CtrlB-Decompose: A Streaming Pipeline for Structural Log Compression and LLM-Ready Analysis

Adarsh Srivastava
CtrlB Explore Inc.
adarsh@ctrlb.ai

August 2026

Abstract

Modern software systems produce logs that are large in volume but low in structural entropy: millions of messages often reduce to a small number of recurring templates with changing timestamps, identifiers, addresses, durations, and status values. Feeding raw logs directly to analysts or language models wastes context on repeated structure and obscures anomalies. We present CtrlB-Decompose, an open-source Rust pipeline that converts raw log messages into structural patterns, typed variables, bounded statistical summaries, anomaly signals, and machine-readable output. The pipeline combines normalized token encoding and incremental Drain3 clustering with semantic variable classification, approximate statistics, severity scoring, and correlation signals. Each message is processed with a linear token scan, while clustering and statistical state remain bounded by configurable limits. We formalize the clustering stage as a bounded join-semilattice under wildcard generalization, show that the Drain merge step is exactly its join, and prove that the similarity threshold induces a hard *wildcard budget* limiting cluster generalization. CtrlB-Decompose supports terminal, Markdown, JSON, Rust, and WebAssembly interfaces. This paper isolates the decomposition and analysis pipeline from the larger CtrlB datalake and search system, describing its representation, streaming algorithm, memory model, output contract, and limitations.

1 Introduction

Logs are simultaneously one of the most valuable and least efficient sources of operational information. A production service can emit millions of messages during a short incident, yet most messages are generated by a small set of code paths. The messages differ in dynamic values such as timestamps, request identifiers, IP addresses, durations, and status codes, while their static structure is repeated.

This repetition creates a problem for both human investigation and automated reasoning. A person looking at a raw log file must mentally group related messages before identifying the important deviations. A language model receives many copies of the same template and spends context on repeated syntax instead of the variables and distributions that distinguish one event from another.

CtrlB-Decompose addresses this problem with a streaming transformation from raw messages to structural patterns and typed variable summaries. It is designed as a standalone command-line tool, a Rust library, and a browser-compatible WebAssembly module. It can read from standard input or a file and emit human-readable terminal output, compact Markdown optimized for language-model context, or structured JSON.

This paper focuses on the decomposition and analysis pipeline. The larger CtrlB research system additionally includes Parquet storage, file- and block-level search indexes, object-storage

coordination, and decoupled ingestion and query services. Those components are outside the scope of this paper.

The contributions of this work are:

1. a formal pattern-variable representation for semi-structured log messages (Section 3);
2. a two-stage streaming pipeline combining lightweight token normalization with incremental structural clustering (Section 4);
3. a formal model of the clustering stage, including a lattice-theoretic characterization of the Drain merge step and a threshold-induced bound on cluster generality (Section 5);
4. semantic typing and type-aware summaries for common log variables, with bounded-memory accumulation of quantiles, cardinalities, top values, temporal buckets, examples, and anomaly signals (Section 6); and
5. output modes that preserve the same analysis for terminal users, LLM context, and downstream programs (Section 8).

The implementation is available at `https://github.com/ctrlb-hq/ctrlb-decompose`.

2 Motivation and Design Goals

The design is guided by five goals.

Streaming operation. The tool should process a log file sequentially without loading the complete corpus into memory. This makes it suitable for large files and piped operational workflows.

Structural reduction. Repeated messages should be represented by a small number of patterns rather than reproduced line by line.

Typed context. A duration should not be treated as an opaque string when it can support percentile queries. An IPv4 address should be distinguishable from a UUID, an integer, or a low-cardinality status value.

Bounded state. Statistics and examples must have explicit memory bounds. A log analyzer that prevents context overflow by accumulating an unbounded in-memory representation has only moved the problem.

Multiple consumers. The same analysis should be useful at a terminal, in an LLM prompt, and in an automated pipeline without requiring separate parsers.

3 Representation

Let $L = \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$ be a sequence of log messages. The conceptual decomposition maps each message to a pattern-variable pair:

$$D(\ell_i) = (p_i, V_i), \tag{1}$$

where p_i is a structural pattern containing typed placeholders and

$$V_i = \langle v_1^{(i)}, v_2^{(i)}, \ldots, v_{k_i}^{(i)} \rangle \tag{2}$$

is the ordered sequence of extracted variable values.

