Enabling Programmable Metric Flows

Aishwariya Chakraborty[1], Chander Govindarajan[2], Kavya Govindarajan[1], Priyanka Naik[1], and Seep Goel[2]

IBM Research, India

[1]{*aishwariya.chakraborty1,kavya.g,priyanka.naik*}*@ibm.com*, [2]{*chandergovind,sgoel219*}*@in.ibm.com*

***Abstract*—In the evolving computing landscape, extending from centralized clouds to multi-cloud and edge, the need for adaptable observability is becoming increasingly critical. Traditional static monitoring approaches grapple with inefficient data transfer, limited scalability, heterogeneous environments, and rigid metric processing pipelines. This paper introduces a novel metric processing system, Programmable Metric Flows (PMF), which is rooted in the principle of dynamism. PMF is a first-of-its-kind, light-weight, SQL-based metric processor. It empowers optimization-driven transformations of metrics, tailored to evolving resource availability and application requirements. This paper demonstrates how PMF enables various transformations for dynamic and fine-grained metric collection. We also showcase the capability of PMF for dynamically tuning the frequency of metrics to reduce the WAN cost in edge environments. Our experiments show that PMF performs at par with state-of-the-art techniques in terms of metric processing capability, with 10X lesser resource utilization. We envision PMF to usher in an era of lightweight programmability for observability platforms.**
PMF is open-source and available at: https://github.com/observ-vol-mgt/PMF.

I. Introduction

The continuous expansion of cloud computing is motivating enterprises to adopt a multi-cloud strategy [1]. The multi-cloud strategy provides organizations the flexibility to operate within the most optimal computing environment for each workload. Gartner [2] projections indicate that by 2025, 75% organizations will implement services across clouds. For enterprises utilizing cloud services across geographical locations, selecting a single public cloud infrastructure provider that meets all their requirements is a challenge, and thus, a multi-cloud approach becomes a necessity. Moreover, a multi-cloud strategy enables enterprises to capitalize on the capabilities, including resiliency, performance, and pricing, offered by the selected platforms. This approach also safeguards against vendor lock-in, preventing a scenario where an organization is compelled to incur higher costs due to reliance on a single cloud service and be vulnerable to potential constraints.

As organizations increasingly distribute their workloads across multiple cloud platforms, the need to gain comprehensive insights into the performance, behaviour, and interactions of these systems is paramount. Observability provides a holistic view of the entire infrastructure, facilitating real-time monitoring, optimal resource utilization, and proactive issue identification. Observability is central to all cloud deployments, facilitated by tools such as Prometheus [3], Graphite [4], VictoriaMetrics [5], and Application Performance Monitoring (APM) solutions [6–8] that seek to provide detailed analytics. These solutions are widely used across cloud deployments and are a cost overhead for enterprises. For example, Amazon Cloudwatch [9], Azure Monitor [10] and Google Cloud monitoring [11] adopt a per metric pricing model where enterprises incur charges for exported metrics, dashboards and queries executed against these metrics. This implies a direct correlation between the volume of metrics monitored and the associated costs borne by the enterprises.

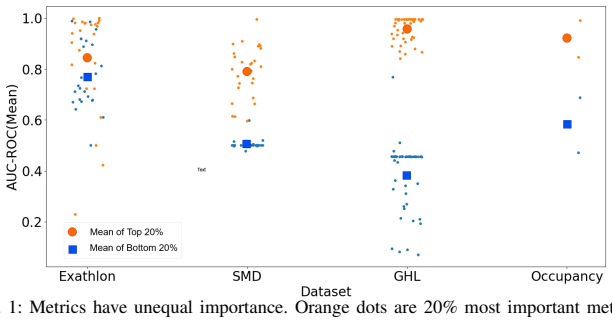

Fig. 1: Metrics have unequal importance. Orange dots are 20% most important metrics & blue dots are 20% least important metrics.

The transition to multi-cloud has resulted in an explosion in the scale of observability data. This data originates from various sources, the application containers, orchestration middleware and underlying compute hosts. It is crucial to aggregate observability data from these diverse sources to provide a unified view of the system and to enable timely detection of the root cause of failures [12]. The larger volume of observability data in multi-cloud leads to higher collection, storage and processing costs and also causes a substantial increase in Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR). Furthermore, it introduces complexity to all downstream observability tasks, ranging from the manual setup of dashboards to automated learning systems. Additionally, bandwidth constraint, a problem even in single-cloud environments, is aggravated in a multi-cloud setting. Cross-cloud observability plumbing incurs high egress costs and limits valuable bandwidth, especially in edge clouds with constrained resources. This emphasizes the importance of effectively managing the volume of observability data.

A simple approach to decrease metrics volume is to reduce their collection frequency. However, metrics serve varying observability objectives, such as anomaly detection, optimization and planning, and different metrics contribute differently to these goals. We show that metrics contribute unequally to different objectives by analysing several multivariate time series anomaly datasets [13–16] and report the AU-ROC [17]

All authors contributed equally

scores for anomaly detection (COPOD [18, 19]). We calculate each metric's importance as the Pearson correlation between the metric and the ground truth indicating system anomalies at a timestamp. Figure 1 shows the AU-ROC when only the top 20% most(least) important metrics are considered, with the remaining metrics replaced by their average value in the absence of anomalies. We observe that the top 20% most important metrics consistently outperform the least important 20%, indicating that treating all metrics equally is not optimal.

Metrics critical for real-time anomaly detection and performance issues may need to be collected more frequently while those required for bookkeeping purposes may be collected less frequently. Tuning optimal frequency for individual sensors is a well-studied problem for IoT data collection [20, 21]. These works can be considered as adaptive mechanisms; for instance, they transmit a data point only if it crosses a threshold from the mean of previous values. There is also prior art [22] that looks at the optimal collection frequency for different metrics based on the historical rate of change of the metric.

Existing solutions fare poorly in multi-cloud scenarios on two main dimensions. Firstly, these approaches are typically bottom-up, adaptive schemes where the collection rates of individual metrics are based on their local history. These approaches lack a global view, thereby, failing to consider factors such as the available bandwidth or the importance of a metric to all others for downstream tasks. Secondly, these solutions do not dynamically respond to changing underlying conditions. Multi-cloud environments, more than single clouds, are characterized by dynamically shifting conditions, including changing traffic, workload deployments due to cloud bursts, and varying available bandwidth. Traditional metric collection tools, such as Prometheus, OpenTelemetry [23], and Datadog, lack support for dynamic, on-demand changes in metric collection frequency and do not offer the capability to configure different frequencies per metric.

We propose a first-of-its-kind metric processing system that adopts a top-down approach by dynamically selecting per-metric frequency in a fine-grained manner. Our approach involves modelling the trade-off between frequencies of individual metrics, accuracy of downstream tasks, and the available bandwidth at a cloud location. We formulate this as a linear programming problem to solve for optimal frequencies, considering that both metrics and bandwidth change over time. Additionally, we advocate for larger frequencies for the long tail of unimportant metrics for higher bandwidth savings with minimal impact on the observability stack.

