LogSense: Scalable Real-Time Log Anomaly Detection Architecture

Abhay Srivatsa, Venkatasai Gudisa

Abstract

Most scalable systems lack an equally scalable anomaly detection architecture for their logs, which is crucial to maintaining integrity. This paper introduces LogSense, an all-in-one solution that leverages state-of-the-art LLM-based Machine Learning models and advanced stream processesing frameworks to swiftly identify and address anomalies within complex log data streams. By focusing on an easy-to-use plug-and-play configuration, LogSense differentiates itself by inviting and effectively treating distributed systems. Orchestrated by efficient messaging and backed by user-driven refinement, LogSense offers a robust architecture capable of delivering rapid and accurate insights into system anomalies.

I. Background

What are Logs?

The key to understanding any computer system is through its logs: chronological records of system events. While an overwhelming majority of these logs are generated by system startups, resource accesses, and other mundane events, deviations from the standard flow result in anomalies. The term "normal" refers to logs associated with mundane, expected events while "anomalies" are unwanted logs that indicate unexpected system events. Analogous to a human body, normal logs represent expected health indicators, while anomalies serve as symptomatic signals that may lead to adverse consequences. Consequently, as doctors, programmers must accurately diagnose and promptly treat the anomalies through the use of Log Anomaly Detection – a pivotal process in system management.

Log Anomaly Detection

In an ideal world, programmers can simply monitor logs manually, identify any anomalies as they occur, and treat them efficiently. It would be like standing on the shore of a beach and picking up seashells from a soft wave as it brushes your feet. Unfortunately, the reality is much more grim. The wave of logs current systems usually generate resembles less of a ripple and more of a tsunami, necessitating automated detection methods.

Before we delve into algorithms we can utilize, it's important to keep in mind that while log anomalies are analogous to symptoms of illness, we can't simply define a set of rules to conclude. Given the unpredictable nature of anomalies, traditional rule-based approaches fall short. For example, let's say you implement a rule-based system that scans network logs for occurrences such as continued failed logins from an IP or login attempts from known malicious IPs. While this schema can detect well-known attacks that match the predefined signatures, it would be unable to detect any novel or sophisticated threats. Attackers could very easily tailor an attack to your system's infrastructure, bypassing the IP rules or any other rules you've defined previously. This necessitates the adoption of continually improving algorithmic strategies capable of establishing norms within the logs and discerning any deviations therein.

Traditional Approaches

Algorithmic approaches to Log Anomaly Detection can be split into two subsets – supervised and unsupervised. Supervised learning uses annotated data to learn the patterns in both normal and abnormal data and classify any new data as normal or anomalous. In contrast, unsupervised models internalize the structure of normal logs to define any heavy deviations as anomalous. While both schemas present their own set of advantages and disadvantages, supervised models are usually unrealistic due to their dependence on annotated data. The incredible imbalance between normal and anomalous log data along with a general lack of labeled data limits the effectiveness of Supervised Learning. On the other hand, Unsupervised Learning is largely based on statistical knowledge, causing it difficulties in capturing and interpreting complex anomalies. Therefore, a mix of both must be employed to find the few needles in the enormous pin stack.

Historically, Principal Component Analysis (PCA), Support Vector Machines (SVMs), and Invariant Mining have been employed for log anomaly detection. Principal Component Analysis (PCA) is a dimensionality reduction technique that is widely utilized in the world of data. It identifies and projects data onto the

directions (principal components) of maximum variance within, reducing its dimensionality and removing noise. In the case of log anomaly detection, popular PCA-based approaches reduce the dimensionality of event-count vectors, identifying outliers as anomalies [1]. Another approach that utilizes event counts is a Support Vector Machine. SVMs work by examining normal and anomalous events and then defining a boundary in a high-dimensional space that best separates them. Conventionally, SVMs are most effective in supervised learning environments, but a variation known as the One-Class SVM [1] has been shown to work well by defining a boundary that encompasses the most normal logs possible, defining any logs outside that boundary as anomalous. The primary drawback of these methods is the loss of semantics – treating log lines as simple event counts disregards any meaning encoded in them. Invariant Mining, on the other hand, takes us a step closer in gaining a deeper understanding of the logs. Invariant Mining [1] discerns linear relations between log occurrences and decides if a new log follows predefined workflows of execution. Gaining a deeper understanding of the logs necessitates more sophisticated techniques rooted in Deep Learning.

