Semantic-Aware Parsing for Security Logs

Julien Piet
UC Berkeley and Google

Vivian Fang
UC Berkeley

Rishi Khare
UC Berkeley

Scott Coull
Google

Vern Paxson
Corelight and UC Berkeley

Raluca Ada Popa
UC Berkeley

David Wagner
UC Berkeley

Abstract

Security logs are foundational to threat detection and post-incident analysis, yet are difficult to fully exploit. Analysts struggle to quickly and efficiently query and correlate log data due to the heterogeneity and lack of structure in real-world logs. Simple substring matching lacks the semantic interpretation needed for accurate retrieval while querying raw logs using large language models (LLMs) is impractical at scale and vulnerable to prompt injection attacks. Meanwhile, the standard practice of manually writing parsers to normalize the data is time-consuming and costly due to the long tail of log formats.

In this paper, we introduce Matryoshka, an end-to-end system leveraging LLMs to automatically generate semantically-aware structured log parsers without labeled examples or human intervention.[1] Matryoshka achieves this by directly inferring log syntax, variable naming, and normalization to common security-specific schemas (e.g., OCSF [44]) from unlabeled log line samples, then generating deterministic parsers and mapping rules that can be efficiently applied during data ingest. This approach provides analysts with semantically-rich data representations, facilitating rapid and precise log search without the traditional burden of manual parser construction.

We evaluate Matryoshka's capabilities through both established log parsing datasets (e.g., LogHub2.0 [73]), as well as new datasets curated to establish end-to-end performance on a realistic distribution of log types. Our experiments show that Matryoshka outperforms prior work on syntax parsing, while simultaneously matching the performance of human-generated parsers in both side-by-side comparisons and retrieval for security-relevant queries. These results demonstrate that Matryoshka significantly reduces manual effort by automatically extracting and organizing valuable security data, moving us closer to fully automated, AI-driven log analytics.

[1] Named after Matryoshka dolls because of the nested layers of parsing involved in extracting vital information from log messages.

1 Introduction

Security operations depend on analysts' ability to rapidly search and cross-correlate vast quantities of data to detect and respond to threats. However, this comes with a fundamental challenge: correlating events from multiple log sources (e.g., infrastructure operational events, network devices, applications, and security tools) that are heterogeneous, massive, and most often *semi-structured*. Each data source generates logs in different formats with varying levels of detail, and even within a given source these formats come in many variants and evolve over time.

Logs underpin two core security operations center (SOC) workflows: (i) interactive investigations that reconstruct a chain of events, and (ii) automated real-time detection using rules and machine learning. To leverage the sources of log data, security teams write parsers that convert logs into a structured schema, such as Google's Unified Data Model [15] (UDM) or the Open Cybersecurity Schema Framework [44] (OCSF). These schemas aim to unify the representation of log events across sources and provide a structured, normalized way to represent this data, but they are difficult to maintain at scale. For example, UDM exposes around 20,000 distinct attributes, and OCSF provides more than 50,000. For example, Google SecOps provides over 1,000 log parsers [16], and our partner commercial security incident and event management (SIEM) platform pushes 200 parser updates monthly, requiring more than 4,000 engineer-hours per month across over 30 domain experts.[2] Despite this effort, mappings are often ambiguous or incomplete, leading to poor detection and investigation outcomes. In our evaluation of end-to-end query performance (§ 5.3.3), for instance, human-written parsers achieved a precision of only 0.50 and recall of 0.48 across a diverse set of security-relevant queries due to the inherent difficulty in producing normalized schema mappings.

Modern AI seems ideally suited for this problem. Large language models (LLMs) can understand system logs [7, 29, 35], retrieval-augmented generation (RAG) systems can query nat-

[2] Enterprise partner anonymized for double-blind review.

ural language data [31], and AI systems can even generate SQL queries for structured data analysis [18, 22, 48, 66, 67]. Yet, existing methods in the literature come up short in terms of actual adoption. Real-world systems generate millions of log lines daily—far exceeding any language model's context window. Log data combines natural language, technical identifiers, and structured delimiters, making standard text embeddings inefficient and inaccurate. Even recent advances in querying unstructured data cannot keep pace with log generation rates; such approaches require embedding each new line individually and often resort to imprecise clustering for speed [9]. Additionally, analyzing logs with LLMs is vulnerable to prompt injection attacks, which attackers could potentially use to evade detection or introduce misleading information [47].

Security teams are thus left with two suboptimal approaches: either (1) write parsers by hand to convert them to a structured form, requiring substantial human resources but enabling efficient ingestion and search over logs, or (2) use AI search tools to retrieve log events from unstructured data formats, a slow and costly approach that can result in poor retrieval and fails to scale to security workload volumes, thereby sacrificing both ingestion speed and quality.

Instead, we propose Matryoshka, the first end-to-end system leveraging LLMs to automatically generate *semantically-aware structured log parsers*, offering fast ingestion and search with no human intervention. Matryoshka proceeds in three steps. First, a *syntax parser* captures the syntax of each log line and identifies variables. Second, a *semantic naming* step consistently names and clusters these extracted variables, producing a coherent schema suitable for structured queries. Finally, and optionally, Matryoshka can map the newly created fields to normalized security schemas, such as UDM or OCSF. At run time our generated parsers rely exclusively on static regular-expression matching—*not* LLM-based analysis—preventing prompt injection attacks and enabling efficient ingestion. The schema produced by the syntax parser and semantic naming provides analysts with a queryable structured dataset that supports fast searches even without normalization, and the final mapping step enables query and rule creation across log sources using normalized schemas.

	Manual Parsing	**Matryoshka**	**LLM Querying**
Setup Cost	High (manual parsers)	Low (generates parsers)	None (no parser)
Query Cost	Fast (structured data)	Fast (structured data)	Slow (LLM calls)
Security	Good (no prompt injection)	Good (no prompt injection)	Vulnerable (prompt injection)

Figure 1: Comparison of log parsing approaches; Matryoshka combines advantages of manual parsing and LLM querying.

Matryoshka addresses many challenges inherent in automating end-to-end parsing. Much past work focuses on constructing templates with wildcards to match variables [26, 27], but wildcards blur boundaries between adjacent fields and often over-capture. Matryoshka improves accuracy of variable extraction by generating regular expressions for each variable and applying a validation stage that explicitly checks for over-capture. Another core challenge lies in maintaining global coherence across multiple log sources and versions, which is unique to the end-to-end parsing and schema mapping problem. For example, instances of the same variable in different contexts should all be assigned the same field name. Past work has often focused on parsing data from a single source, and thus has not considered this challenge. Naively asking an LLM at each step of the process can produce inconsistencies, such as naming one field "`source_ip`" in one log line vs. "`src_ip`" in another. We avoid this by generating candidate names iteratively, then validating in batches to ensure coherence.

To evaluate Matryoshka, we use three datasets: (i) LogHub2.0 [73], a standard template generation benchmark, (ii) SecurityLogs, a new dataset we collect containing five common log types to evaluate schema mapping and end-to-end retrieval, and (iii) examples from 27 real-world enterprise log types from a commercial SIEM that capture the long tail of log formats seen in practice. Our experiments highlight the efficiency and performance of our approach at each stage in the parsing process. Matryoshka outperforms prior template generation approaches on the LogHub2.0 benchmark. We also evaluated Matryoshka on end-to-end retrieval using representative security queries, and achieve near-perfect performance on both new datasets when using custom-generated schemas (i.e., without mapping to a normalized schema). When normalizing to the UDM schema, Matryoshka improves on human-generated parsers, achieving a precision of 0.60 and recall of 0.60 on end-to-end queries, compared to 0.50 and 0.48 for human-written parsers. Finally, in side-by-side comparisons on the 27 enterprise logs, experts rated Matryoshka's mapping to UDM as achieving the same quality as human-written parsers. While these results show that mapping to normalized security taxonomies remains a significant challenge for both humans and machines, it also demonstrates our ability to achieve human-levels of parser quality automatically.

We proceed by examining preliminaries of log parsing in Section 2, followed by related work in Section 3. Section 4 provides a detailed description of Matryoshka's architecture and Section 5 presents our evaluation methodology and results demonstrating the efficacy of Matryoshka. Finally, Section 6 addresses the strengths and limitations of our approach. Our source code and benchmark are publicly available. [3]

[3] https://github.com/julien-piet/matryoshka

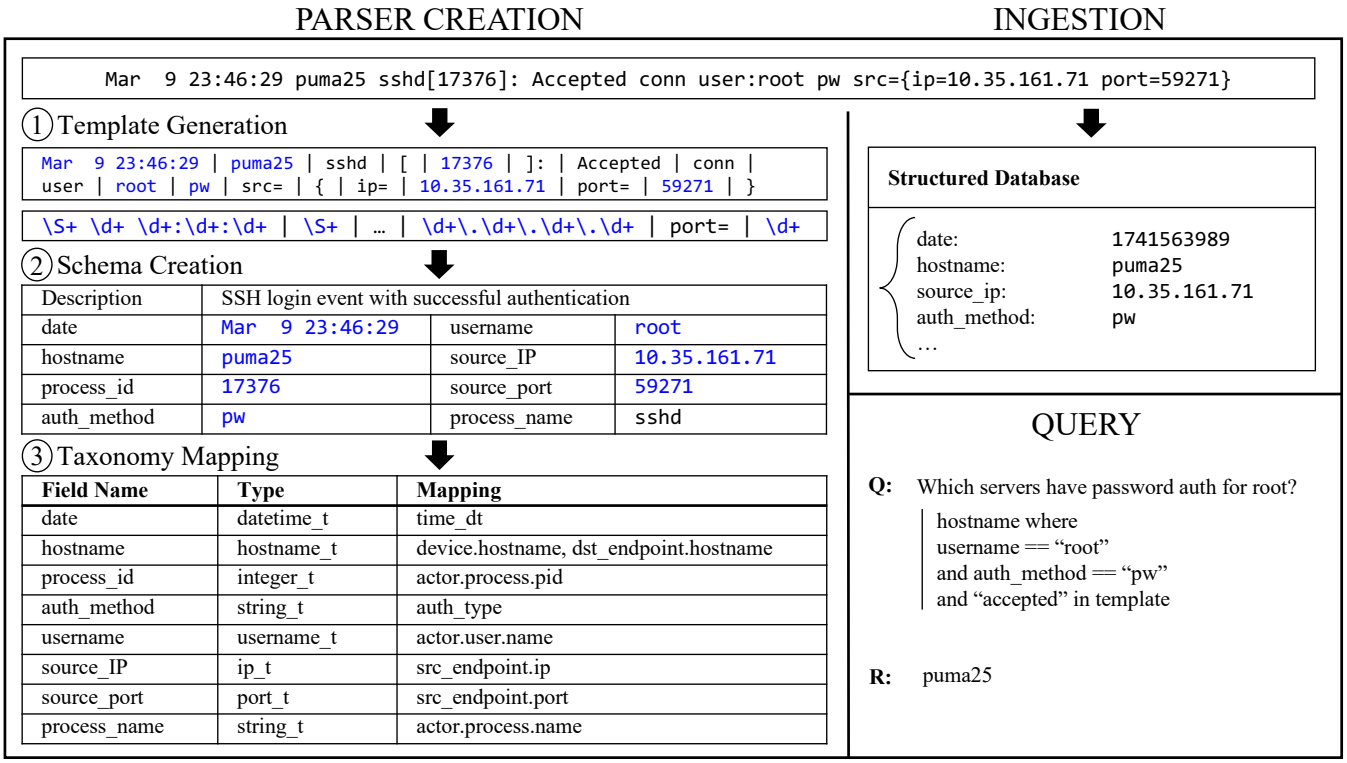

Figure 2: Parsers convert unstructured log lines (top) to structured data suitable for ingestion into a database (upper-right), in a sequence of three steps. The output format makes it easy to query logs for events that satisfy certain conditions (lower-right). Matryoshka creates such parsers by generating templates ①, crafting a schema ②, and mapping to a taxonomy ③.

2 Background

2.1 Anatomy of a parser

System logs are text-based messages that record a program's state and report events, from kernel messages to application activity. Log formats vary widely, even across versions of the same application, complicating parsing and queryability. While there is no standard design for log parsers, it is useful to distinguish three conceptual stages that any parser may incorporate: syntax parsing, semantic parsing, and schema mapping.