Enterprises can use approaches other than optimizing metric collection frequencies to reduce metric volume. For instance, they could aggregate metrics across devices in edge deployments and only collect individual metrics when anomalies arise. Enterprises may also seek to selectively offload the processing of observability data based on resource availability at the clouds. A disaggregated approach, where processing occurs at individual clouds and a summarized view is transmitted to the central cloud, during intervals of ample compute and memory resources could switch to a centralized approach during resource constraints, where a reduced set of metrics are collected at the edge and processed centrally. Most enterprises navigate through these architectures based on resource availability and downstream task requirements, executing selective partial processing at the edge and handling the remaining processing centrally.

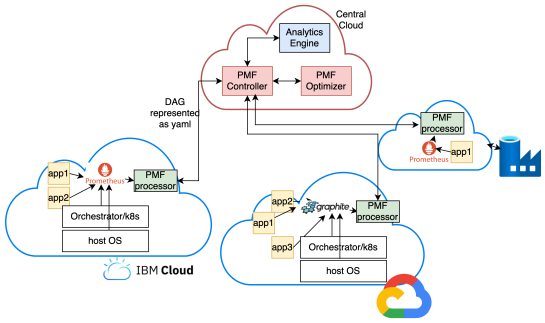

Fig. 2: PMF Framework Overview

We present the design and implementation of Programmable Metric Flows (PMF), shown in Figure 2. PMF is a dynamic metric flow processor that is built for transforming metrics with support for on-demand reconfigurations. PMF takes a Directed Acyclic Graph (DAG) of the transformations to be performed on the metrics data. These transformations could be popular operations such as aggregation, filtering as well as our novel transformation to set metrics' collection frequency. PMF translates these transformations into SQL queries that are executed on a database of ingested metrics. A key requirement for such a system is to be generic and adaptable to any metric format and orchestration platform, to be lightweight but still support various transformations required to handle the dynamic nature of multi-cloud metric pipelines. PMF is based on an in-memory SQL database and is fast, minimal and generic - allowing easy programming of metric flows.

To summarize, our contributions in this work are:

1. We present PMF, a novel system for fast, programmable and dynamic transformation of metric flows (§ III).
2. We propose a novel top-down approach to manage metric data volume by dynamically adjusting the frequency of collection, based on metric importance and resource availability (§ IV).
3. We show the programmability of PMF with support for various transformations like filter, frequency, aggregate and adaptive. PMF works on metric transformation DAGs, supporting combinations of these transformations for real-world use cases (§ III-B).
4. We evaluate PMF 's performance and show that it can scale up to 200K metrics per node. PMF is light weight with a low resource consumption of ∼2.5% CPU and ∼300MB memory. The low control plane latency (in the order of microseconds) renders PMF highly responsive and adaptable to the user's dynamic requirements (§ V-A).
5. While no prior art supports dynamic metric transformation, we compare the performance and resource overhead of PMF with OTel. For comparable execution time, OTel requires 10x more CPU than PMF for filter transformation (§ V-B).
6. We showcase the applicability of PMF for the use case of

dynamically adjusting the frequency of metrics in an emulated multi-cloud environment. We demonstrate bandwidth savings and negligible impact on downstream tasks. We compare our approach with the state-of-the-art baseline, which collects metrics at the frequency of 30s. Compared to the baseline, we reduce the loss of significant information by ∼600x (§ V-D).

II. Background and Related Work

Efficient monitoring of cloud environments is necessary for timely root cause analysis and low MTTD and MTTR. The metric observability pipeline for a multi-cloud environment consists of three core components:

(i) Metric Collector is the set of monitoring agents placed across nodes in a multi-cloud or edge setup. These agents capture real-time metrics with cloud-native and open-source solutions such as OpenTelemetry [23] (OTel) and Prometheus [24]. OTel is an instrumentation standard that focuses on providing a unified approach to collect traces, metrics, and logs. It includes libraries for different programming languages, agents for data collection and transformation, and exporters for transmitting data to observability backends. Prometheus enables the collection of metrics from various sources like the application, operating system, and network. Metrics are scraped from these sources periodically, and are stored locally in a time-series database (TSDB). A metric consists of a timestamp, metric name, value and a set of labels represented in the form of a key and value, allowing transformations based on labels.

(ii) Metric Aggregator Since local storage is limited and metric analyzers rely on historical data for proactive failure detection, a long-term storage mechanism is essential. Tools such as Thanos [25] and Cortex [26] enable long-term multi-tenant storage of metrics, and can consume metrics from multiple sources. Thanos acts as a global query and storage aggregator for Prometheus instances, with metric federation across different cloud or edge nodes ensuring a unified and coherent view of the entire infrastructure. It supports horizontal scalability, enabling organizations to handle large volumes of metrics efficiently. However, the cost overhead of communication from a collector, like Prometheus to an aggregator, like Thanos is over WAN. Managing the volume of metrics and reducing this overhead is one of our key contribution.

(iii) Metric Analyzer derives insights from the collected data. Metrics can be used to define alerts and be visually monitored through dashboards in Grafana [27]. Integration of machine learning algorithms into processing pipelines enhances the capability for anomaly detection and predictive analytics, enabling proactive issue resolution.

Related Work Efficient monitoring implies fast identification of application, system, and performance issues with low overhead on the environment, which is the aim of many prior arts in academia and industry. Most monitoring frameworks work on static KPIs or performance analysis to set the metric sampling rate [9, 28–31]. Viperprobe [12], an eBPF based monitoring tool, defines critical metrics based on overhead of collecting each metric. However, it does not consider the impact of these metrics for a downstream task. Volley [32] adjusts the sampling rates of metrics based on overhead on the node from which the metrics are collected. Sieve [9] clusters and aggregates metrics across microservice chains based on dependencies to reduce the number of metrics. Toni et.al. [33] propose preventing metrics from being sent to the metric collection system from an application if the metric value is not changing over time. This is efficient for getting maximum storage and bandwidth saving benefits, but it assumes control over the application, and thus, may not work as a plug and play to any existing metric collection system. Moreover, all these frameworks focus on the entire set of metrics collected without considering the impact of individual metrics on downstream tasks. SkyView [34] proposes constraint-based transformations to reduce query cost for metrics. The authors suggest dropping the non-useful metrics for bandwidth reduction, but do not consider the need for these metrics with addition of new anomaly functions or new application deployments, and thus, may not be applicable for the dynamic cloud environment.

Efficient monitoring in dynamic and resource constrained IoT networks is achieved using various data-driven techniques. These include basic threshold-based filtering approaches [35, 36] which sense data continuously and forward it for processing only when the data value exceeds a given threshold. Since sensing also consumes significant energy, some works [37, 38] propose adaptive sampling techniques that utilize the statistical aspects like, spatial and temporal correlation of sensed data. Another category of work [21] on data transmission reduction in IoT explores machine learning techniques to predict the future sensor data values and decide whether to sample a data point. PREMON [39] proposes generation of a model on historical sensor data at a centralized location, distributing the model to the sensors, and the sensors sending the readings only when different from the model values. To reduce data transmissions further, DBP [40] suggests the use of sliding windows for buffering and waiting for multiple occurrences of divergent data points before transmission. In another stream of work, the data collection and transmission rate of sensors is determined using optimization. These schemes establish a trade-off between energy consumption [41], network throughput [42], and data quality [43] but, are specific to sensor networks/IoT environment. We build on similar principles intending to cause minimum impact on anomaly detection algorithms that use these metrics. OTel provides various transformation like filtering and aggregation, but does not provide a framework for adjusting these transformations in real-time.