Deep Learning-based Approaches

Deep Learning leverages interconnected layers of nodes that perform matrix multiplications to learn increasingly abstract forms of representing data. Due to its adaptability and effectiveness, it has found widespread applications in various domains, including Predictive Analysis, Image Recognition, and Natural Language Processing (NLP). NLP enables computers to interpret human language contextually, encoding as much meaning as possible. Since the majority of a log is composed of human language written by programmers, NLP-based models can treat it as a linguistic entity. This, in turn, helps NLP facilitate a deeper understanding of system behaviors, enabling more accurate anomaly detection and diagnosis.

II. Models

Preliminary

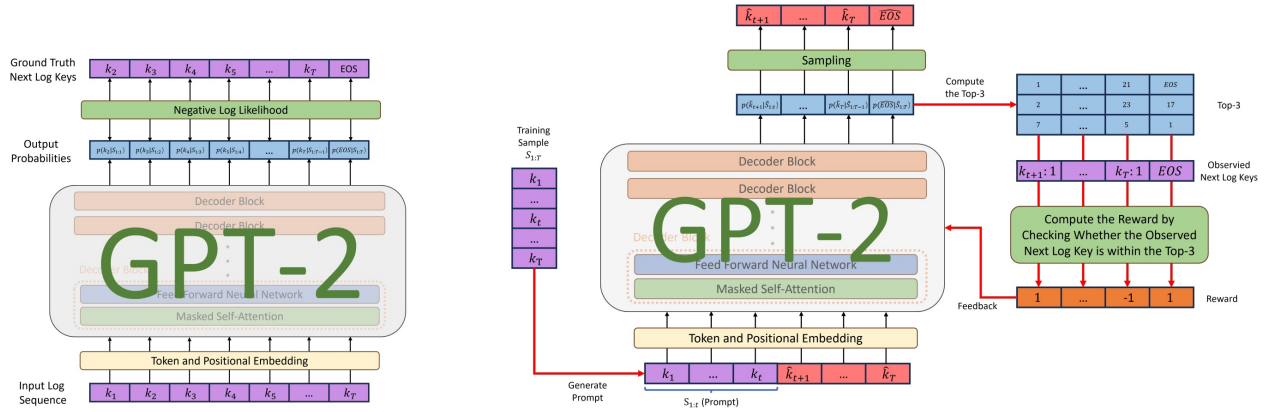
Before delving into NLP-based models, it's imperative to establish foundational knowledge. All of the following models typically undergo a process involving preprocessing of log lines, embedding using transformer-based architectures like BERT or GPT, and subsequent inferencing using these embeddings. State-of-the-art NLP, models such as BERT and GPT, employ transformers to encode text comprehensively, leveraging vast pre-training on immense corpora to capture the essence of log lines. Ultimately, anomaly detection techniques examined in this study determine the regularity of log entries by extracting their meaning from learned embeddings.

LAnoBERT

LAnoBERT [2] is a relatively recent approach rooted in the idea that anomalous logs often exhibit discernible contextual clues. If we can understand what normal contexts look like, we can flip this knowledge to define logs as abnormal based on their context. To this end, the model is trained on normal logs using Masked Language Model (MLM) techniques, where tokens representing log lines undergo masking, fostering context learning. Initially, preprocessing steps filter out extraneous information such as numerical values and IP addresses due to their irrelevance to the task at hand. Next, a predefined percentage of logs are masked at random, forcing a wide comprehension of the corpora. Similarly, during testing, every token in a sequence undergoes masking one at a time – tokens assigned with anomalously low probabilities are deemed as such due to their deviation from learned contexts. In essence, if a log is given a low probability of occurrence with its context, it's most probably anomalous.

