arXiv:2601.00633v1 [cs.DB] 2 Jan 2026

KELP：通过演化分组树实现稳健的在线日志解析

Satyam Singh*StoneBuck Labs*

Sai Niranjan Ramachandran*StoneBuck Labs**

摘要

实时日志分析是现代基础设施可观测性的基石。然而，现有在线解析器在架构上并不适用于生产环境的动态变化。它们基于本质上静态的模板模型构建，因而极其脆弱：轻微的模式漂移就会悄无声息地破坏解析流水线，导致告警丢失和 运维负担。我们提出 **KELP** (**K**elp **E**volutionary **L**og **P**arser)，这是一种高吞吐量解析器，基于一种新颖的数据结构构建： 演化分组树。与依赖固定规则的启发式方法不同，KELP 将模板发现视为持续进行的在线聚类过程。随着日志不断到达，树结构随之演化，节点会根据变化的频率分布进行拆分、合并并重新评估根节点。要验证这种适应能力，需要一个能够模拟真实生产环境复杂性的数据集，但我们发现，标准基准测试依赖静态的、基于正则表达式的真实标注，无法反映这种复杂性。为支持严格评估，我们推出了一个新基准测试，旨在体现现代生产系统中的结构歧义。评估结果表明，在这个严格的数据集上，KELP 能够保持高准确率，而传统启发式方法则表现不佳，同时吞吐量不受影响。我们的代码和数据集可在以下地址获取：codeberg.org/stonebucklabs/kelp

1 引言

多租户云基础设施的有效管理，取决于能否将海量非结构化日志数据流提炼为结构化事件。对于站点可靠性工程师（SREs）而言，日志解析器不仅仅是一种工具；它还是异常检测和合规审计的第一道防线 [6, 11, 19]。

过去，日志被视为粗糙但大体稳定的半结构化消息。早期的解析系统假定分隔符、格式字符串和字段布局会在数月内保持不变。这些假设曾经合情合理：单体应用很少更改其日志模式；运维人员手动编写模板；下游分析流水线则可以依赖这些模式的持久性。

然而，当代云部署的面貌已截然不同。微服务经历持续部署、滚动升级、部分回滚、A/B测试、多版本并发、紧急补丁、流量切换、故障转移引发的行为突变，以及架构层面的重构。这些操作都会导致日志格式发生变化：字段出现或消失，标识符更换形式，字符串断裂，参数顺序改变，异常被包装，容器 ID 发生变化，整个模板族也会毫无预警地分化。

这导致了日益扩大的、如今已成为结构性的问题。一方面是**启发式解析器** [1, 5, 15, 21], , 它们吞吐量高，却依赖脆弱的静态深度规则。一次改变日志格式的软件更新就可能使解析逻辑失效，迫使人们手动更新规则。另一方面是**机器学习解析器** [7, 9, 10, 23], , 它们试图从训练数据集中学习语义结构，但其计算和资源开销比实时摄取流水线在每日 PB 级规模下所能承受的高出几个数量级。这些系统通常需要借助图形处理器或张量处理器进行推理，占用大量内存，并需要进行繁复的重新训练周期，因此无法用于必须在线解析日志、满足严格延迟约束且不牺牲吞吐量的环境。

协调这些约束从根本上说是一项极具挑战性的问题，其成因包括

1. 标识符、计数器、时间戳和路由标签等偶然变化通常具有较高基数，但在结构上并无意义。相比之下，真正的模式演

*本研究在 StoneBuck Labs 完成。作者同时隶属慕尼黑工业大学；该隶属关系与本研究无关。

化表现为新增字段、结构重排或消息分化，表面上可能与偶然变化相似，但从根本上改变了令牌分布。经典解析器无法可靠地区分这两种现象。

2. 生产环境解析器无法在格式变化时重新处理数百万条历史日志，也无法从头开始重新训练。每项决策都必须在摄取关键路径上作出，同时满足严格的延迟目标。

3. 过于激进的解析器会拆分模板，使下游分析受到噪声干扰。相反，过于保守的解析器则会合并不兼容的模式，导致特征提取结果受损、异常检测器失效、基数估计错误，以及审计轨迹产生误导。

4. 格式会发生漂移、分叉、回退和重新汇合。因此，解析器不仅必须支持结构扩展，也必须支持结构收缩。此外，生产环境中的标识符（哈希值、UUID、探针 ID、容器 ID、内部请求签名）会产生极其稀疏的分布。这些分布与结构字段相互作用，令基于分隔符或手工制定的规则难以奏效。

因此，在当前情境下，我们需要一种全新的视角。我们提出**KELP**，这是一个旨在弥合这一日益扩大的差距的系统。现有日志解析器将日志消息视为静态模板，或视为需要离线推断的语义序列；而 KELP 则基于一个不同的前提：大规模日志解析本质上是一个动态在线聚类问题。KELP 通过**演化分组树** （EGT）实现这一目标。EGT 是一种分层数据结构，能够根据列级冗余模式增量组织日志。传统模板树假设日志遵循固定分隔符或人工构建的层次结构，而 EGT 会随工作负载的变化而演化结构。每次插入都是轻量且幂等的操作，只更新局部树区域，无需重新解析历史日志，也无需启动代价高昂的重新训练周期。这样，即使服务快速迭代，KELP 也能维持稳定的解析模板。