III. Programmable Metric Flows (PMF)

Figure 2 shows the typical architecture of a cloud-native metric collection stack extended to multi-cloud scenarios. The applications are instrumented with preferred observability libraries. The metrics are collected at a local collector, either via push or periodic pull, over the local network. The metrics can also be forwarded not only to the central cluster but also to an alerting agent or local storage. The central metric aggregator periodically receives updates from the individual collectors over the WAN. The Prometheus Remote Write protocol, for

example, sends batches of metrics using a common protocol, understandable by a wide range of aggregators.

We enable dynamic control on the transmission of metrics from local collectors to the central aggregator. We have the following requirements from our metric processing system:
(i) **Generic**: Support a wide range of metric collection stacks and frameworks. It should be plug-and-play and work with a variety of solutions like Prometheus, Graphite, Influx, etc.
(ii) **Fast & Minimal**: Add minimal overheads to cloud deployments and the collection times of metrics.
(iv) **Programmable**: Enable new approaches towards programming metric flows. Beyond our use case of programming frequencies, PMF should allow fine-grained control and intelligent metric collection based on end-user requirements.
(v) **Fine-grained**: Allow the selection of a specific subset of metrics based on their label keys and values for programming.
(vi) **Dynamic**: Support on-demand updates, making the process of changing flow processing quick and easy.

Figure 3 shows the system architecture of our proposed PMF processor. Specifically, we envision that our PMF processor is deployed on edge locations as a proxy-like middlebox between the local controller and central aggregator.

The Processor comprises the following subcomponents:
(i) **Receiver**: Decodes transmitted metrics to a processing-friendly format. For e.g., Prometheus Remote Write conveys compressed metrics as batches encoded in a Protobuf [44] format. Similar formats exist for platforms like OTel.
(ii) **Executor**: Core component responsible for transforming the metrics, e.g., changing collection frequency of metrics.
(iii) **Transmitter**: Encodes updated metrics back into the expected format and transmits to the central aggregator.
(iv) **Controller**: The user-facing component with an interface to create(update) the metrics programming flow.
(v) **Configurator**: Converts the YAML specification of the transformation into transformation queries that can be applied to the metrics data stored in the PMF processor.

The Receiver, Transmitter, Controller and Configurator are straightforward components, grounded to actual implementation requirements. However, due to the inherent complexity of the Executor, we prioritize a more detailed explanation of its design and implementation for the brevity of space.

To implement the Executor, we propose a SQL-based system. The Receiver inserts metrics into SQL tables and the processing of metrics is achieved using SQL queries. This approach allows us to rely on a standard framework for efficient programming of metric flows.

Prior art in TSDB avoided SQL databases for 3 reasons: (a) difficulty in scaling, (b) poor suitability for append-only data and (c) archiving old data [45]. All these issues pertain to the collection, storing, and querying of metrics for long-term use. Metrics, in our context, have a limited span. The metrics in the edge location are stored in the SQL database for the duration of processing and are finally either dropped or forwarded.

Using streaming data solutions is yet another option for time series data. However, they are unsuitable for 3 main reasons: (a) such systems are designed for large scales of data, the so-called big data, and violate our minimal and fast requirements, (b) streaming vs batching: though metrics are themselves streaming, the transfer of metrics between from edge collector to central aggregator is usually in large batches (to minimize the transfer overheads), and (c) SQL provides a more standard interface for the creation of processing programs. SQL also allows wider adoption by enabling easy porting of prior processing approaches to our platform.

Figure 4 shows the proposed schema to store metrics. We model metrics using 2 tables: (a) **metrics** table to store metric instances with timestamp and value, and (b) **labels** table to hold metric metadata in the form of labels as key-value pairs. These tables are linked using the phash column which is a unique hash string representing the exact label set, key-values for a metric. For inserting an incoming metric, we first compute the metric's phash. Then, we insert each label corresponding to this metric into the **labels** table, only if the corresponding phash does not already exist in the table. Since label key-values for a metric stream are fixed, this insertion happens only once at the start of a metric flows. Finally, a new row is added to the **metrics** table corresponding to the metric. The **metrics** table has an additional *markForExport* column that is used to indicate rows that will be forwarded to the central aggregator at the end of the execution.

A. Transforms

PMF supports processing of the metric data on the edge cloud before forwarding them to the central cloud. Consider the smallest unit of operation a user may want to apply on the metric flows. We term these operations as **Transforms**. In our primary use case, this operation reduces the number of metrics being sent by applying a sampling frequency. Each Transform is realized as a SQL transaction, comprising standard SQL queries. These queries create/update/delete rows in the metrics and labels tables. The SQL-based design of PMF enables support for generic transforms. The transforms supported by PMF are a superset of capabilities available in prior art [23], as well as novel transforms such as Frequency Transform that is based on our primary use case to manage the volume of metric data. We support the following transforms:

Frequency Transform: The Frequency Transform allows controlling the sampling frequency for a set of metrics. It has 2 parameters, a selector parameter and a frequency parameter. The selector parameter is a SQL select statement that allows to choose a set of metrics in a fine-grained manner using labels. Example selectors could be (a) all metrics of *app1* (based on the label key app), or (b) all network related metrics for an application (regex on label keys which match *net.**). The frequency parameter specifies the frequency of sending metrics. For example, if metric samples come every 10 seconds and the user programs a frequency of 5 min, only 1 data point is transmitted in a 5 min window. The optimizer discussed in § IV dynamically determines the frequency for each metric based on its importance towards downstream tasks being executed in the central cloud, such as anomaly detection, while considering external factors such as resource constraints.

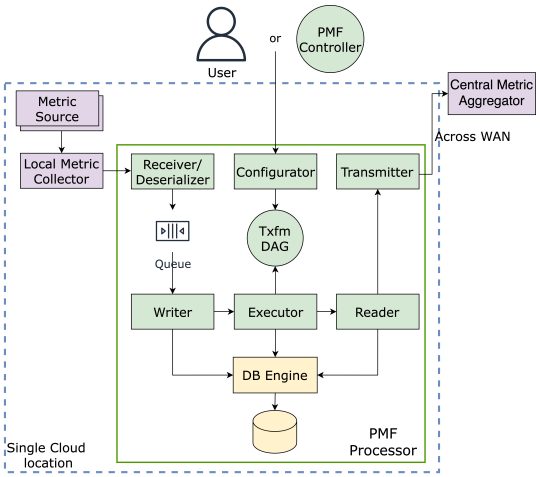

Fig. 3: PMF Architecture

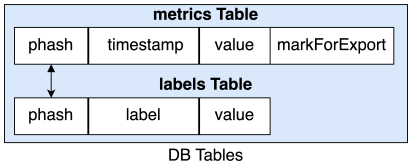

Fig. 4: Table Schema in PMF processor

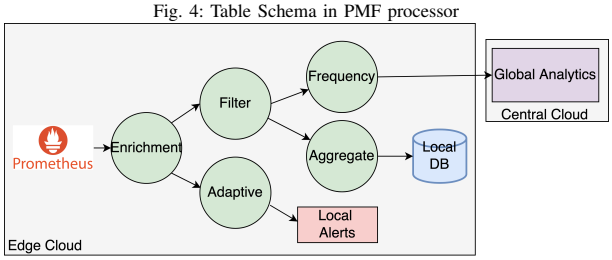

Fig. 5: Example of a DAG for Edge Cloud Environment

Filter Transform: Filter drops all metric datapoints matching a selector. This can be used to drastically reduce the amount of data being sent. For instance, all metrics pertaining to a particular namespace can be dropped. Since PMF allows on-demand programming, the Controller can later re-enable these metrics as and when needed.