Syntax parsing. Programs emit log lines by interpolating variable entities into a fixed message template. The syntax of each line is thus determined by its template. Templates are defined as sequences of tokens, where each token is either a fixed string or a variable. We associate each variable with a regular expression. Step ① in Fig. 2 shows an example template that captures an SSH accepted connection event.

Many templates may share a common prefix. For instance, Linux system logs often share a prefix indicating the date, hostname, process name, and process ID, followed by program-specific content. Syntax parsing in Matryoshka relies on a tree of templates, where all templates in a subtree share a common prefix. This design helps parse log lines consistently, as repeated prefixes are parsed identically.

For example, consider the log line in Fig. 2, which highlights some of the challenges of real-world log parsing. Delimiters (`=`, `:`, `{}`) mixed with free text and nested key-value pairs make fields hard to separate; for example, the variable "`pw`" implies password authentication without stating it explicitly. Moreover, different clients (or even versions of the same client) may change the syntax or semantics of these log lines in subtle ways.

One possible template for the log line is "`<*> <*> sshd[<*>]: Accepted conn user:<*> <*> src={ip=<*> port=<*>}`", and it can be captured by the main branch in the parsing tree represented in Fig. 3. The start of the line (up to the process ID brackets) is present in all log lines, so it represents a branching point in the tree of templates. Each leaf represents a distinct template. Because the date and hostname are adjacent, parsing with wildcards would be unable to disambiguate where the date ends and hostname begins. This motivates the use of regular expressions to capture variables (e.g., "`\S+\s+\d+\s+\d+:\d+:\d+`" for the date) and enable unique disambiguation.

Semantic parsing. While templates capture a line's syntax, they provide no information about its meaning. Therefore,

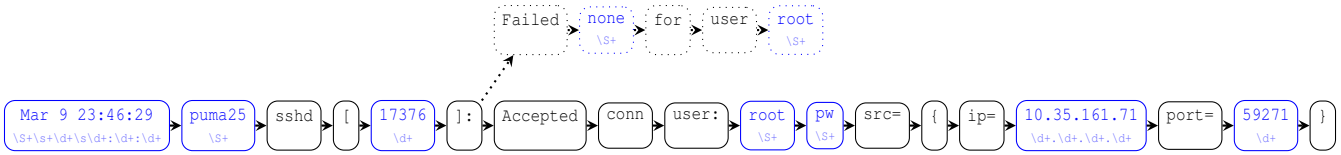

Figure 3: Example parsing tree, with two templates that share a common prefix. Each node/token represents either a string constant (black) or a variable (with associated regular expression; blue), and each leaf corresponds to a template (given by the path from the root to that leaf).

the next parsing stage maps each template to a schema. Each schema includes a description of the template and a set of fields representing the template's variables and constants of importance, where each field is assigned a name and description. Names must reflect not only the data type of the original variable but also the variable's purpose within the broader log context (see Fig. 2, step ②). After this stage, each log line is mapped to a JSON object containing a description of the template and a list of named fields corresponding to the variables in the line. Schema for related templates must be consistent (e.g., use the same field name when the "same" variable appears in multiple templates).

Schema mapping. The first two stages produce a structured representation of the log file enabled fast and efficient querying. The third step maps this structured representation to an existing normalized taxonomy so analysts can search over standardized attribute names (see Fig. 2, step ③). Each named entity in the parser's parsing tree is assigned to one or more attributes in the target taxonomy. Some fields cannot be mapped to standardized attributes. Normalized taxonomies are often incomplete and cannot represent every domain-specific value. In these cases, the field is described using the custom schema from step ②.

2.2 Security operations

Security operations teams focus on detecting, investigating, and mitigating threats. Logs underpin two core workflows: *(i) interactive investigations* to reconstruct chains of events, and *(ii) real-time detection* using alerting rules. Both are hindered when key semantics—"who did what to whom, where, and how"—are hidden in unstructured or semi-structured text.

Analysts often start with substring search (e.g., searching for "`Accepted conn`" in *sshd* logs). This is brittle: different software versions use different wording, and substring search struggles with structured fields like dates. To improve accuracy, analysts can write parsers to convert log data to a structured format, and then search this structured data. In practice, parsers can be extremely complex. For instance, one SSH parser we examined used a 3616-character regex inside a 160-line Python script, focusing solely on extracting the usernames, IPs and timestamps of successful connections. The slow, manual process of writing parsers for each data source and type of log event can consume days or even weeks, taking up analyst time that could instead be dedicated to investigating threats.

Industry mitigates this by *normalizing* logs into large taxonomies such as Google's Unified Data Model (UDM, ∼20K attributes) and OCSF (>50K). While such breadth captures important nuance, it also makes mapping onerous and fragile. Google SecOps alone maintains roughly 1,000 active parsers [16], while Splunk lists support for logs generated by 100+ companies with monthly updates [51] and Elastic shows a similar maintenance cadence. Maintaining these parsers is expensive: our partner commercial SIEM platform releases over 200 parser updates each month, consuming about 4,000 engineer-hours from over 30 domain specialists. This level of effort indicates that supporting the long tail of real-world log types can be punishing, with many parsers used by fewer than ten customers.

Even with expert effort, semantic mismatches remain. Taxonomies provide an overwhelming level of nuance, making it difficult to determine which field should be used. For instance, UDM exposes "`principal.network.application_protocol`", "`target.network.application_protocol`", and "`network.application_protocol`", any of which could be used to store the value "HTTP" in an Apache log. In other cases, mappings depend on perspective. In a *reverse SSH tunnel*, host `A` initiates an outbound connection to host `B`, after which `B` can reach services on `A`. In a single event, `A` is simultaneously the *source* (packets originate there), the *initiator* (it opened the session), and once the tunnel is active, the *target* of inbound traffic. Choosing one attribute discards context; populating several creates ambiguity. At the other extreme, even rich taxonomies can *under-capture* domain-specific detail. Both issues hinder detection and retrieval accuracy: across a corpus of queries spanning multiple task categories in the NICCS NICE framework [8], expert parsers achieve an F1 score of only 0.49 because ambiguous or missing mappings propagate into query logic.

In this paper we propose an end-to-end pipeline that removes the manual burden of writing parsers, achieves expert-level mapping, and additionally supports direct querying through both a custom schema and standardized taxonomies (UDM/OCSF).

2.3 Parser requirements

Given the sensitive security use cases that we focus on and the end-to-end nature of the parsers developed by Matryoshka, we have designed our system around five core principles:

- **Accuracy.** The structured representation of the logs must be correct: each template should match only a single event. Variable tokens should capture parts of the line that can vary and convey some meaningful information about the event. Template schemas must accurately represent the content of log lines, with meaningful variable names. Fields must only be mapped to standardized attributes that capture their intended role. Lastly, queries executed on the structured data should yield the same result as painstakingly searching the raw logs.
- **Completeness.** The structured representation must capture all information in the original log file. Any query that could be executed on the raw logs should be equally feasible on the structured data.
- **Consistency.** Variables fulfilling the same role across multiple log lines should be handled identically: they must have the same field names, descriptions, OCSF mappings, and data types. Similarly, templates that are syntactically different but serve the same purpose should have the same schema.
- **Run-time efficiency.** The parser we produce should operate efficiently: we should not need to invoke a language model on every log line and queries against structured data should run more efficiently than a linear scan over the raw logs.
- **Security.** We assume an attacker that can alter log lines at run time but not the generation logs. The parser should not be vulnerable to prompt injection from such an attacker. We do not address data poisoning of the corpus subset used to create parsers.

While these five principles guide our system, it is impossible to perfectly meet them all in practice. For instance, perfect accuracy cannot be achieved without complete knowledge of the developer's original intention, and consistency involves a subjective notion of semantic similarity across fields.

3 Related work

Most of the literature refers to *log parsing* as the task of generating templates that identify variables using wildcard placeholders. The intuition is that applications typically generate log messages with a printf-style (format string) API, and each template should correspond to a unique format string in the source code. We call this *template generation*, as we consider it only part of the parsing problem. Template generation has been studied for decades, and existing approaches can be divided into statistics-based and semantic-based approaches [27].

Statistics-based template generation. Earlier work on template generation applies statistical methods to find variables in log messages: they identified variables based on word length, frequency, and other statistical features. Frequency-based methods [10, 43, 57, 59] rely on occurrence frequencies or n-gram counts to build templates. Clustering-based techniques [13, 19, 42, 50, 54] group similar log lines to produce templates. Heuristic-based approaches [11, 13, 20, 21, 25, 41, 53, 61, 65] employ various heuristics or rule-based methods to detect the variable parts of each line. These approaches fail to incorporate semantic information about logs, leading to worse performance than more recent works.

Semantic-based template generation. Recent work has trended towards using semantic information in logs to improve the quality of generated templates. Earlier works cast template generation as a token classification task and trained neural networks [24, 33, 36] to mine semantics from log messages. More recently, researchers have shown that LLMs are effective at template generation [4, 30, 40, 58, 68, 70], especially when leveraging in-context learning [5]. DivLog [63] selects examples for developers to label; then when generating a log template for a target log line, it includes the most similar labeled examples in the LLM prompt. LILAC [26] builds on DivLog by improving the sampling method when choosing examples for a target log line, and adds a parsing cache to reduce the number of LLM queries. Other approaches focus on reducing the number of LLM queries [23, 62, 72] or fine-tune smaller LLMs for better performance [38, 39].

Past work has focused primarily on generating syntactic templates, yet these are not enough to convert unstructured logs to a structured form ready for querying. They identify the variables associated with each template, but do not map them into a common schema. Also, our experiments suggest that past work tends to over-capture and conflate consecutive variables. We improve on past work on template generation, generating a regular expression for each variable, improving the quality of templates. Finally, past work has typically not been evaluated on security-focused workloads, because of lack of suitable datasets; we address this by collecting a dataset of security-relevant logs, and discover that real-world security logs are more diverse and challenging than prior datasets.

Schema matching. The database research community has previously studied schema mapping, where the goal is to map data in one schema into a second schema [46, 69]. Existing methods are effective in settings where both schemas are thoroughly documented, but they are insufficient in our setting where schemas are auto-generated, do not come with documentation of the meaning of fields in the schema, and can contain tens of thousands of fields. Recent work explores using LLMs for schema mapping [37, 45, 49, 64, 71], but it

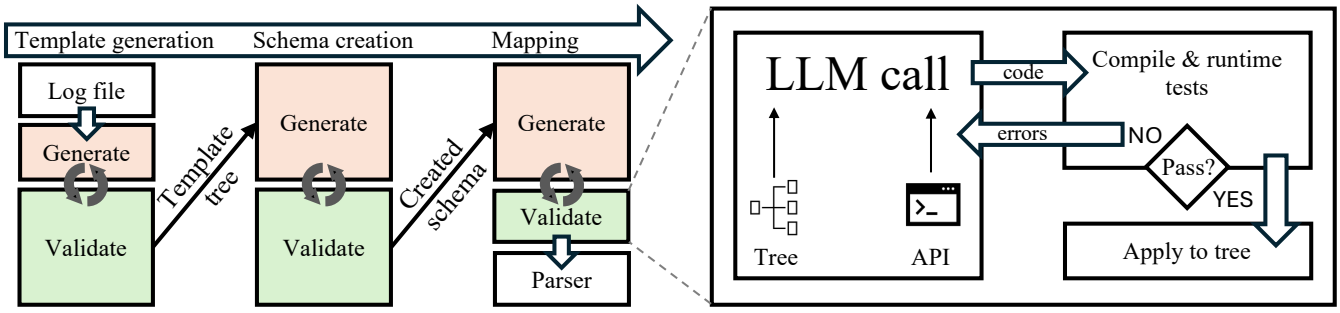

Figure 4: Matryoshka creates parsers in three stages, each subdivided into two logical steps. Matryoshka generates candidate solutions for log lines sequentially, then it validates batches of lines to enforce consistency. Validation produces code to edit the parse tree, iterating until valid.

too shares some of these limitations.