为区分变化与结构，KELP 引入了实时**频率映射**，用于追踪每个节点和列中的令牌分布。这一机制赋予树“呼吸”的能力：当令牌基数上升，显示出行为演化的迹象时，树便会扩展，以创建或细化变量节点。当分布趋于集中时，树会将节点向上拉取，合并冗余模式并恢复结构规律性。这种双向演化是 KELP 鲁棒性的核心。它体现了系统的设计理念：解析逻辑必须以与产生日志的软件相同的时间尺度进行适应。通过将轻量级统计反馈直接嵌入数据结构，KELP 无需承担机器学习驱动的解析器所需的计算开销，便能提供同等的适应能力。

为了构建稳健且可扩展的解决方案，我们还需要重新审视基于现有基准测试进行评估的长期假设（第 5 节）[8, 22, 24]。受近期系统研究方法的启发 [15, 17, 20],，我们开发了一种新的基准测试，以反映站点可靠性工程师依赖解析器进行异常检测、可审计性和事后分析时的真实运行条件。

总而言之，我们的工作作出了以下贡献：

1. **进化分组树（EGT）**：一种新的在线解析数据结构，支持幂等写入、列级冗余跟踪和动态结构演化。

2. **鲁棒演化算法**：我们开发了拉取、根验证和重新评估机制，用于控制树的重构，使 KELP 能够实时适应模式漂移，同时保持稳定性。

3. **解析器鲁棒性新基准测试**：我们提出了一种纳入真实世界软件演化的评估方法，展示了现有系统在哪些方面会失效，以及 KELP 如何在模式漂移下保持准确率和吞吐量。

本文其余部分安排如下。第2 节介绍日志解析的背景并讨论相关工作。第3 节详细介绍 KELP 的设计，包括演化分组树、频率映射，以及拉取、根验证和重新评估算法，并附有 Rust 代码示例。第4 节介绍实现细节，评估方法和结果见第5 节，理论分析见第6节。最后，第7 节总结全文并展望未来方向。

2 背景与相关工作

日志解析问题最好从日志生成的逆过程来理解。在现代生产系统S中，开发人员会在代码中插入日志语句，以捕获运行时状态。日志语句通常由一个常量字符串字面量（即模板）和一组动态变量（即参数）组成。

2.1 日志解析抽象

形式化地，设 $\mathcal{L} = \{l_1, l2, \ldots, l_n\}$ 是由 S 发出的日志消息数据流。解析器P 是一个函数，它将日志消息l_i 映射为一个元组（T_i, V_i），其中 T_i 是静态模板标识符

而V_i 是从消息中提取出的有序动态参数列表。

例如，考虑一个记录服务连接的系统：

1. “已连接到内部服务的 8080 端口” 2. “已连接到外部服务的 443 端口”

简单的解析器可能会严格地将每个不同的字符串标记都视为重要内容，因此由于“内部”和“外部”之间的差异而生成两个不同的模板。然而，更通用的解析器则会构建出一个层次化模板：

$$T_{gen} = \text{“Connected to } \langle \text{var} \rangle \text{ service on port } \langle \text{var} \rangle \text{”}$$

这种层次化视角表明，日志模板并非扁平字符串，而是由令牌类别构成的有向无环图（DAG）或树。在这一层级结构中，静态前缀“已连接到”构成根节点，随后分支为“内部/外部”，再汇聚到“服务的端口”。

因此，在线日志解析的核心挑战在于：仅给定数据流$T\ \mathcal{L}$，在无法访问源代码的情况下，发现这种潜在结构，并在单次遍历中以最低延迟完成解析。

2.2 相关工作

日志解析领域主要由离线算法和静态启发式方法主导。

离线与静态启发式方法。 早期方法（如 SLCT） [16] 和 LFA [12]依赖离线的多遍频繁项集挖掘，并面临 $O(N^2)$ 复杂度问题。尽管较新的方法（如 Drain） [5] 和 Spell [3] 提高了吞吐量，但它们依赖于僵化的假设：Drain 使用固定深度树，在可变长度前缀下会产生碎片化；而 Spell 的最长公共子序列计算会在高多样性数据流中造成延迟峰值。更为关键的是，它们的解析逻辑仍然是静态的；如果没有人工干预，它们无法针对不断变化的格式进行结构化适应。

离线训练密集型机器学习方法。 深度学习模型（NuLog [13]）和基于大语言模型的解析器 [10] 提供了稳健的语义理解能力，但在实际运行中并不可行。它们需要耗费大量资源的离线训练阶段，以及开销高昂的推理过程（通常需要图形处理器），这与实时摄取流水线严格的延迟预算不相容。

两类方法共同面临的一种关键失效模式是概念漂移。当软件发生演化时（例如，错误消息附加了一个新字段），静态启发式方法会将该日志误分类为新模板，而机器学习模型则会遭遇分布偏移，需要重新训练。现有系统实际上将解析视为静态分类。KELP 通过将解析视为在线动态聚类来应对这一问题，根据数据流熵实时扩展和收缩模板树。

3 系统设计

3.1 系统概述

KELP 的高层架构如图 1 所示。该流程分为三个阶段：

1. **分组：** 传入的日志会按词元数量进行分词并分桶。这种优化也被 Drain 等系统采用 [5], ，它假设由同一日志语句生成的日志通常具有相同的长度。

2. **演化：** 在每个桶内，日志由 EGT 处理。KELP 会更新局部频率映射、验证树根，并将数据推送到叶节点。

3. **重构：** KELP 会定期（或按批次）触发重新评估 过程，以纠正结构漂移、合并分散的分支或提取新的静态列。

3.2 数据表示：频率映射

KELP 的一个核心洞见是，日志语句会始终将常量词元置于特定列位置。为此，我们为每个桶维护一个**频率映射** 。该映射会跟踪每个列索引处每个词元的基数。

考虑以下数据流：

1. “已连接到端口 8080 上的客户端 Sid” 2. “已连接到端口 8000 上的客户端 Luke”

