Building a scalable real-time ML inference platform for AIOps

Praveen Manoharan
Applied AI & Discovery
Comcast India Engineering Center (I) LLP
Chennai, India
Praveen_Manoharan@comcast.com

Nilesh Nayan
Applied AI & Discovery
Comcast India Engineering Center (I) LLP
Chennai, India
Nilesh_Nayan@comcast.com

Aaditya Sharma
Applied AI & Discovery
Comcast India Engineering Center (I) LLP
Chennai, India
Aaditya_Sharma@comcast.com

Aravindakumar Venugopalan
Applied AI & Discovery
Comcast India Engineering Center (I) LLP
Chennai, India
Aravindakumar_Venugopalan@cable.comcast.com

***Abstract*—In this paper, we present the method of building a scalable, real-time inference platform for large-scale time-series anomaly detection and root-cause analysis solutions, built as a part of AI For Operations (AIOps) tool. AIOps is a tool built to ease the manual and time-consuming activities of DevOps engineers involved in monitoring and troubleshooting production systems. Such a system has to be operated in real-time to detect anomalies in a plethora of time-series metrics and logs from the productions systems in order to provide timely alerts and possible root causes for quick remediation and thus requires a low-latency operation. This system must be scalable for the vast amounts of data involved for ETL and ML inference jobs that the solution needs. In this work, we show how we engineered and scaled up the AI research POC to a solution that supports a massive search engine system, where we achieved reduction in latency by 30x. We also evaluate different tools for inference such as Apache Airflow, Serverless REST API and Spark engine and demonstrate our improvements achieved and our estimations of these different commonly used platforms for ML inference, in terms of feasibility and cost for an AIOps solution.**

Keywords—AIOps, time-series, anomaly-detection, root-cause analysis, model inference

I. Introduction

"AIOps" [1] is a portmanteau of Artificial Intelligence (AI) which comprises Machine Learning (ML) and Deep Learning (DL) and Information Technology (IT) Operations. As per Gartner's 2022 Market Guide [2], AIOps is going to be future of IT Operations. AIOps platforms analyze application, system metrics, telemetry events and application logs to identify patterns and trends to understand the behavior of the system. AIOps platform is designed with the following characteristics [3][4]: (1) ability to ingest and analyze cross-domain data, (2) perform dependency graph analysis across different stacks and services, (3) correlate events to understand the patterns to detect incidents, and (4) predict possible hypothesis towards the root cause analysis and recommend solutions. AIOps refers to the timely identification of IT operation issues using AI/ML. While MLOps refers to the deployment and operations of ML models and DevOps refers to software development and operations, AIOps is different from them as its applicable to MLOps or DevOps to make them more robust and cost effective (Fig. 1).

Availability of mission critical applications are crucial, and any outages can directly influence business revenue and customer satisfaction. A reliability engineer may need to get into a crisis call in the middle of the night to fix a critical issue in the field. AIOps is designed to act as a third hand to the engineer to help in resolving the issue faster and thereby, reducing mean time to failure (MTTF) and to resolve (MTTR) the issue [5]. AIOps operates with continuous time series data. Since the volume of data processed is huge and the system must operate in real time, scalability of the system across different stages of AIOPs is critical. This paper discusses on resolving challenges related to scalability and the best practices for achieving a highly scalable real-time AIOps platform.

The rest of the paper is organized as follows. In Section II, we discuss related work in literature on AIOps and scalability. In Section III, we explain the scope and requirements of the AIOps product and the various problems and challenges that we encountered in the practical real-life implementation. In Section IV, we propose solutions and elucidate how we tackled each of the problems present in different components of our AIOps platform. Finally, in Section V, we provide our conclusion and also discuss other potential options for further optimizations and our future evaluation considerations with respect to building a scalable real-time AIOps product.

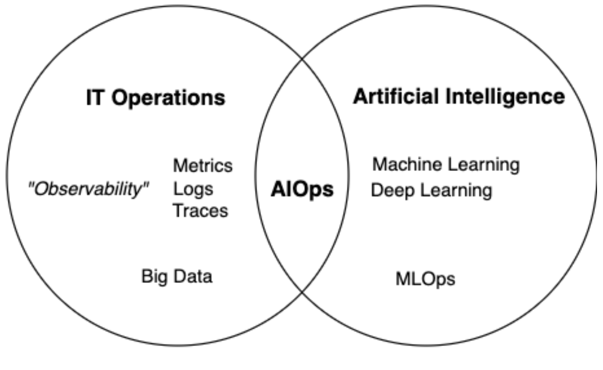

Fig. 1. AI For IT Operations (AIOps)

II. Literature Review

The different features and requirements for an AIOps solution have been discussed in multiple forums and in literature such as [3] and [4]. These works discuss the impact of AIOps product in an IT organization and the value that it brings to DevOps teams, the quality of service, reduction in operations cost, and customer satisfaction achieved. Only very few works discuss the challenges involved in bringing valuable research knowledge into a scalable viable product that can add real business value for an organization.

