Intelligent Log Analysis & Anomaly Detection Tool: A Technical Report (C++)

MD Ziad Bin Sorwar

Abstract

This report documents the internal design and implementation of the Intelligent Log Analysis & Anomaly Detection Tool *implemented in modern C++. The system targets* ***GB-scale offline log processing*** *using a* ***streaming*** *(line-by-line) architecture, and combines lightweight parsing with multiple anomaly detectors (frequency spikes, statistical outliers, burst repetitions, and rare IP indicators). The goal of this document is to explain the project end-to-end for a reader unfamiliar with the codebase, including workflow, data flow, statistical methods, and architectural decomposition.*

1. Project Overview

1.1. What the project does

The project is a command-line log analysis tool that ingests raw log files (plain text and mixed-format content), parses them into structured events, computes summary statistics, and detects anomalies. The implementation is designed for **offline batch processing**: a user points the tool to one log file (potentially large), and the tool produces (i) human-readable output, and (ii) machine-readable artifacts such as JSON/CSV reports and optional time-series files for plotting.

The core processing loop uses **streaming input** (e.g., `std::getline` over an `ifstream`) rather than loading the entire file into memory. This makes the tool suitable for large logs and aligns with memory-efficient design goals: state is kept only for summary counters, bounded sliding windows, and a small number of representative samples per anomaly.

1.2. Why it matters (problem statement)

Operational systems generate enormous volumes of logs (application traces, security audit trails, service health events). Manual inspection is slow, subjective, and frequently infeasible at scale. The tool addresses two common requirements:

1. **Fast offline triage:** summarize a large log file and highlight suspicious regions without requiring specialized infrastructure.
2. **Robustness to heterogeneous data:** handle mixed formats (standard timestamped text, JSON logs [5], and corrupted/malformed lines) and still produce actionable reports.

By using simple statistical detectors (e.g., Z-score on event rates) and sliding-window logic, the system can surface deviations that are often early indicators of incidents: bursts of repeated failures, sudden surges of activity, long silences, or rare security-related tokens (e.g., previously unseen IPs).

1.3. Main components (code-level)

The project is organized into the following major module families (mirrored in `include/` and `src/`):

- **Core** (`core/`): fundamental data types such as `LogEntry`, `Anomaly`, and the aggregated `Report`.
- **Input** (`input/`): streaming file access and robust parsing (`LogParser`).
- **Analysis** (`analysis/`): offline analyzers that aggregate statistics and emit anomaly descriptions after full-file processing.
- **Anomaly** (`anomaly/`): online/streaming detectors that operate during ingestion using sliding windows and online statistics.
- **Reporting** (`report/`): renderers for console, JSON, and CSV output; plus an orchestrating `ReportGenerator`.
- **Utilities** (`utils/`): timestamp parsing/formatting, string helpers, simple logging, and configuration parsing.

2. Code Workflow & Data Flow

2.1. End-to-end execution workflow

At a high level, `src/main.cpp` implements the CLI entry point and wires the pipeline:

1. **Parse CLI arguments:** input file path plus optional flags (`--json`, `--csv`, `--graphs`, `-o` output directory, `-v` verbosity).
2. **Initialize modules:** create one instance each of `LogParser`, analyzers (`FrequencyAnalyzer`, `TimeWindowAnalyzer`, `PatternAnalyzer`), and detectors (`SpikeDetector`, `StatisticalDetector`, `BurstPatternDetector`, `IpFrequencyDetector`, plus a `RuleBasedDetector` scaffold).

3. **Stream input file:** open the log file using `std::ifstream` and iterate line-by-line via `std::getline`.
4. **Parse each line:** call `LogParser::parseLineDetailed` which returns either a structured `LogEntry` or a "malformed" classification.
5. **Update report counters:** maintain global statistics (total, by level, by source) and time-bucketed counters (events/errors/warnings/anomalies per minute).
6. **Run streaming detectors:** feed each parsed `LogEntry` to the online detectors that maintain bounded windows and emit anomaly objects as they occur.
7. **Run offline analyzers:** after ingestion completes, call each analyzer's anomaly routine (e.g., `FrequencyAnalyzer::detectAnomalies`) to generate summary anomalies based on whole-file aggregates.
8. **Generate outputs:** write console output and optionally JSON/CSV reports. If `--graphs` is enabled, the tool also exports time-series CSVs and a small Python script that can plot the exported data.