频率映射记录了表 1. 中所示的状态。此映射为树提供了“真实值”：基数较低且频率较高的列（例如“已连接”“端口”）是静态节点的候选项，而基数较高的列（例如“Sid”“Luke”）则被识别为动态变量。

Col	令牌	计数
0	“已连接”	2
2	“客户端”	2
3	“Sid”	1
3	“Luke”	1
5	“端口”	2

表1：**频率映射状态。** 跟踪令牌分布，以区分静态模板部分与动态变量。

3.3 演化分组树（EGT）

EGT 是一种按公共子序列组织日志的层次结构。与固定深度树不同，EGT 具有可变深度和以下节点类型：

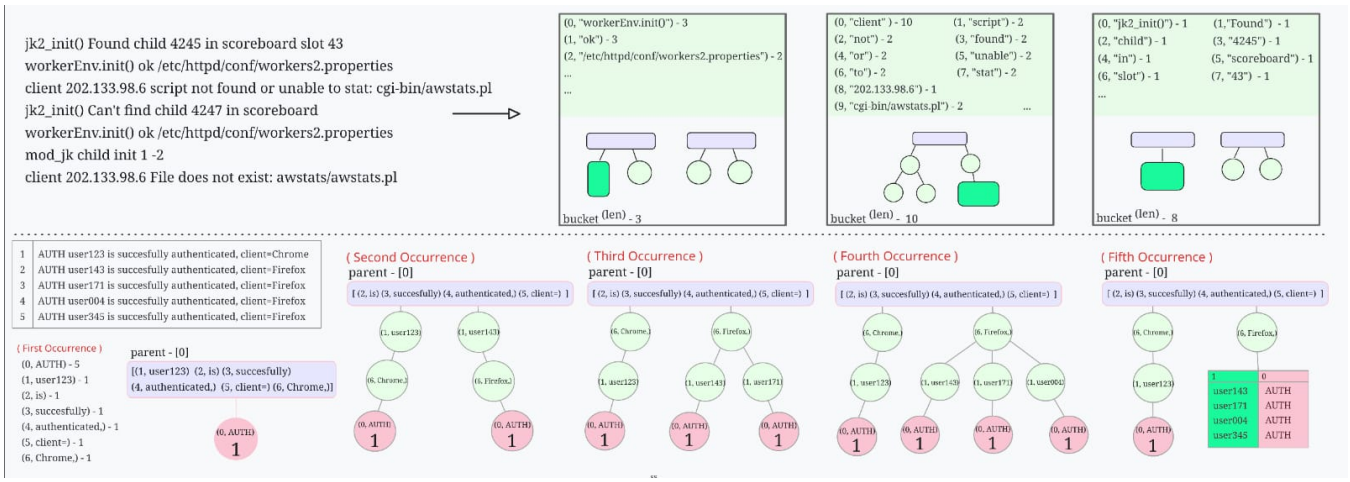


图1：**KELP 系统架构**。日志经长度分组后流入特定的桶。每个桶都维护一个频率映射和一棵演化分组树（EGT）。重新评估引擎会定期优化树结构。

1. **根节点：** 表示当前被视为静态的最长列子序列（依据频率映射判定）。2. **静态节点：** 表示特定列中的特定令牌值的内部节点（例如，列$_2$ = “客户端”）。3. **动态节点：** 作为原始日志行容器的叶节点。这些节点表示尚未被“拉取”到静态结构中的模糊或可变数据。

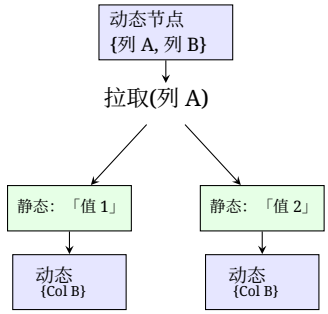


图2：**提取操作**。从动态节点提取一列会创建分支静态节点，从而细化模板结构。

该树保证从根节点到叶节点的任意路径都表示一种特定的日志模式。然而，由于数据是流式传入的，最初作为动态容器的节点之后可能会演化为一组静态分支。

3.4 基本操作：提取

树演化的基本操作是**提取**。这类似于决策树中的分裂操作。当动态节点累积数据时，KELP 会检查特定列。如果某列的基数较低，就会将其“提取”出来，把动态节点转换为具有分支的静态节点。

从算法角度看，拉取操作会从原始数据帧中提取一列，按唯一值对行索引分组，并为每个唯一令牌创建新的静态节点。此操作具有递归性：如果从层级结构深处提取某一列，该操作会逐层向上传递，重构树结构，使判别列显现出来。

```
函数 拉取(&可变自身, 列: usize) -> Vec<StaticNode> {
    如果自身.列() == 列 { 返回 vec![自身]; }

    // 从子节点递归拉取
    自身.子节点.pull(col).into_iter().map(|mut node| {
        // 将当前节点添加到拉取节点的子节点前面
        节点.子节点.前置添加(自身.列, 自身.value);
        node
    }).collect()
}
```

