Scalability:** Containerized services allow the system to scale horizontally based on workload demands.
- **Polyglot Persistence:** The architecture employs multiple data storage technologies, including a relational database and an object storage system, chosen based on data access patterns.

3 Frontend Architecture

3.1 Technology Stack:

- **React with Vite:** Enables fast development builds and an efficient modern frontend developer experience.
- **Monaco Editor:** Provides an in-browser code editing environment with a Visual Studio Code–like experience.
- **Yjs:** CRDT-based framework enabling real-time collaborative code editing with conflict-free synchronization.
- **WebSockets:** Facilitates low-latency, bidirectional communication for real-time synchronization between clients.

3.2 Responsibilities:

The frontend is responsible for:

- **User Authentication Flows:** Handles email-based user login and session initiation.
- **Collaborative Editor Rendering:** Renders real-time collaborative code editor sessions for multiple users.
- **Problem View and Submission Management:** Manages problem descriptions, code submissions, and related user interactions.
- **Execution Results and Verdict Display:** Displays code execution outputs, verdicts, and detailed logs to the user.
- **Real-Time Collaboration:** Supports real-time text synchronization across users during collaborative editing sessions.

The frontend remains logic-light, delegating all business rules to backend services.

4 Backend Architecture (Microservices):

We follow a monorepo based architecture...
We have a single kubernetes cluster deployed via kind locally for the entire setup. The backend is decomposed into domain-driven microservices, each independently deployable.
We have the following services:

4.1 API Gateway

In this deployment, an API Gateway is used as the central entry point for all backend services in a Kubernetes-based microservices architecture. Since the API Gateway runs inside a private cluster and is not directly accessible from the internet, Ngrok is employed to securely expose it via a public HTTPS endpoint.
This setup is primarily intended for development, testing, and external integrations, such as webhook callbacks and third-party service access.
Ngrok is used in this deployment to securely expose the internal API Gateway service running inside a Kubernetes cluster to the public internet. This is particularly useful for development, testing, and integrating third-party services (such as webhooks) that require a publicly reachable endpoint.

4.1.1 API Gateway Role and Responsibilities

The API Gateway serves as the core access layer of the system and performs the following functions:

- Acts as a single entry point for all client requests.
- Routes incoming traffic to appropriate backend microservices.
- Abstracts internal service topology from external clients.
- Enables centralized handling of: Request routing, Authentication/authorization (where applicable), API versioning and request validation.

The API Gateway is deployed as a Kubernetes ClusterIP service, ensuring it remains internal to the cluster and accessible only to other services unless explicitly exposed.

4.1.2 Purpose of using Ngrok:

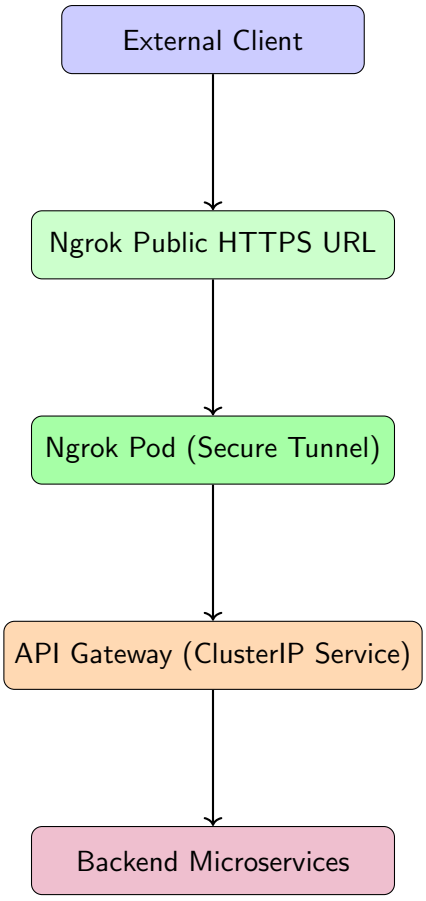
Since the API Gateway is not publicly exposed by default (in our current code its inside our k8s cluster):

- The API Gateway runs inside a private Kubernetes cluster and is not directly accessible from outside.
- Ngrok creates a secure public tunnel that forwards external HTTPS traffic to the internal API Gateway service.
- This avoids the need for cloud load balancers or public ingress during development.