As we can see from [6] where a systematic mapping study has been done on the various literary works available related to AIOps, we see how the focus has been on building different valuable solutions in the IT Operations space such as Resource Provisioning, Failure Detection, Failure Prediction, Root Cause Analysis, Prevention and Remediation that are used in the AIOps platform development. Such solutions have been offered by multiple AIOps platforms that may offer a subset of these solutions or all of them. One such AIOps platform development can be seen in [7] where an AIOps product has been built to address solutions ranging from Anomaly Detection and Smart Alerting, Automatic Root Cause Analysis (RCA) from Logs, RCA through multiple Metric Correlations using dependency graphs of services, Auto-remediation and Failure Prevention solution, Intelligent Orchestration for Cost Efficiency and Release Management.

As noted in [2], the main barrier for AIOps platforms is the difficulty in measuring the value created from the usage of such products. To fully measure the impact created by such a tool, the tool has to be designed to operate at scale at nominal costs and offer quick results for the operators to find meaning in the adoption. But it is to be noted that enterprises are rapidly adopting AIOps solutions; and the costs and complexities involved in managing, storing and handling data at scale are still major challenges for AIOps platform developers. In [8], the authors discuss some of the real-world challenges that they faced while implementing their AIOps solution in production and discuss the research innovations performed to create a few successful AIOps solutions that they built in their products. The various challenges related to data quality, efficiency of AI and ML, limitation in use case availability, etc. are also reviewed in detail in [9].

This paper discusses the practical challenges involved in building an AIOps product, specifically in terms of infrastructure scalability, that must be handled while building such innovative ML products that most of the works in literature do not discuss explicitly. This way, we complement the existing research works by providing a more detailed paper on the various techniques and solutions we have engineered to solve some of the complexities involved in building a production-grade AIOps platform from a research Proof-of-Concept (POC). We have evaluated different tools and engineering solutions while trying to scale up the product to handle large-scale loads involved in an application that caters to the massive search and browse engine workloads of a real-world commercial product and have presented the results that we obtained.

III. Scalability Challenges

The AIOps solution requires a highly scalable, real-time operations in multiple verticals such as: (1) data ingestion or extract-transform-load (ETL) pipeline, (2) Machine Learning (ML) model inference, (3) alerting or notification scheme, and (4) root cause analysis and reporting solutions. There are multiple challenges to be addressed in implementing these solutions for the product to be effective.

Operation metrics are time-series in nature as the telemetry data are collected from across multiple services at different instances in time. The nature of the time-series data vary vastly from metric to metric, and they are also dynamic in nature and can change over time. Most of the system metrics are collected from multiple sources, and such metrics collected and stored in an operations data collection and monitoring solution like Prometheus [10], have different labels which denote the metadata for identifying the source of the time-series. In real production systems such as the one we handle, we observe that one metric can have even 1000 time-series data or more at a given instance, which are collected from different sources. If the sources are dynamically destroyed and created, such as in the case of Kubernetes Pods when a new release deployment takes place, the existing metric time-series data from the destroyed instances cease to appear and new ones appear. Thus, over a period, the number of unique time-series data within a single metric keeps growing. We also observe that some metrics disappear and re-appear after some time, especially in case a service is restarted after a maintenance period. We require operations monitoring on 100s or more of such metrics, which is a perfect example of the extent of complexities involved in the data management.

To counter these challenges, data ingestion or ETL pipeline must be scalable to support such large volumes of data. It should be a low-latency streaming solution, but operations metrics stored in databases like Prometheus can only be fetched by querying data over its REST API as there is no push mechanism to send the telemetry data collected to some external streaming service. These offer limitations in the choice of ETL service. Also, performing a very large number of metric queries to the source metric storage can overwhelm the system, leading to increased latency, more frequent HTTP request failures, and slow response times in dashboards used by operators manually monitoring other system metrics. Thus, the ETL operation must prevent such client-side resource (CPU or memory) overloading too.