清单 1：递归拉取逻辑

3.5 流式处理摄入与根验证

随着日志不断到达，KELP 会直接将它们推入树中。然而，新数据会更新频率映射，这可能使当前的根定义失效。例如，一个最初被认为是静态的令牌（如“用户：管理员”），可能会在之后出现“用户：访客”时退化为变量。

KELP 会在每个批次上执行**根验证**。它会计算动态频率阈值，以区分信号与

噪声。我们通过分析前三位令牌频率的衰减，并采用自然对数启发式方法来确定该阈值：

$$Threshold = \left\lfloor \frac{e^{slope(top_3)} + 1}{2} \right\rfloor \tag{1}$$

如果当前根列相对于全局最大值低于此阈值，则会将其降级。验证_根函数（列表2）确保树的主干始终代表日志流中最稳定的特征。

```
函数验证_根（&可变自身，映射：&FrequencyMap）{
    // 按频率对列进行分组
    令链 = 分组_依据_频率（自身.值, 映射）；
    令阈值 = 计算_阈值(链);

    // 查找超过阈值的最长序列
    令 (最佳_频率，最佳_列) = 查找_最佳_根(链,
        阈值);

    如果 最佳_列 !=自身.当前_列 {
        自身.重新_评估 = 真; // 标记为重构
        自身.降级_列（最佳_列）；
    }
}
```

清单 2：动态根验证

3.6 树重构：重新评估

KELP 稳健性的核心是**重新评估**机制。将数据推入树中是一种乐观操作；随着时间推移，树结构可能变得次优。例如，一个“静态”节点可能会分裂成数千个子节点，这表明它实际上是一个高基数变量。

该重_新评估过程（清单3）会递归遍历树以恢复不变量：

- **折叠（泛化）：** 如果一组分支超过基数阈值（高熵），就会将它们合并回单个动态节点。这实际上是“学习”一个通配符。

- **拉取（特化）：** 如果动态节点包含基数（熵）较低的列，就会将其“拉取”出来，创建特定的静态分支。

```
fn reeval(branches , threshold) -> Children { // 基本情况：如果
branches.all_leaf(), 则合并叶节点 { return merge(branches); } // 找到
基数最小的列 let (col, words) = find_min_cardinality(branches); //
高熵 -> 转换为动态变量 if words.len() > threshold { return
Children::Dynamic( branches.into_iter() .map( |b| b.into_
dynamic()) .reduce( |a, b| a.append(b)) ); } // 低熵 -> 拉取列以形成静
态分支
```

```
    let拆分 = branches.pull(col);
    子节点::静态(split.map( |head, kids| {
        head.with_child(reeval(kids , threshold))
    }))
}
```

列表 3：树重新评估

3.7 内存管理：裁剪

为支持无限流，KELP 无法无限期存储历史原始数据。我们采用一种“遗忘”策略。当动态节点达到容量上限（例如，k 行），或达到指定时间窗口时，就会丢弃原始数据数据帧或将其“滚动更新”。结构知识（树节点）和统计知识（频率映射）会予以保留，从而确保解析器在内存不会无限增长的情况下仍能保持准确。

4 实现

我们使用 Rust 实现了 KELP（≈3,500 代码行数）。实现策略优先考虑内存紧凑性和指令缓存局部性，避免面向对象解析器常见的指针间接寻址开销。该系统架构依托三种紧密耦合的机制，旨在维持高吞吐量摄取。

4.1 通过 RleVec 实现列式压缩

生产日志的一个基本特征是存在大量连续冗余。在包含 10,000 个请求的数据流中，“状态码”列可能有 99% 的条目都是整数 200。将日志存储为 ‘Vec<Vec<String»‘ 的朴素实现会产生巨大的开销：一个 4 字节字符串需要 24 字节的向量头部，还需要进行堆分配，导致内存碎片化和较差的缓存局部性。

为缓解这一问题，KELP 实现了自定义的 RleVec（游程编码向量），作为‘DynamicN-ode’叶节点的底层存储。

```
结构体 RleRun<T> { len: usize, val: T }结构体
RleVec <T>{ inner: Vec<RleRun <T>>}
```

‘RleVec’采用密集型列式存储布局。推入新的日志令牌时，系统会检查向量末尾。如果该令牌与前一项匹配（通过指针比较），我们只需将‘usize’计数器加一。这使稳态日志的写入复杂度降低为一条算术指令。

此外，‘RleVec‘支持高效的“拆分”。当树重新评估算法(§5)确定必须拉取某一列时，可以对‘RleVec‘进行切片和重组，而无需深度复制底层令牌数据，只需操作轻量级的‘RleRun‘标头。

4.2 使用双向映射实现零拷贝驻留

字符串比较是日志解析中的主要开销。为将其从关键路径中消除，KELP 使用由‘Slab‘分配器支持的专用双向映射，实现了全局驻留层。

```
pubstruct GlobalFreqMap {ids: Slab<()>, // O(1) 槽位分配器pub map: BiMap<String, TokenValueRef >,pub col_freq: Vec<HashMap<TokenValue, usize>>,}
```

摄取路径： 当一行日志到达时，会对字符串进行哈希并在‘BiMap’中查找。若已存在，则返回一个‘TokenValueRef’（一个轻量级包装器，封装了‘Arc<usize>’）。若不存在，则只分配一次该字符串，将其插入‘Slab’中的空闲槽位，并建立映射。