Ngrok allows external systems to interact with the API Gateway as if it were publicly hosted.

4.1.3 Deployment Architecture

1. **API Gateway:**
   - Deployed as a Kubernetes ClusterIP service.
   - Routes requests to backend microservices inside the cluster.
2. **Ngrok Pod:**
   - Deployed as a standalone pod within the same Kubernetes namespace.
   - Authenticates using an Ngrok auth token.
   - Establishes a tunnel from a public Ngrok URL to the API Gateway's internal service and port.



4.1.4 Benefits:

- Securely exposes the API Gateway without modifying cluster networking configurations.
- Automatically provides HTTPS support through Ngrok.
- Well-suited for webhook integrations, external API testing, and development demos.
- Eliminates the need for `NodePort` or `LoadBalancer` services during development.

4.1.5 Limitations:

- Public URLs generated by Ngrok are temporary unless a paid plan is used.
- Not intended for production-scale traffic.
- Tunnel reliability depends on the Ngrok service.
- Lower throughput and higher latency compared to cloud-native load balancers.

By combining an internal API Gateway with Ngrok, the system achieves a clean separation between internal service architecture and external accessibility. Ngrok provides a lightweight, secure, and flexible mechanism to expose the API Gateway for development and testing, while keeping the Kubernetes cluster private and uncluttered by permanent ingress infrastructure.

4.2 Boilerplate Generation Service (`boilerplate-server`)

4.2.1 Overview

The Boilerplate Generation Service is an AI-driven microservice designed to automatically generate competitive programming boilerplate code. It uses the Gemini 2.5 Flash large language model to convert problem statements written in Markdown into executable starter code for a specified programming language.

This service focuses strictly on generating standardized scaffolding rather than complete solutions.

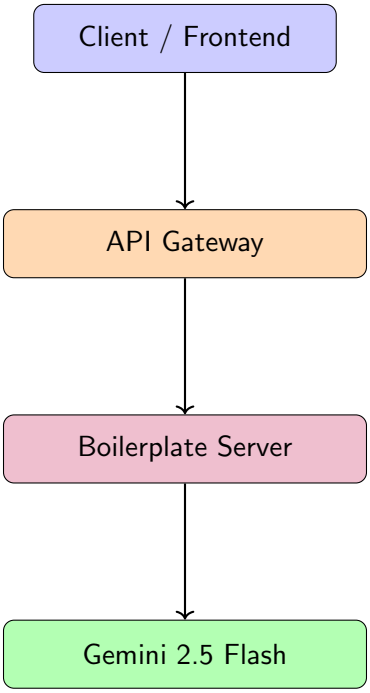
4.2.2 Inputs and Outputs

- **Input:**
  - Problem description in Markdown format
  - Target programming language
- **Output:**
  - Function boilerplate containing the required function signature
  - Main function handling input from `stdin` and output to `stdout`

4.2.3 Request Processing Logic

- Requests are validated for required fields.
- A fixed 15-second delay is enforced before each LLM call to avoid rate bursts.
- A structured prompt is sent to the Gemini model.
- The LLM response is constrained using a strict JSON schema.
- The generated boilerplate is returned as a JSON object.

4.2.4 Architecture and Data Flow



4.2.5 Design Constraints

- Stateless service design.
- Strict JSON schema enforcement for predictable output.
- Subject to LLM quota and rate limits.

4.2.6 Conclusion

The boilerplate-server automates repetitive setup work for competitive programming, allowing users to focus entirely on problem-solving logic while maintaining language-agnostic support.

4.3 Problem AI Service (`problem-ai-service`)

4.3.1 Overview

The Problem AI Service provides AI-assisted learning support for users solving programming problems. Unlike solution generators, this service is designed to guide users through hints, explanations, and debugging assistance without directly revealing answers.

4.3.2 Supported Endpoints

`/hint` Provides a single, concise, non-spoilery hint based on the problem description and the user's current code.