Also, the nature of the time-series in terms of magnitude, periodicity or seasonality, trend, stochasticity, etc., vary from metric to metric and between the different time-series within a metric. Hence, we don't have a "*one-fit-for-all*" ML model solution for all time-series data. Since the operations data is unlabeled by default, unsupervised time-series anomaly detection algorithms are used to fit on the data over a suitable time period, to capture the trends and seasonality present in the data. By allowing an annotation option through a user-interface (UI), we also make use of supervised classification algorithms to detect anomalies using the labelled anomalous points. A plethora of algorithms are available to fit on multiple metrics and even if a model fits on a particular pattern of data, when deployed for inference on multiple time-series metrics data, the threshold chosen for detecting anomalies using the anomaly score computed by the model needs some careful setting for each of the data. Therefore, even if our solution enables us to choose certain trained models to be deployed for multiple time-series inference, we still may have to fine-tune the model or adjust the thresholds for each of them by having a human-in-the-loop for more accurate inference. A system with automated unsupervised model training and inference can still benefit from a feedback mechanism for fine-tuning models or adjusting thresholds that improves the overall performance in terms of prediction accuracy. Thus, it is difficult to go for a distributed environment like Apache Spark [11] for this operation as there is no single model used for multiple data inference (single big and complex task), rather multiple fine-tuned specific models, performing inference on individual or a small subset of related data. The conclusion — it is not straightforward to use that distributed framework for inference.

The main goal of an AIOps tool is to provide operators with timely alerts that notify the team of an anomaly in some monitored metric that helps them attend to and rectify quickly before a major outage occurs. Our infrastructure has to be robust in handling the rules for checking and sending alerts and must be scalable to support the operations for all inference jobs that would be making predictions every minute. It must have a very low latency and send alerts instantly.

With the addition of features that are targeted to reduce the MTTR of the operators such as RCA solutions that correlate detected metric anomalies with error log trends or other metric anomalies, we have a need to perform this at scale and send the RCA report instantly for a quick fix. A naive, simple solution of sequentially checking the anomaly trend correlations with every metric (ignoring the complexities involved in fetching a block of data from feature store and in computing correlation coefficient) will have a linear time complexity of O(n), where 'n' is the number of time-series metrics. Since 'n' could be as large as one million and this computation has to happen for an anomaly detected on a selected metric at a selected minute, getting a quick RCA report for an operator to act upon, is highly difficult in this setup. It becomes more impractical as we can have multiple alerts and this can occur every minute. Thus, we require a better optimal approach for handling such computations.

The following Table I displays the initial operational statistics of the actual use cases 1 and 2 which are explained in Section IV. The numbers represent the scale at which the AIOps platform had to operate for real-time operations at reduced latencies and thus presented us with the challenges during the initial POC stage that we just discussed.

TABLE I. INITIAL OPERATIONAL STATISTICS

Use Case	**Data**		**Average Airflow Tasks (Per Minute)**		
	Metric	***Time-series***	***ETL***	***Inference***	***Alerting***
1	1	961	3 x 1	3 x 1	4 x 1 + 1
2	38400	73633[a.]	3 x 38400	3 x 8400	4 x 8400 + 1

[a.] Without including count of time-series in the metrics removed from the 38400 metrics

IV. Solution Methodology and Results

As the tool must operate in real-time, there is a requirement to periodically fetch data from source and perform inference on them. Thus, we need a scheduler. Initially, we used a cron-based scheduled script to fetch the operations data, transform and store the results in comma-separated values (CSV) files in disk in a monolithic architecture where a UI dashboard, the training, inference and alerting jobs (cron), and a rules engine, were all hosted within the same instance and coupled tightly together. As this solution limits the operations to just one instance, to deploy a solution in cloud that is independently horizontally scalable and loosely coupled, we went for a microservices architecture with separate services for different components. The various services perform their tasks independently and access shared resources such as the feature store and metadata store. To streamline the operations, we used a workflow management tool Apache Airflow [12] which was used as a scheduler and an orchestrator of various tasks such as data ingestion, log data mining and log metrics creation, model inference, alert rules evaluations and notifications, by dynamically defining tasks for different use cases and metrics as Directed Acyclic Graphs (DAG). This migration from a monolithic to a microservice architecture hosted in cloud, helped us have the AIOps solution built for simpler use cases with few metrics (say, <100). In particular, we used it to monitor the load of prediction requests made to a computer vision ML model deployed in a home security product (use case 1). This helped DevOps team detect anomalies in the observed seasonal load pattern and find out root cause error logs for immediate remediation. However, we faced the scalability challenges (Section III) when we onboarded use case 2, to monitor the operations of different microservices available in a large-scale relevance engine application that powers the search and browse features in products like voice remote, personalized content discovery services, etc. The following subsections discuss how we addressed the different challenges and the performance gains achieved in building the AI powered operations monitoring and automated RCA tool having an impactful usage in IT observability domain.

A. Scalability in Data Ingestion

The scheduling interval used in Airflow was one (1) minute and so the data ingestion task was expected to get completed in <1 minute. But, when we ingested data from a large-scale system, we ended up with more than 38400 metrics that resulted in 3 x 38400 data ingestion tasks. As we had to train and deploy models for ~1050 time-series metrics per data-center (we had 8 such data-centers), we had a very large number of tasks for model inference and alerting as shown in Table I. Even though Airflow is theoretically capable of handling such loads, practically with limited number of worker nodes, schedulers, and worker resources, especially when operated in a managed cloud environment, Airflow gets overwhelmed and freezes. Such a large number of metric requests cannot be handled by the client Prometheus or Thanos instance too.