解析路径： 完成令牌化后，所有内部树操作（即分支遍历、相等性检查和频率计数）都只对 8 字节整数（‘usize’）进行。这使热点路径中的数据结构完全容纳于 L1/L2 缓存中，相比‘strcmp’，比较速度可提升数个数量级。

重构路径： 在生成模板时，‘BiMap’支持 O(1) 反向查找，可根据整数 ID 重构原始字符串。

4.3 多态节点布局

传统解析器通常使用类层次结构（虚函数表）表示树节点，因此每一步遍历都会产生动态分派开销。KELP 利用 Rust 的‘枚举’创建内存安全的带标签联合体：

```
pub枚举 ChildEither {
    计数(usize),
    静态(Vec<StaticNode>),
    动态(DynamicNode),
}
```

这种结构将日志簇的生命周期直接编码到类型系统中：

1. **计数（零开销）：** 当模板完全匹配（没有差异）时，叶节点会折叠为一个单独的 usize 计数器。这是可能达到的最紧凑表示。

2. **动态（累积）：** 当新的模糊数据到达时，节点会转变为“动态”状态，使用 RleVec 缓冲原始数据，以便进行统计分析。

3. **静态（分支）：** 通过“拉取”算法确认差异后，节点会转变为“静态”状态，并创建显式分支。

这种设计确保树中的稳定区域只产生可忽略的内存开销，而复杂区域只需为必要的结构付出代价。

4.4 内存生命周期与垃圾回收

在流式系统中，活跃词汇表（IP 地址、请求标识符）的集合是无界的。只插入条目的驻留器最终会耗尽内存。KELP 实现了一种与“FrequencyMap”关联的引用计数机制。

当树执行“Trim”操作（丢弃旧日志以维持滑动窗口）时，会递减相关词元的引用计数。当某个词元在所有列中的频率降为零时，KELP 会回收其在“Slab”中的槽位，并从“BiMap”中移除相应条目。这确保了解析器的内存占用与模板的活动工作集成正比，而非与日志流的总历史记录量成正比。

5 评估

我们将 KELP 与三个基线进行评估： **Drain** [5], **Lo-gram** [1],和 **LogMine** [4]。我们将比较范围限定在这些启发式解析器，因为深度学习替代方案会带来高昂的推理开销和离线重新训练周期，无法满足实时日志摄取的严格延迟约束。

5.1 基准测试方法：零偏差协议

本研究的一项主要贡献，是指出现有日志解析研究在结构化方法上存在缺口。我们认为，标准基准测试，尤其是 Loghub [8, 24], , 存在真实标注泄漏，因此不适合用于评估在线解析器在流式生产环境中的稳健性。

真实标注泄漏。 Loghub 数据集通过半自动流程进行标注，而这些流程高度依赖特定领域的正则表达式（例如，专门匹配 IP 地址、UUID 或日期格式）。因此，“真实值”模板实际上与特定的令牌化规则隐性关联。在 Loghub 上评估解析器，往往考验的是正则表达式覆盖率，而非结构推断。采用与标注者相似预处理规则的解析器会获得虚高的分数；而那些试图从原始分布中学习结构的解析器，则会因未能识别出一些变量而受到惩罚——在小样本中，这些变量在统计上与静态文本无法区分。

为说明真实标注泄漏的严重性，请看以下来自 Loghub 中 Linux 数据集的日志条目：

日志：“身份验证失败；登录名= 用户标识符=0 有效用户标识符=0终端=无设备ssh 远程用户名=远程主机=207.243.167.114 用户=root”

真实模板数：“身份验证失败；登录名= 用户标识符=<*>有效用户标识符=<*>终端=无设备ssh 远程用户名=远程主机=<*> 用户=<*>”

真实值明确将 uid=0 标记为动态变量（<*>）。然而，在该数据集的特定范围内，此模板通常呈现**零方差**，uid 始终为 0。

对于统计型或分布式解析器而言，令牌 0 的熵为零；从统计角度看，它与“failure”等静态关键词无法区分。因此，任何纯粹依赖数据证据的解析器都必须将其归类为静态内容。Loghub 的真实标注并非源自数据集的分布，而是来自外部语义先验（即“UID”意味着变化这一人类知识）或预先编写的正则表达式规则。使用语义真实标注评估分布式解析器，会因为它们缺乏原始数据流中不存在的“预言机”知识而受到惩罚。

解决方案：合成注入。为了将解析算法的性能与其预处理启发式方法隔离开来，我们构建了一个“零知识”基准测试。我们的目标是创建一个数据集，使静态模板与动态变量之间的区别完全由词元基数定义，而不是由词元格式定义。

我们通过一个三阶段流水线构建了该基准测试：

1. **模板提取：**我们从 Loghub 中的 Apache、BSL 和 Linux 数据集中提取了 165–180 个独特的真实世界事件模板。这确保了结构复杂性（消息长度、词语位置）仍能代表生产系统。

2. **变量擦除：**我们从这些模板中剥离了原始动态变量，并用通用占位符替换它们。

3. **高熵注入：**我们通过向变量槽位注入高基数随机字符串，生成了合成日志流。

零偏差约束。至关重要的是，我们的评估协议禁止使用领域特定预处理（例如，“将所有数字替换为*”）。解析器必须处理原始的高熵日志。这迫使系统完全依赖分布信号，即频率、基数和位置，来发现潜在结构。这模拟了冷启动 [14]问题；在多租户软件即服务环境中，站点可靠性工程师无法为每个新租户的日志格式手动编写正则表达式。