`/explain` Analyzes an accepted solution and returns:

- Step-by-step explanation

- Time complexity
- Space complexity
- Optimization suggestion

`/debug` Analyzes a failed submission by comparing:

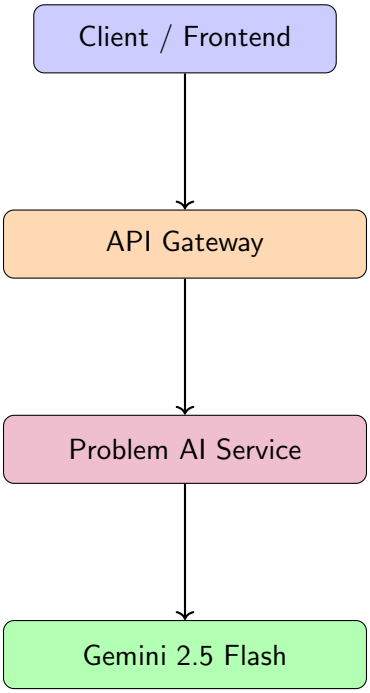
- Input
- User output
- Expected output
- Compilation errors (if any)

and provides corrective guidance.

4.3.3 LLM Interaction Model

- Each endpoint uses a specialized prompt.
- Responses are strictly enforced using JSON schemas.
- A 15-second delay is applied before every Gemini request.

4.3.4 Architecture and Data Flow



4.3.5 Design Philosophy

- Learning-first approach.
- No direct solution leakage.
- Deterministic JSON responses for frontend integration.

4.3.6 Conclusion

The problem-ai-service functions as an intelligent tutoring layer that enhances learning outcomes while preserving the integrity of the problem-solving process.

4.4 URL Parser Service (`url-parser-server`)

4.4.1 Overview

The URL Parser Service is a specialized microservice responsible for extracting competitive programming problem statements from external URLs. It combines browser automation with LLM-based content extraction to generate clean, structured Markdown.

4.4.2 Responsibilities

- Fetch problem pages using a headless browser.
- Extract full HTML content reliably.
- Convert unstructured HTML into structured Markdown.
- Normalize problem data across different platforms.

4.4.3 Implementation Details

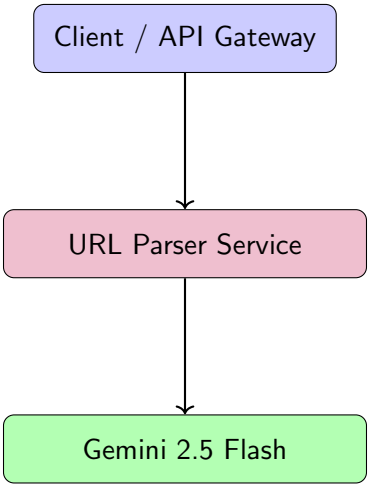
- Uses **Puppeteer** for browser-based scraping.
- Handles JavaScript-heavy websites.
- Extracts raw HTML from the page body.
- Sends HTML content to Gemini for semantic parsing.

4.4.4 LLM Output Structure

The parsed output includes:

- Problem title
- Description (Markdown)
- Constraints (Markdown)
- Examples (Markdown)
- Combined final Markdown document

4.4.5 Architecture and Data Flow



4.4.6 Conclusion

The URL Parser Service enables seamless ingestion of external competitive programming problems by converting arbitrary web pages into structured, platform-ready Markdown, while keeping scraping logic isolated from core services.

4.5 Submission Orchestration and Webhook Service (`webhook-server`)

4.5.1 Overview

The Webhook Server is the core execution and orchestration service of the platform. It manages the entire lifecycle of a code submission, from problem creation and submission handling to test case execution, real-time result streaming, and final verdict computation.

This service integrates multiple subsystems including Judge0, Redis, WebSockets, PostgreSQL, and object storage, making it the most stateful and coordination-heavy microservice in the architecture.

4.5.2 Primary Responsibilities

- Accept and validate code submissions from users
- Dispatch test cases to Judge0 for execution
- Receive asynchronous execution results via webhooks
- Aggregate and evaluate test case results
- Stream real-time execution updates to clients
- Persist final verdicts and execution metrics

4.5.3 Problem Creation and Boilerplate Integration

When a new problem is created:

- Test cases are uploaded to object storage (MinIO for local, GCS for cloud).
- Problem metadata is stored in PostgreSQL.