2.2. Data model and key invariants

Structured event: `core::LogEntry`. `LogEntry` is the canonical representation of one parsed log record. It stores: (1) a normalized timestamp (`std::chrono TimePoint`), (2) a severity level (`core::LogLevel`), (3) an optional logical source (service/component), (4) the extracted message, and (5) the original raw line for context. This design keeps parsing lightweight while retaining enough context for downstream reporting.

Anomaly representation: `core::Anomaly`. Detected anomalies are stored as `core::Anomaly` objects with: type (e.g., `FrequencySpike`, `StatisticalOutlier`, `Silence`), severity (Low/Medium/High/Critical), a time window (start/end), a detector-specific score (e.g., Z-score or spike ratio), a human-readable description, and optional sample events. This supports both human inspection and machine consumption via JSON/CSV output.

Aggregated report: `core::Report`. `Report` is a mutable aggregation container: it stores the analysis time range, total parsed entries, per-level and per-source counters, and the anomaly list. The pipeline updates counters incrementally during streaming, and anomalies are appended as detectors fire.

2.3. How raw log data flows from input to output

Figure 1 illustrates the main data movement: raw text is streamed from disk, parsed into `LogEntry`, distributed to analysis/detection modules, and finally emitted as reports and artifacts.

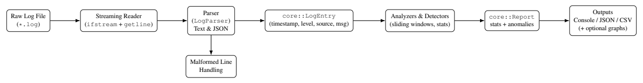

Raw Log File (*.log)
Streaming Reader (ifstream + getline)
Parser (LogParser) Text & JSON
Malformed Line Handling
core::LogEntry (timestamp, level, source, msg)
Analyzers & Detectors (sliding windows, stats)
core::Report stats + anomalies
Outputs Console / JSON / CSV (+ optional graphs)

Figure 1. Data flow from raw log input to analysis outputs.

2.4. Module interactions during anomaly detection

During streaming ingestion, each parsed `LogEntry` is fed into multiple detectors that operate independently:

- **SpikeDetector:** compares a short window event rate against a longer baseline for each source and emits frequency-spike anomalies.
- **StatisticalDetector:** converts timestamps into an estimated event-rate signal and applies Z-score thresholding using online mean/variance updates.
- **BurstPatternDetector:** detects repeated occurrences of the same normalized message signature within a short window.
- **IpFrequencyDetector:** extracts IP tokens from the message and flags rare IPs based on frequency thresholds.

In addition, the tool records malformed lines as low-severity anomalies to keep track of data quality issues (missing fields, unrecognized formats, corruption).

3. Methods and Algorithms

3.1. Parsing strategy for mixed-format and malformed logs

The parser (`input/LogParser`) is intentionally conservative: it attempts to extract required fields and returns a structured event only if a timestamp, a level, and a message can be derived.

Text logs. For non-JSON lines, parsing is heuristic:

- Timestamp: the parser expects a `YYYY-MM-DD HH:MM:SS` prefix (19 characters) and uses `utils/TimeUtils::parseTimestamp`.
- Level: it performs a case-insensitive substring search for tokens such as `INFO`, `WARN`, `ERROR`, `CRITICAL`/`FATAL`.
- Source: it tries simple patterns such as `source: message` or `[source]` brackets; if extraction fails, it falls back to `"unknown"`.
- Message: it trims the timestamp prefix and heuristically skips the first two tokens (commonly level and source) before reconstructing the remainder as the message body.

JSON logs. If a trimmed line begins with { it is treated as JSON. To avoid external dependencies, JSON parsing is "best-effort" string extraction: it searches for string-valued keys with common aliases (timestamp: `timestamp/time/@timestamp`; level: `level/severity`; message: `message/msg`; source: `service/component/sour` ce). Timestamps accept either the space-separated format or an ISO-like YYYY-MM-DDTHH:MM:SS prefix (by replacing T with a space before parsing). Missing required keys or invalid timestamp formats cause the line to be marked malformed.

Malformed/corrupted entries. When parsing fails, the tool increments a malformed counter and emits a low-severity anomaly with a short description (e.g., "No matching pattern" or "JSON missing required fields"). This design choice is pragmatic: in large-scale offline processing, corrupted entries are themselves valuable signals, and tracking them avoids silently discarding potential evidence.

3.2. Sliding windows and why they are used

Many anomaly classes are defined relative to a recent baseline. A sliding window converts an unbounded stream into a bounded memory footprint while retaining temporal locality. In this codebase, sliding windows appear in multiple forms:

- **TimeWindowAnalyzer:** maintains a fixed-duration time bucket (default window size in seconds) to compute error rate and detect silence gaps.
- **SpikeDetector:** keeps a *short* deque of recent events and a *baseline* deque to compute a rate ratio (current vs baseline).
- **StatisticalDetector:** keeps a deque of recent timestamps per source to estimate event rate in a configurable time window.
- **BurstPatternDetector:** keeps a deque of recent occurrences per normalized message signature and triggers when repeats exceed a threshold.