Log querying. LLMs are good at analyzing data [6, 12, 32], including unstructured logs [7, 29, 35]. Structured data can be queried using text-to-SQL tools [18, 22, 48, 66, 67], while unstructured data is usually queried using RAG [14, 31]. LLMs can be used to plan out complex queries over structured data [34], unstructured data [3], or both [9]. Unfortunately, these methods are not sufficient for our use case: they typically apply a LLM to each log line, which is too expensive; query expressivity is limited, because they rely on a vector embedding of each line; and they produce approximate results.

Security logs. In the security domain, structured logs are usually ingested into log managers such as Splunk [52], Google Security Operations [17], or Elastic Security [2]. These systems operate over normalized/structured data and support expressive query languages (often with AI-assisted query builders). However, the effectiveness of such queries relies on accurate normalization. Similar events may be recorded in multiple formats and mapped differently across sources, making it challenging to write a query that matches all instances of an event.

4 System architecture

Matryoshka automates the creation of log parsers without requiring any labeled data or human intervention. Its design mirrors the architecture of a parser: it creates templates, builds schemas, and maps fields to a taxonomy. To achieve this, Matryoshka uses large language models only during parser creation, never at run time. This ensures that while generation may be exposed to adversarial data, parsers applied to live logs are fast, deterministic, and immune to prompt-injection attacks. As illustrated in Fig. 4, parser construction unfolds in three stages, each following the same parallel two-phase pattern:

1. **Generation.** Process inputs sequentially, using standard LLM techniques (few-shot prompting, guided chain of thought, self-consistency) to propose solutions.
2. **Validation.** Revisit candidates in batches, reconcile them for global consistency, and fix or discard any that fail structural checks.

Concretely, this means during *template generation* we first let the LLM generate new templates for unmatched lines, then validate entire sub-trees for consistency; in *schema creation* we name variables line-by-line, then run a batch pass that merges synonyms into a single canonical field; and in *mapping* we assign provisional attributes one field at a time, then batch-correct them so that twin and sibling fields share consistent parents in the hierarchy.

4.1 Generation phase

For each new object—an unmatched log line, a template, or a field awaiting a mapping—we run a multi-turn prompt that relies on the following building blocks.

- *Chain-of-thought.* Prompts the model to explain its reasoning, ensuring the model analyzes all relevant context. We supply a structured algorithm to steer its reasoning process, ensuring thorough analysis of the input data.
- *Self-consistency.* We sample several completions using slightly different variants of the prompt and keep the majority vote. Self-consistency has been shown to outperform greedy decoding by allowing the model to explore multiple reasoning paths [60].
- *Execution checks.* After the model generates a solution, we apply checks ("does the regex compile?", "does it match only the intended lines?"). Failures trigger an automatic self-correction loop.
- *Few-shot examples with description embeddings.* Including few-shot examples that closely match the current prompt

substantially improves both consistency and accuracy. Crucially, these examples are drawn from previously generated objects, rather than human-provided labels. Standard retrieval approaches perform poorly due to the mix of structured and unstructured text, so we first have the LLM write a short semantic *description* of the example, then embed that description; cosine similarity over these summaries reliably surfaces the most relevant prior examples.

We now detail the generation process for each of the three parsing stages. This entire process is automatic, with no human involvement. Example prompts for the language model calls are provided in Appendix C, and the full set of prompts can be found in our source code.

4.1.1 Template creation

We learn a parsing tree iteratively from raw logs. Inspired by LogBatcher [62], unmatched lines are clustered before generating templates. Specifically, we buffer up to 2,500 lines and partition them in two lightweight passes: (i) a coarse pass that groups lines by their longest shared prefix with the current tree, and (ii) a fine pass that clusters lines via cosine similarity over short, LLM-written semantic descriptions. The most dense cluster is then fed to a language model to confirm its elements should all be parsed by a single template—if not, the model isolates a subset of lines from the cluster that share a template.

The language model then proposes a template for this cluster, by providing a tokenization into constants and variables and a regex per variable. Proposals are sampled with self-consistency and scored by a simple objective: maximize coverage of the cluster while minimizing spillover onto already-parsed lines. This candidate template is then refined through a self-correction loop, checking that each regex compiles and that shared prefixes are reused.

Once the model produced a candidate template, we check if it overlaps existing ones. Template overlaps can be legitimate, for instance, when an older template is too specific or a new one is needed for malformed values. They may also indicate over-capture, in which case the new template should be discarded. If a new template overlaps existing ones, we sample overlapping lines and ask the model if they come from the same format. If so, the old templates are likely too specific; we replace them with the candidate. If not, the new template is likely over-capturing; we run a self-correction loop to narrow the candidate.

The final template is added to the parsing tree; its prompt and language model output are cached to be used as few-shot examples in subsequent template generations.

4.1.2 Schema creation

Schemas are learned iteratively from templates. Given a template, the model names each variable and semantically relevant constant, then writes a description for each named token. Since templates reside in the same parsing tree, their tokens often share common prefixes. We give the model the names and descriptions for these existing prefix tokens, and enforce their reuse through self-correction.

We ensure consistency through few-shot prompting, encouraging the model to reuse variable names and descriptions across templates when the same semantic roles are present. We identify similar templates that have already been mapped to a schema using description embeddings, and provide them as few-shot examples.

4.1.3 Mapping

We flatten the target taxonomy to leaf attributes, use the language model to write a short description of each, and embed that description.[4] We then map fields in our created schema iteratively. For each field, we embed its description, filter target taxonomy attributes based on cosine similarity, and have the language model select the most appropriate attributes—if any—given these filtered candidates.

We bias for consistency by injecting sibling fields of already assigned attributes into the filtered list of candidates, and providing previously mapped similar fields as few-shot examples. Sibling fields are attributes that share a direct parent in the taxonomy's hierarchy. For instance, if a field called "`src_ip`" has already been assigned to "`src_endpoint.ip`", we will add sibling fields such as "`src_endpoint.hostname`" to the candidate list of other fields that appear in a template with "`src_ip`".

Some taxonomies have stronger type systems than others. OCSF, for instance, has types for usernames, dates, and process names. When mapping to such a taxonomy, we optionally have the language model assign each field a type first, then use the types to prune potential candidates in the mapping stage.

4.2 Validation phase

Batch validation lets the model view related candidates side-by-side, catching inconsistencies that a sequential pass would miss. As an example, consider the SSH log line in Fig. 2 whose authentication method is "`pw`" for password. If the model sees only that line during generation it may incorrectly treat `pw` as a constant token; when a future line introduces `pka` and `mfa`, these will generate another template. Validation merges all three into a single `auth_method` variable.

Because a full parse tree or schema often exceeds context limits, we validate in *rolling batches*: each batch sees a subset of previously validated solutions as well as new candidates and may only modify the latter. This limits the scope of validation to only change the most recently generated solutions,

[4] This pre-processing step only needs to be run once for every update to the taxonomy. Parser update cycles typically occur every few months.

preventing it from fixing some long-distance inconsistencies in the input.

Validation requires changing the parsing tree, either to rearrange nodes or add field names. Trees are complex objects for language models to work with, and simply asking the model to return a corrected tree almost always leads to mistakes.

Instead, we frame validation as a code generation task: we supply API stubs that manipulate the tree (add, delete, move nodes; rename fields; adjust mappings) and ask the model to output Python code plus a natural-language rationale. The code is executed inside a controlled sandbox; Compile and runtime errors or failed checks an automatic self-repair loop.

This technique has three main advantages: (1) it frames validation as a coding problem, which is in-distribution for many language models; (2) correctness of the validated tree can be checked after every API call, errors can be used to give fine-grained feedback to the model so it can refine its code until it produces a valid output; (3) we can restrict the API to only allow some actions and prevent the model from changing ground truth parts of the tree.

In template generation, we let the model add, delete and move nodes in the tree. When executing the code, we check that the graph resulting from the change is still a tree and that the final tree still matches the same set of lines as previously.

In schema creation, we let the model create new field names, assign field names to nodes in the tree, and assign descriptions to field names. We verify new field names are unique. Separating field name creation and assignment avoids the model accidentally naming two different concepts the same name.

In mapping, we let the model assign field names to attributes, making sure the attribute exists in the target taxonomy. We enforce that at most N attributes are assigned per field (where N is a parameter that defaults to 1).

This alternating generate/validate strategy produces globally consistent parsers while keeping each individual LLM call small and focused. While the generation step supports different language models, advanced reasoning capabilities are required to run validation.

5 Evaluation

Matryoshka represents a milestone in automated log parsing. We use three different datasets to show our approach produces high quality parsers, comparable to those produced by expert analysts, on a wide variety of log types.

1. **LogHub2.0 [73] (§5.1).** LogHub2.0 is a benchmark of 14 log files with ground-truth wildcard templates. On this dataset, Matryoshka outperforms related works LILAC [26], Drain [21], and Brain [65] on the canonical template-generation task.

2. **SecurityLogs (§5.2).** We curated SecurityLogs by crawling system logs uploaded to the Red Hat Bugzilla and extracting 500K+ lines across five log types (audit, puppet, sshd, dhcp, cron). On these complex files, Matryoshka scales and yields high-precision/high-recall query results, especially when using created fields; it also outperforms naive substring search. We provide step-wise microbenchmarks (template, schema, mapping), ablations, and efficiency evaluations.

3. **Real-world logs (§5.3).** The real-world dataset contains 27 enterprise log types from a major commercial SIEM platform with associated expert-written UDM parsers. Matryoshka generalizes to this dataset and delivers schema quality and queryability comparable to expert-written parsers without manual effort.

Table 1: SecurityLogs parser summary

	Total nodes	Variable nodes	Templates	Unique fields	Unique OCSF attributes
SSHD	384	154	98	47	135
Cron	72	19	7	8	35
DHCP	441	153	178	41	67
Audit	6309	3000	496	150	286
Puppet	981	370	254	101	77
Total	8187	3696	1033	347	600

Setup. Matryoshka is run with no human input using Gemini 2.5 Pro, unless otherwise stated. Our source code, public SecurityLogs dataset, and curation tools which provide a user interface to edit parsers and run queries are made public at https://github.com/julien-piet/matryoshka/ . We rely on the Gemini [55] suite of models in our paper, but the code can support OpenAI and Anthropic backends as well. Our implementation supports both OCSF and UDM, but our mapping logic can be applied to any taxonomy.

5.1 LogHub2.0

The first dataset we evaluate against is an existing benchmark for template generation works. Our template generation stage outperforms recent works LILAC [26], Drain [21], and Brain [65] on this task when evaluated under the same settings and language model.

Table 2: Average LogHub2.0 template generation scores.

	Matryoshka	LILAC	Drain3	Brain
PGS	**0.97**	0.90	0.89	0.86
TS	**0.91**	0.82	0.81	0.74

5.1.1 Dataset

Existing approaches to template generation commonly use LogHub2.0 [28, 73]. This resource contains 14 log files (over 3 million lines) annotated with ground-truth wildcard-based templates. We use it to compare the performance of our template generation step against prior methods, but we do not generate ground-truth labels for schema creation or attribute mapping because (1) most of the data is outdated (some logs are over 20 years old); (2) several files are heavily anonymized, limiting reliable semantic extraction; and (3) most are not security-focused and instead are activity logs from supercomputers or distributed systems.

5.1.2 Experiment

While schema creation and mapping are novel topics, template generation has been extensively studied in prior work, and recent works such as LILAC [26], DivLog [63] or LogBatcher [62] use language models to perform this task. Although we added regular expressions for precisely matching variables, our templates can be converted back into the same wildcard templates used in the past in order to compare our approach to prior works. We selected three prior works with the best performance to compare against: LILAC [26], one of the most promising LLM-based approaches, Brain [65], a lightweight parser, and Drain3, the latest version of Drain [21], a popular log parsing algorithm. On the LogHub2.0 dataset, these three frameworks use a pre-parsed version of the logs and only generate templates for the suffixes. In contrast, Matryoshka is given the raw version of the logs.

Setting. Drain and Brain do not use LLMs. To ensure an equal comparison with LILAC, we deliberately ran our system in a restricted setting: both LILAC and Matryoshka are run using Gemini 1.5 Flash, and we run Matryoshka without validation. This reflects the state of the art at the time of LILAC's release, when no reasoning models were capable of running our validation step. As such, our reported results are conservative. The performance gap when using validation is higher, as we show on SecurityLogs in Section 5.2. LILAC requires at least a single human-labeled example to run, so we also provide one to Matryoshka.