Adaptive Transform: In this transform, a metric is selected for forwarding based on its value, specifically if it provides distinctive information to the time series. An example adaptive scheme only forwards metrics when their value at that timestamp is significantly different (such as by 2 standard deviations) from the previously observed value. This approach helps avoid unnecessary transmission of numerous small updates.

Aggregate Transform: Aggregate sends fewer metrics that are derived from the full data. For instance, when metrics are coming in every 10 sec, we may choose to send a single value every 1 min, calculated as the mean of the values within that time window. Prior art § II has studied advanced variants of aggregation that can be used to send approximate updates.

No-Op Transform: Generally, a user specifies transforms on a subset of metrics. The remaining metrics are assumed to require no transformation, i.e. are forwarded without any processing. For such metrics, we introduce a No-Op transform, implying no transformation is to be performed on the metrics. This is the default mode for metrics not selected by any transforms, ensuring that unspecified metrics are not dropped.

Table I lists the details of our semantically equivalent Transform to SQL translation strategy. Our SQL-based approach allows users to craft custom transformations easily. A transform is succinctly represented as a parameterized set of SQL statements. These parameters typically fall into two categories: (a) the selector for selecting relevant metrics and (b) variables controlling the transform, like the frequency window, aggregate threshold etc.

We can further classify Transforms as stateless or stateful. The Transforms may require state, for instance, some adaptive algorithms store past historical data to compute parameters such as threshold and standard deviation. In these cases, since our executor is SQL-based, stateful transforms can create tables for additional state management. Each Transform unit is free to define its own schemas for these state tables.

B. *Transform DAGs*

To support generic programming of metric flows, we model the input as a Directed Acyclic Graph (DAG) of transformation, where each node in the DAG is a transform unit. The transformation DAG is specified through a YAML configuration. Specifying DAGs through yamls is a well known technique in prior art such as Kubeflow [46]. DAGs require changes to the Executor detailed earlier. The executor now processes a DAG starting from the input table. Each Transform node in the graph selects a subset of the table according to its logic and forwards a view to their children nodes. Each child node operates on this reduced input set, and creates new views for its children. Finally, the Executor exports all the rows selected in the leaf nodes and sends one batch.

This design of modelling the transformation operations as a DAG of individual Transform units, where each Transform unit is a reusable block of SQL statements, allows a wide range of possible metric flow programming using simple components.

Example DAG: We explain the need for DAG to model PMF transformations with the help of an example application running in an edge cluster. The application sends its metrics to the central cluster every 5 seconds. The collected metrics from multiple such edge clouds are then analyzed by an analytic engine at the central cluster for a recommendation based on geo locations. Since the analytics algorithm requires some additional insights into the metrics, the metrics need to be enriched with additional metadata. Moreover, due to limited bandwidth availability, the user requires filtering to send only metrics relevant to global analytics to the central cloud. Further, as the analytics functions operate on incoming data at per-minute granularity, the user wants to send the metrics at per-minute frequency. In addition, there are two local edge cloud operations that the user wants to perform. The first is to use some of the metrics for local alerting, for example, health check metric, CPU utilization overload metric, etc. The second is to store and maintain some additional metrics locally for a certain duration for audit purposes.

The above edge cloud metric requirements can be achieved using the PMF transformation DAG capability as shown in Figure 5. The user can specify an enrich transform, which adds metadata available on the edge cloud as new labels. This can be another transformation in addition to the ones

Transform	Transform Query	Operation
No-Op	select * from metrics and markForExport=1	The default transform for metrics not affected by any transform. The selected metrics are exported out of the processor.
Filter	select metric from metrics where label_1 != "A" and forward; no-op (label_1 == "A")	Drop metrics with matching labels and forward the remaining metrics
Frequency	select metrics where label_1 == "A" current_timestamp - last_timestamp >= metric_frequency_interval and forward; no-op (label_1 != "A")	Select metrics with matching labels and send if the timestamp difference is greater than frequency interval of the metric
Adaptive	select metrics where label_1 == "A" and metric_value > val_threshold and forward; no-op (label_1 != "A")	Forward metric only when the metric value is greater than the threshold value specified
Aggregate	select avg(metric_value) where label_1 == "A" and forward; no-op (label_1 != "A")	Forward only the aggregate (avg) of the metric value

TABLE I: Transform to SQL translation in PMF . The transforms typically have parameters such as selectors (we have taken a simple selector on a single label) and variables.

described above III-A. The DAG can then fan out to serve the 3 requirements – (i) using adaptive transform for local alerts; (ii) using the aggregate transform to store locally for audit metrics; and, (iii) adding a filter transform followed by frequency transform to send only the required metrics for global analytics at a frequency of 1 min.

C. Dynamism

The modelling of metric transformations as a DAG automatically supports one of our core desired properties of dynamism. In this model, dynamism can be achieved in two ways:

The first and more common requirement is when we need to update the parameters of a Transformer - the selectors and the variables. For example, suppose a family of metrics with label *app="foobar"* is being collected at a frequency of 5 minutes. Due to changing conditions such as bandwidth or metric importance for some downstream tasks, there is a requirement to increase the frequency to once a minute. Such a change does not require an update to the entire DAG, but to a single node and, the rest of the DAG may operate unchanged.

The second form of dynamism is when the user wishes to dynamically modify the structure of the DAG, such as by adding or removing nodes in the DAG. An example of such dynamism is that under certain bandwidth restrictions, the user may wish to Filter all metrics with the label *instance="secondary"* of the *app="foobar"*, thus adding a new child to the previously seen Transform dynamically (and removing it later, when bandwidth is available).

PMF supports both forms of dynamism. The PMF Controller receives requests to update the DAG from its northbound API. It then reprograms only the relevant portions of the DAG by changing variables, resetting state, adding or removing children nodes, while preserving the rest of the DAG structure. In the next batch processing, the executor uses the updated DAG, leading to minimal disruption of operations.

D. Implementation

We implement our PMF processor as 678 lines of Go program, with the executor built on top of embedded SQLite running in in-memory mode. All of the database operations are done in SQLite using the Go bindings. The transformation DAG is modelled using structures in Golang, which store and update SQL queries to be used for execution.