We initially dropped about 74% of the metrics from the total based on the understanding that they carry redundant information captured by other metrics or were not significant for the operators. This reduced count of metric is still not feasible for data ingestion and so we worked on optimizing the number of queries we make to the HTTP endpoint by combining metric queries using clauses such as group by, and thereby increasing the number of time-series per metric query and thus ending up with the same total number of metric data that we want to ingest and monitor. This way, we reduced the number of metrics significantly by up to a further 97% but we found that certain queries are computationally expensive at the client end and split some metrics to finally end up with 425 metrics — a 95% reduction in number of metric queries that we have to make. This way, we have provided a user-configurable settings for ingesting metric data that can perform the ETL task without overloading both the client side as well as the Airflow environment. We also reduced the count of tasks to 426 per minute by combining them.

As we moved from CSV files in cloud object storage to a feature store more suited for metric time-series as explained in the next section, where the write operation is optimized by having a separate service that is independently scalable and performs parallel write through multiprocessing over multiple cores, we achieved a reduction in ETL latency for a single metric for a single DAG Run from ~20 minutes to ~10 seconds which is almost ~99% improvement. The CSV-based feature store required additional overheads of reading and writing files and offered limited concurrency because of an aggregation that we also do over all the fetched metric time-series.

However, to have an automated way of onboarding a client application without the need to specify optimal configurations, we are setting up an instance of Thanos at our end to replicate the data at the client side by having a sidecar container at the client end that pushes the metrics data to this instance. This will be configured to scale independently depending on the load of ETL activities, thereby not affecting any client operations. This ensures a smooth scalable, low-latency data ingestion pipeline for transforming and storing as time-series data as needed for visualization and ML models.

B. Scalability in Feature Store

The choice of a time-series database (TSDB) as a feature store is crucial from the perspective of scalability, number of read-write threads hitting the feature store, and the required latency in read-write operations. Also, the database used for feature store is expected to scale up horizontally according to the number of read-write requests made and vertically according to the required granularity and period of data per request.

- *TSDB vs other DBs*: Compared to other databases, TSDBs provides faster and economic read-write operations for time-series data with proper data schema specific to time-series data. They also help in performing efficient aggregation and imputation queries over time-series data. Some TSDB solutions are optimized for tier-based data access where read operations are optimized for both recent and historical data while write operations are optimized for recent data, making both ingestion and reading faster, along with a common query engine for all the storage tiers. TSDB are timestamp indexed, making them efficient for large-scale sequential data. They also avoid data duplication and maintains data order despite ingestion request order. This allows clients to run parallel workers for the data ingestion jobs without worrying about the order in which the jobs are executed.
- *Serverless vs Server-based*: Serverless TSDB allows us to have a highly available database service without having to maintain a server — this is becoming a popular choice in cloud-native deployments. Serverless DB services can scale as per the fluctuations in demand and can process millions of queries per day.

An application programming interface (API) built as a layer between AIOps platform as a client and TSDB as a server is used to transform and load data to the TSDB such that it can divide the data to run multiple parallel processes based on available resources on the host. The API layer was built in such a way that it could scale horizontally depending upon the number of requests made for different activities like (1) pushing metric data from data ingestion tasks, (2) pulling data for inference jobs or showing in dashboard user-interface (UI) for visualization, etc., (3) pushing predictions data from inference jobs, and (4) pulling predictions data for checking alerts or for RCA. A scalable container orchestration service is used to deploy, manage, and scale the containerized application on a cluster with a serverless configuration that helps us achieve an interface that is independently scalable with respect to the load of read and write operations, which in turn depends on the number of use-cases and metrics configured for real-time anomaly inference. The API layer processes the data for writing and pushes the records in batches through multiprocessing over available cores by dividing the data into equal length chunks. TSDB v1 implementation does not maintain common attributes for dimensional values common for a particular time-series, causing highly redundant write operations, while TSDB v2 implementation uses common attributes saving writing costs and improving latency in write operations. The improvements in read and write latencies with the changes made in the feature store backend have been presented in Table II.