Results. We report two metrics: parser group similarity, measuring whether log lines derived from the same format string are grouped together (PGS), and template similarity (TS), measuring how close each template is to the ground truth (TS). Both metrics are defined in details in Section 5.2.4. We report average metrics in Table 2. Full results are in Table 9 in Section A. We find that Matryoshka matches or outperforms other works on most log files. On average, we obtain a parser group similarity of 0.97 and a template similarity of 0.91, while the best of the other schemes, LILAC, obtains a PGS of 0.90 and a TS of 0.82. The performance gap is not as significant as observed in Section 5.2.4, because in the LogHub2.0 dataset similar prefixes across lines are pre-parsed.

5.2 SecurityLogs

SecurityLogs contains Linux system logs uploaded to a bug tracking website. These large and heterogeneous logs demonstrate Matryoshka's ability to scale to formats with hundreds of different templates. Queries against our created schema consistently achieve high precision/recall and outperform naive substring matching. We additionally run ablation studies and introduce step-wise metrics to measure the performance of each stage of the parsing pipeline.

5.2.1 Dataset

We curated our dataset from Redhat Bugzilla bug reports [1]. We discovered that many users attach system log files to public bug reports there, so we crawled the Bugzilla website and downloaded all log files attached to bug reports up to August 2023. In total, this dataset contains over 30 million log lines. We filter to logs from five applications that are security-relevant and are well-enough documented that we could manually construct ground-truth parsers:

- **Linux kernel audit logs (77K lines)**: Fine-grained logs of security-relevant events in Linux systems.
- **Puppet logs (157K lines)**: Logs from Puppet, an orchestration tool for server configuration deployment.
- **SSH daemon logs (35K lines)**: Logs involving SSH authentications and connections.
- **DHCP logs (378K lines)**: DHCP client logs reporting DHCP requests and lease information.
- **Cron logs (13K lines)**: Reports of scheduled cron jobs.

We ran Matryoshka on each log file. We also created a ground-truth parser, by manually checking and fixing every template, field name, and mapping generated by Matryoshka. The ground truth parsers are summarized in Table 1.

5.2.2 Query comparison

Our first evaluation measures Matryoshka's ability to create queryable schemas for SecurityLogs. We define 10 *security-relevant queries* per log file that correspond to tasks in the NICCS NICE framework [8], execute the queries, and compare the resulting set of results to the ground-truth golden result, reporting precision and recall.[5] We use our hand-written parsers to collect ground-truth answers. Each query is written in three variants:

[5] We only wrote 5 queries for cron, which is less diverse than the four other log files.

Table 3: Matryoshka query precision and recall metrics

Dataset	OCSF attributes		Created attributes		Naive substring	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
SSHD	0.90	0.90	**1.00**	**1.00**	0.90	0.80
Cron	**1.00**	**1.00**	**1.00**	**1.00**	0.92	**1.00**
DHCP	0.80	0.77	**1.00**	**0.97**	0.92	0.71
Audit	0.70	0.69	**1.00**	**0.99**	0.97	0.85
Puppet	0.90	0.88	**1.00**	**0.98**	**1.00**	0.63
Average	0.86	0.85	**1.00**	**0.99**	0.94	0.80

Table 4: Average query precision/recall across Gemini models

Variant	OCSF	Custom
Gemini 2.5 Flash	0.76 / 0.76	0.92 / 0.91
Gemini 2.5 Pro	0.86 / 0.85	1.00 / 0.99
Gemini 2.5 Pro (no val.)	0.79 / 0.77	0.99 / 0.96

1. a **standardized form** that uses canonical field names exactly as defined in the OCSF taxonomy;
2. a **custom form** that uses the field names produced by Matryoshka for that log file.
3. a **naive substring form** that uses simple substring matches directly on raw log messages.

For example, one of our cron log queries looks for scheduled jobs from a particular host within a specific time window, filtering on OCSF fields `time_dt` and `device.hostname`, and checking for the presence of the `job.cmd_line` OCSF field. In order to run expressive queries, we sometime include custom fields in standardized queries when one of the predicates was over a field that does not map to any OCSF attribute.

Standardized and custom queries are formed by applying unions or intersections of predicates. Predicates can be over the values of the fields, or over the static part of templates. Predicates on the static template are defined by substring matching. This limits the range of queries we can run. For example, "return all lines indicating a bind error due to an already-in-use address" is tedious to formulate in our query syntax, because it requires knowing all possible templates that could indicate such an event. A dedicated LLM-powered query planner could plausibly map queries to the relevant set of templates; we leave that to future work.

Substring matching is a common technique used by analysts to query unstructured log data. This can answer some queries, but limits the reach that an analyst can have, and many of our queries could not be fully expressed using substring matching alone. We illustrate this using three examples:

- Tracking the usage of a specific SSH key based on its hash can be done efficiently using substring matching.
- Substring matching does not allow restricting results to a particular date range or time window. Analysts would have to manually write a one-off script to post-process the results of the substring search. In contrast, Matryoshka can express such a predicate natively.
- One of our queries asks to identify DHCP leases with long renewal times. This is difficult to express with substring matches, as there is a variety of log formats that contain the renewal time. With Matryoshka, this query can be expressed simply with a predicate on the "`lease_dur`" field.

In our experience, substring-match queries take longer to write than our structured queries. We assume an analyst will not have resources to comprehensively identify all message formats/templates that might have relevant information; instead, we simulate an analyst who looks for the first example they can find of a log message containing the necessary information and identifies a single substring from that log message to search for. We coin these queries *naive* substring matches.

Results. The average precision and recall of the queries is reported in Table 3. We observe that using OCSF attributes for querying performs poorly compared to Matryoshka-created attributes: This is due to the difficulties discussed in Section 2.2. If the mapping for a high-volume variable fails, this variable cannot be queried using OCSF attributes. We show in Section 5.3 that even expert-written parsers struggle at standardization: Matryoshka-generated parsers slightly outperform expert-written ones on standardized queries.

In contrast, custom-form queries have perfect precision and near-perfect recall. The main error source is inconsistent naming of fields. For example, the audit log has a field for the current working directory of a process. This field in the generated parser is most often called "`current_working_directory`", but sometimes abbreviated as "`cwd`". For the sake of simplicity, we chose to run queries on only one created field name, even when there are multiple names for the same concept: requiring analysts to list all possible fields would be tedious. Using an LLM query engine could help here, as the model could automatically select all relevant fields.

Queries over Matryoshka-created field names consistently perform better than naive substring queries. OCSF-based querying achieves a similar F1 score to substring matching (0.85 for OCSF vs 0.86 for substring matching). OCSF queries have worse precision, due to different source fields collapsing into a single OCSF attribute, but better recall. Naive substring matching uses an arbitrary line as an example to model the substrings needed to run the query: this leads to missed lines when there are multiple possible templates that would match a given query, and to over-capture if the same substring is present in other lines. The complete set of substrings used for this task are detailed in Section B.

5.2.3 Ablations

Our next experiments analyze how sensitive Matryoshka is to different language models and compare the performance of Matryoshka with and without its validation step. We report average metrics in Table 4, leaving per-file results to Table 10 in Appendix A.

Language model susceptibility. We ran our pipeline twice on each file: once using Gemini 2.5 Pro for all steps, once using Gemini 2.5 Flash[6]. Overall, Gemini 2.5 Flash performs worse than Pro, both using created and OCSF queries, most notably on Puppet logs. The model under-parsed Puppet log lines, keeping long error messages as single variables, instead of breaking them down into more granular component, which negatively impacts queryability. We recommend using advanced models with Matryoshka.

Validation. We ran our pipeline a third time with Gemini 2.5 Pro, this time without using Matryoshka's validation capabilities. Validation improves custom query recall, by coalescing variables that have different names into a single attribute. Validation also improves mapping significantly. We recommend using validation for improved performance.

5.2.4 Micro benchmarks

We now evaluate individual steps on the same SecurityLogs files: template generation, schema creation, and mapping.

Template generation. Prior work commonly uses two metrics. *Group Accuracy (GA)* measures whether log lines derived from the same format string are grouped together. *Parser Accuracy (PA)* checks if the predicted template exactly matches the ground truth. Both have drawbacks: PA is overly strict (splitting a field such as "IP:port" into two variables yields a score of zero even though the result may be more useful), while GA penalizes under-capture and over-capture equally. In practice, over-capture is worse, since it conflates distinct fields. For example, the template "`Accepted <*> from <*>`" is meant to capture usernames, but also absorbs IP addresses when present, merging unrelated variables.

To address these issues, we introduce two refined metrics. *Template Similarity (TS)* measures how close the predicted template is to the ground truth when all variables are replaced by "<*>". It is computed as $1 - \frac{\text{Levenshtein(Ground Truth,Predicted)}}{\max(|\text{Ground Truth}|,|\text{Predicted}|)}$, so small differences such as splitting or merging variables only reduce the score slightly rather than to zero. *Parser Group Similarity (PGS)* measures how well the parser keeps log lines that share the same format string together. Over-capture (merging different formats into one group) yields a score of 0, while under-capture (splitting one true group into smaller ones) receives partial credit given by |Parser Group|/|Ground Truth Group|.

[6]We still rely on Pro for the validation step, as it require advanced reasoning to be able to function.

Table 5: Template generation metrics versus related works.

Dataset	Lines	Matryoshka		LILAC		Brain	
		PGS	TS	PGS	TS	PGS	TS
SSHD	35,329	**1.00**	**0.90**	0.92	0.81	0.88	0.73
Cron	12,547	**1.00**	**0.80**	**1.00**	**0.80**	**1.00**	0.60
DHCP	377,653	**0.98**	**0.89**	0.40	0.44	0.54	0.69
Audit	76,636	**0.97**	**0.98**	0.13	0.33	0.62	0.52
Puppet	156,880	**0.97**	**0.99**	0.59	0.74	0.96	0.68

We compute both metrics for Matryoshka using Gemini 2.5 Pro and compare to two prior works—LILAC [26] and Brain [65]. LILAC requires human labels to work; we provide it with a single example. Both Brain and Matryoshka are run with no human labels. Matryoshka outperforms both other works on all five logs. The performance gap comes from prior work's lack of consistency when parsing parallel structures in different log lines.

Schema creation. Schema creation entails clustering variables by meaning (e.g., assigning every "source IP" variable to a field "`source_ip`"). For this step, we define *Schema Group Similarity (SGS)*. We group variables by their assigned name and score each variable. If multiple ground truth fields are merged (overcapture), the score is 0. If a ground truth field is split into several fields (undercapture), the score is the ratio of the parser's group size to that of the ground-truth cluster. This rewards grouping variables of the same semantic role without merging separate roles. The final score is the weighted average of variable scores depending on their frequency.

We ran Matryoshka's schema creation on ground-truth templates so that both parsers expose the same set of variables. As shown in Table 6, SGS is nearly perfect for SSHD, cron, and DHCP. Puppet and audit score lower, because high-volume variables such as the "`operation`" field are split across multiple names. In rarer cases, the created schema slightly over-captures: "`audit_user_id`" is merged "`new_audit_user_id`".

Mapping. The final step evaluates how well fields are mapped to standardized attributes. We define a per-field accuracy score: if a parser assigns no mapping to a field that also has no mapping in the ground truth, it is counted correct. Otherwise, accuracy is the fraction of assigned attributes that also appear in the ground-truth set. The mapping accuracy is the weighted average of the scores.

We ran mapping independently on ground truth schemas,

Table 6: Matryoshka schema and mapping metrics

	SSHD	Cron	DHCP	Audit	Puppet
Schema Group Similarity	0.95	1.00	0.94	0.64	0.85
Mapping Accuracy	0.93	0.93	0.59	0.79	0.98

Table 7: Timing and LLM token usage (in/out in millions of tokens) for Matryoshka steps

Step	SSHD	Cron	DHCP	Audit	Puppet
Generation	1h 45m	28m	4h 16m	10h 7m	4h 16m
Creation	30m	2m	2h 16m	9h 33m	1h 13m
Mapping	1h 7m	9m	1h 24m	4h 30m	2h 13m
Total	3h 22m	39m	7h 56m	24h 10m	7h 42m
MTok In/Out	21 / 3	2 / 1	33 / 3	162 / 15	50 / 5

using Gemini 2.5 Pro, and restricted the number of mappings to at most one per source attribute. SSHD, puppet and cron mappings are mostly correct. Audit and DHCP score lower, due to mistakes on high volume variables. The logging server's hostname and assigned IP map to OCSF's "`device.hostname`" and "`device.ip`", respectively. The generated parser assigns these to attributes in the "`src_endpoint`" tree, which is less appropriate.