We selected SQLite as the engine since it can be embedded into our program and meets our fast and minimal requirements. Alternative implementations on top of other SQL database engines may be considered, where our system is implemented as a plugin to minimize the data transfer time to and from the database. We opted for in-memory operations to further speed up processing, since the metrics are being stored and processed only temporarily, but show the impact of using a files system in the evaluations § V.

Since SQLite does not support concurrent transactions, currently, our executor operates on the DAG in a depth-first manner. This restricts our processing speed and does not allow realizing the full potential of our design. We can easily switch to other SQL engines to overcome this limitation.

Adding a new transform in OTel requires adding the specification using their own grammar, OpenTelemetry Transformation Language [47], whereas PMF simplifies this task with the use of simple SQL queries. This is evident from the fact that PMF implements a filter transform in just 14 lines of code, whereas OTel requires 221 lines of code to implement the same [23]. Thus, PMF provides a plug-and-play mechanism to perform dynamic transformations to the metrics pipeline.

IV. Dynamic Frequency Optimization

The PMF Optimizer is a centrally located component responsible for dynamically determining the optimal frequency for every metric. In this section, we discuss how the Optimizer computes the optimal frequency.

System Model: We model the problem of determining the optimal frequency of metrics as an optimization problem. Typically, each cluster produces a large number of metrics of different types such as app-level and node-level metrics. The number of metrics generated per cluster varies dynamically depending on its workload. In this work, we consider that, at a particular time t, there are N clusters such that cluster i generates a set $\mathcal{M}_i$ of M_i metrics. These metrics are stored locally and periodically sent to a centralized controller for further processing. For the metric j of cluster i, denoted by m_{ij}, we denote its collection frequency and data-point size to be f_{ij} and d_{ij}, respectively. Thus, the total bandwidth consumption b_i of cluster i for the transmission of metrics and the constraints that it must satisfy are presented below.

$$b_i = \sum_{j=1}^{M_i} d_{ij} f_{ij}, \quad b_i \leq B_i, \quad \forall i \in [1, N] \text{ and } \sum_{i=1}^{N} b_i \leq B \qquad (1)$$

Here, B_i and B denote the available upload and download bandwidth of cluster i and the controller, respectively. To analyse the impact of varying the collection frequency of metrics, we consider the most commonly used downstream application of anomaly detection.

Modelling of Anomalies: Typically, anomalies are of 3 types [48], point anomalies where individual data points are anomalous with respect to other data points, contextual anomalies where data points are anomalous depending on the

context (flat period in an oscillatory curve), and collective anomalies where a collection of data points are anomalous together, but not individually. It is implicit that each anomaly impacts a subset of available metrics. A large number of time series anomaly detection algorithms have been proposed by extensive research in this area. We treat the anomaly detection algorithm as a black box for the rest of our work.

We consider that the anomaly detection algorithm is able to detect a set $\mathcal{A}$ of K anomalies. Each anomaly k impacts a subset $\mathcal{M}_k$ of metrics either directly (#requests that error out) or indirectly (queue build up due to slow processing causing high memory utilization). Here, $|\mathcal{M}_k| \geq 1$, i.e., each anomaly impacts atleast 1 metric. The anomalies have varying levels of significance, and impact each metric to a different degree, which is quantified using *weights*. We denote the weight of anomaly k as v_k, and that of m_{ij} for detection of k as w_{ijk}, such that $v_k, w_{ijk} \in [0, 1]$. The following constraints hold.

$$\sum_{k=1}^{K} v_k = 1 \text{ and } \sum_{i=1}^{N}\sum_{j=1}^{M_i} w_{ijk} = 1 \quad \forall k \in [1, K] \tag{2}$$

The fast and successful detection of an anomaly relies heavily on the availability of updated metrics after the anomaly occurs. Thus, in a system which collects metrics at different frequencies, the time to detect (TTD) an anomaly is limited by the refresh time of the metric collected at the lowest frequency. Mathematically, we express TTD T_k for anomaly k as follows.

$$T_k = \max_{m_{ij} \in M_k} \frac{1}{f_{ij}} \text{ and } T_k \leq T_k^{max}, \quad \forall k \in [1, K] \tag{3}$$

Here, T_k^{max} is a given threshold for successful detection of k.

Problem Formulation: In this work, our aim is to decide the optimal metric collection frequencies such that the overall bandwidth consumption for metric data collection as well as average TTD of all anomalies are minimized. Hence, the objectives of this problem are as follows.

$$\text{OBJ1}: \ \arg\min_{\mathcal{F}} \sum_{i=1}^{N} b_i, \quad \text{OBJ2}: \ \arg\min_{\mathcal{F}} T_k, \quad \forall k \in [1, K] \tag{4}$$

where $\mathcal{F}$ denotes the frequency matrix of all metrics in $\mathcal{M}$. By careful inspection of this equation, it is observed that the information regarding the importances or weights of each metric and each anomaly is not captured in either of the objectives. However, we analyse that this information can help to further improve the efficiency of anomaly detection by selecting more nuanced frequency distributions: (i) Some anomalies are more significant than others, the metrics affected by the more significant anomalies can be collected at higher frequencies, thereby reducing their TTDs in comparison to less significant ones. (ii) A particular metric may be affected by a number of anomalies to varying degrees. It may be beneficial to collect such metrics, i.e., with high cumulative weights, at higher frequencies. To capture these, we introduce a parameter, termed as *Freshness Index*, to quantify the freshness of significant metrics. Denoted by FI, it is defined the the weighted average of the frequencies of all metrics. Based on this term, we define the third objective of this problem as follows:

$$\text{OBJ3}: \ \arg\max_{\mathcal{F}} FI, \text{ where } FI = \sum_{k=1}^{K} v_k \sum_{m_{ij} \in \mathcal{M}_k} w_{ijk} f_{ij} \tag{5}$$

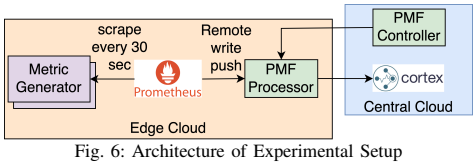

Fig. 6: Architecture of Experimental Setup

We combine the aforementioned objectives into a single objective function $\mathcal{O}$ by rewriting the terms in maximization form and calculating their weighted sum, as shown below:

$$\mathcal{O}: \ \arg\max_{\mathcal{F}} \left(\alpha \sum_{k=1}^{K} \sum_{m_{ij} \in \mathcal{M}_k} v_k w_{ijk} f_{ij} + \beta \sum_{k=1}^{K} \frac{1}{T_k} - \gamma \sum_{i=1}^{N} b_i \right) \tag{6}$$

subject to bandwidth and TTD constraints mentioned in Equations (1) and (3) and $F_{min} \leq f_{ij} \leq F_{max}$. Here, α, β, and γ are scaling constants, and F_{min} and F_{max} denote the minimum and maximum possible metric collection frequencies.