LogGPT

LogGPT [3], akin to LAnoBERT, employs transformer-based embeddings, albeit using GPT-2. Noteworthy distinctions lie in LogGPT's preprocessing, which entails log parsing to derive log templates that can be mapped to event ids within the sequence being considered. Next-token prediction during pre-training replaces MLMs since LogGPT argues that an MLM would reduce the model's ability to comprehend and exploit the natural flow of normal logs. This pre-training is coupled with a Reinforcement Learning fine-tuning system that sharpens model understanding. The model is led to predict top-k (k being a configurable integer) logs for the next log in the sequence given all of the previous logs. If this prediction contains the actual log line, it's rewarded with a +1, otherwise, it's punished with a -1. Likewise, our analysis during testing relies on token predictions within a top-k framework to identify anomalies – if the log isn't in the model's prediction of the top-k, it's an anomaly. The logic behind this decision is pretty simple – the model first learned sequential information during the pre-training stage and then got sharper during the fine-tuning stage. So if the model can't predict a token in its top-k after all the rigorous learning it went through, there's a very high



(a) Pre-training (b) Fine-tuning

Figure 1: *Pre-training and Fine tuning process of GPT-2 used in LogGPT. Pre-training aims to deepen understanding in the area while Finetuning sharpens its ability on downstream tasks (such as anomaly detection in our case)* ***[3].***

chance of it being anomalous.

RAPID

Our implementation of RAPID [4] diverges markedly by focusing on individual log entries devoid of sequential context. After employing regex cleaning and tokenization, RAPID utilizes BERT embeddings to perform inference. It first compares each log's [CLS] token, a token at the beginning of a sentence's encoding that captures the meaning of the sentence as a whole, to a database of normal logs to obtain a core set of its closest neighbors. In this case, tokens refer to the individual parts of one log line rather than entire log lines like in LogGPT. If the max similarity score from the current log to any of its closest neighbors is past a certain threshold, it's an anomaly. By eschewing sequential context considerations, RAPID optimizes for speed without sacrificing significant accuracy.

$$maxSim(q,d) = \sum_{i \in [|E_q|]} \max_{j \in [|E_d|]} E_{q_i} \bullet E_{q_j} \qquad (1)$$

Differences

Treatment of semantics varies among models. LAnoBERT and LogGPT prioritize contextual cues at the expense of computational efficiency, while RAPID emphasizes individual log analysis for speedy anomaly detection. Balancing contextual and token-level information is pivotal for accuracy, with LogGPT emerging as a pragmatic choice to accompany RAPID. Although LAnoBERT boasts higher scores, it considers both past and future logs as part of its inference. This highly contrasts usability in the real world, since present log lines should never be analyzed in the context of future logs. Therefore, LogGPT's efficient utilization of past log contexts for prediction is preferred over LAnoBERT. Additionally, the tradeoffs between RAPID and LogGPT are also what unites them into the perfect tandem.

III. LogSense

Getting Started

Configuration

To initiate their journey with LogSense, users engage with the platform's web application to configure a service tailored to their specific needs. Beyond providing basic details like the service's name and description, users can define various hyper-parameters specific to RAPID and LogGPT, streamlining the process for those versed in machine learning while offering guidance for newcomers. The configuration process integrates seamlessly with the Promtail setup on each node, ensuring a plug-and-play experience aimed at facilitating user adoption and fostering a positive initial interaction.