- Boilerplate generation is triggered asynchronously.
- Boilerplates for multiple languages are generated in the background by invoking the Boilerplate Generation Service.

This design ensures fast API responses while offloading expensive LLM operations to background execution.

4.5.4 Submission Flow

The submission pipeline follows a strictly ordered sequence:

1. User submits source code and language selection.
2. Test cases are fetched from object storage.
3. A submission record is created in PostgreSQL.
4. Redis is initialized to track test case counts and results.
5. Each test case is submitted sequentially to Judge0.
6. Judge0 executes code and sends results to the webhook endpoint.

The client receives an immediate acknowledgment with a unique submission identifier while execution continues asynchronously.

4.5.5 Judge0 Webhook Processing

The webhook endpoint performs the following:

- Matches Judge0 tokens to original test cases using Redis.
- Combines execution output with input and expected output.
- Publishes incremental test results to Redis Pub/Sub channels.
- Aggregates all test case results once execution completes.

Execution limits such as time and memory are enforced during result processing.

4.5.6 Real-Time Result Streaming

To enable live feedback:

- Clients establish WebSocket connections with the server.
- Each client subscribes to a Redis Pub/Sub channel scoped to a submission ID.
- As Judge0 results arrive, updates are streamed instantly to connected clients.

Redis Pub/Sub ensures scalability across multiple server instances without relying on in-memory state.

4.5.7 Final Verdict Computation

Once all test cases are received:

- Each test case verdict is computed independently.
- A global verdict (`Accepted` or `Rejected`) is derived.
- Maximum time and memory usage are calculated.
- Final results are uploaded to object storage.
- Submission records are updated in PostgreSQL.

A final completion message is published to notify all subscribed clients.

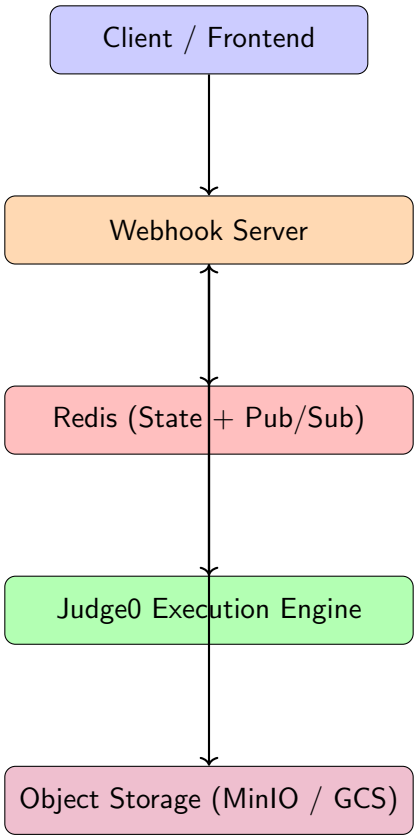
4.5.8 Storage Abstraction

The service uses a unified storage adapter that supports:

- Google Cloud Storage (GCS)
- S3-compatible object storage (e.g., MinIO)

This abstraction allows seamless deployment across local, cloud, and hybrid environments without code changes.

4.5.9 Architecture and Data Flow



4.5.10 Scalability and Fault Tolerance

- Redis Pub/Sub enables horizontal scaling.
- WebSocket state is isolated per instance.
- Background jobs do not block API responsiveness.
- Object storage decouples large payloads from the database.

4.5.11 Conclusion

The webhook-server acts as the backbone of the platform, coordinating execution, persistence, and real-time feedback. Its event-driven architecture ensures scalability, responsiveness, and reliability while maintaining strict separation of concerns across microservices.

5 Monitoring (ELK Stack)

We employed an ELK stack to ingest logs from all the microservices. Every microservice ran a filebeat sidecar container. The updating logs were relayed to a singleton ELK stack server. This further was augmented with automated logs classification for AIOps.