For example, the following messages:

```
2026-04-12T10:15:03Z ERROR request from 10.0.1.15 completed in 45ms status=500
2026-04-12T10:15:04Z ERROR request from 10.0.1.22 completed in 61ms status=500
```

share a representation similar to:

```
<TS> ERROR request from <IPv4> completed in <Duration> status=<Integer>
```

with the values retained in their original positions. If the pattern and the complete ordered variable sequence are retained, the original message can be reconstructed. Summary output, however, is intentionally lossy with respect to the full corpus: it retains bounded examples and aggregate statistics rather than every original line.

The representation separates two sources of information:

$$H(L) = H(P) + H(V \mid P), \tag{3}$$

where $H(P)$ is the entropy of the pattern distribution and $H(V \mid P)$ is the conditional entropy of the variables given the pattern. Patterns are generally low-cardinality and repeat heavily; variables carry the changing information and can be summarized according to type. Writing P for the set of distinct patterns discovered over L, the structural reduction ratio is

$$\rho(L) = \frac{|P|}{n}, \tag{4}$$

and the useful regime for this pipeline is $\rho \ll 1$. Section 10 discusses what happens when it is not.

Table 1 fixes the notation used in the rest of the paper.

4 Streaming Pipeline

The implementation is organized as a two-stage normalization and clustering pipeline followed by typed statistics and analysis. Algorithm 1 gives the per-message control flow.

4.1 Timestamp Extraction

The first stage recognizes common timestamp forms, including ISO 8601, Apache-style timestamps, syslog timestamps, and Unix epoch values. Recognized timestamps are normalized to a timestamp placeholder while the parsed value remains available for temporal statistics and bucketing.

4.2 CLP-Style Token Encoding

The tokenizer scans each message using configurable delimiters. Each token is tested with lightweight character-class predicates. Integers, floating-point values, IP addresses, UUIDs, hexadecimal identifiers, and dictionary-like values are replaced by compact placeholders; static tokens remain

Table 1: Notation.

Symbol	Meaning
ℓ	a raw log message
Σ	token alphabet after stage-one normalization
$*$	wildcard symbol, $* \notin \Sigma$
Σ_*	$\Sigma \cup \{*\}$
$s = \langle t_1, \ldots, t_m \rangle$	normalized token sequence of arity m
$T = \langle \theta_1, \ldots, \theta_m \rangle$	cluster template, $T \in \Sigma_*^m$
$W(T)$, $w(T)$	wildcard positions of T and their count
$\mathcal{L}(T)$	set of token sequences covered by T
σ	Drain similarity threshold, $\sigma \in (0, 1]$
d	parse-tree depth
τ_{child}	maximum children per internal node
τ_{leaf}	maximum clusters examined per leaf
$C_{\max}$	LRU capacity on active clusters
α	DDSketch relative accuracy

Algorithm 1 Per-message processing

```
function PROCESS(ℓ)
    (ℓ̂, ts) ← EXTRACTTIMESTAMP(ℓ)
    (s, V) ← TOKENIZE(ℓ̂)                    ▷ CLP-style, O(|ℓ̂|)
    c ← ASSIGNCLUSTER(s)                    ▷ Drain3, Algorithm 2
    V′ ← RECONCILE(V, T_c)                  ▷ align values to W(T_c)
    for all (j, v) ∈ V′ do
        τ ← CLASSIFYTYPE(v)
        UPDATESTATS(c, j, τ, v, ts)         ▷ sketches, top-k, reservoir
    end for
    UPDATESIGNALS(c, ts)                    ▷ severity, spikes, co-occurrence
end function
```

part of the pattern. The encoding strategy follows CLP [1], which separates a log message into a logtype and a sequence of encoded variables.

The classifier is evaluated in priority order so that a token matching a more specific type is not also classified as a generic hexadecimal or string value. For a token t of length $|t|$, recognition is $O(|t|)$. The total normalization work for a message is therefore linear in its byte length:

$$T_{\text{norm}}(\ell) = O(|\ell|). \tag{5}$$

No regular-expression engine, backtracking automaton, or full-corpus pass is required for this stage.