Solution: The optimization problem defined above is a non-linear optimization problem with non-linear constraints. It can however be linearized easily by introducing introduce auxiliary variables $z_k = 1/T_k \ \forall k \in [1, K]$. The resulting problem with linear objectives and constraints can be solved using any linear optimization algorithm such as Simplex and Interior Point method. In our implementation, we use Gurobi [49] optimizer which implements these algorithms under the hood. Specifically, we use the gurobipy Python library to implement the optimization code. The framework to modify the metrics as per the output of the optimizer is achieved using PMF.

V. Evaluation

Experimental Setup: The PMF Processor is built over SQLite [1] [50]. We use Prometheus [2] [51] and Cortex [3] [52]. Comparison experiments are with OpenTelemetry [4] [53] and we use Gurobi [5] [54] for optimization. Our experiments were run on a cluster of baremetal servers [6]. We describe our evaluation setup in Figure 6.

Metric Generator: We used a custom metric generator that varies the number of metrics, the number of labels and the size of each metric. The metrics are in the Prometheus metrics format, but m can be easily extended to other formats like OTel. We run a Prometheus instance that scrapes metrics from the metric generator every 30 seconds and then pushes them in the remote write format to the PMF Processor.

We evaluate our PMF Processor on the following axes: (i) Throughput, (ii) Per-component and end-to-end system latency and (iii) CPU and Memory Utilization. Insert time is the time taken by the Reader to insert a batch of metrics into the table. Exec time is the time taken by the Executor to execute the SQL queries on that batch and Export time is the time taken by the Writer to export the metrics in their original format. The total Batch Execution Time (BET) is the sum of these. Throughput is the total number of metrics in a batch divided by the BET ($Throughput = \#metrics/BET$). The resource overhead of the PMF Processor is consistently low and experiences

[1] v3.31.1 [2] v2.43.0 [3] v1.14.1 [4] v0.92.0 [5] v10.0.1 [6] Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz processor with 64 cores and 2 threads/core, and 800 GB RAM, running 64-bit Ubuntu 20.04.6

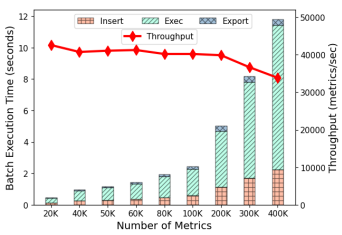

Fig. 7: BET & Thrpt. on varying #metrics

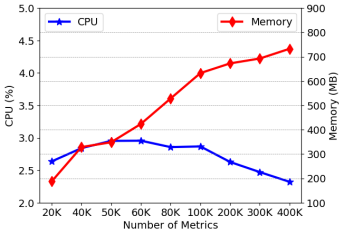

Fig. 8: Effect of #metrics on resource util.

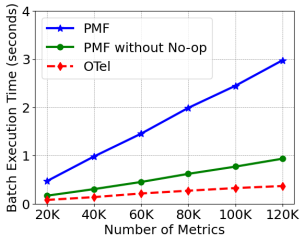

Fig. 9: Performance Comparison: OTel

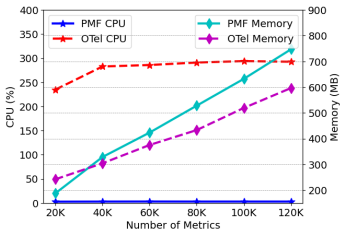

Fig. 10: Resource Comparison: OTel

spike only during batch metric processing. Therefore, for an accurate representation of the resource overhead, we report the 99th percentile for both CPU and memory overheads. Unless otherwise specified, experiments are run with 40K metrics and 10 labels generated every 30 seconds and PMF Processor executes a Frequency Transform with selector defined on one label. We use number of metrics in similar ballpark as production cloud environments as shown in [55].

A. PMF Performance and Scalability

DAG Type	No. of Units	No. of Selectors	Creation Time (µs)
Singular	1	1	30.785µs
	1	7	51.099µs
	1	15	51.38µs
Parallel	7	1	106.853µs
	15	1	187.809µs
Sequential	7	1	127.463µs
	15	1	288.854µs

TABLE II: DAG creation Latency

Throughput and Batch Execution Time: In Figure 7, we show the performance and scalability of the PMF Processor by plotting the throughput and per-component execution time while varying the number of metrics processed in a batch. Notably, the Exec latency is the highest, followed by the insertion latency. PMF exhibits linear scalability up to 200K metrics, with a linear increase in end-to-end BET with the number of metrics. Throughput is stable at around 40K metrics/sec up to 200K metrics, gradually declining beyond that threshold. For a batch of 40K metrics, the BET is ∼0.9s.

Resource Overheads: Figure 8 shows the resource utilization of the PMF Processor. The CPU usage remains consistently low at around 2.5-3%, exhibiting minimal variation with the number of metrics per batch. On the other hand, memory scales linearly with the number of metrics due to the batch processing nature of the PMF Processor, where all arrived metrics are stored before processing. For a batch of 40K metrics, the memory overhead is ∼328MB.

Control Plane Latency: The PMF control plane initializes and updates the DAG. Initialization of the DAG happens only once for conversion to corresponding SQL queries. However, the user can update the DAG dynamically. This update could be minor, such as updating a parameter of a Transform Unit or adding/deleting Transform units in the DAG. The updation time for DAGs is negligible, measuring around ∼12µs. Table II shows the creation time of DAGs of varying complexities. A DAG with a single Transform Unit with multiple selectors consolidates those selectors into a single SQL condition. A Parallel DAG has multiple disparate queries and a Sequential DAG consists of nested queries. We observe that the DAG creation time is quite small, a Sequential DAG with 15 Transform units taking the maximum time of 288µs. The Fast control plane latency renders PMF highly responsive and adaptable to the user's dynamic requirements.

B. Comparison with State-of-the-art

We compare our system with the closest state-of-the art for metric transformation, OTel. It does not support all the transformations described in § III-A. Therefore, we focus on the end-to-end filter transformation latency, varying the number of metrics as shown in Figure 9. We use the same metric generator and run OTel on the same setup as PMF . However, a significant operational difference between OTel and PMF is that by default PMF exports metrics that are not transformed (i.e. metrics not matching the selector), while OTel drops non-transformed metrics unless explicitly specified. Consequently, PMF has a higher BET when the default behaviour (No-Op transform) is enabled, which increases with the number of metrics. For a fair comparison, we configure PMF's default behaviour to drop non-transformed metrics (PMF without No-Op) and observe comparable execution times for both PMF and OTel. The slightly higher execution time of PMF can be attributed to our more generic and programmable design. Figure 10 shows the resource utilization comparison of PMF and OTel. Currently, PMF processor is not optimized for efficient memory utilization. In future, we plan on working on optimizations targeted at enhancing table structures. Interestingly, PMF can process the same number of metrics with significantly lower (10X) CPU consumption compared to OTel, demonstrating PMF's applicability in resource-constrained environments such as edge clouds. It's worth noting that dynamic transformation updates are not supported by OTel and are thus not part of the comparison.