This approach is well-suited for GB-scale logs because memory consumption depends mainly on window sizes and thresholds rather than total file size.

3.3. Statistical anomaly detection [1] (mean, std, Z-score)

The anomaly/StatisticalDetector implements online estimation of mean and variance using **Welford's algorithm [2, 3]**. For each source (and globally), it maintains:

- a running mean μ,
- an accumulated second moment m_2,
- a count n,
- and a bounded window of recent values (for potential future windowed statistics).

For a new value x:

$$\delta = x - \mu, \quad \mu \leftarrow \mu + \delta/n, \quad m_2 \leftarrow m_2 + \delta(x-\mu)$$

and the sample variance is $\sigma^2 = m_2/(n-1)$ when $n \geq 2$.

Event-rate signal. Rather than using raw counts, the detector computes an *event rate* from the timestamps observed for each source: it keeps a deque of timestamps within a time window and estimates events-per-minute based on the span between the oldest and newest retained timestamps. This yields a continuous signal suitable for Z-score normalization.

Z-score test. Once the detector has accumulated sufficient statistics (the implementation checks for at least 10 observations and non-zero standard deviation), it computes:

$$z = \frac{x-\mu}{\sigma}$$

and classifies the observation as anomalous if $|z| > \tau$, where τ is a configurable threshold (m_zScoreThreshold). The anomaly severity is scaled relative to τ and capped at 1.0 for reporting.

3.4. Spike detection (short-vs-baseline rate ratio)

The anomaly/SpikeDetector performs a classic "rate ratio" test per source. It keeps two deques per source:

- a short window (m_shortWindow) capturing the most recent activity,
- and a longer baseline window (m_baselineWindow) representing typical behavior.

For each entry it updates both deques (evicting old timestamps), then computes:

$$\text{spikeRatio} = \frac{\text{currentCount}/|\text{shortWindow}|}{\text{baselineCount}/|\text{baselineWindow}|}.$$

A spike is emitted if the ratio exceeds a threshold and there are enough events to establish a baseline (minimum-event safeguards reduce false alarms on sparse sources).

3.5. Burst repetition detection

The anomaly/BurstPatternDetector detects rapid repetition of "the same" message. It first maps each log entry to a *signature* (e.g., a normalized representation of message text and metadata), then retains recent occurrences of that signature in a time window. If the count exceeds m_minRepeats, it emits a burst anomaly with representative samples and then trims its internal deque to avoid emitting on every subsequent repeated event.

3.6. Rare IP detection (security-oriented heuristic)

The anomaly/IpFrequencyDetector extracts IP-like patterns from log messages and tracks counts per IP. When an IP's count is below a rarity threshold (or appears for the first time), it emits a low-severity anomaly indicating that a rare IP has been observed. This is useful for security-heavy logs (e.g., brute-force attempts) where the presence of novel IPs can be meaningful even without complex models.

3.7. Offline analyzers (whole-file aggregation)

In addition to streaming detectors, the system includes analyzers in `analysis/` that are executed after all entries have been processed:

- **FrequencyAnalyzer:** maintains counts per source and per message hash, and flags (i) sources whose total count is much larger than their moving average, and (ii) rare message hashes that occur fewer than a minimum threshold.
- **PatternAnalyzer:** tracks message sequence signatures (n-gram-like transitions) and emits descriptions for novel high-severity patterns and never-before-seen transitions.
- **TimeWindowAnalyzer:** buckets events into fixed time windows, computes error rates, and emits anomalies for error spikes and silence gaps between consecutive windows.

These analyzers are useful for producing summary insights in offline mode, but they do not require per-entry anomaly emission during the ingestion loop.

3.8. Performance optimization techniques in C++

The codebase prioritizes predictable memory use and low overhead:

- **Streaming I/O:** processing uses `std::getline` to avoid loading the entire file.
- **Bounded state:** deques used for sliding windows are capped by time-window eviction; burst detectors cap sample storage; statistical windows use a fixed maximum queue size for values.
- **Thread-safety for future scaling:** analyzers and detectors wrap updates in `std::mutex` locks, enabling future parallel ingestion designs without rewriting core logic.
- **No heavy dependencies:** JSON extraction is implemented with string scanning rather than an external JSON library, trading completeness for lightweight deployment.