4.3 Incremental Drain3 Clustering

Normalization alone can produce patterns that differ only in the location or presentation of dynamic values. The second stage uses incremental Drain3-style clustering [2, 3] to group similar normalized logtypes. Drain maintains a bounded fixed-depth parse tree over observed structures and assigns an

Table 2: Variable classes used by the decomposition pipeline.

Type	Example	Use
IPv4/IPv6	`10.0.1.15`	Address counts and top values
UUID	`550e8400-e29b-...`	Identifier cardinality
Duration	`45ms`, `3.2s`	Quantiles and anomalies
Integer	`500`	Counts, ranges, and distributions
Float	`3.14`	Quantiles and numeric detection
HexID	`0x1a2b3c`	Identifier statistics
Enum	`ERROR`, `INFO`	Low-cardinality distributions
Timestamp	RFC 3339 value	Temporal buckets
String	Other values	Fallback representation

incoming message to an existing cluster when its token similarity exceeds a configurable threshold. Divergent positions become wildcards.

This stage is streaming rather than batch: a message is assigned as it arrives, and the cluster state is updated incrementally. The implementation applies LRU eviction to bound the number of active clusters. The default repository configuration limits the active cluster state to approximately $C_{\max} = 10{,}000$ entries; deployments can adjust this according to workload diversity and memory budget.

The distinction between the two stages is important. Token normalization is a per-message linear scan with no cross-message state. Clustering is incremental and maintains bounded state across messages. Section 5 models the second stage precisely.

4.4 Variable Reconciliation and Typing

After a cluster is selected, the extracted values are reconciled with the wildcard positions in the cluster template. Variables are classified into the semantic types used by the statistical layer, listed in Table 2.

The type is part of the analysis contract, not merely a display annotation. It determines which sketches, encoders, anomaly checks, and output fields are meaningful for a variable.

5 Formal Model of the Clustering Stage

The clustering stage is the component whose behaviour is least obvious from its description, and the component whose parameters most directly control both memory and correctness. This section models it precisely. Unless otherwise noted, the model describes Drain [2] as implemented in Drain3 [3], restricted to the configuration used by CtrlB-Decompose.

5.1 Templates and the Generalization Semilattice

Definition 5.1 (Token sequence)**.** Stage one maps a raw message ℓ to a token sequence $s = \langle t_1, \ldots, t_m \rangle \in \Sigma^m$, where Σ contains both literal tokens and the type placeholders of Table 2. We call $m = |s|$ the *arity* of s.

Definition 5.2 (Template, coverage, language)**.** A *template* of arity m is a sequence $T = \langle \theta_1, \ldots, \theta_m \rangle \in \Sigma_*^m$. Its wildcard set is $W(T) = \{\, j : \theta_j = * \,\}$ with cardinality $w(T) = |W(T)|$. The

template T *covers* s, written $s \models T$, iff

$$\forall j \in \{1, \ldots, m\} : \quad \theta_j = * \ \vee \ t_j = \theta_j.$$

The *language* of T is $\mathcal{L}(T) = \{\, s \in \Sigma^m : s \models T \,\}$, and $|\mathcal{L}(T)| = |\Sigma|^{\,w(T)}$.

Definition 5.3 (Generalization order)**.** For $T, T' \in \Sigma_*^m$, write $T \sqsubseteq T'$ iff for every j either $\theta'_j = *$ or $\theta_j = \theta'_j$. Read T' as *at least as general as* T.

Proposition 5.4 (Semilattice structure)**.** *Let $|\Sigma| \geq 2$. Then $T \sqsubseteq T'$ iff $\mathcal{L}(T) \subseteq \mathcal{L}(T')$, and $(\Sigma_*^m, \sqsubseteq)$ is a bounded join-semilattice with top element $\top_m = \langle *, \ldots, * \rangle$ and positionwise join*

$$(T \vee T')_j = \begin{cases} \theta_j & \text{if } \theta_j = \theta'_j, \\ * & \text{otherwise.} \end{cases} \tag{6}$$

Proof. Each coordinate carries the flat order in which every element of Σ is below $*$ and elements of Σ are pairwise incomparable. This is a join-semilattice with top $*$, and its binary join is exactly Equation (6). The product of join-semilattices ordered componentwise is a join-semilattice with componentwise join, giving the second claim.