C. Microbenchmarks

Effect of Number of Labels A metric comprises of a metric name, a set of labels (in key:value format), a timestamp and a value, collectively contributing to the metric's size. The number of labels in a metric significantly impacts not only the metric size but also metric processing time and memory overhead. To assess this impact, we vary the number of labels in a metric from 5 to 20, aligning with the typical label count (10-15) observed in prior art and production environments [56, 57]. A metric generated by the metric generator with 10 labels is ∼450B in size. In the PMF Processor, an increase in the

number of labels corresponds to more entries in the labels table § III, resulting in higher exec times, as depicted in the green segment of the bar graphs in Figure 11. This increase in table size also leads to the rise in memory consumption, as illustrated in Figure 12, while the CPU consumption remains consistently low and stable.

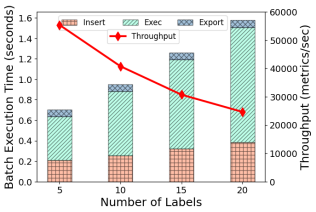

Fig. 11: BET & Thrpt. with #labels

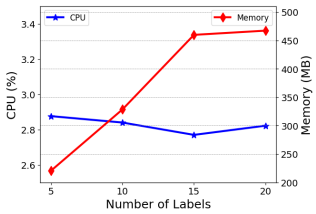

Fig. 12: CPU and Memory with #labels

Effect of Transformation Type In Figure 13 we plot the BET for different Transform Units § III-A. All experiments are on a single Transform with a single selector. While insertion latency is independent of the Transform, exec latency depends on the Transform being executed. The Filter Transform has the least execution latency. Frequency, Adaptive and Aggregate Transforms are more complex as they perform non-trivial computation on the metrics data and have higher exec latency. No-op Transform Unit updates every entry and marks them for export, thus having the highest exec and export latency. CPU and Memory overheads (Figure 14), are stable across different Transforms. These overheads are primarily a function of the number of metrics and their corresponding sizes, rather than the transformation being applied.

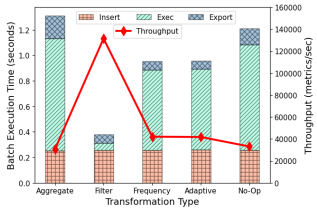

Fig. 13: Effect of Transformation Type on Batch Execution Time and Throughput

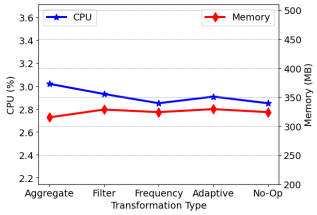

Fig. 14: Effect of Transformation type on CPU and Memory Overhead

Effect of DAG size and Type Table III shows the BET, component wise latency, Throughput and resource overhead by: (i) varying the DAG size and (ii) comparing Parallel vs Sequential DAG Type. All units in these DAGs are Frequency Transform. Increasing the DAG size from 3 to 7 units has a negligible impact on resource utilization.

Each transform in Parallel DAGs operates on all the metrics, exhibitng a higher execution latency than Sequential DAGs, wherein each Transform operates on the output of the preceding Transform. The higher latency of Parallel DAGs could potentially be offset by parallel processing.

We also observe that Sequential DAGs have higher export latency. This is due to our No-Op Transform Unit, where for every transform selecting a subset of metrics, the complementary set of metrics is processed using the No-Op transform (currently set to send all). In the parallel DAG, a single No-Op unit handles the complement of the intersection of all selectors. However, in the Sequential DAG, there is a No-Op Transform at every level of the chain, sending all metrics not Transformed. Figure 15 provides a detailed explanation of this reasoning. Consequently, Sequential DAGs export a larger number of metrics due to the No-Op transform and incur higher export latency as demonstrated in Figure 13.

Effect of Number of Selectors As mentioned in § III-A, each transform has one or more selectors, defined on labels, which act as conditions in the SQL query (WHERE clause). Selectors allow users to select groups of metrics using label keys and values, a Transform Unit can have queries on multiple labels. In Table IV, we evaluate the overhead of union of multiple selectors (performed as an INTERSECT operation over all the selectors), for a Frequency Transform. We observe that only the exec time varies with the number of selectors. The insert latency and resource usage remain constant, as expected.

#Selectors	Insert (sec)	Exec (sec)	Export (sec)	E2E (sec)	Throughput (metrics/sec)	CPU (%)	Memory (in MB)
3	0.284	1.155	0.140	1.580	31638.89	2.56	223.67
5	0.296	1.529	0.147	1.972	25348.54	2.54	286.16
7	0.295	1.786	0.143	2.225	22467.87	2.39	341.53

TABLE IV: Effect of #Selectors on PMF Performance and Resource Overhead

Effect of In Memory vs File storage The use of SQLite gives us the benefit of using either RAM or disk storage for the metrics and labels tables § III. As disk storage incurs an expensive I/O operation, the insert operation is expensive as seen in Table V. But it also tends to have low memory(RAM) pressure, since the tables are not maintained in-memory as in the alternative case. Given the relatively lower memory overhead and the faster compute, in-memory storage emerges as the preferred choice for implementing PMF processor, and it is the default choice for all other experiments.

Storage Type	Insert (sec)	Exec (sec)	Export (sec)	E2E (Sec)	Throughput (metrics/sec)	CPU (%)	Memory (MB)
File	0.824	0.68	0.066	1.571	25448.53	2.23	221.91
Memory	0.254	0.629	0.066	0.951	42043.30	2.84	328.52

TABLE V: Effect of Storage Type on PMF Performance and Resource Overhead

D. Evaluation of Dynamic Frequency Usecase

To show the impact of metric collection with dynamic frequencies, we simulate a multi-cloud system with 1 central cloud and 10 edge clouds running 100 clusters each. Metrics are generated at a frequency of 1 second by the clusters. The PMF Optimizer runs at the central cloud. We also simulate each metric time series as a step function with an increment every 10 seconds with some probability. We assume *random* weight distribution between anomalies and *Pareto* [58] (80-20) weight distribution between metrics per anomaly. The detailed simulation parameters are presented in Table VI.

For comparison, we consider the following two baselines and analyse their performance in our simulation settings: (a) Prometheus, which scrapes metrics once every 30 seconds, and consumes a total bandwidth of 13.2 MB/sec, and (b) MaxFreq, which scrapes metrics at the highest frequency, i.e., once every second, and consumes a total bandwidth of 400 MB/sec.