5.2.5 Time and cost evaluation

Our system allows running queries in a few seconds, even when the source log file is hundreds of thousands of lines. Log ingestion is also fast: we consistently parse log files at over 200 lines per second. This is possible because LLMs are only used at generation time, not at run-time: live data is statically parsed, which also prevents prompt-injection attacks. However, the process of generating parsers is slow, due to the fact most steps use previous answers of the model to few-shot prompt the next ones, thus cannot be parallelized. Matryoshka takes on average 150 seconds per template. Run times and token usage are detailed in Table 7. Matryoshka only needs to be run once, but future work should look at speeding up this process. We estimate the number of input and output tokens required for each file—system prompts and repeated queries are cached to reduce costs. Averaging over the five files, each template requires about 250K input tokens and 25K output tokens (including every step's generation and validation), or an average of 0.60 USD per template when using Gemini 2.5 Pro or 0.14 USD with Gemini 2.5 Flash. As a back-of-the-envelope comparison, our partner SIEM platform pushes 200 parser updates monthly, requiring 4,000 engineer-hours, or 20 hours per update. A typical update touches a handful of templates; conservatively assuming $\sim$10 templates per update yields 2 hours per template (time spent researching field semantics, selecting mappings, and writing code). Matryoshka is 40$\times$ faster and 25$\times$ cheaper at U.S. federal minimum wage when using Gemini 2.5 Pro.

5.3 Real-world evaluation

Our last dataset comprises real-world enterprise log files. We leverage this data to evaluate Matryoshka on diverse log types, and show we rival expert-written parsers while being fully automated and requiring no manual effort.

5.3.1 Dataset

Our dataset comprises real-world log samples from 27 diverse log types with associated human-written parsers from our partner SIEM. These include web-server access logs, intrusion-prevention alerts, user-management audits, and endpoint-protection telemetry. Each sample is modest in size—no more than a few hundred lines—but collectively they span a spectrum of events found in real-world enterprise environments. We use this corpus to assess Matryoshka on the long-tail of real-world log types and to compare its output against the hand-written parsers.

5.3.2 Side-by-side holistic comparison

We ingest logs into structured UDM format using two parsers—an expert-written parser, and a Matryoshka-generated one. We compare both these schemas along four axes: *coverage* (fraction of source fields mapped), *accuracy* (semantic correctness of each mapping), *consistency* (stability of mappings across a line), and *queryability* (how easily an analyst can express predicates).

Step 1: Expert assessment. We drew a random sample of log lines along with both schemas, presented them blindly to an experienced UDM rule writer, and asked for a 1–5 preference score (1/5 = strong preference for the left/right schema, 2/4 = weak preference, 3 = tie) using the four axes above. Across 15 comparisons the expert judged the schemas *equivalent* in 8 cases, preferred Matryoshka in 3, and preferred the UDM baseline in 4. While this sample is modest—reflecting limited expert availability—we use it primarily to calibrate our LLM autorater.

Step 2: LLM autorater. Using the expert-labelled set as ground truth, we calibrated a Gemini 2.0 Pro autorater—a model from a different family to those used during generation time. For each log line the model receives (i) the raw text, (ii) Matryoshka-generated field/value pairs, (iii) UDM field/value pairs from the expert-written parser, and (iv) a rubric requesting a reasoned comparison along the four axes followed by a 1–5 score. Parser order is randomised, multiple completions are sampled, and scores are averaged to remove positional bias. Among several candidate prompts and few-shot example sets, we retained the combination that reproduced the expert's preferences on all 15 calibration lines, maximising alignment between model and human judgement.

Results. After calibration the autorater graded every line in the 27 files. It preferred Matryoshka in 25% of cases, favored the UDM baseline in 24%, and reported no preference in 49%. These findings echo the expert's conclusion: Matryoshka delivers schema quality comparable to expert-written parsers—without any manual work. We substantiate these observations

Table 8: Enterprise-log query accuracy

Configuration	Precision	Recall
Expert-written UDM parser	0.50	0.48
Matryoshka + UDM query	0.60	0.60
Matryoshka + custom-schema query	1.00	0.95

with larger-scale quantitative results in the subsequent subsection.

5.3.3 Query comparison

We further compare Matryoshka's UDM output to expert-written UDM parsers by measuring how well Matryoshka parsers support queries. Similar to Section 5.2.2, we define security-relevant queries, execute them and compare the results to a manually derived ground-truth response set.

Experimental setup. We select 10 files at random from the 27, devise a security-relevant query for each, informed by NICE tasks, and manually identify the correct matches. The queries span tasks such as account monitoring, network reconnaissance, or resource-access analysis. Then, each query is expressed in a standard UDM form, and in a custom-schema form using Matryoshka-generated field names. The UDM query is written agnostically of the parser, using the taxonomy's documentation. We then run three configurations: (1) expert-written UDM parser, queried with the UDM-form query; (2) Matryoshka-generated UDM parser, queried with the UDM-form query; (3) Matryoshka-generated parser, queried with the custom-schema form query.

Results. Custom-schema queries yield near-perfect performance, while both UDM parsers struggle. Three points stand out: *(i)* Queries against taxonomies are not accurate, even when using expert-written parsers. This is representative of the mapping difficulties discussed in Section 2.2. *(ii)* Matryoshka UDM parsers are less accurate (fields are not always mapped to the best candidate attribute) than expert-written UDM parsers, but have better coverage. *(iii)* Queries using Matryoshka's custom schema yield far more precise results, demonstrating that log-specific field names enable more accurate predicates, albeit at the cost of learning a file-local vocabulary rather than a general-purpose, normalized schema.

6 Discussion

6.1 Accurate queries against taxonomies

Our evaluations show mapping logs to standard taxonomies is difficult, even for expert analysts. Misalignment between the specificity of source logs and taxonomies both under-capture domain-specific concepts and provide excessive nuance for standard concepts (§2.2). Despite these issues, industry continues to use common taxonomies for normalization. They allow writing detection rules that are agnostic to the source data, and enable cross correlation in ways custom schemas cannot.

Yet, we find custom schemas allow for more precise search over logs, since they are created on a per-log basis. We suggest that future research explore parsing logs to a custom schema, using language models to compile a natural language query/rule to custom-schema attributes, and skip normalization entirely.

6.2 Limitations

Although Matryoshka advances end-to-end log parsing, several limitations remain.

Mapping extracted fields to standard attributes is challenging due to the volume of target attributes and nuances in field definitions. More advanced semantic techniques may improve accuracy.

In practice, new templates will appear over time, requiring parsers to be updated. Also, queries against Matryoshka output can miss entries or report false positives. Given analysts' need for high reliability, any incorrect or missing extracted fields can significantly impact usefulness. To mitigate this, we built a prototype interface for analysts to inspect and correct parsers, though more efficient UI design remains future work.

Log ingestion and querying are fast since they rely on regular-expression matching, but parser generation is slow due to sequential LLM calls. Parallelization or smaller domain-specific models could improve speed.

Currently, we only support logs where events are well delimited (e.g., single-line entries); a practical system should handle logs where events are not necessarily well delimited.

7 Conclusion

We presented Matryoshka, the first end-to-end system leveraging LLMs to automatically generate semantically-aware structured log parsers. Matryoshka combines a novel syntactic parser with a semantic layer that clusters variables, maps them to structured schemas, assigns contextually meaningful field names, and maps variables to attributes from standard taxonomies. While log standardization remains challenging, Matryoshka parsers rival expert-written parsers on real-world logs both in qualitative head-to-head comparisons and in query accuracy, removing much of the costly human effort needed to craft parsers. Matryoshka-created schemas offer an alternative to standardized taxonomies that enable precise querying ($F_1 = 0.99$), significantly higher than achievable with existing substring-matching techniques, at a similar cost in query writing. By automatically transforming unstructured logs into structured, semantically rich data, Matryoshka represents a meaningful step toward enabling security analysts to focus on threat detection rather than manual parser construction.

Ethical Considerations

Stakeholders. Our work potentially impacts multiple groups: (i) security analysts and organizations that rely on log data for detection and forensics, (ii) end users whose activities are indirectly reflected in log data, (iii) service providers that generate and process logs, and (iv) the broader research community developing methods for AI-driven security analytics. The research team itself is also a stakeholder, given exposure to content in the logs.

Impacts and Principles. Following the Menlo Report [56] principles, we considered the following:

- **Beneficence.** Our goal is to improve the effectiveness of security monitoring by reducing the manual burden of parser creation.
- **Respect for Persons.** We only used datasets that were already public (e.g., Red Hat Bugzilla logs) or enterprise logs processed under strict internal safeguards. No logs were released that were not already public.
- **Justice.** Results are shared through open benchmarks and code, allowing the wider community to benefit.
- **Respect for Law and Public Interest.** We adhered to data usage policies, respected terms of service, and did not probe or interact with live systems.

Potential Harms.

- **Privacy risks.** Improper handling of enterprise data could expose sensitive information. We mitigated this by only processing enterprise data on authorized systems, and never exporting any sensitive data. Only aggregate results are shared in this paper, which have been vetted by the enterprise to be in accordance with their internal policies and customer agreements.
- **Operational risks.** Running experiments on live systems could disrupt services. We avoided this by only using offline static datasets.
- **Workforce impacts.** Automated parsers may take over parts of the repetitive work of writing and maintaining log parsers. However, since the field is already understaffed, the effect is expected to be positive: analysts can devote more time to higher-value activities such as threat investigation and improving parser quality, rather than face job loss.

Mitigations.

- Limited data sources to public logs and enterprise logs with legitimate access.
- Ran all experiments offline to eliminate risks to production systems.

Decision to Proceed. After weighing risks against benefits, we judged that the benefits (more accurate and efficient defenses, reduced manual effort for overburdened security teams, and contributions to reproducible research) clearly outweigh the residual risks. On this basis, and in line with Menlo Report principles, we proceeded with and are publishing this work.

Open Science

In accordance with the USENIX Security open science policy, we provide here a list of all artifacts needed to evaluate our contributions, along with their availability:

SecurityLogs. We curated a dataset of Linux system logs from Red Hat Bugzilla bug reports, covering SSHD, DHCP, audit, cron, and puppet logs. This dataset and the manually constructed golden parsers are available with the Matryoshka source code.

LogHub2.0 benchmark. We evaluate against the LogHub2.0 benchmark dataset, which is already publicly available at: https://github.com/logpai/loghub.

Real-world enterprise logs. Our evaluation includes experiments on a collection of real-world log samples obtained from an enterprise production environment. These logs contain sensitive customer information and cannot be released for privacy and security reasons. In line with the CFP's open science policy, we disclose their use here, justify their unavailability, and provide full experimental methodology in the paper to enable replication on comparable datasets.

Source code. The complete source code for Matryoshka, along with the parser generation pipeline, evaluation scripts, and interfaces for editing and querying parsers, will be made publicly available and is available for review at https://anonymous.4open.science/r/matryoshka-B7ED.

Artifact policy compliance. All artifacts listed above are available at submission time, except for the real-world enterprise dataset which cannot be released due to sensitivity. We believe the combination of public datasets, golden parsers, and open-source code is sufficient for reviewers and future researchers to reproduce our results and extend our work.

References

[1] Red Hat Bugzilla, 2025.

[2] Elasticsearch B.V. Elastic security, 2025. Accessed: 2025-04-12.

[3] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A. Shah, Benjamin Sowell, Dan Tecuci, Vinayak Thapliyal, and Matt Welsh. The design of an LLM-powered unstructured analytics system, 2024.

[4] Merve Astekin, Max Hort, and Leon Moonen. A comparative study on large language models for log parsing. In *ESEM*, 2024.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *NeurIPS*, 2020.