For the first claim, the forward direction is immediate: if $T \sqsubseteq T'$ and $s \models T$, then at every j with $\theta'_j \neq *$ we have $\theta_j = \theta'_j$, and $s \models T$ forces $t_j = \theta_j$, hence $s \models T'$. For the converse, suppose $T \not\sqsubseteq T'$, so some j has $\theta'_j \neq *$ and $\theta_j \neq \theta'_j$. If $\theta_j \in \Sigma$, any $s \in \mathcal{L}(T)$ has $t_j = \theta_j \neq \theta'_j$, so $s \notin \mathcal{L}(T')$. If $\theta_j = *$, then because $|\Sigma| \geq 2$ we may choose $s \in \mathcal{L}(T)$ with $t_j \neq \theta'_j$, and again $s \notin \mathcal{L}(T')$. Either way $\mathcal{L}(T) \not\subseteq \mathcal{L}(T')$. □

Let $\iota : \Sigma^m \to \Sigma_*^m$ be the identity embedding that reads a concrete token sequence as a wildcard-free template. The image of ι is exactly the set of minimal elements of Σ_*^m, and $\mathcal{L}(\iota(s)) = \{s\}$.

5.2 Routing: the Fixed-Depth Parse Tree

Drain avoids scanning all clusters by routing each message to a leaf through a tree of fixed depth d. The routing key is derived from the arity and the leading tokens.

Definition 5.5 (Digit mask)**.**

$$\kappa(t) = \begin{cases} * & \text{if } t \text{ contains a decimal digit or is a type placeholder,} \\ t & \text{otherwise.} \end{cases}$$

Definition 5.6 (Routing path)**.** The path of s is

$$\pi(s) = \langle\, m,\ \kappa_{v_1}(t_1),\ \kappa_{v_2}(t_2),\ \ldots,\ \kappa_{v_{d-2}}(t_{d-2}) \,\rangle, \tag{7}$$

where v_i is the node reached after $i - 1$ token layers and the node-local mask

$$\kappa_v(t) = \begin{cases} \kappa(t) & \text{if } \kappa(t) \in \mathrm{ch}(v) \ \vee \ |\mathrm{ch}(v)| < \tau_{\text{child}}, \\ * & \text{otherwise} \end{cases} \tag{8}$$

routes an overflowing token to the reserved wildcard child. If $m < d - 2$ the path terminates early at the last available token.

Two consequences follow directly and are worth stating because they are frequent sources of surprise in production.

Proposition 5.7 (Arity is a hard partition)**.** *If $|s| \neq |s'|$ then s and s' never share a cluster, regardless of σ.*

Proof. The first component of π is the arity, so the two messages reach disjoint subtrees. Every template has a fixed arity and simSeq is defined only between sequences of equal arity. □

Proposition 5.8 (Tree size)**.** *With Λ distinct observed arities, the number of internal nodes is at most*

$$1 + \Lambda \cdot \sum_{i=0}^{d-3} \tau_{\text{child}}^{i} \;=\; O\Big(\Lambda\, \tau_{\text{child}}^{d-2}\Big)\,,$$

and the depth is exactly d irrespective of the number of messages processed.

Proposition 5.7 explains why stage-one normalization must be arity-stable. A normalizer that emits a different token count for the same code path, for instance by splitting an address on a delimiter in one case and not another, fragments a single logical template across several clusters and no choice of σ can repair it.

5.3 Match Score and Selection

Definition 5.9 (Sequence similarity)**.** For $s \in \Sigma^m$ and $T \in \Sigma_*^m$,

$$\text{simSeq}(s, T) \;=\; \frac{1}{m} \sum_{j=1}^{m} \mathbf{1}[\theta_j \neq *] \cdot \mathbf{1}[t_j = \theta_j]\,. \tag{9}$$

Wildcard positions contribute zero. This is the choice that makes the rest of this section work, and Remark 5.17 shows what breaks without it.