TABLE II. OPERATION LATENCY IN DIFFERENT FEATURE STORE FOR A SAMPLE 25KB DATA PROCESSED PER REQUEST

Operation	**Database**		
	Simple cloud storage	***TSDB v1***	***TSDB v2***
read	≅ 1 s	≅ 1 s	≤ 10 ms
write	≅ 15 min	≅ 36 s	3 s

C. Scalability in Model Inference

As we have multiple models used for inference for multiple time-series metrics, we need a low-latency, real-time, scalable solution. When we used Airflow for this task, we performed all the heavy activities like loading the data and model in memory and performing model inference within Airflow tasks. Since Airflow is designed primarily to serve as an orchestrator and not intended for heavy tasks, the inference performance was poor. The following sections discusses the performance improvements obtained in moving from Airflow to further optimized, scalable solutions that we designed and deployed.

a) Initial Airflow-based Inference (I1): From Fig. 2, we observe a very large memory consumption of ~1823 MB for a single cold-start (typically ~4 weeks) prediction as the task loads model and data in memory, performs anomaly predictions and saves results back to feature store. To accommodate such scenarios of peak memory when a new data is scheduled for detecting anomalies, we had to restrict the number of tasks per worker node in Airflow as all tasks within a worker node share the resources. If resources aren't enough, tasks get killed and the DAGRun fails. Therefore, I1 limited the task concurrency which in turn increased the execution times of Airflow DAGs. We also observed a significant latency of 4s for a single real-time inference task. To avoid overwhelming Airflow, we reduced the number of tasks by having concurrency at metric level instead of at individual time-series level. So, we had to perform inference for each scheduled time-series metric present in a metric linearly, as worker nodes have limited cores which are already shared amongst tasks. This leads to large delays as the individual inference latency is scaled linearly by the number of time-series data configured with anomaly detection models trained, at different granularities and aggregations.

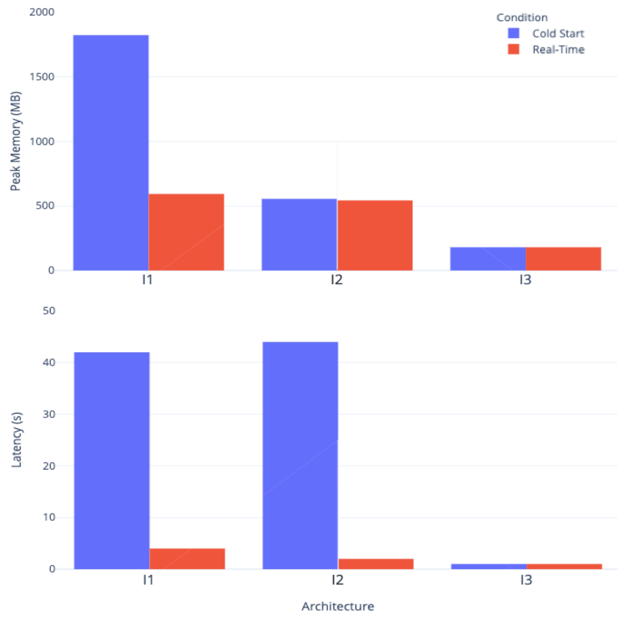

Fig. 2. Memory (top) and Latency (bottom) in Airflow Task in I1, I2, I3

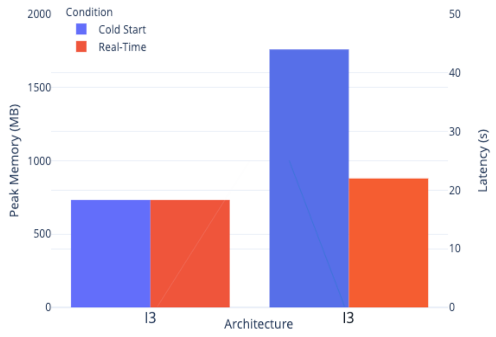

Fig. 3. Memory (left) and Latency (right) in Serverless Instance in I3

b) Inference through scalable, serverless REST API compute instances (I2): In this modification to I1, we deployed 2 serverless compute instances that are independently horizontally scalable based on the number of function invocations (load). One gets triggered when new models are saved and deployed in an object store and downloads them into an elastic storage which provides a network file system (NFS) interface. The second loads models from this storage into memory and performs the inference job when invoked via HTTP REST API, reads the metadata of the metrics and models from the API request and sends the results back to the client to save into feature store. This function also caches the models to quickly serve in subsequent invocations of the same model and only has to load from the storage when a new version of a model is deployed. From Fig. 2, we can see that this architecture has reduced the peak memory consumption in Airflow tasks during cold start significantly and the model inference job is offloaded out of Airflow. This helps us have an Airflow environment with more concurrent operations. The reduced load also makes the system more reliable and helped us in saving Airflow costs by upto 55% as we can reduce the resources of worker nodes. However, the latency of the overall DAGRun was still significant even though the individual task latency dropped by 50% as the inference job for each time-series still had to be executed sequentially in the Airflow tasks as each inference job is a blocking operation and the task waits for API response that needs to be saved to feature store before going to the next job.