我们生成了三个数据集（Synthetic-1、-2、-3），其变量复杂度逐步提高，以压力测试解析器区分信号（模板）与噪声（高基数变量）的能力。

5.2 结果

数据集 1：Synthetic-1（基线）。表2 展示了基线数据集上的性能。KELP 的执行时间（0.22 秒）与 Drain 相当，但准确率显著更高。Drain 识别出 1,091 个模板，而真实值为 180 个，因此 F1组准确率（FGA）较低，仅为 0.223。这证实了在没有正则表达式辅助的情况下，Drain 固定深度的启发式方法在高基数数据上会失效。KELP 则保持了结构一致性，FGA 达到 0.817。

解析器	时间（秒）	ID 模板	GT	GA	PA	FGA
Drain	0.40	1091	180	0.790	0.766	0.223
LogMine	37.11	232	180	0.863	0.870	0.752
Logram	0.65	544	180	0.778	0.593	0.384
KELP	**0.22**	**207**	180	**0.878**	**0.912**	**0.817**

表 2：**Synthetic-1 结果**。KELP 的速度与 Drain 相当，同时避免了模板爆炸。

数据集 2：Synthetic-2（中级）。在表 3中，KELP 的解析准确率接近完美（0.956）。LogMine 的准确率具有竞争力，但会带来 160×延迟惩罚（35.27 秒对比 0.22 秒），因此不适合流式处理。Drain 仍然存在过度划分问题（1057 个模板）。这表明，即使变量分布变得更加复杂，KELP 仍能保持精准度。

解析器	时间（秒）	ID 模板	GT	GA	PA	FGA
Drain	0.40	1057	165	0.812	0.803	0.219
LogMine	35.27	223	165	0.888	0.911	0.747
Logram	0.62	511	165	0.819	0.641	0.405
KELP	**0.22**	**182**	165	**0.905**	**0.956**	**0.853**

表 3：**Synthetic-2 结果**。 KELP 的解析准确率达到 0.956，且开销极小。

数据集 3：Synthetic-3（高熵）。表 4 代表了难度最高的等级。在此数据集上，LogMine 遭遇灾难性失败，仅识别出 1 个模板（准确率为 0.0），这可能是因为其聚类算法未能在高熵噪声中收敛。Drain 的表现依然稳定，但结果较为零散（753 个模板）。KELP 展现出很强的鲁棒性，识别出 181 个模板（真实模板数：165），且准确率较高（解析准确率为 0.915），验证了 EGT 的重新评估机制在最大熵条件下的稳定性。

解析器	时间（秒）	ID 模板	GT	GA	PA	FGA
Drain	0.39	753	165	0.816	0.806	0.294
LogMine	0.07	1	165	0.000	0.000	0.000
Logram	0.58	577	165	0.745	0.608	0.340
KELP	**0.21**	**181**	165	**0.899**	**0.915**	**0.867**

表 4：**Synthetic-3 结果**。LogMine 无法保持一致性；KELP 则能保持一致性。

在全部三个数据集上，KELP 是唯一能在生产流处理中兼顾吞吐量与准确率的可行方案。它处理日志的速度比 Drain 快 $\approx 2\times$，同时模板数量与真实值相差不超过 10-15%。这证实，KELP 解决了“模板爆炸”问题，而该问题

困扰着固定深度启发式方法，且无需承担更复杂聚类方法带来的延迟成本。

6 理论分析

本节假设读者对鞅理论有所了解 [18]，以便形式化收敛界限；主要关注系统实现和结果的读者可直接跳至第7节，不会影响阅读连贯性。为了严格理解 KELP 与嵌套解析器（如 Drain）的延迟特性，我们将模板发现过程建模为一个**随机停止时间问题**。我们运用鞅理论推导出完全识别一个含有N 个日志行的模板所需的期望日志行数，该模板具有m个动态变量。

6.1 问题表述

设一个日志模板T包含m 个动态令牌。解析器要识别动态令牌i，必须至少观察到 τ 个不同值（即分支阈值）。设 X_t 表示读取t行后，在某个令牌位置观察到的不同值数量。当所有m 个动态位置均满足$X_t \geq \tau$ 时，解析过程停止（收敛）。

我们对比两种架构模型：

1. **嵌套划分（例如 Drain）：** 解析器先根据令牌i 的值对数据集进行划分，再分析令牌$i+1$。令牌$i+1$的概率空间取决于由令牌 $1 \ldots i$定义的子集。

2. **并行识别（KELP）：** 解析器通过频率映射全局跟踪令牌分布，或按列独立跟踪。识别令牌i不会缩小令牌j的样本量。

6.2 嵌套划分的鞅分析

考虑一个包含m 个动态字段的模板，其中每个字段i 以均匀概率$p = 1/k$取k 个可能值之一。在嵌套解析器中，识别第d个动态令牌需要分析日志$S_d \subset \mathcal{L}$的一个子集。随机日志行落入正确路径并到达深度d 的概率为：

$$P(\text{reach depth } d) = \prod_{i=1}^{d} p_i = \left(\frac{1}{k}\right)^d \tag{2}$$