Definition 5.10 (Selection rule)**.** Let $\mathcal{C}(\pi(s))$ be the cluster list at the leaf reached by s. The candidate is

$$c^\star \;=\; \operatorname*{arg\,max}_{c \in \mathcal{C}(\pi(s))} \big(\,\text{simSeq}(s, T_c),\; w(T_c)\,\big), \tag{10}$$

maximized lexicographically. The message is assigned to $c^\star$ iff $\text{simSeq}(s, T_{c^\star}) \geq \sigma$; otherwise a new cluster is created with template $\iota(s)$.

The secondary key breaks ties toward the more general template, on the reasoning that a position already treated as variable is more likely to be genuinely variable than one that has merely not yet been contradicted.

5.4 The Merge Step is a Join

Lemma 5.11 (Merge is least common generalization)**.** *The Drain template update*

$$\theta_j \leftarrow \begin{cases} \theta_j & \text{if } t_j = \theta_j, \\ * & \text{otherwise} \end{cases} \tag{11}$$

is exactly $T_c \leftarrow T_c \vee \iota(s)$ in the semilattice of Proposition 5.4. Consequently, if A_c is the multiset of messages assigned to cluster c so far, then

$$T_c \;=\; \bigvee_{s \in A_c} \iota(s), \tag{12}$$

the least element of Σ_*^m that covers every message assigned to c.

Proof. The update rule is Equation (6) instantiated with $T' = \iota(s)$, since $\iota(s)$ has no wildcards. The closed form follows by induction on $|A_c|$: the base case is $T_c = \iota(s_1)$ at creation, and the inductive step uses associativity of $\vee$. Leastness is the defining property of the join. □

Corollary 5.12 (Monotone generalization)**.** *T_c is non-decreasing in $\sqsubseteq$ over the lifetime of the cluster, $w(T_c)$ is non-decreasing, and a cluster of arity m undergoes at most m template mutations in total. Every assigned message is covered by the cluster's final template.*

Corollary 5.12 is what makes the structure safe to expose downstream: a cluster's template never becomes *less* representative of its own history, so a template read at the end of a run is a sound description of every line that reached it.

5.5 The Wildcard Budget

The similarity threshold is usually presented as a tuning knob. It is stronger than that: it caps how general any cluster can ever become.

Proposition 5.13 (Score ceiling)**.** *For all s and T of arity m,* $\mathrm{simSeq}(s, T) \le 1 - w(T)/m$.

Proof. By Equation (9) the summand vanishes at each of the $w(T)$ wildcard positions and is at most 1 elsewhere. □

Theorem 5.14 (Wildcard budget invariant)**.** *Under the scoring rule of Equation (9), every cluster template satisfies, at all times,*

$$w(T_c) \;\le\; B(m) \;\coloneqq\; \lfloor m\,(1-\sigma) \rfloor. \tag{13}$$

Proof. Induction over the assignments to c. At creation $T_c = \iota(s)$ and $w(T_c) = 0 \le B(m)$. Suppose the invariant holds and s is accepted into c. Let

$$D = \left|\{\, j : \theta_j \neq * \;\wedge\; t_j \neq \theta_j \,\}\right|$$

be the number of concrete disagreements. Every position is exactly one of: a wildcard, a concrete agreement, or a concrete disagreement, so the numerator of Equation (9) equals $m - w(T_c) - D$. Acceptance requires

$$\frac{m - w(T_c) - D}{m} \;\ge\; \sigma \quad\Longleftrightarrow\quad w(T_c) + D \;\le\; m(1-\sigma).$$

By Lemma 5.11 the update sets exactly the D disagreeing positions to $*$, so the new count is $w(T_c') = w(T_c) + D \le m(1-\sigma)$. Since w is an integer, $w(T_c') \le B(m)$. □

Corollary 5.15 (Bounded over-merging)**.** *Any two messages assigned to the same cluster agree at every non-wildcard position of its final template, and therefore differ in at most $B(m)$ token positions. Equivalently, the cluster's Hamming diameter is at most $B(m)$ and*

$$|\mathcal{L}(T_c)| \;\le\; |\Sigma|^{\,B(m)}.$$

Proof. By Lemma 5.11 every assigned s satisfies $\iota(s) \sqsubseteq T_c$, so all assigned messages agree outside $W(T_c)$. Apply Theorem 5.14. □