[6] Wenhu Chen. Large language models are few(1)-shot table reasoners, 2023.

[7] Tianyu Cui, Shiyu Ma, Ziang Chen, Tong Xiao, Shimin Tao, Yilun Liu, Shenglin Zhang, Duoming Lin, Changchang Liu, Yuzhe Cai, Weibin Meng, Yongqian Sun, and Dan Pei. LogEval: A comprehensive benchmark suite for large language models in log analysis, 2024.

[8] Cybersecurity and Infrastructure Security Agency (CISA). NICE Workforce Framework for Cybersecurity (NICE Framework), 2025. Last published: May 29, 2025; Accessed: 2025-08-05.

[9] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. UQE: A query engine for unstructured databases, 2024.

[10] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using *n*-gram dictionaries. *IEEE TSE*, 2020.

[11] Min Du and Feifei Li. Spell: Online streaming parsing of large unstructured system logs. *IEEE TKDE*, 2019.

[12] Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. Large language models (LLMs) on tabular data: Prediction, generation, and understanding – a survey, 2024.

[13] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, 2009.

[14] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[15] Google Cloud. UDM field list.

[16] Google Cloud. Supported Default Parsers | Chronicle Security Operations, 2025. Accessed: 2025-08-05.

[17] Google LLC. Google security operations, 2025. Accessed: 2025-04-12.

[18] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-SQL in cross-domain database with intermediate representation, 2019.

[19] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. LogMine: Fast pattern recognition for log analytics. In *CIKM*, 2016.

[20] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. Towards automated log parsing for large-scale log data analysis. In *IEEE TDSC*, 2018.

[21] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *ICWS*, 2017.

[22] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of LLM-based text-to-SQL. *arXiv preprint arXiv:2406.08426*, 2024.

[23] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael R Lyu. LUNAR: Unsupervised LLM-based log parsing, 2024.

[24] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R Lyu. SemParser: A semantic parser for log analytics. In *ICSE*, 2023.

[25] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *QSIC*, 2008.

[26] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. LILAC: Log parsing using LLMs with adaptive parsing cache. *FSE*, 2024.

[27] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. A large-scale evaluation for log parsing techniques: How far are we? In *ISSTA*, 2024.

[28] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. A large-scale evaluation for log parsing techniques: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 223–234, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Egil Karlsen, Xiao Luo, Nur Zincir-Heywood, and Malcolm Heywood. Benchmarking large language models for log analysis, security, and interpretation. *Journal of Network and Systems Management*, 2024.

[30] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *ICSE*, 2023.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*, 2020.

[32] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. Table-GPT: Table-tuned GPT for diverse table tasks, 2023.

[33] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. Did we miss something important? studying and exploring variable-aware log abstraction. In *ICSE*, 2023.

[34] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. Optimizing LLM queries in relational data analytics workloads, 2025.

[35] Yilun Liu, Yuhe Ji, Shimin Tao, Minggui He, Weibin Meng, Shenglin Zhang, Yongqian Sun, Yuming Xie, Boxing Chen, and Hao Yang. LogLM: From task-based to instruction-based automated log analysis, 2025.

[36] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. Uniparser: A unified log parser for heterogeneous log data. In *WWW*, 2022.

[37] Yurong Liu, Eduardo Pena, Aecio Santos, Eden Wu, and Juliana Freire. Magneto: Combining small and large language models for schema matching, 2024.

[38] Lipeng Ma, Weidong Yang, Sihang Jiang, Ben Fei, Mingjie Zhou, Shuhao Li, Mingyu Zhao, Bo Xu, and Yanghua Xiao. Luk: Empowering log understanding with expert knowledge from large language models, 2024.

[39] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. LLMParser: An exploratory study on using large language models for log parsing. In *ICSE*, 2024.

[40] Zeyang Ma, Dong Jae Kim, and Tse-Hsun Chen. LibreLog: Accurate and efficient unsupervised log parsing using open-source large language models, 2024.

[41] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. Clustering event logs using iterative partitioning. In *KDD*, 2009.

[42] Masayoshi Mizutani. Incremental mining of system log format. In *SCC*. IEEE, 2013.

[43] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *MSR*, 2010.

[44] Open Cybersecurity Schema Framework. OCSF Schema – Categories, 2025. https://schema.ocsf.io/1.4.0/.

[45] Marcel Parciak, Brecht Vandevoort, Frank Neven, Liesbet M. Peeters, and Stijn Vansummeren. Schema matching with large language models: an experimental study, 2024.

[46] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB*, 2001.

[47] Mikel Rodriguez, Raluca Ada Popa, Four Flynn, Lihao Liang, Allan Dafoe, and Anna Wang. A framework for evaluating emerging cyberattack capabilities of ai, 2025.

[48] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models, 2021.

[49] Eitam Sheetrit, Menachem Brief, Moshik Mishaeli, and Oren Elisha. ReMatch: Retrieval enhanced schema matching with llms, 2024.

[50] Keiichi Shima. Length matters: Clustering system log messages using length of words, 2016.

[51] Splunk Inc. Add-ons and CIM | Splunk Supported Add-ons Documentation, 2025. Accessed: 2025-08-05.

[52] Splunk Inc. Splunk, 2025. Accessed: 2025-04-12.

[53] Byungchul Tak and Wook-Shin Han. Lognroll: Discovering accurate log templates by iterative filtering. In *Middleware*, 2021.

[54] Liang Tang, Tao Li, and Chang-Shing Perng. LogSig: Generating system events from raw textual logs. In *CIKM*, 2011.

[55] Gemini Team and Google. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[56] U.S. Department of Homeland Security. The Menlo report: Ethical principles guiding information and communication technology research. Technical report, Department of Homeland Security, August 2012.

[57] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IPOM*, 2003.

[58] Risto Vaarandi and Hayretdin Bahsi. Using large language models for template detection from security event logs, 2025.

[59] Risto Vaarandi and Mauno Pihelgas. LogCluster - a data clustering and pattern mining algorithm for event logs. In *CNSM*, 2015.

[60] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[61] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, Saravanakumar Rajmohan, and Dongmei Zhang. SPINE: a scalable log parser with feedback guidance. In *ESEC/FSE*, 2022.

[62] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. Demonstration-free: Towards more practical log parsing with large language models. In *ASE*, 2024.

[63] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. DivLog: Log parsing with prompt enhanced in-context learning. In *ICSE*, 2024.

[64] Yongqin Xu, Huan Li, Ke Chen, and Lidan Shou. KcMF: A knowledge-compliant framework for schema and entity matching with fine-tuning-free llms, 2025.

[65] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. Brain: Log parsing with bidirectional parallel tree. *IEEE TSC*, 2023.

[66] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task, 2018.

[67] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task, 2019.

[68] Xian Yu, Shengxi Nong, Dongbiao He, Weijie Zheng, Teng Ma, Ning Liu, Jianhui Li, and Gaogang Xie. LogGenius: An unsupervised log parsing framework with zero-shot prompt engineering. In *ICWS*, 2024.

[69] Jing Zhang, Bonggun Shin, Jinho D. Choi, and Joyce C. Ho. SMAT: An attention-based deep learning solution to the automation of schema matching. In *ADBIS*, 2021.

[70] Wei Zhang, Hongcheng Guo, Anjie Le, Jian Yang, Jiaheng Liu, and Zhoujun Li. Lemur: Log parsing with entropy sampling and chain-of-thought merging, 2025.

[71] Yu Zhang, Mei Di, Haozheng Luo, Chenwei Xu, and Richard Tzong-Han Tsai. SMUTF: Schema matching using generative tags and hybrid features, 2024.

[72] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. LogParser-LLM: Advancing efficient log parsing with large language models. In *KDD*, 2024.

[73] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics . In *ISSRE*, 2023.

A Additional results

We provide here detailed metrics for (1) the template generation metrics for each file in the LogHub2.0 dataset (Section 5.1), and (2) precision and recall for each file in the SecurityLogs dataset for our ablation studies (Section 5.2).

Table 9: Comparison of Matryoshka and prior template generation on LogHub2.0 data

Dataset	Matryoshka		LILAC		Drain3		Brain		Dataset	Matryoshka		LILAC		Drain3		Brain	
	PGS	TS	PGS	TS	PGS	TS	PGS	TS		PGS	TS	PGS	TS	PGS	TS	PGS	TS
Apache	**1.00**	**1.00**	0.99	0.99	0.99	0.89	0.99	0.88	HPC	**1.00**	0.99	**1.00**	**1.00**	0.97	0.96	0.78	0.75
BGL	**0.99**	**0.98**	0.97	0.96	0.95	0.96	0.88	0.79	Linux	0.85	**0.93**	0.83	0.86	**0.87**	0.80	0.81	0.71
Hadoop	**0.97**	**0.91**	0.86	0.81	0.88	0.71	0.35	0.29	Mac	**0.96**	**0.91**	0.78	0.75	0.83	0.82	0.95	0.89
HDFS	**1.00**	**0.96**	0.96	0.84	0.97	0.93	0.97	0.84	OpenSSH	0.97	**0.97**	0.72	0.68	0.96	0.92	**0.99**	0.92
HealthApp	**1.00**	**0.94**	**1.00**	**0.94**	0.97	0.92	0.54	0.44	OpenStack	0.93	0.86	0.93	0.88	0.66	0.67	**1.00**	**0.97**
Proxifier	**1.00**	**0.50**	0.97	0.38	0.52	0.31	0.99	0.35	Spark	**1.00**	**0.97**	0.91	0.93	0.97	0.79	0.97	0.91
Thunderbird	**0.98**	**0.82**	0.82	0.68	0.87	0.68	0.79	0.69	Zookeeper	**0.99**	0.96	0.86	0.83	**0.99**	**0.98**	**0.99**	0.95

Table 10: Matryoshka query precision and recall metrics across Gemini models on SecurityLogs

Dataset	Gemini 2.5 Flash				Gemini 2.5 Pro				Gemini 2.5 Pro Without Validation			
	OCSF		Custom		OCSF		Custom		OCSF		Custom	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
SSHD	0.70	0.70	**1.00**	**1.00**	0.90	0.90	**1.00**	**1.00**	0.80	0.80	**1.00**	**1.00**
Cron	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
DHCP	0.80	0.77	0.90	0.87	0.80	0.77	**1.00**	**0.97**	0.54	0.57	0.94	**0.97**
Audit	0.60	0.60	**1.00**	0.96	0.70	0.69	**1.00**	**0.99**	0.70	0.68	**1.00**	0.95
Puppet	0.70	0.71	0.70	0.71	0.90	0.88	**1.00**	**0.98**	0.90	0.79	**1.00**	0.89
Average	0.76	0.76	0.92	0.91	0.86	0.85	**1.00**	**0.99**	0.79	0.77	0.99	0.96

B Evaluation queries

B.1 DHCP Log Queries

Assigned addresses for a given hostname	Description	Identifies IP addresses assigned to a specific host.
	Query	`assigned_ip exists AND log_host is laphroaig`
	Naive substring	`fgrep "bound to" | fgrep "laphroaig"`
List all server IPs	Description	Enumerates all DHCP server IP addresses that are not broadcast addresses.
	Query	`server_ip != 255.255.255.255`
	Naive substring	`fgrep "port" | fgrep -v "255.255.255.255"`
	Note	The word port seems to always be present next to server IPs
Track all log messages for a given transaction ID	Description	Follows the complete lifecycle of a specific DHCP transaction.
	Query	`transaction_id is 0x6520bf0e`
	Naive substring	`fgrep "xid=0x6520bf0e"`
List the MAC addresses used by a specific host	Description	Retrieves entries containing MAC address information for a particular hostname.
	Query	`mac_address exists and log_host is laphroaig`
	Naive substring	`fgrep "laphroaig" | fgrep "Listening"`
Entries with high renewal times	Description	Identifies DHCP leases with renewal times exceeding 1 day (86400 seconds).
	Query	`renewal_time > 86400`
	Naive substring	`fgrep "renewal"`
	Note	We cannot compare numbers so we can only search for lines that include renewal times.
Servers on non-standard ports	Description	Lists DHCP servers operating on ports other than the standard port 67.
	Query	`server_port is not 67`
	Naive substring	`fgrep "port" | fgrep -v "67"`
Specific client version usage	Description	Identifies log entries associated with a specific DHCP client version (3.0.1).
	Query	`client_version is 3.0.1`
	Naive substring	`fgrep "3.0.1"`
DHCPDISCOVER messages	Description	Lists clients that have sent DHCPDISCOVER messages.
	Query	`DHCP_message_type is DHCPDISCOVER`
	Naive substring	`fgrep "DHCPDISCOVER"`
XMT Renew messages	Description	Lists clients that have issued renewal requests.
	Query	`DHCP_message_type is Renew`
	Naive substring	`fgrep "Renew"`
Bad IP checksums	Description	Identifies packets with incorrect IP checksums.
	Query	`bad IP checksums`
	Naive substring	`fgrep "bad IP checksums"`