设 $Y_n^{(d)}$为达到深度 d 的行数，n 条总行数之后。这构成一个次鞅，其中 $\mathbb{E}[Y^{(d)}{}_{n+1} | Y_{(d)n} \ ] = Y_n^{(d)} + (1/k)^d$。根据 **可选停止定理** [18]，累积 $\mathbb{E}[N]$个样本所需的预期行数，在深度m 处为：

$$\mathbb{E}[N_{\text{nested}}] \approx \tau \cdot \left(\frac{1}{P(\text{reach depth } m)}\right) = \tau \cdot k^m \tag{3}$$

这揭示了相对于模板深度而言的**指数级复杂度**。随着动态复杂度（m）增加，数据需求会激增，这解释了 Drain 在高熵数据集（Synthetic-3）上出现的模板爆炸。

6.3 并行识别的鞅分析

在 KELP 中，频率映射独立跟踪列统计信息。将列i 识别为动态列，仅取决于全局边缘概率$p_i = 1/k$，而非条件路径。当所有m 个独立随机游走都越过阈值 τ 时，该过程完成。设N_i 为识别令牌i所需的时间。由于各分布相互独立：

$$\mathbb{E}[N_{\text{parallel}}] = \mathbb{E}[\max(N_1, \ldots, N_m)] \tag{4}$$

对于均匀概率， $\mathbb{E}[N_i] = \tau \cdot k$。将每个令牌的识别时间$N_i$建模为均值为$\mu = \tau k$的独立同分布指数随机变量。次序统计量中的标准结果 [2] 表明，最大值的期望与调和数成正比

$$\mathbb{E}[N_{\text{parallel}}] \approx \tau \cdot k \cdot H_m \approx \tau \cdot k \cdot \ln(m) \tag{5}$$

其中H_m 是m阶调和数。

这项分析证明了 KELP 设计的一项根本优势。嵌套解析器会受到 $O(k^m)$数据需求的困扰，需要指数级增长的日志才能在深层模板上收敛；而 KELP 的并行演化以 $O(k \ln m)$的复杂度扩展。图 3直观展示了这一差异，证实了 KELP 为何能在零偏差基准测试中保持稳定，而嵌套启发式方法却会失效。

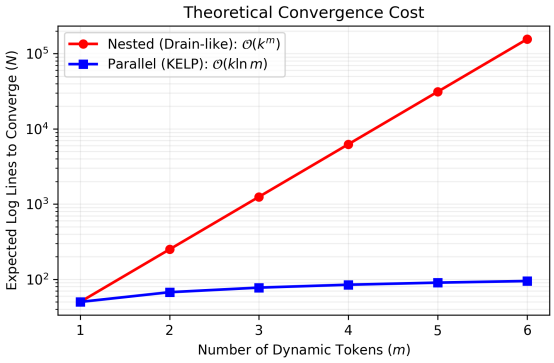


图3：**收敛复杂度**。理论比较识别一个模板所需的日志行数。嵌套方法（红色）随着模板复杂度（m）的增加，需要指数级的数据量，而 KELP 的并行方法（蓝色）则保持近线性。

7 结论与未来工作

本文指出，现代日志解析基础设施之所以脆弱，根源在于静态解析算法与动态生产环境之间存在根本性错配。我们推出了**KELP**这一系统，将解析重新构想为在线演化过程。通过将**演化分组树** 与实时频率分析相结合，KELP 使模板能够随着软件行为的变化自然扩展和收缩。我们的贡献有三方面：揭示了 Loghub 等标准基准测试中存在的真实值泄漏问题，因此有必要建立严格的**零偏差基准测试**；通过实证表明，KELP 在防止模板爆炸的同时，能够达到启发式吞吐量；并提供了理论上的鞅分析，证明 KELP 的收敛复杂度随模板深度呈对数增长（$\ln m$），而嵌套方法则会遭受指数级退化（k^m）。

尽管这些结果证明 KELP 是自主可观测性的坚实基础，我们的研究也揭示了若干值得进一步完善的方向。首先，我们依赖基于长度的分组，这可能会将包含多词变量的模板拆分到不同的桶中；我们计划通过引入跨桶协调层来解决这一局限。其次，贪心演化逻辑有时可能陷入局部最优，例如将低方差变量提升为“伪根节点”；我们计划通过分析静态节点链，回溯并纠正结构错位，以缓解这一问题。最后，我们设想通过对频率映射和演化分组树进行分片，将 KELP 的单节点效率扩展至分布式架构，从而实现超大规模环境下的水平扩展。

参考文献

[1] 戴和同、李恒、陈哲绍、尚伟毅和Tse-Hsun Chen。Logram：使用*n* 元语法字典进行高效日志解析。*IEEE*软件工程汇刊，48(3):879–892，2020年。

[2] Herbert A David 和 Haikady N Nagaraja。顺序统计。John Wiley & Sons，2004年。

[3] 杜敏和李飞飞。Spell：系统事件日志的流式解析。载于*2016*年*IEEE*第*16*届国际数据挖掘会议 *(ICDM)*，第859–864页。IEEE，2016年。

[4] Hossein Hamooni、Biplob Debnath、徐建武、张辉、姜国飞和Abdullah Mueen。Logmine：面向日志分析的快速模式识别。载于第*25*届*ACM*国际信息与知识管理会议论文集，第1573–1582页，2016年。

[5] Pinjia He、Jieming Zhu、郑子彬和Michael R Lyu。Drain：一种采用固定深度树的在线日志解析方法。载于*2017*年*IEEE*国际*Web*服务会议 *(ICWS)*，第33–40页。IEEE，2017年。