Corollary 5.16 (Choosing σ from a tolerance)**.** *To admit at most δ variable positions in messages of arity m, it suffices to set*

$$\sigma \;\ge\; 1 - \frac{\delta}{m}.$$

Table 3: Wildcard budget $B(m) = \lfloor m(1-\sigma) \rfloor$: the maximum number of positions a cluster of arity m may generalize.

	arity m			
σ	8	16	32	64
0.4	4	9	19	38
0.5	4	8	16	32
0.7	2	4	9	19
0.9	0	1	3	6

Algorithm 2 Cluster assignment and update

```
function AssignCluster(s)
    v ← Descend(π(s))                          ▷ O(d) node steps, Definition 5.6
    (c*, best) ← (⊥, −∞)
    for all c ∈ C(v) do                        ▷ at most τ_leaf clusters
        score ← (simSeq(s, T_c), w(T_c))
        if score >_lex best then
            (c*, best) ← (c, score)
        end if
    end for
    if c* ≠ ⊥ and best_1 ≥ σ then
        T_c* ← T_c* ∨ ι(s)                     ▷ join, Lemma 5.11
        Touch(c*)                              ▷ LRU
        return c*
    end if
    return NewCluster(v, ι(s))                 ▷ evict if |C| = C_max
end function
```

Table 3 instantiates this. The practical reading is that a low threshold is much more permissive on long messages than on short ones, because the budget scales with arity. A threshold chosen against 10-token messages behaves very differently on 40-token messages from the same stream, which is the usual explanation for otherwise unrelated lines sharing a cluster.

Remark 5.17 (The parameter-credit variant). Drain3 offers a scoring variant in which wildcard positions count as matches, giving $\text{simSeq}'(s, T) = \text{simSeq}(s, T) + w(T)/m$. Under that rule Proposition 5.13 fails, Theorem 5.14 fails with it, and $\top_m$ scores 1 against every message of arity m. A cluster that drifts far enough then absorbs all subsequent traffic of its arity, and the partition collapses. CtrlB-Decompose therefore scores without parameter credit.

Remark 5.18 (Why eviction is still needed). Theorem 5.14 bounds the generality of each cluster but not their number. Distinct structures arrive indefinitely in a long-lived stream, so the cluster count is bounded only by the LRU capacity $C_{\max}$. Eviction trades recall on rare recurring patterns for a hard memory ceiling; a pattern evicted and later re-seen reappears as a new cluster with a reset count.

5.6 Cost

Proposition 5.19 (Per-message cost)**.** *Assignment and update of a message ℓ of arity m costs*

$$O(|\ell| + \tau_{\text{leaf}} \cdot m)$$

time, independent of the number of messages processed so far.

Proof. Computing $\pi(s)$ evaluates κ on at most $d-2$ tokens, bounded by $O(|\ell|)$, and performs $O(d)$ hash lookups with d constant. The leaf scan evaluates Equation (9) on at most τ_{leaf} templates at $O(m)$ each. The join in Lemma 5.11 is $O(m)$. No step consults state proportional to n. □

Corollary 5.20. *Processing L costs $O(\sum_i |\ell_i| + n\,\tau_{\text{leaf}}\,m_{\max})$ with resident state $O(\Lambda\,\tau_{\text{child}}^{d-2} + C_{\max}\,m_{\max})$, both independent of the corpus size held in memory at any instant.*

5.7 Interaction With Stage One

The two stages are not independent, and the direction of the coupling matters.

Stage one collapses high-cardinality tokens onto a small set of placeholders, which shrinks the effective alphabet seen by stage two. Two messages from the same code path therefore agree at more positions after normalization than before, raising simSeq and consuming less of the wildcard budget of Theorem 5.14 for the same σ. The same collapse makes κ nearly redundant on the routing prefix, since digit-bearing tokens have already become placeholders; the routing key degenerates gracefully into a type signature over the leading tokens.