When the service configuration is altered in the future, Kafka is leveraged to propagate these changes throughout the LogSense architecture. The backend system dispatches a message to Kafka, instigating an event that seamlessly incorporates the updated configuration settings into subsequent Flink preprocessing tasks. As we will discuss later on, this integration underscores Kafka's robust capabilities within the context of Flink's stream processing

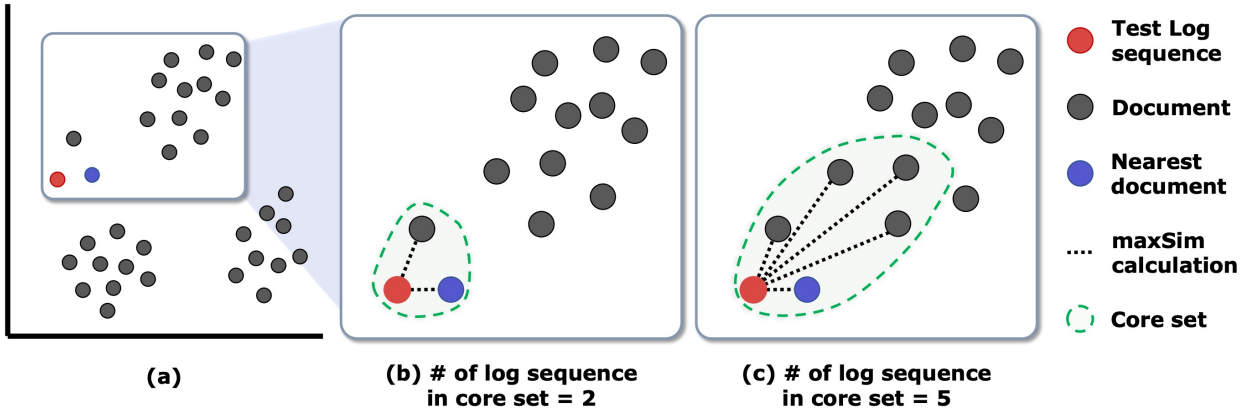


Figure 2: *Visual 2D representation of how the core set is derived in RAPID. Neighbors obtained through the core set are then used to calculate the maxsim score from above* ***[4]****.*

paradigm, facilitating instantaneous adaptation to evolving user-defined configurations.

Train Mode

Deep-learning-driven log anomaly detection relies on comprehending the structure of normal logs to identify deviations indicating anomalies. This requires supplying the models with labeled normal data at the outset, a prerequisite addressed through LogSense's innovative Train Mode. By temporarily suspending inference, Train Mode allows incoming logs to be treated as normal data for model bootstrapping. This kick-starts model training while providing ongoing opportunities for model refinement, amplifying their effectiveness over time. Feeding normal logs into these models is the equivalent of providing greater support to the foundation of a home – the greater the quantity the more robust the models will be.

Online Architecture

Loki Aggregation

Upon configuration, logs flow into Loki, a robust aggregation system optimized for storage and indexing. Leveraging tagging capabilities and an efficient Minio object storage database, Loki organizes logs for efficient querying and analysis, laying the groundwork for subsequent querying. Tagging refers to associating logs with data such as their filename, node, service, and timestamp all of which lead the way for metadata utilization later on. This data is then ingested by Kafka, a high-throughput messaging platform that orchestrates seamless data movement across LogSense's architecture, ensuring scalability and fault-tolerance. The importance of Kafka to LogSense can't be overstated. It's essentially the backbone of the entire operation, alleviating the stress from producers and consumers by providing a streamlined approach for communication within the architecture's components.

Stream Processing

The core of LogSense's stream processing infrastructure relies on Apache Flink, a robust stream processing framework renowned for its capability to handle high-throughput, low-latency, and continuous data streams while ensuring fault tolerance and exactly-once processing paradigm. LogSense's intricate architecture necessitates the coordination of 2 flows of data within the Flink preprocessing.

RAPID's Flink flow operates on a per-log basis, prioritizing low latency at the expense of semantic richness. This approach aligns with LogSense's overarching design philosophy, leveraging LogGPT to compensate for any loss in semantic context. The preprocessing pipeline for RAPID within Flink follows a meticulously orchestrated sequence:

1. Each log undergoes regex-based cleaning, employing user-defined service-based regex patterns configured during service setup.
2. The cleaned log is subsequently tokenized utilizing the WordPiece Tokenizer and hashed using a 64-bit hash function.
3. The resulting tokens, along with their corresponding hash values and log tags, are dispatched to Kafka for transmission to the inference cluster, where they are utilized for generating embeddings and updating the QDrant vector DB during the inference phase.