c) Inference through scalable, serverless message queue triggered compute instances (I3): The motivation on further improvement in architecture from I2 was to reduce the latency of the overall operations by having more concurrency in the individual time-series inference jobs. For this, we replaced the REST API trigger with a message queue based event-driven architecture for the 2nd serverless compute to have non-blocking invocations decoupling Airflow from model inference step. Loading data from feature store, saving predictions back and sending alert checking messages to its queue, are handled within the compute instances making the Airflow tasks fast and extremely light-weight as can be seen from Fig. 2 that shows an execution time of ~1s and peak memory consumption of ~180MB. To avoid race condition of making multiple inference job invocations before the previous job is completed for a given time-series, the status of the running state and the last predicted timestamps are stored and handled separately through a relational database, making the entire inference job scalable, reliable and fast. The operational costs involved in the serverless instances offset the cost savings obtained from Airflow operations. The latency and peak memory consumption in the serverless compute functions do not exceed any limits that can cause any deterioration in performance as seen from Fig. 3. Due to the decoupled architecture, the inference jobs have more concurrency and were observed to complete within 1 minute for the entire metric, which helped in achieving 30x reduction in latency from I1 architecture.

D. Scalability in Alert Notification Scheme

The major objective of AIOps is to reduce the crucial MTTR metric for operation teams in their triaging activities [5] since its reduction directly correlates to customer experience and revenue in terms of customer retention or acquisition. The most intuitive way to reduce MTTR is by making our AIOps platform operations (near) real-time by sending timely alerts and reports to operators if abnormality is observed. This reduces the delay between the actual issue occurrence and the acknowledgement by the concerned team for triage. In this subsection, we discuss the approaches taken to evolve our platform for scalability with respect to this alerting need. We can roughly estimate MTTR as

$$o + a + ack + t \approx MTTR \tag{1}$$

o – AIOps platform operations delay (ETL, Inference),

a – Alerting lag,

ack – Operator acknowledgement delay,

t – Triage delay

The first part of (1) i.e., o has been discussed in earlier subsections. In this subsection, we focus on reducing a and ack. While the initial POC had an acceptable delay of ~1-5 mins w.r.t. alert notifications, we faced the following critical challenges while onboarding the large-scale application:

1) Huge delays in alerting: Alerting delay a in (1) has to be as low as possible (near real-time) so that an issue is acknowledged quickly to start the triaging activity. Using CSV files in cloud store, we faced delays of over 30 mins for getting alerts, which dissolves the purpose of our tool for reducing MTTR. Majority of this delay was introduced by file input/output (I/O) and network delays in API calls. Also added to this, the nature of data (time-series) stored in CSV was not ideal for the scale of operations we target since we require additional post-processing for using that data. All these delays cascaded when we used Airflow scheduler for alerting too, where the the limited worker resources get blocked out for other critical functions and so affected the total latency.

2) Race condition: We maintained states essential for alerting feature (like last alerted timestamp, pause status and duration if alert is paused) in CSV files stored in cloud object storage. When multiple Airflow tasks access the state file simultaneously (race condition), it causes inconsistencies in file leading to EOF (End of File) errors while parsing a read CSV file, being updated at the same time.

For these observed scalability challenges, the following approaches were adopted to tackle them (refer Fig. 4, Table III):

a) Using RDBMS for states: Since using CSV files for maintaining alerting states was not optimal, we resolved it by shifting our state storage to Postgres RDBMS (Relational Database Management System) which was able to handle the scale of CRUD (Create Read Update Delete) operations with all desired properties we wanted like consistent state, atomic operations etc. Thus, the flaw of race condition was eliminated, and alerting feature operated smoothly in the AIOps platform.

b) Migration to TSDB: This was a major step taken to migrate from conventional CSV file-based storage to a highly scalable TSDB as discussed in subsection B. Due to this feature store, we reduced the delays in data fetching by eliminating excessive file I/O operations and data post-processing steps.

c) Message queue based event-driven architecture: The alerting delays were still unacceptable (>10 mins) after *b)* in which the alerting was tightly coupled with Airflow DAGs and operational delay (say, ETL lag, resource unavailability, etc) cascaded to alerting lag. In the revamped architecture (Fig. 5), we send message with relevant metadata for alerting to a queue from the inference function whenever anomalies are detected which eliminated the redundant alerting DAG. This enqueue triggers a low memory serverless compute instance that can scale horizontally (high concurrency) based on load. Due to the relevant metadata provided, no additional API calls to TSDB is made. They directly perform the alerting check and send alert metadata to another queue if condition for alerting is passed. Using this separate queue, we send alerts in a desired sequential manner using high memory compute instances with single concurrency (no scaling required to send notifications in correct order) but high batch size to process the queue at low latencies. This reduced the latency to < 1 minute which is near real-time.

Apart from the reduction in a, we reduced ack by providing a customized link to the UI in the alert message which was dynamically generated while processing an alert. The message has relevant details of the issue allowing the operator to land at the correct metric at the alert timestamp and so saves their time.