The risk runs the other way. Because ι is arity-preserving and arity is a hard partition (Proposition 5.7), any normalization decision that changes token count for structurally identical inputs splits a template irrecoverably. Delimiter configuration is therefore a correctness parameter of the clustering stage, not only of the tokenizer.

6 Bounded Statistical Accumulation

The analyzer accumulates statistics per pattern and variable without retaining all raw values.

6.1 Quantiles

Numeric variables such as durations and response sizes are summarized with DDSketch [4], which provides quantiles with a *relative* error guarantee rather than a rank error guarantee. For a target accuracy α, values are bucketed on a logarithmic scale with ratio

$$\gamma = \frac{1+\alpha}{1-\alpha}, \qquad i(x) = \left\lceil \log_\gamma x \right\rceil, \tag{14}$$

and bucket i reports the representative $2\gamma^i/(\gamma+1)$. The estimate $\tilde{x}_q$ then satisfies

$$|\tilde{x}_q - x_q| \;\le\; \alpha\, x_q \tag{15}$$

for every quantile q. This is the property that matters for latency work: rank-error sketches can be arbitrarily wrong in relative terms on the heavy tail, which is exactly the region an operator reads. Storage is the number of occupied buckets, which grows logarithmically in the dynamic range of the data rather than linearly in the sample count.

Table 4: Bounded state components.

Component	Bound	Controlled by
Parse tree	$O(\Lambda\, \tau_{\text{child}}^{d-2})$	depth d, fan-out τ_{child}
Active clusters	$O(C_{\max}\, m_{\max})$	LRU capacity $C_{\max}$
Template width	$w(T_c) \leq B(m)$	threshold σ (Theorem 5.14)
Quantiles	occupied buckets, $O(\log_\gamma R)$	accuracy α, dynamic range R
Cardinality	$O(m_{\text{r}} \log\log N)$ bits	register count m_{r}
Examples	$O(k)$ lines per pattern	reservoir capacity k
Output	$O(\lvert P\rvert)$ records	emitted from summaries, not input

6.2 Cardinality

High-cardinality variables such as trace IDs, span IDs, and IP addresses use HyperLogLog [5]. With m_{r} registers the estimator has standard error approximately $1.04/\sqrt{m_{\text{r}}}$ at a cost of $O(m_{\text{r}} \log\log N)$ bits, giving a fixed footprint per variable instead of an unbounded set.

6.3 Top Values and Temporal Buckets

For low-cardinality or operationally important variables, the analyzer retains top values and counts. Timestamped events are assigned to temporal buckets to expose changes in frequency and behaviour over time.

6.4 Examples

The analyzer uses reservoir sampling to retain a uniform random sample of size k from an unbounded stream in $O(k)$ space, bounding the raw lines kept per pattern. This preserves useful context for investigation without allowing a high-volume pattern to dominate memory.

6.5 Memory Model

Table 4 summarizes the principal state components and their bounds.

This model is particularly important when the output is intended for an LLM. The system reduces the number of lines presented while also preventing its own intermediate representation from growing without limit.

7 Anomaly Detection and Correlation

Pattern counts provide a natural baseline for several lightweight signals. The analyzer can identify frequency spikes, error cascades, low-cardinality variables, bimodal numeric distributions, and clustered numeric anomalies.

Severity scoring begins with keyword-based classification, with terms such as `ERROR`, `WARN`, `INFO`, and `DEBUG` providing an initial ordering. The system then combines temporal co-occurrence, shared-variable relationships, and error-cascade signals to surface patterns that are more likely to be connected.

These signals are deliberately descriptive rather than prescriptive. They identify patterns and relationships for investigation; they do not claim to infer a root cause from log text alone.

8 Output Contract

The same internal representation supports three primary output modes:

Human ANSI terminal output with frequency bars, severity, variables, quantiles, and representative examples.

LLM Compact Markdown designed to reduce repetitive context while preserving patterns, typed variables, counts, anomalies, and relevant examples.

JSON Structured output for pipelines, tests, indexing, and downstream applications.

Typical command-line usage is:

```
cat server.log | ctrlb-decompose
ctrlb-decompose --llm app.log
ctrlb-decompose --json app.log
ctrlb-decompose --top 10 --context 3 app.log
```

The command reads from standard input when no file is supplied, allowing it to be composed with existing Unix tools. The project also includes a Claude Code plugin that installs and invokes the analyzer through natural-language requests.