In contrast, the Flink preprocessing flow for LogGPT diverges significantly due to the model's unique requirements, particularly its reliance on fine-tuning and pre-training mechanisms to enhance perfor-

Figure 3: *Seperate flows of Flink. RAPID is short and simple, aimed at quick preprocessing while LogGPT is patient to gather a minute's worth of logs before preprocessing.*

mance. The preprocessing workflow for LogGPT is intricately tailored to accommodate these needs:

1. Upon data ingestion, logs are segregated based on their associated service, facilitating granular processing.
2. When Train Mode is enabled, a stateful mapping operation is performed on each log key to determine if the current sequence must be used for pre-training or fine-tuning. Essentially, if a stateful iterator increases past a certain threshold, subsequent sequences are marked to be fine-tuned. The threshold requirement is rooted in the fact that extensive pre-training leads to over-generalization of the model, leading to adverse effects on its F1 score. As shown in Figure 5, numerous datasets benefit from limiting the number of logs used during pre-training.
3. Logs earmarked for pre-training or fine-tuning are stored in MongoDB for subsequent offline training sessions, ensuring regular updates to model parameters.
4. Logs earmarked for regular processing (train mode turned off) undergo sequential processing in chunks from 1-minute windows, involving regex-based cleaning followed by hashing and individual parsing using the Drain3 Log Parser. Chunks within the 1-minute window are necessary due to the limitations of GPT in considering context size.
5. Parsed logs are mapped to their respective log templates, yielding a sequence of log keys that serve as inputs for subsequent inference tasks.
6. Hashed log lines and associated metadata are routed to the inference cluster for real-time analysis and decision-making.

Real-time Inference

The beauty of Kubernetes and Docker is on full display in LogSense as they're used to facilitate the horizontal scaling of the inference process to accommodate the substantial volume of incoming logs. Docker orchestrates the creation of container images containing the requisite code for each model, which are subsequently pushed onto a designated container registry. Kubernetes, in turn, interacts with the container registry to retrieve these images and deploy them dynamically, optimizing resource allocation to efficiently handle the incoming data streams. Within LogSense, Docker and Kubernetes collaborate to instantiate new model instances capable of efficiently processing large volumes of inputs.

The disparities between RAPID and LogGPT are most pronounced in the context of inference. RAPID benefits from the expedited arrival of pre-processed logs, resulting in quicker inference decisions compared to LogGPT. This deliberate architectural design underscores RAPID's emphasis on low latency, while LogGPT prioritizes semantic comprehension.

RAPID leverages a Qdrant vector database to store normal logs uniquely identified with the text's hashed representations, enabling efficient identification of logs we've regarded as normal previously and their subsequent judgment. For unidentified logs, RAPID employs an efficient Approximate Nearest Neighbors (ANN) algorithm to obtain the log's core set across the Qdrant database. This approach, supported by vector databases like Qdrant, enables quick retrieval of nearest neighbors due to the efficient storage of vectors. The user-configurable core-set size provides flexibility in defining the anomaly tolerance level. Following the core-set determination, RAPID computes the max-similarity metric by aggregating token similarities and finally applying a dynamically adjusted threshold to classify logs as normal or abnormal. This threshold is continually refined based on feedback, ensuring improved accuracy over time. To initialize it, we must first compute the highest max sim score from the normal database by masking each log line from its own core set. This max similarity is used as the first threshold, meaning any log line that scores above it is displayed as an anomaly. If the user sends feedback stating it was a false positive, the log is marked as normal in the anomaly database and the threshold is pushed to cover the max sim of that log line. Essentially, the threshold is meant to be at the maximum similarity of the least similar log we've encountered thus far. As more and more false positives are detected, the false positive rate is continually improved.