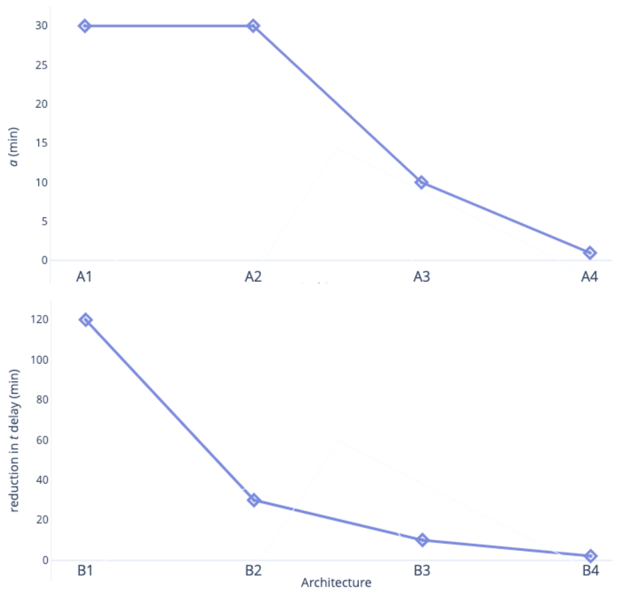

Fig. 4. a (top) and reduction in t delay (bottom) (Refer Table III)

TABLE III. X-AXIS TICK LABEL REFERENCE FOR FIG. 4

Label	***Definition***
A1	CSV cloud storage with Airflow DAG
A2	CSV cloud storage with Airflow DAG and Postgres for state
A3	TSDB with Airflow DAG using Postgres for state
A4	TSDB with Message queue based event-driven architecture using Postgres for state
B1	CSV cloud storage with 1xn approach
B2	TSDB with dependency graph without filtering
B3	TSDB with dependency graph with filtering
B4	TSDB with dependency graph with filtering and vectorized operations

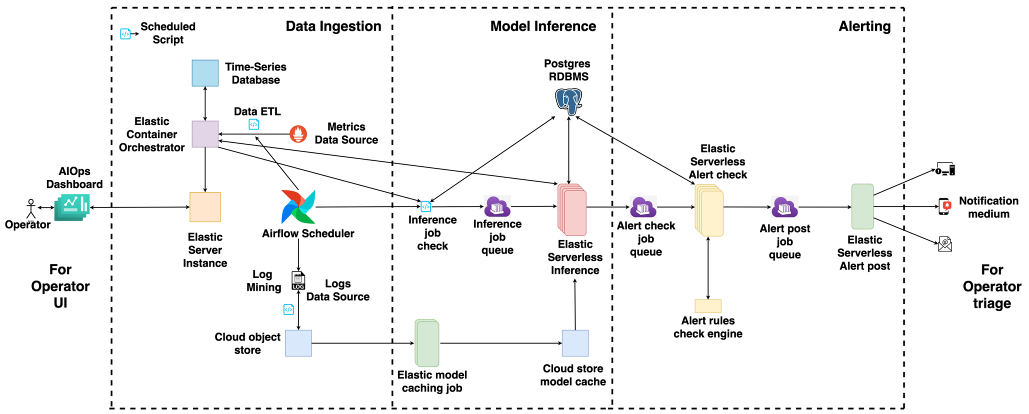

Fig. 5. Scalable Real-Time AIOps Platform Architecture

E. *Scalability in Root Cause Analysis*

Operators trace the root cause of an identified issue in high-pressure environments during time-critical on-call activities scheduled outside regular work-hours to resolve and notify stakeholders. Automatic RCA feature is thus an essential feature of AIOps for their triaging activities as it can greatly reduce *t* in (1). RCA involves large-scale time-series correlation calculations which must be performed optimally. Initially, we suboptimally correlated the concerned metric with all the time-series available with O(n) complexity (~4 minutes for 50 time-series). To improve efficiency, we filtered and performed only required correlations using dependency graph (a hierarchical topological graph representing the entire system service dependencies) to avoid redundant calculations of unrelated time-series. RCA took >2 hours for the large-scale 2nd use case as there were high latencies in handling CSV files (I/O) and in data processing while calculating these time-series correlations.

Since RCA has to work at scale (>100K time-series), migration to TSDB along with dependency graph (hierarchical representation of service dependencies) helped reducing complexity to O(h * L) where ‘L’ is the average number of time-series in one level of graph and ‘h’ is the the total height of graph where h~log(n) on average case. The latency reduced to ~20-30 minutes. We further optimized RCA by writing custom filtering queries in TSDB to only fetch the anomalous time-series to eliminate redundant calculations. This reduced latency to <10 minutes. Finally, using vectorized approach for correlation calculation resulted in latency dropping to <2 minutes i.e., near real-time (refer Fig. 4, Table III). Here, a customized single query was used to fetch an aggregated filtered time-series matrix from TSDB. The complexity remains same as O(h * L) but this implementation reduces the ‘L’ part due to multiple small units calculated simultaneously since it uses internal hardware parallelism to achieve quick results. So, the program behaves like O(h) but this is subject to the scale of data. For further scaling up, upgradation of hardware in our instances (high core CPU or GPU) is required to take advantage of vectorization.