When compiled with optimization flags (e.g., `-O2`), the tool is intended to sustain high throughput on commodity CPUs.

4. System Design

4.1. High-level system architecture

Figure 2 shows the overall structure of the tool from the perspective of major subsystems. This is the most appropriate abstraction for communicating "what talks to what" without going into per-class details.

4.2. Logical architecture (responsibilities)

The logical architecture decomposes the system by *responsibility* rather than by file layout:

- **Ingestion:** open file, stream lines, maintain parsing success/failure counters.
- **Normalization:** transform heterogeneous log formats into a unified `LogEntry`.
- **Detection:** produce `Anomaly` objects using independent detectors operating on bounded state.
- **Aggregation:** update report-level statistics (by level, by source, total processed).
- **Presentation:** emit user-facing console summaries and structured exports.

4.3. Module-level architecture (detailed)

Table 1 summarizes the primary modules and their roles. This view is helpful when extending the tool (e.g., adding new detectors, improving parsing, or integrating external dashboards).

Table 1. Module-level architecture and responsibilities.

Module	Responsibilities (selected)
`core/`	`LogEntry` (normalized events), `Anomaly` (type/severity/window/score), `Report` (aggregate stats + anomaly list).
`input/`	`LogParser` parses text/JSON logs; classifies malformed lines with explicit errors; optional `FileReader` abstraction for streaming.
`analysis/`	Whole-file analytics: top sources/messages, rare patterns, error-rate windows, novelty in sequences; emits anomaly descriptions after ingestion.
`anomaly/`	Streaming detectors with bounded state: spike ratios, Z-score outliers, burst repetition, rare IPs; rule-based detector scaffold with caching and hot-reload hooks.
`report/`	`ConsoleReporter` (readable output), `JsonReporter` and `CsvReporter` (structured exports), `ReportGenerator` (sorting/filtering).
`utils/`	Timestamp parsing/formatting, string escaping for JSON/CSV, logger with severity filtering, simple key-value config parsing.

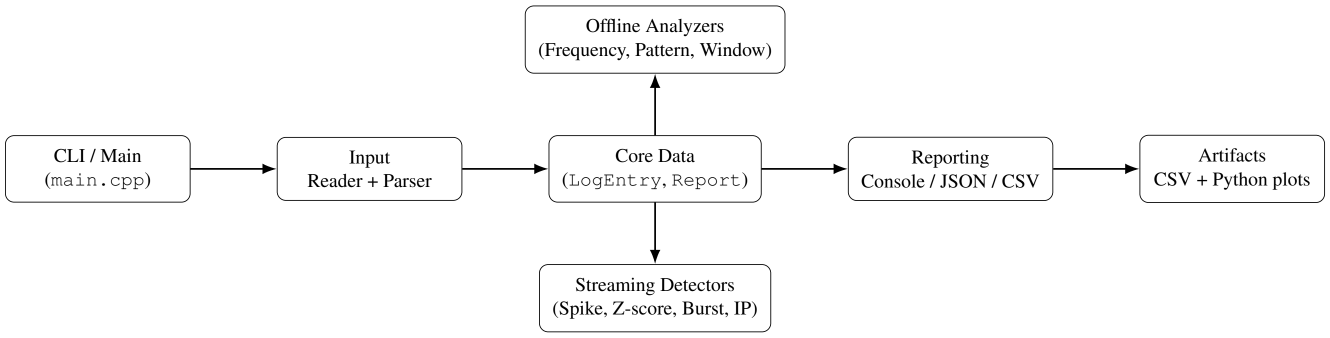
5. Evaluation & Benchmarking (Optional)

Context in the literature. While this tool uses transparent statistical detectors rather than learned models, the evaluation protocol can be compared against established log-anomaly baselines in the literature (e.g., experience reports and sequence-model approaches) [6, 7].