In contrast, LogGPT adopts a top-k system to evaluate log sequences, leveraging a pre-trained and fine-tuned GPT-2 model to predict the top-k tokens for the subsequent log entry. Anomalies are identified when the actual token falls outside the predicted set, reflecting a departure from the anticipated sequence pattern. LogGPT's anomaly predictions are stored in MongoDB, facilitating user feedback integration for anomaly refinement and offline model enhancement.

HITL Updating

Human-in-the-loop (HITL) updating represents a pivotal mechanism whereby users possess the capability to rectify any false positives displayed on the LogSense platform. Upon the cluster's determination of a log as anomalous using RAPID, it is presented to the user via the website interface for validation. Should the user confirm that the log is indeed normal, the model service is invoked to rectify the anomaly database accordingly and append the log entry to the normal database. This

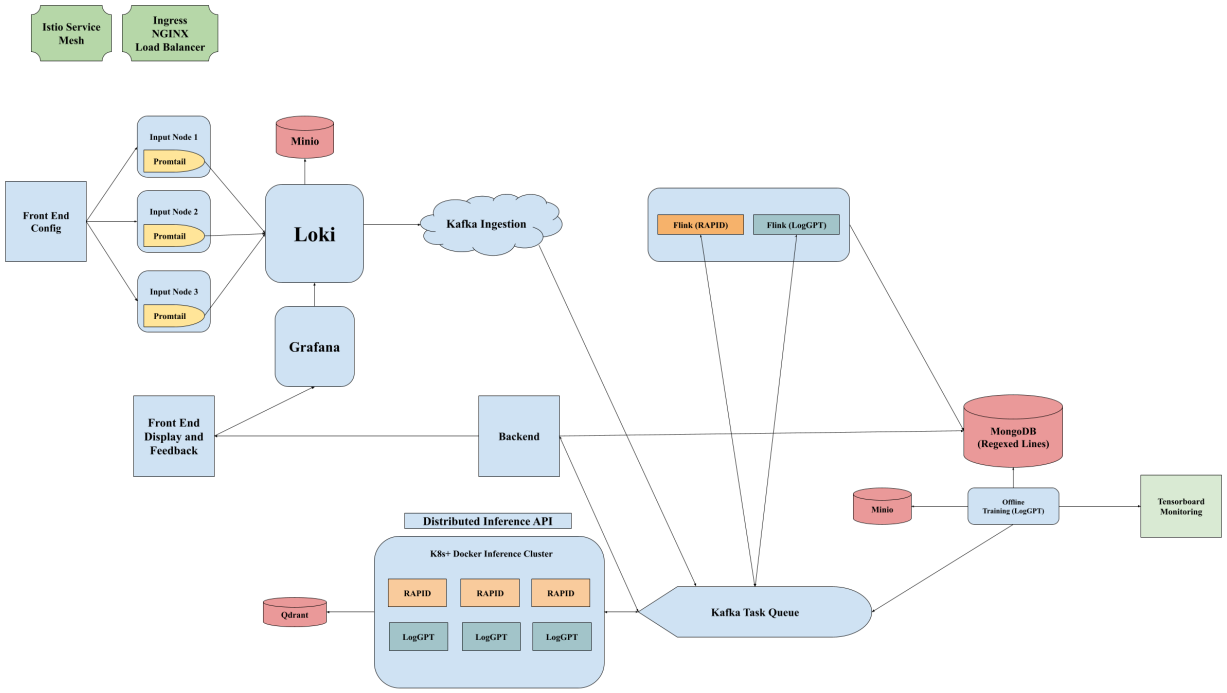


Figure 4: *Diagram of the complete architecture that composes LogSense*

iterative process enables the model to assimilate corrective feedback, enhancing its performance over time by augmenting its knowledge base with accurate representations of normal logs. Concurrently, this approach facilitates continuous monitoring and evaluation of the model's efficacy through the tracking of normalized log counts within the anomaly database.