RCA includes dependency graph traversal where we only traverse next level nodes (services) if that path has high anomaly correlation with the concerned metric. The operator can set the Pearson correlation threshold and control the traversal beam width (number of paths to follow) which controls the number of RCA hypothesis. This is practically important as the highest correlated time-series might not be the root cause but the metric present at say, 3rd rank might be. Apart from these *t* reduction, we also provide graph traversal visualizations to the operator to trace the root cause path more effectively reducing the overall *t*.

V. Conclusion and Future Work

According to the current trend, new generation applications are following microservices architecture having complex infrastructure and managing that has been a challenge for the operators. AIOps brings the power of AI to operations data to assist operators in taking better decisions and to achieve lower MTTR. From the perspective of real-time, streaming and big data-based solutions like Spark can be effective but considering the need for local or single AI models for specific metrics, such Spark based processing may not be viable and cost effective. Achieving scalability and cost effectiveness for such real-time solution is challenging, but the solutions we have proposed and built helps to achieve a balance with cost and performance.

Future work involves automatic model training, thresholding, deployment, and fine tuning based on feedback for continuous improvement to reduce false positive alerts (noise). Utilizing model that generalizes pattern based on multiple time series analysis is also seen as a challenge and research is happening towards deep learning models with transformer-based time-series forecasting to improve generalization. Model explainability to provide proper justification to IT operators will also help them to understand why AI made a specific decision and in turn, help to receive feedback for improving model performance. For RCA, caching reports to further improve operations in real time is being considered and we’re also researching and implementing RCA at scale using application logs, apart from using metric time-series correlations.

Acknowledgment

We would like to thank Nicholas Pinckernell and Mateja Putic for their guidance and review on the architectural changes, Hongcheng Wang and Abhijeet Mulye for helping us initiate the project and find use cases, Gourav Gupta and Rajabhupati K S for helping us with a deployment, and Kolammal Sankaranarayanan for the editorial review. We are also grateful to Jan Neumann, Amit Bagga, Harish Jayesh and Kannan Subramaniam for their continuous support.

REFERENCES

[1] A. Lerner. “AIOps Platforms”, gartner.com, https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/ (accessed Dec. 8, 2022).

[2] “Market Guide for AIOps Platforms”, gartner.com, https://www.gartner.com/en/documents/4015085 (accessed Dec. 8, 2022).

[3] S. J. Bigelow, “AIOps (artificial intelligence for IT operations)”, techtarget.com, https://www.techtarget.com/searchitoperations/definition/AIOps (accessed Dec. 8, 2022).

[4] B. Linders, “Artificial Intelligence for IT Operations: an overview”, infoq.com, https://www.infoq.com/news/2021/07/AI-IT-operations/ (accessed Dec. 8, 2022).

[5] “MTBF, MTTR, MTTA, and MTTF”, atlassian.com, https://www.atlassian.com/incident-management/kpis/common-metrics (accessed Dec. 8, 2022).

[6] P. Notaro, J. Cardoso, and M. Gerndt, “A systematic Mapping Study in AIOps”, Service-Oriented Comp. – ICSOC 2020 Workshops, pp. 110–123, 2020, doi: 10.1007/978-3-030-76352-7_15.

[7] H. Wang, P. Manoharan, N. Nayan, A. Venugopalan, A. Mulye, T. Chen, and M. Putic, “AI for IT operations (AIOps) – Using AI/ML for improving IT Operations”, Fall Tech. Forum Proc. NCTA Tech. Papers, 2022.

[8] Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-World Challenges and Research Innovations", 2019 IEEE/ACM 41st Int. Conf. Soft. Eng.: Comp. Proc. (ICSE-Comp.), 2019, pp. 4–5, doi: 10.1109/ICSE-Companion.2019.00023.

[9] L. Rijal, R. Colomo-Palacios, and M. Sánchez-Gordón, “AIOps: A Multivocal Literature Review”, Artif. Intell. Cloud Edge Comp. Internet of Things. Springer, Cham, pp. 31–50, 2022, doi: 10.1007/978-3-030-80821-1_2.

[10] Prometheus - Monitoring system & time series database, prometheus.io, https://prometheus.io/ (accessed Dec. 8, 2022).

[11] Apache Spark™ - Unified Engine for large-scale data analytics, spark.apache.org, https://spark.apache.org/ (accessed Dec. 8, 2022).

[12] Apache Airflow, airflow.apache.org, https://airflow.apache.org/ (accessed Dec. 8, 2022).