5.1. Performance evaluation methodology

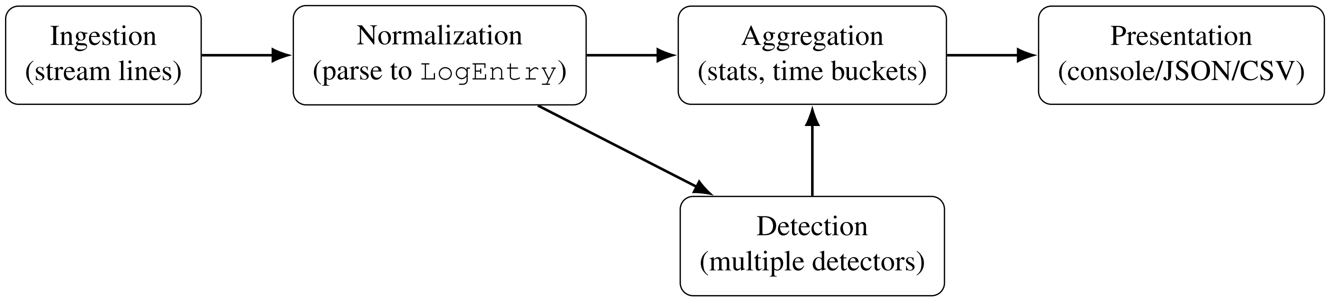
A typical evaluation of a GB-scale offline log tool should report:

- **Throughput** (logs/sec): parsed entries divided by wall-clock time.
- **Runtime** (ms or s): total processing time per file size.
- **Peak memory usage**: maximum RSS during processing; expected to be bounded by sliding-window sizes.



Offline Analyzers (Frequency, Pattern, Window)
CLI / Main (main.cpp)
Input Reader + Parser
Core Data (LogEntry, Report)
Reporting Console / JSON / CSV
Artifacts CSV + Python plots
Streaming Detectors (Spike, Z-score, Burst, IP)

Figure 2. High-level system architecture.



Ingestion (stream lines)
Normalization (parse to LogEntry)
Aggregation (stats, time buckets)
Presentation (console/JSON/CSV)
Detection (multiple detectors)

Figure 3. Logical architecture: responsibilities and information flow.

- **Scalability**: how throughput changes as file size grows (e.g., 5 MB → 1 GB+).

The current implementation already records per-run benchmarking fields (file size, wall time, parsed entries, malformed lines, and emitted anomalies) into an append-only CSV (`benchmark_runs.csv`), which can be aggregated across repeated runs.

5.2. Detection quality evaluation

To evaluate anomaly detection accuracy, a standard approach is to use labeled test logs (or synthetic logs with injected anomalies):

- **Precision:** fraction of detected anomalies that correspond to true injected/known anomalies.
- **Recall:** fraction of true anomalies that are successfully detected.
- **F1-score:** harmonic mean of precision and recall.

The project includes sample datasets (e.g., mixed-format logs, corrupted logs, and security-oriented logs) that can serve as a starting point for controlled experiments by injecting known spikes, bursts, timestamp irregularities, and rare tokens.

6. Figures and Diagram Placeholders

If you prefer to replace the TikZ figures with external images (e.g., diagrams made in draw.io), keep the following placeholders and insert your exported PDFs/PNGs:

- **High-Level System Architecture**: Figure 2.
- **Logical Architecture**: Figure 3.
- **Module-Level Architecture**: Table 1 (or replace with a diagram).
- **Data Flow Diagram**: Figure 1.
- **Benchmarking Graphs (optional)**: add a figure that plots throughput vs. file size using `benchmark_runs.csv`, and a time-series plot using `timeseries_per_minute.csv`.

7. Conclusion and recommended next steps

The project provides a practical, dependency-light baseline for offline log analysis on large files. Its design emphasizes streaming ingestion, bounded state via sliding windows, and a modular detector architecture with both online and offline components. Several extensions would increase correctness and coverage without sacrificing performance:

- Improve **source extraction** for common formats like `service - message` using regex-based parsing.
- Replace best-effort JSON extraction with an optional, feature-flagged JSON library for strict parsing when needed.
- Implement **rule-to-anomaly conversion** in `RuleBasedDetector::matchesToAnomalies` to activate rule matches in reporting.
- Extend timestamp parsing to support additional formats (timezone offsets, milliseconds).

- Add an explicit evaluation harness (synthetic injection + label files) to measure precision/recall consistently.

Build note. The repository contains example `g++` build commands using `-std=c++17` and `-O2`. If using CMake, ensure that subdirectories under `src/` (e.g., `src/anomaly/`, `src/analysis/`) are included in the build target.

References

[1] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3), 2009. 3

[2] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. 3

[3] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison–Wesley, 3rd edition, 1997. 3

[4] D. C. Montgomery. *Introduction to Statistical Quality Control*. Wiley, 7th edition, 2012.

[5] T. Bray. The JSON Data Interchange Format. RFC 8259, Internet Engineering Task Force (IETF), 2017. 1

[6] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2016. 4

[7] M. Du, F. Li, G. Zheng, and V. Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017. 4