In the context of LogGPT, anomalies are presented as temporal windows rather than individual log entries, which inherently provide less granular information. Both setups necessitate contextualization to aid with Root Cause Analysis (RCA) and debugging, prompting users to utilize Grafana for further elaboration. When users opt for "More Context" within the anomaly window, LogSense queries Grafana using the tagging schema set up in Loki, directing users to Grafana Explorer with the query results, and furnishing them with more information to aid their decision-making. This feature is equally applicable to RAPID's conclusions, offering users comprehensive insights to facilitate informed assessments of LogSense's inference judgments. In both cases, the beginning of the logs' journey is poetically tied to its end as users can see both sides when deciding the validity of LogSense's inference.

Offline Architecture

MongoDB serves as the repository for RAPID's and LogGPT's distinct anomaly predictions, segregated to facilitate efficient retrieval and management. Additionally, MongoDB accommodates the storage of logs generated during train-mode operations, earmarked for subsequent LogGPT offline training and fine-tuning conducted at the end of the day. Notably, the correction of false positives triggers a mechanism that promptly updates affected logs to reflect their normal status within the database. This corrective action serves as a potent instrument for model refinement, enabling iterative improvement through fine-tuning processes. By iteratively fine-tuning LogGPT based on identified false positives, the model is compelled to internalize the structural nuances of normal log sequences, thus augmenting its comprehension of typical log patterns and iteratively enhancing its predictive capabilities.

Facilitating the monitoring and assessment of model evolution, Tensorboard provides metrics to evaluate the models' performance – any abnormal patterns within the evaluation scores will be promptly dealt with. To ensure streamlined accessibility and updateability, the model under training is versioned and securely stored within a Minio database. Daily, the model undergoes iterative training cycles using fresh data extracted from MongoDB, perpetually updating its version to reflect the latest insights gleaned from the accumulated data corpus.

The efficacy of Kafka shines bright once again as it orchestrates the update of the online model

weights with seamless integration with the versioned Minio database. Following each training iteration, a notification is propagated through a separate Kafka topic designated to model updates, thus orchestrating the subsequent update of the online model. This seamless synchronization mechanism, powered by Kafka's robust messaging capabilities, ensures the timely dissemination and integration of model updates across the LogSense architecture, thereby continuing the cycle of model refinement and improvement.

Applications in Distributed Systems
Distributed systems represent a collection of autonomous computing nodes collaborating towards shared objectives, often by allocating tasks across the available nodes. This architectural paradigm fosters scalability, fault tolerance, and parallel processing capabilities by spreading computational workloads across a networked infrastructure. The resultant log data is significantly complex compared to single-system logs due to factors such as concurrent processes, heterogeneous environments, and intricate error-handling mechanisms, necessitating an equally complex analysis architecture. LogSense emerges as a formidable solution against this backdrop, tailored to navigate the intricacies of intertwined logs inherent in distributed systems.

At the heart of LogSense lies its robust aggregation capabilities, enabling centralized monitoring of logs produced by diverse nodes within the distributed system. By consolidating log data into a unified platform, LogSense removes the need for users to painstakingly assemble disparate log fragments, offering instead a cohesive visual representation of system-wide log anomalies. Armed with intuitive configuration tools, users can swiftly configure LogSense to survey their service and nodes, instantly gaining insights into anomalous activities spanning the entire network upon accessing the platform. In essence, LogSense offers a comprehensive log monitoring solution, affording users a singular vantage point for overseeing log activity across distributed infrastructures.

A future distinguishing hallmark of LogSense lies in its sophisticated integration of trace analysis methodologies, indispensable for detecting anomalies within distributed systems. Trace IDs allow us to map log lines to specific processes within the distributed architecture, allowing us to view logs contextually. For example, if a map-reduce job is run across multiple nodes, Trace IDs are what allow us to piece together the multi-node process. Using this mapping, we can take log analysis to a whole new level, getting useful insights into high level processes rather than picking up granularities from individual logs. Trace analysis would inform users of any anomalous processes rather than anomalous log lines, providing deeper, more productive information about the systems. LogSense is en route to implementing a Distributed Tracer that not only notifies users about anomalous traces, but actually lets them visualize them. Through the Trace Visualizer, LogSense depicts the cross-node traces in a digestible way for users to respond effectively.