[6] Shilin He、Pinjia He、Zhuangbin Chen、杨天一、苏雨欣和Michael R Lyu。面向可靠性工程的自动化日志分析综述。*ACM*计算调查 *(CSUR)*，54(6):1–37，2021。

[7] Junjie Huang、Zhihan Jiang、Zhuangbin Chen 和 Michael R Lyu。Lunar：无监督的基于大语言模型的日志解析。*arXiv* 预印本 *arXiv:2406.07174*，2024年。

[8] Zhihan Jiang、Jinyang Liu、Junjie Huang、Yichen Li、Yintong Huo、Jiazhen Gu、Zhuangbin Chen、Jieming Zhu 和 Michael R Lyu。日志解析技术的大规模评估：我们进展到哪一步了？载于第*33*届 *ACM SIGSOFT*国际软件测试与分析研讨会论文集，第223–234页，2024年。

[9] Yudong Liu、Xu Zhang、Shilin He、Hongyu Zhang、Liqun Li、Yu Kang、Yong Xu、Minghua Ma、Qingwei Lin、Yingnong Dang 等。Uniparser：面向异构日志数据的统一日志解析器。载于*2022*年 *ACM*网络大会论文集，第1893–1901页，2022年。

[10] Zeyang Ma、An Ran Chen、Dong Jae Kim、Tse-Hsun Chen 和 Shaowei Wang。Llmparser：使用大语言模型进行日志解析的探索性研究。载于*IEEE/ACM* 第*46*届软件工程国际会议论文集，第1–13页，2024年。

[11] Haibo Mi、Huaimin Wang、Yangfan Zhou、Michael Rung-Tsong Lyu 和 Hua Cai。面向生产云计算系统的细粒度、无监督、可扩展性能诊断。*IEEE*并行与分布式系统汇刊，24(6):1245–1255，2013年。

[12] Meiyappan Nagappan 和 Mladen A Vouk。将日志行抽象为日志事件类型，以挖掘软件系统日志。载于*2010*年 *IEEE* 第七届软件仓库挖掘工作会议（*MSR 2010*），第114–117页。IEEE，2010年。

[13] Sasho Nedelkoski、Jasmin Bogatinovski、Alexander Acker、Jorge Cardoso 和 Odej Kao。自监督日志解析。载于欧洲机器学习与数据库知识发现联合会议，第122–138页。施普林格，2020年。

[14] Chu Qiao、Cong Wang、Zhenkai Zhang、Yuede Ji 和 Xing Gao。缓存辅助的多租户无服务器计算。*arXiv* 预印本 *arXiv:2408.00957*，2024年。

[15] Kirk Rodrigues、Yu Luo 和 Ding Yuan。 {CLP}: 在压缩文本日志上进行高效、可扩展的搜索。载于第 *15*届 {*USENIX*} 操作系统设计与实现研讨会（{*OSDI* } *21*），第183–198页，2021年。[16] Risto Vaarandi。一种用于从事件日志中挖掘模式的数据聚类算法。载于*IEEE* 第三届 *IP* 运维与管理研讨会论文集（*IPOM 2003*）（*IEEE* 目录号 *03EX764*），第119–126页。IEEE，2003年。[17] Abhishek Verma、Luis Pedrosa、Madhukar Korupolu、David Oppenheimer、Eric Tune 和 John Wilkes。谷歌使用 Borg 进行大规模集群管理。载于第十届欧洲计算机系统会议论文集，第1–17页，2015年。[18] David Williams。《鞅与概率》。剑桥大学出版社，1991年。[19] Wei Xu、Ling Huang、Armando Fox、David Patterson 和 Michael I Jordan。通过挖掘控制台日志检测大规模系统问题。载于*ACM SIGOPS* 第*22*届操作系统原理研讨会论文集，第117–132页，2009年。[20] Zongheng Yang、Zhanghao Wu、Michael Luo、Wei-Lin Chiang、Romil Bhardwaj、Woosuk Kwon、Siyuan Zhuang、Frank Sifei Luan、Gautam Mittal、Scott Shenker 等。 {SkyPilot}：用于天空计算的跨云代理。载于第*20*届 *USENIX* 网络系统设计与实现研讨会（*NSDI 23*），第437–455页，2023年。[21] Siyu Yu、Pinjia He、Ningjiang Chen 和 Yifan Wu。Brain：基于双向并行树的日志解析。*IEEE* 服务计算汇刊，16(5):3224–3237，2023年。[22] Chenbo Zhang、Wenying Xu、Jinbu Liu、Lu Zhang、Guiyang Liu、Jihong Guan、Qi Zhou 和 Shuigeng Zhou。Logbase：面向语义日志解析的大规模基准测试。*ACM* 软件工程会议录，2(ISSTA):2091–2112，2025年。[23] Aoxiao Zhong、Dengyao Mo、Guiyang Liu、Jinbu Liu、Qingda Lu、Qi Zhou、Jiesheng Wu、Quanzheng Li 和 Qingsong Wen。Logparser-llm：利用大语言模型提升日志解析效率。载于第*30*届 *ACM SIGKDD* 知识发现与数据挖掘会议论文集，第4559–4570页，2024年。[24] Jieming Zhu、Shilin He、Pinjia He、Jinyang Liu 和 Michael R Lyu。Loghub：用于人工智能驱动的日志分析的大型系统日志数据集合集。载于*2023*年 *IEEE* 第*34*届软件可靠性工程国际研讨会（*ISSRE*），第355–366页。IEEE，2023年。