5.1 Tech Stack

Elasticsearch A distributed, document-oriented search and analytics engine used for indexing, storing, and querying log data in near real-time. It provides full-text search, structured queries, and aggregation capabilities over time-series data, enabling efficient log analysis and correlation across microservices.

Logstash A centralized data processing pipeline used to ingest, parse, filter, and enrich log events before indexing them into Elasticsearch. Logstash enabled flexible transformation of heterogeneous log formats into a unified schema suitable for downstream analytics.

Filebeat A lightweight log shipper deployed as a sidecar container alongside each microservice. Filebeat continuously monitored log files, efficiently forwarded incremental updates to the ELK stack, and ensured low-overhead, reliable log ingestion.

Kibana A visualization and analytics interface used to explore indexed logs, construct dashboards, and perform ad-hoc queries. Kibana facilitated real-time monitoring, debugging, and inspection of system behavior through interactive visualizations.

AIOps Layer An automated log classification and analysis layer built on top of the ELK stack. This component applied machine learning techniques to clustered and categorized log streams, enabling anomaly detection, pattern discovery, and proactive operational insights.

5.2 Implementation Overview

The implementation follows a modular and containerized design to enable centralized logging and monitoring across multiple microservices.

5.2.1 System Architecture

The overall architecture is composed of the following key components:

- A centralized ELK stack server for log ingestion, storage, and analysis.
- Multiple microservices, each paired with a Filebeat sidecar container.
- A shared logging interface based on append-only log files.

This design enables decoupled observability without modifying application-level logic.

5.2.2 Log Collection Mechanism

Log collection is performed using Filebeat sidecar containers deployed alongside each microservice.

- Each Filebeat instance monitors service-specific log files.
- Log updates are detected incrementally to avoid redundant reads.
- Events are forwarded to the central ELK stack using the Beats protocol.

This approach ensures low-overhead and continuous log shipping across services.

5.2.3 Centralized Log Ingestion

The ELK stack is deployed using Docker Compose and acts as the aggregation layer.

- Elasticsearch indexes incoming log events in near real-time.
- Indexed logs are stored as structured documents.
- Time-based indexing enables efficient querying and aggregation.

This centralized ingestion pipeline enables unified access to distributed logs.

5.2.4 Log Analysis and Visualization

Kibana is used as the primary interface for log exploration and monitoring.

- Interactive dashboards provide a system-wide view of logs.
- Filters and queries enable targeted inspection of events.
- Visualizations support debugging and performance analysis.

This facilitates rapid identification of anomalies and operational issues.

5.2.5 AIOps Augmentation

The logging pipeline is augmented with an automated log classification layer.

- Collected logs are grouped based on structural and semantic patterns.
- Classification enables identification of recurring event types.
- The processed outputs support higher-level operational insights.

This AIOps layer enhances the monitoring system with intelligent analysis capabilities.

5.2.6 Deployment and Usage Workflow

The deployment process follows a simple and reproducible workflow:

- The ELK stack is launched as a standalone service using Docker Compose.
- Filebeat sidecars are started alongside each microservice.
- Logs become available for querying and visualization in Kibana.

This workflow allows rapid setup and experimentation with centralized logging.