Limitations
As promised above, let's discuss the limitations of Train Mode. Foremost among these limitations is LogSense's presumption that every log generated during train mode adheres to normal behavior. This assumption is inherently flawed, as anomalies may inadvertently slip through the filtering process. While this introduces cracks in the foundational knowledge of the models, it is preferable to operate with a flawed foundation than none at all. After model initialization, user feedback on false positives becomes instrumental in refining model performance, thereby justifying this trade-off. Notably, LogSense deliberately permits a higher percentage of false positives to empower users with a direct mechanism for model refinement. By prioritizing the correction of false positives over false negatives, users are empowered to rectify model inaccuracies, ensuring a safer and more effective learning process. Unlike false positives, which can be rectified, false negatives remain undetected, underscoring the rationale behind prioritizing the mitigation of false negatives within the very structure of LogSense.

Another pertinent limitation of LogSense pertains to its reliance on a predefined vocabulary size within its LogGPT implementation. Fundamentally, LogGPT determines log sequence anomalies based on the model's inability to predict the subsequent log line within its top-k predictions. Consequently, the model necessitates a predetermined range of keys for top-k selection. Presently, LogSense rigidly enforces a maximum vocabulary size hardcoded at 500; should a 501st unique log template emerge from the parsing process, unexpected behavior may emerge. With this in mind, we're exploring multiple avenues of action such as extending the hardcoded vocab size past any feasible system's possibility. While retraining the model with an expanded key range appears as an intuitive approach, it disregards the necessities of real-time stream processing. The dynamic nature of log generation renders real-time model updates unfeasible within LogSense's architecture, where thousands of logs are generated per second. Consequently, a nuanced solution must be devised

to surmount the constraints imposed by hardcoded vocabulary size limitations.

IV. Further Study

While LogSense represents a significant stride in comprehending and managing system intricacies, there are notable areas that demand further refinement. One such area revolves around the generalization of models. Given the diverse nature of the services supported, LogSense requires distinct modeling approaches to cater to each service. This diversity may potentially undermine overall effectiveness, impeding its ability to generalize efficiently, thereby raising concerns regarding false negatives. One possible solution for this problem is to display every unique normal log per service in LogSense. This would empower users with the capability to flag anomalies based on an aggregate view of unique normal logs, with an obvious trade-off in increased human involvement. Additionally, augmenting LogSense's capabilities in trace visualization holds promise for future development, as it would enable users to gain deeper insights by visualizing intricate interactions and logs across system nodes. These areas underscore the ongoing evolution of LogSense and signify opportunities for continued enhancement and innovation.

V. Conclusion

In conclusion, LogSense presents a cutting-edge approach to stream processing log anomaly detection, harnessing the capabilities of state-of-the-art machine learning models and continuous user feedback to deliver rapid (ha!) and accurate insights. By integrating advanced algorithms with user-driven refinement, LogSense offers a robust architecture capable of swiftly identifying and addressing anomalies within complex log data. Through its tandem of cutting-edge models and efficient allocation of resources, LogSense stands strong in the face of any tsunami.

References

[1] Max Landauer et al. "Deep learning for anomaly detection in log data: A survey". In: *Machine Learning with Applications* 12 (June 2023), p. 100470. ISSN: 2666-8270. DOI: 10.1016/j.mlwa.2023.100470. URL: http://dx.doi.org/10.1016/j.mlwa.2023.100470.

[2] Yukyung Lee, Jina Kim, and Pilsung Kang. *LAnoBERT: System Log Anomaly Detection based on BERT Masked Language Model*. 2023. arXiv: 2111.09564 [cs.LG].

[3] Xiao Han, Shuhan Yuan, and Mohamed Trabelsi. *LogGPT: Log Anomaly Detection via GPT*. 2023. arXiv: 2309.14482 [cs.LG].

[4] Gunho No et al. *RAPID: Training-free Retrieval-based Log Anomaly Detection with PLM considering Token-level information*. 2023. arXiv: 2311.05160 [cs.LG].