Table 11: DHCP Log Queries

B.2 SSHD Log Queries

Password authentication for root	Description	Identifies instances where the root account attempted to authenticate using a password.
	Query	`authentication_method is password and user_name is root`
	Naive substring	`fgrep "password" | fgrep "root"`
Unusual server ports	Description	Detects SSH servers operating on non-standard ports (not port 22).
	Query	`bind_port is not 22`
	Naive substring	`fgrep "port" | fgrep -v "22"`
Usage of specific key	Description	Tracks usage of a particular SSH key based on its fingerprint.
	Query	`key_hash is SHA256:iJC3O+heWZCsp5···+VwGmmcFhEnc`
	Naive substring	`fgrep "SHA256:iJC3O+heWZCsp5···VwGmmcFhEnc"`
Root user SSH keys	Description	Retrieves all SSH key fingerprints associated with root user logins.
	Query	`key_hash exists and user_name is root`
	Naive substring	`fgrep "root" | fgrep "publickey"`
Activity from specific IP on specific date	Description	Monitors all SSH activity from IP 61.143.236.193 on September 25.
	Query	`remote_ip is 61.143.236.193 and log_timestamp > sept. 25 00:00:00 and log_timestamp < sept. 26 00:00:00`
	Naive substring	`fgrep "61.143.236.193" | fgrep "sept. 25"`
Non-SSH terminals for root user	Description	Identifies root logins through terminal types other than SSH.
	Query	`terminal_type is not ssh and user_name is root`
	Naive substring	`fgrep "root" | fgrep "tty" | fgrep -v "tty=ssh"`
Root sessions initiated by non-root accounts	Description	Detects when standard users escalate to root privileges.
	Query	`initiating_user_name is not root and user_name is root`
	Naive substring	`fgrep "user root" | fgrep -v "root("`
	Note	This matches the format of the main template that contains this information
"None" authentication attempts	Description	Identifies login attempts using the "none" authentication method.
	Query	`authentication_method is none`
	Naive substring	`fgrep "none"`
Activity for specific system and process	Description	Tracks SSH activity related to a specific process ID on a particular host.
	Query	`process_id is 4317 and log_host is LIPC003.intranet.local`
	Naive substring	`fgrep "4317" | fgrep "LIPC003.intranet.local"`
Host key mentions	Description	Finds log entries that reference host key files or paths.
	Query	`host_key_path exists`
	Naive substring	`fgrep "host key"`

Table 12: SSHD Log Queries

B.3 Audit Log Queries

Sudo/su usage by non-root users	Description	Identifies when standard users attempt to use sudo or su commands.
	Query	`(executable_path contains /sudo or executable_path contains /su) AND user_id is not 0`
	Naive substring	`fgrep "/su" | fgrep -v "uid=0"`
Denied write operations for rsync	Description	Detects when rsync processes are denied write permissions.
	Query	`avc_operation contains write and process_name contains rsync`
	Naive substring	`fgrep "write" | fgrep "rsync"`
Specific Target SELinux context	Description	Finds log entries with a specific target SELinux context.
	Query	`target_context is system_u:system_r:udev_t:s0-s0:c0.c1023`
	Naive substring	`fgrep "system_u:system_r:udev_t:s0-s0:c0.c1023"`
Devices that had denied calls to mount	Description	Identifies log entries related to mounting storage devices.
	Query	`device_name exists and process_name is "mount"`
	Naive substring	`fgrep "mount" | fgrep 'dev='`
Root user logins	Description	Captures all direct login events for the root user.
	Query	`audit_type is LOGIN and user_id is 0`
	Naive substring	`fgrep "LOGIN" | fgrep "uid=0"`
Audit rule removal	Description	Detects when audit rules are removed from the system.
	Query	`operation contains "remove rule"`
	Naive substring	`fgrep "remove rule"`
SELinux permissive mode setting	Description	Identifies when SELinux is set to permissive mode rather than enforcing.
	Query	`selinux_permissive is 1`
	Naive substring	`fgrep "permissive=1"`
Root directory as working directory	Description	Finds processes operating with root (/) as their current working directory.
	Query	`current_working_directory is '/' or current_working_directory is '"/"'`
	Naive substring	`fgrep "cwd=/ " and fgrep 'cwd="/"'`
Non-binary audit enabled flags	Description	Detects when the audit enabled flag is set to a value other than 0 or 1.
	Query	`audit_enabled is not 0 and audit_enabled is not 1`
	Naive substring	`fgrep "audit_enabled=" | fgrep -v "audit_enabled=1" | fgrep -v "audit_enabled=0"`
Remote SSH connections to specific host on specific date	Description	Tracks remote hosts that established SSH connections to a particular server on a specific date.
	Query	`audit_datetime >= Aug 3 and audit_datetime < Aug 4 and terminal contains ssh and remote_hostname exists and audit_host is perfc-380g8-01`
	Naive substring	`fgrep "perfc-380g8-01" | fgrep "Aug 3" | fgrep "terminal=ssh" | fgrep "hostname"`

Table 13: Audit Log Queries

B.4 Cron Log Queries

List all executed jobs on a host at a specific date	Description	Identifies entries with executable paths on a particular host within a specific date range.
	Query	`executable_path exists and log_timestamp >= 2017-07-14 and log_timestamp < 2017-07-15 and log_hostname is httpboot`
	Naive substring	`fgrep "CMD" | fgrep "2017-07-14" | fgrep "httpboot"`
Entries with scaling factor	Description	Locates log entries that contain scaling factor information.
	Query	`scaling_factor exists`
	Naive substring	`fgrep "factor"`
Specific process ID	Description	Finds entries related to a specific process ID.
	Query	`process_id is 24225`
	Naive substring	`fgrep "24225"`
CRON session openings for root	Description	Lists all session openings for the root user before a specific date.
	Query	`opened and username is root and log_timestamp < 2017-07-15`
	Naive substring	`fgrep "opened" | fgrep "root" | fgrep "2017-07-14"`
	Note	We cannot compare dates so we look for the day before
CRON session closings	Description	Lists all session closings before a specific date.
	Query	`closed and log_timestamp < 2017-07-15`
	Naive substring	`fgrep "closed" | fgrep "2017-07-14"`

Table 14: Cron Log Queries

B.5 Puppet Log Queries

Resource-specific failures for a host	Description	Retrieves failure reports about a specific Puppet resource on a particular host.
	Query	`puppet_resource is Service[galera] AND has failures AND log_hostname is controller1`
	Naive substring	`fgrep "Service[galera]" | fgrep "has failures" | fgrep "controller1"`
Revoked certificates	Description	Identifies logs reporting revoked certificates.
	Query	`certificate_common_name exists AND revoked`
	Naive substring	`fgrep "revoked"`
Specific error code on similar hosts	Description	Finds hosts with similar naming patterns experiencing a specific error code.
	Query	`error_code is 14 and log_hostname contains maca`
	Naive substring	`fgrep "14" | fgrep "maca"`
Host associated with specific request ID	Description	Identifies the host linked to a particular Puppet request ID.
	Query	`request_id is req-9ac8edb7-f81f-44a7-9f34-9a375e7df573`
	Naive substring	`fgrep "req-9ac8edb7-f81f-44a7-9f34-9a375e7df573"`
Interval value changes	Description	Tracks changes to interval values in Puppet configurations.
	Query	`attribute_name is interval and new_value exists`
	Naive substring	`fgrep "interval"`
Specific SQL password hash detection	Description	Checks if any SQL-related resources contain a specific password hash value.
	Query	`attribute_name is password_hash and new_value contains D602AB02F4227D3EBF5FE6EA0323BD6D586A7454 and reporting_resource contains sql`
	Naive substring	`fgrep "D602AB02F4227D3EBF5FE6EA0323BD6D586A7454" | fgrep "sql"`
Extended Puppet run durations	Description	Identifies Puppet runs that took longer than 1 hour (3600 seconds).
	Query	`run_time > 3600`
	Naive substring	`fgrep "catalog run"`
	Note	We cannot compare numbers without parsing
Non-localhost server connections	Description	Lists connections to non-localhost servers by a Puppet agent on a specific date.
	Query	`server_ip is not 127.0.0.1 and log_hostname is puma03 and log_timestamp >= Jan 8 and log_timestamp < Jan 9`
	Naive substring	`fgrep "127.0.0.1" | fgrep "puma03" | fgrep "Jan 8"`
Firewall persistence failures	Description	Identifies cases where firewall rules cannot be persisted.
	Query	`Unable to persist firewall rules`
	Naive substring	`fgrep "Unable to persist firewall rules"`
HTTP URL targets	Description	Lists log entries with HTTP URL targets.
	Query	`target_url contains http://`
	Naive substring	`fgrep "http://"`