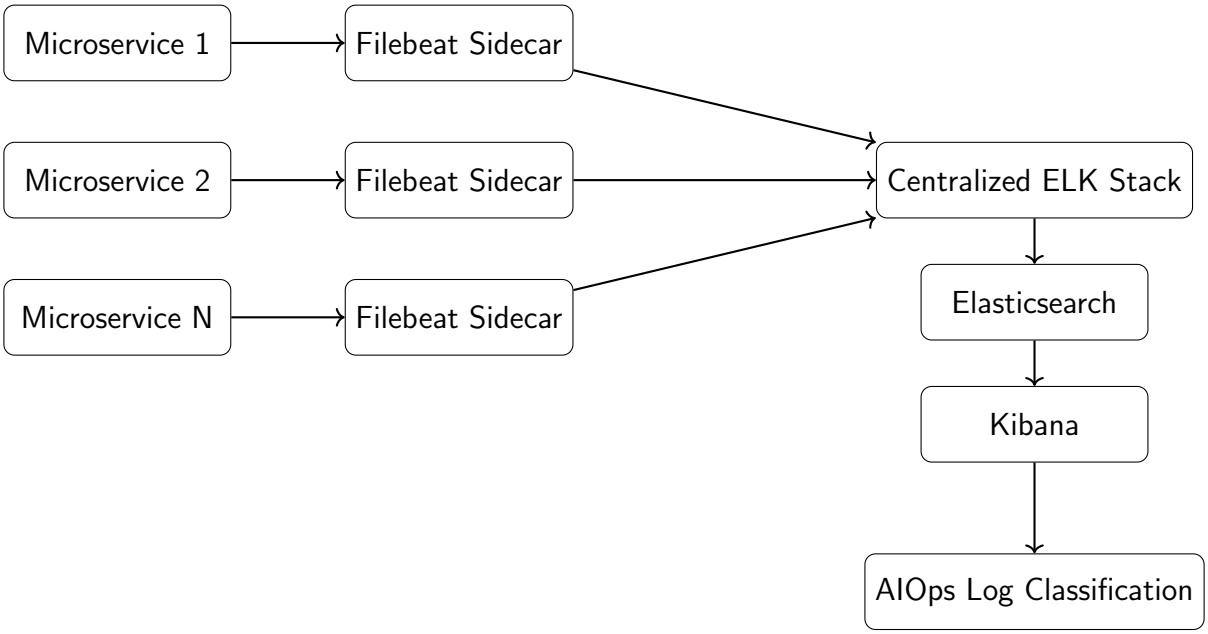


Figure 5.1: Centralized Logging Architecture using ELK Stack with Filebeat Sidecars

6 AIOps-Based Log Intelligence

To enhance the observability pipeline beyond manual inspection and rule-based alerts, the system was augmented with a lightweight AIOps layer for automated log analysis. This layer performs sentiment-based classification of log messages to identify potentially critical or anomalous events.

6.1 Motivation

In large-scale microservice systems, raw logs often contain early indicators of failures, degradations, or abnormal behavior. However, these signals may not always be explicitly structured or tagged. Automated log intelligence helps in:

- Reducing manual effort required for log inspection.
- Highlighting potentially critical events from large log volumes.
- Enabling proactive identification of operational incidents.

6.2 Log Retrieval from Elasticsearch

The AIOps pipeline retrieves recent logs directly from Elasticsearch using its search API.

- Logs are queried over a rolling time window (last 24 hours).
- Retrieved entries are filtered to extract relevant textual fields.
- This enables near real-time post-ingestion analysis.

The centralized indexing provided by Elasticsearch allows efficient access to distributed log data across all microservices.

6.3 Sentiment-Based Log Analysis

Natural language processing is applied to log messages using a sentiment analysis model.

- Each log message is analyzed using the VADER sentiment analyzer.
- A compound sentiment score is computed for every log entry.
- Scores capture the overall emotional polarity of the message text.

While logs are primarily technical, negative sentiment often correlates with error states, failures, or abnormal execution paths.

6.4 Automated Severity Classification

Based on sentiment scores, logs are categorized into severity levels.

- Strongly negative sentiment is classified as an *incident*.
- Neutral sentiment is treated as informational.
- Positive sentiment is considered normal operational behavior.

This classification provides a simple but effective abstraction over raw logs, enabling faster identification of problematic events.

6.5 Incident Indexing and Feedback Loop

Logs identified as incidents are indexed back into Elasticsearch under a dedicated index.

- Classified incident logs are stored with timestamps and metadata.
- This creates a curated index of high-severity events.
- The indexed results can be visualized directly in Kibana dashboards.

This feedback loop integrates machine-assisted intelligence directly into the existing ELK workflow.

6.6 Role in the Monitoring Pipeline

The AIOps component operates as an auxiliary analytics layer on top of centralized logging.

- It complements traditional monitoring and dashboards.
- It enables prioritization of logs based on inferred severity.
- It supports scalable monitoring as log volumes increase.

Overall, this approach demonstrates how lightweight machine learning techniques can be seamlessly integrated into observability pipelines to improve operational awareness.