9 Implementation

The implementation is written in Rust. The core can be used as a command-line binary or library, and the project includes a WebAssembly target for browser execution. Rust provides explicit control over allocation and binary representations while retaining portability across local developer environments and server-side pipelines.

The architecture separates parsing, clustering, statistics, anomaly analysis, and rendering. This allows the same parsed representation to feed human, LLM, and JSON renderers without repeating the log scan.

The repository is released under the MIT license and includes source code, tests, benchmarks, a browser target, and the Claude Code plugin integration:

https://github.com/ctrlb-hq/ctrlb-decompose

10 Limitations

The pipeline depends on the regularities of the input logs. If a log stream contains mostly unique messages then $\rho \to 1$ and pattern reduction is limited. Aggressive thresholds can merge semantically different messages within the bound of Corollary 5.15, while conservative thresholds produce more patterns and increase pressure on $C_{\max}$.

The guarantees of Section 5 are structural, not semantic. Theorem 5.14 bounds how far a template can generalize; it does not establish that the messages sharing a cluster came from the same code path. Nothing in the model prevents two unrelated call sites of equal arity from merging when they happen to agree on a σ fraction of their tokens.

Type classification is heuristic. A token that resembles an identifier may not carry identifier semantics, and a numeric token may be a code rather than a measurement. The fallback string type

is therefore necessary, and downstream users should treat inferred types as useful signals rather than a schema guarantee.

Summary output is not a substitute for retaining raw logs when exact forensic reconstruction is required. Reservoir samples and sketches preserve bounded evidence and distributions, but they do not preserve every original row. The lossless property applies to the complete pattern-plus-variable representation, not to every compact summary emitted by the analyzer.

Finally, the single-pass claim applies to per-message normalization and incremental clustering, as formalized in Proposition 5.19. Some larger systems built around the decomposition may perform additional sampling, indexing, compaction, or query-planning passes. Those operations should not be conflated with the message parser itself.

11 Conclusion

CtrlB-Decompose treats logs as semi-structured data rather than opaque strings. A streaming pipeline first separates recurring structure from variable values, then clusters related logtypes, assigns semantic types, accumulates bounded summaries, and emits representations for people, language models, and programs.

Modelling the clustering stage as a join-semilattice makes the parameters legible. The merge step is the least common generalization of a cluster's own history, the similarity threshold is a hard budget on generality rather than a soft preference, and the LRU capacity is the only parameter that trades recall for memory.

The practical consequence is not merely fewer lines. It is a different unit of analysis: patterns with distributions, variable semantics, representative examples, and anomaly signals. That unit is compact enough for large-scale investigation while retaining the information needed to decide which raw data deserves deeper inspection.

The decomposition pipeline is intentionally independent of the larger CtrlB datalake system. It can be used as a local command-line tool today, while the same structural representation can serve as a foundation for storage, search, and AI-assisted debugging systems.

Acknowledgments

This paper's decomposition model builds on prior work in compressed log processing, online log parsing, approximate quantiles, and cardinality estimation.

References

[1] K. Rodrigues, Y. Luo, and D. Yuan. CLP: Efficient and Scalable Search on Compressed Text Logs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, pages 183–198, 2021.

[2] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *Proceedings of the 24th IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[3] IBM Research Haifa and LogPAI. Drain3: A robust streaming log template miner based on the Drain algorithm. https://github.com/logpai/Drain3.

[4] C. Masson, J. E. Rim, and H. K. Lee. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proceedings of the VLDB Endowment*, 12(12):2195–2205, 2019.

[5] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the Conference on Analysis of Algorithms (AofA '07)*, pages 137–156, 2007.

[6] J. S. Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[7] CtrlB. ctrlb-decompose: LLM-ready reasoning surface over logs. https://github.com/ctrlb-hq/ctrlb-decompose.

[8] A. Srivastava. CtrlB: Elastic Full-Text Search Over Petabyte-Scale Logs on Object Storage. CtrlB Explore Inc., 2026.