Table 15: Puppet Log Queries

C Example Prompts

Template generation system prompt

```
Your are an expert systems analyst. Your role is to create log parsers. Log files are collections of entries, where each entry is
produced by a program to record its state. Log lines can be grouped according to the formatting string that produced them in their
original program. These formatting strings define the template for each group of lines. Since we only observe the log entries, we
must infer these templates.

Templates consist of static parts (fixed text from the formatting string) and variable parts (variable entries in the formatting
string). Templates can be broken down into tokens, where each token represents a single semantic unit within the log entry. A token
can be a single character, word, or multiple words. Templates aim at capturing as much information as possible from the log entries,
so that the logs can be queried and analyzed.

Your task is to infer a template from a set of log entries. Here are the key concepts about tokens:

### Variability ###

Templates contain two distinct types of tokens:

* Variable tokens, or entities: These represent parameters from the original formatting string. They are parts of the template that
can vary between log lines from the same group. Variable tokens must represent distinct entities like resources, devices, locations,
dates, times, or users. In key-value pairs, the value is always a variable token. Variables cannot be messages, descriptions,
actions, or complete sentences - they must be specific parameters. We also refer to these as entities. Values can be variables even
if they do not change between log lines, as long as they represent entities or parameters. Examples of entities that should always
be variable tokens are: timestamps, IP addresses, hostnames, ports, usernames, paths, URLs, usernames, email addresses, phone
numbers, MAC addresses, UUIDs, values in quotes, unique identifiers, authentication methods, etc.

* Static tokens: These are the fixed parts of the formatting string that appear identically in every line produced by this template.
They typically include descriptions, punctuation marks, or structural elements that provide context and connect variable entities.
Static tokens include verbs describing actions, descriptive messages, and keywords in key-value pairs. They encompass anything that
is not strictly a variable. Entities that do not change between log lines should NOT be treated as static, but as variables.

### Tokenization ###

* Tokens can either be static or variable, but not both.
* Static tokens should be broken down into their most granular components: Tokens should break at punctuation or other separators.
Queriable concepts should be isolated into their own tokens. Action verbs or nouns should be separate tokens. In short, make sure
tokens are semantic groupings of related words into logical phrases.
* Variable tokens that contain multiple types of data must be broken down into their individual components: They can only include
one parameter from the formatting message. For instance, variables that contain two non related data types that can vary
independently should be in different tokens. Pay attention to separators, these often indicate there are multiple variable.
* Punctuation should be kept separate from variables, except if that punctuation is part of the variable (such as punctuation used
to separate multiple items in a single variable).
* Valid json data formats (dictionaries, arrays, lists, etc.) must be kept as one single token.
* If you encounter key-value pairs, the key (or keyword) should be in a separate token from the value. The key should always ve
static, the value should always be variable (even if it does not change in the provided examples). Key value pairs are sets of
tokens in which one represents the name of the field, and the other is the value, such as "port 5836", "uid=0", or "retries: 1".
* If you encounter nested key-value pairs (key-value pairs that are part of the value of a larger key-value pair), the inner-most
key-value pair shouyld be separted. For instance, in "message='port:5836,uid:0'", "port" and "uid" should be separate constant
tokens, and "5836" and "0" should be separate variable tokens.
* Units associated with variables should be included in the variable: for instance, time units, size units, or distance units,
should be part of the entry they follow.
* Fields cannot be optional: variables cannot be empty.
* Some data types have specific rules:
** Hostnames, IP addresses and ports should be in separate tokens. "127.0.0.1:53" should be broken down into two variable tokens:
"127.0.0.1" and "53".
** Paths and URLs should be kept as a single token: do not separate them into sub tokens. "https://google.com/search?q=log+parser"
is a single variable token. So is "/var/log/sshd.log".
** Dates and times should be merged into a single token if adjacent. "Apr 7, 2025 12:30:00" is a single variable token.
** Hex values should include the 0x prefix if present. "0x12345678" is a single variable token.

### Regular expressions ###

Variable tokens are associated with a regular expression. These help capture the expected syntax of the variable. A good regular
expression should match all possible values the variable tokens of a given type can take. They should capture specifics about the
expected data format - for example, a date, a time, a file path, or a user name. They should not be specific to observed subsets of
values, but generalize to any possible value of the same type. Similar values accross templates should reuse the same regex. Do not
use capturing groups in the regex.

### Avoiding Overcapture ###

Avoid overcapture by ensuring the regular expression captures the structure of the variable. For instance, json objects should be
captured using `\\{.*\\}` and arrays should be captured using `\\[.*\\]`.

### Consistency ###
```

```
* When creating templates, ensure that the same entity is represented in the same way across all templates. For example, a timestamp
including a date and time is parsed as a single entity in one template, it should be parsed the same way in all templates.
* If a variable is represented as a single token in one template, it should be represented as a single token in all templates.
* If a variable is broken down into multiple tokens in one template, it should be broken down in the same way in all templates.
* Reuse the same regular expressions for variables that have the same structure across templates.

### Representation ###

* We represent templates as a JSON list of tokens, which are objects with the following fields:
   * "is_variable": a boolean indicating if the token is an entity or a variable (true) or a static part of the formatting string
   (false)
   * "value": the value of the token
* For example, log line `update-alternatives 2022-10-04 22:32:23: uid=0 run with --install git` is produced by template:
'''json
[
   {"is_variable": false, "value": "update-alternatives"},
   {"is_variable": true, "value": "2022-10-04 22:32:23", "placeholder": "<DATETIME>", "regex": "\\d{4}-\\d{2}-\\d{2}
   \\d{2}:\\d{2}:\\d{2}"},
   {"is_variable": false, "value": ":"},
   {"is_variable": false, "value": "uid="},
   {"is_variable": true, "value": "0", "placeholder": "<UID>", "regex": "\\d+"},
   {"is_variable": false, "value": "run with --install"},
   {"is_variable": true, "value": "git", "placeholder": "<COMMAND>", "regex": "\\S+"}
]
````

Rules

I will give you log entries. You will output the template that matches the entries.
Make sure you take into account all parts of the entries, including any punctuation or special characters.
You must adhere to all guidelines about tokenization, and the definition of variables and static tokens.
Most importantly, ensure your responses are consistent with previous templates you have generated.
Entities in new templates should be parsed in the *exact* same way as in previous templates.

In order to complete this tasks, follow the following algorithm:

'''algorithm
print("<explanation>")
First, look at prior templates and explanations to understand the expected format. Ensure entities are parsed in the same way as in
previous templates.
Print a description of the log lines consistent with the previous descriptions you wrote. Do not include more a less information
than necessary and that is provided in previous descriptions.

print("Placeholders")
Print the first line where entities are replaced with placeholders. All entities must be present. Do not include placeholders for
missing values (e.g., if there is a key but no value).

print("Key Value Pairs:")
What is the format of the string that produced these suffixes? Are there any key-value pairs? List all key value pairs. Remeber,
keys are static, values are entities. If you encounted nested key-value pairs, break them down into their most granular components,
separating the inner-most components into their own variables and constants.

print("Static tokens:")
Which part is a static message, description, or action? List all the static tokens.
Concepts that could be queried should be isolated into their own tokens.
Action verbs, nouns and semantic units should be in their own tokens.
Break down static tokens into their most granular components, separating punctuation and other separators into their own tokens.

print("Entities:")
Which part is an entity? List all the entities, including:
* those that do not change.
* any value in a key value pair (except if the value is empty)
* any value what is one of: timestamps, IP addresses, hostnames, ports, usernames, paths, URLs, usernames, email addresses, phone
numbers, MAC addresses, UUIDs, values in quotes, unique identifiers
Remember, entities cannot contain messages, descriptions, actions, or complete sentences - they must be specific parameters.
For each entity, print an example value and a regular expression that matches the expected syntax of the entity. Make sure this
regular expression mathches all possible values the entity can take.

print("</explanation>")

template = Use the explanation and placeholder above when generating the template for this message.
Make sure any token that was assigned a placeholder is a variable token in the template.

template = set all variable values in the template to be equal to their values in the first log entry

print(template) IN JSON
'''
```
```

Template validation system prompt

```
You are an expert systems analyst. Your role is to create log parsers. Log files are collections of entries, where each entry is
produced by a program to record its state. Log lines can be grouped according to the formatting string that produced them in their
original program. These formatting strings define the template for each group of lines. We have identified a set of candidate
templates, along with some lines they match. These templates are structured in a parsing tree, so they share common tokens. Your
task is to correct this parsing tree so all templates are consistent with each other and with the guidelines below.

Templates consist of static parts (fixed text from the formatting string) and variable parts (variable entries in the formatting
string). Templates can be broken down into tokens, where each token represents a single semantic unit within the log entry. A token
can be a single character, word, or multiple words. Templates aim at capturing as much information as possible from the log entries,
so that the logs can be queried and analyzed.

# Guidelines about parsing tree

[same prompt as variability and tokenziation in the template generation system prompt]

## Templates

Templates are branches in the tree. They represent a single formatting string, while the tree represents a set of formatting
strings. The last node in a template is marked with the ID of the template. Nodes that are frozen cannot be changed. Use these as a
reference, and only change nodes that are not frozen.

## Guidelines about tree consistency

The tree was build by adding templates one by one. Each of the templates was generated separately, and may have different
formatting, tokenization, or variable definitions. This means the tree is not consistent. Your task is to make sure the tokens in
the tree follow the guidelines above, but most importantly, that the templates that are part of the tree are consistent with each
other.

## Separation consistency

Ensure tokenization is consistent across tokens. If a token is broken down into multiple tokens in one template, it should be broken
down in the same way in all templates. Use parallel structures for tokens that play similar roles across templates. If you encounter
different tokenization for the same entity across templates, choose the most granular and consistent tokenization across all
templates.

## Variable consistency

* Ensure that variables are consistent across templates. If a token is a variable in one template, it should be a variable in all
templates. If a token is a static token in one template, it should be a static token in all templates. If a token is inconsistently
defined as a variable in one template and a static token in another, choose the most appropriate definition based on the guidelines
above.

## Pitfalls

### Focus on inconsistencies between templates, not within templates

Your task is to enforce consistency among templates, not within templates. Any change must be motivated by the fact there are two or
more templates with similar structures but inconsistent tokenizations.

### Inference from partial information

* Do not assume that the lines provided are fully representative of every possible line that can match the template. As such, do not
change tokens from variables to constants unless you need to do so for consistency amongst templates and you are convinced the value
is static and does not represent any entity or queryable parameter.

### Word-by-word tokenization

* Do not default to splitting constants on spaces. In some contexts, keeping a few constant words together is better than splitting
them. Only split on spaces if you need to isolate one of the constant words because it represents a queryable concept.

### Missed variables

* In some cases, the original templates might miss the fact a token is a variable. This is noticable when you see all of the
following:
  * Multiple templates with parallel structures, which share a common prefix that only differs by a single static token.
  * These templates represent the same type of event.
  * The token fits the definition of a variable, and changing it to a variable would not lead to an overcapturing template.
* If all of these are true, then merge these template prefixes into a single prefix in the tree by replacing the static token with
an appropriate new tokenization, and moving the parallel branches into this new prefix.

### Trying to change frozen tokens


* Do not change tokens that are marked as frozen. These are part of the tree that is fixed, and should be assumed consistent. Use
this part of the tree as a reference, and only change tokens that are not frozen. You might notice some inconsistencies within the
frozen parts of the tree: these can be noted, but they should not be changed.
```

Ignoring whitespaces

* Pay attention to whitespaces: some nodes are identical except for the presence or absence of a single space after.

Task description

You will be given a set of entries, as well as the parsing tree that matches them. Your task is to correct the tree. You can perform the following actions:

```python
node_type = {
    "type": "object",
    "properties": {
        "is_variable": {"type": "boolean"}, # True if the node is a variable, false if it is a static token
        "value": {"type": "string"}, # The observed value of the node. For a constant, this is the value of the constant. For a variable, it's an example value of the variable in one of the lines that matched the template.
        "regexp": {"type": "string"}, # The regular expression used to match the node.
        "id": {"type": "integer"},
        "placeholder": {"type": "string"},
        "trailing_whitespace": {"type": "integer"} # Number of spaces between this token and the next token
    }
}

def CREATE(parent_id: int, value: node_type) -> int:
    """
    Create a new node in the tree and connect it to the parent node.
    If parent node is 0, the new node should be a root node.
    If an ID value is provided in the value, it will be ignored; a new ID will be generated.

    Args:
        parent_id: The ID of the parent node.
        value: The value of the new node.

    Returns:
        The ID of the new node.
    """
    pass

def EDIT(node_id: int, new_value: node_type) -> None:
    """
    Update the value of an existing node in the tree.

    Args:
        node_id: The ID of the node to update.
        new_value: The new value of the node. Non specified fields will not be updated. IDs cannot be changed.
    """
    pass
```

```
def DELETE(node_id: int) -> None:
    """
    Delete an existing node in the tree.
    If it has children, they will be connected to the parent of the deleted node.

    Args:
        node_id: The ID of the node to delete.
    """
    pass

def MOVE(node_id: int, new_parent_id: int) -> None:
    """
    Move an existing node in the tree to a new parent.
    All templates that contain this node will be updated to connect to the new parent.
    This will throw an error if the this creates cycles or if the node_id is the same as the new_parent_id.

    Args:
        node_id: The ID of the node to move.
        new_parent_id: The ID of the new parent node.
    """
    pass

def REPLACE(node_id: int, values: list[node_type]) -> list[int]:
    """
    Replace a node in the tree with a list of nodes.
    The parent of the current node will point to the first node in the list.
    The nodes in the list will form a chain, and the last node in the list will point to the current node's children.
    This will return a list of the new node ids for the new chain.
    The current node will be replaced with the first node in the list.
```

````
    Args:
        node_id: The ID of the node to replace.
        values: The values of the new nodes.

    Returns:
        The IDs of the new nodes.
    """
    pass

def ADD_TEMPLATE(node_id: int) -> int:
    """
    Mark a node in the tree as being the end of a new template.

    Args:
        node_id: The ID of the last node in the new template.

    Returns:
        The ID of the new template.
    """
    pass

def DELETE_TEMPLATE(template_id: int) -> None:
    """
    Delete a template from the tree.
    This will delete the template from the tree.
    All nodes that are part of the template and are not shared with other templates will be removed.

    Args:
        template_id: The ID of the template to delete.
    """
    pass
```

Avoid using DELETE_TEMPLATE and ADD_TEMPLATE if you just need to edit a template and can use MOVE and REPLACE instead.

Please first explain what changed need to be made to the tree for it to be consistent. Then, write some python code in a markdown code block (delimited by ```python and ```) that uses these primitives to edit the tree. Make sure your changes lead to a valid tree, and that the tree still parses the set of entries. Also make sure the code can execute: do not write it inside a function, but instead make sure executing your code as is will work (e.g. running python's exec on your code block). Remember, frozen nodes cannot be changed.
````
````