DAG Type	No. of Units	Insert (sec)	Exec (sec)	Export (sec)	E2E (sec)	Throughput (metrics/sec)	CPU (%)	Memory (MB)
Parallel	3	0.25	1.04	0.024	1.31	30553.01	2.39	323.77
	5	0.24	1.27	0.014	1.52	26178.01	2.13	319.24
	7	0.24	1.49	0.013	1.75	22848.31	2.15	309.67
Sequential	3	0.25	0.64	0.068	0.96	41666.66	2.90	316.24
	5	0.26	0.630	0.064	0.95	42025.63	2.86	321.75
	7	0.26	0.63	0.067	0.98	40682.69	2.82	256.03

TABLE III: Effect of DAG size and type on PMF Performance and Resource Overhead

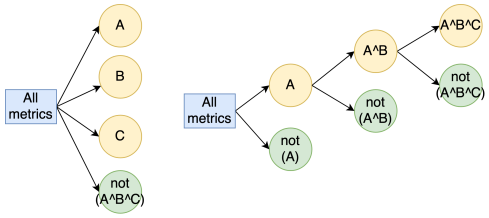

Fig. 15: Effect of No-Op on DAG Types

Parameter	Value
#Metrics per Cluster	[10K−4M]
Size of each Metric	100 Bytes
#Anomalies	50
Metric generation	Per Sec
#Metrics per Anomaly	100
Min Metric Freq	0.001
Max Metric Freq	1
Total Bandwidth	[4 − 400]MB/s

TABLE VI: Simulation Parameters

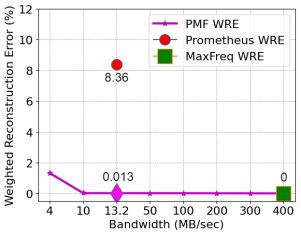

Fig. 16: Effect of Bandwidth on WRE

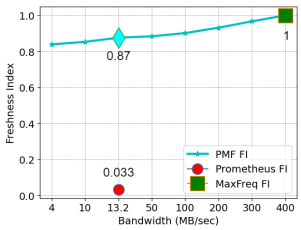

Fig. 17: Effect of Bandwidth on FI

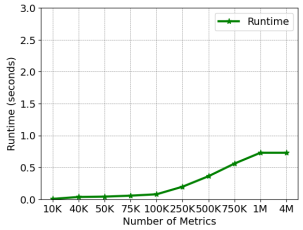

Fig. 18: Optimization Runtime

1) Impact of Dynamic Frequency: We analyse the impact in terms of two major parameters: Weighted Reconstruction Error (discussed later in this subsection) and Freshness Index. We vary the allowable bandwidth for metric collection from 4 MB/sec to 400 MB/sec, calculate the corresponding optimal metric frequencies using our proposed scheme in § IV, and obtain the values of the aforementioned parameters. In this case, we also annotate the values of the parameters for the two baselines (Prometheus and MaxFreq).

a) Weighted Reconstruction Error (WRE): WRE quantifies the average information loss resulting from scraping metrics at a frequency less than the frequency of data generation. It is calculated as the mean average percentage error (MAPE) between the actual time series $x_{ij}(n)$ of metric m_{ij} and the time series $x_{ij}^F(n)$ obtained by sampling m_{ij} at a frequency of F. Mathematically,

$$WRE_F = \sum_{i=1}^{N} \sum_{j=1}^{M_i} z_{ij} \sum_{n=1}^{N} \frac{|x_{ij}(n) - x_{ij}^F(n)|}{x_{ij}^F(n)} \tag{7}$$

Here, z_{ij} denotes the weight of metric m_{ij} calculated as follows: $z_{ij} = \sum_{k=1}^{K} v_k w_{ijk}$. Note that, there are several metrics which are not relevant to any of the anomaly functions, i.e., have weight $w_{ijk} = 0\ \forall k \in [1, K]$. To capture their contribution to the WRE, we assign a minimum weight of 10^{-5} to these metrics.

In Figure 16, we observe that WRE reduces as the bandwidth increases. When the same amount of bandwidth as Prometheus, i.e., 13.2 MB/sec, is made available, our scheme has a lower WRE ($\sim$ 600x reduction) as it collects metrics with higher weights at higher frequencies. When the maximum bandwidth is made available, WRE using our scheme is same as that of MaxFreq. This is intuitive since a larger bandwidth enables the optimization algorithm to assign higher frequencies to more number of metrics.

b) Freshness Index (FI): The variation of FI, as defined in Equation 5, with increasing values of available bandwidth is shown in Figure 17. We observe that FI increases with increasing available bandwidth using our scheme. This can be attributed to the same reason mentioned above. It is noteworthy that for the same bandwidth as that of Prometheus, our scheme results in a higher FI. This shows that our scheme is able to capture fresh values of more important metrics, unlike Prometheus, at the same bandwidth.

2) Optimization Runtime Analysis: Real world multi-cloud deployments can produce millions of metrics per sample. Therefore, the proposed optimization will have to deal with a huge number of variables and constraints. Typically, the complexity of optimization algorithms is polynomial in the number of variables and constraints [59]. However, we were able to linearize the problem and reduce its complexity by majorly using sparse matrices for modelling. To show the impact of the number of metrics on optimization, we vary the total number of metrics from 10K to 4M, assuming metric size of 100B, corresponding bandwidth varies from 1 MB/sec to 400 MB/sec. We study the variation in the solver runtime (Figure 18). We observe that increasing the number of metrics does not result in a significant increase in the runtime, thereby making our scheme scalable. As our maximum optimization runtime is less than a second, this makes the PMF highly adaptable to user's dynamically changing requirements.

VI. Conclusion

In this paper, we propose PMF, a novel metric processing framework for fast, programming and dynamic transformation of metric flows. PMF processes flows using SQL, supporting generic transforms, and enables dynamic metric transformations through reconfiguration. PMF is light-weight, easily programmable, and more resource efficient than OTel. We evaluate PMF extensively for frequency transformation aiming at reducing WAN cost in an edge-cloud environment with minimum impact on any downstream task on the metrics. Our experiments show that dynamic frequency metric collection aids in increasing the efficiency of downstream tasks like anomaly detection. This work is just the start in the era of dynamic programmability for observability starting with the metric pipeline, which we plan to extend to other modalities.

References

[1] "Flexera 2023 State of the Cloud Report." https://info.flexera.com/CM-REPORT-State-of-the-Cloud, Jan. 2023, [Online; accessed 23. Jan. 2024].

[2] "Gartner says cloud will be the centerpiece of new digital experiences," https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences, Jan. 2023, [Online; accessed 19. Jan. 2023].

[3] "From metrics to insight," https://prometheus.io/, Jan. 2024, [Online; accessed 23. Jan. 2024].

[4] "graphite," https://graphiteapp.org/, Jan. 2024, [Online; accessed 23. Jan. 2024].

[5] VictoriaMetrics, "Simple & Reliable Monitoring That Scales," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://victoriametrics.com/

[6] Datadog, "Cloud Monitoring as a Service | Datadog," *Cloud Monitoring as a Service*, Jul. 2016. [Online]. Available: https://www.datadoghq.com

[7] "Modern cloud done right," Mar. 2023, [Online; accessed 25. Apr. 2023]. [Online]. Available: https://www.dynatrace.com

[8] "IBM Instana Observability," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://www.ibm.com/products/instana

[9] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17, 2017.

[10] "Azure Monitor pricing." https://azure.microsoft.com/en-in/pricing/details/monitor/, Jan. 2024, [[I[Online; accessed 23. Jan. 2024].

[11] "Google Cloud's operations suite pricing." https://cloud.google.com/stackdriver/pricing, Jan. 2024, [Online; accessed 23. Jan. 2024].

[12] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020.

[13] V. Jacob, F. Song, A. Stiegler, B. Rad, Y. Diao, and N. Tatbul, "Exathlon: a benchmark for explainable anomaly detection over time series," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2613–2626, 2021.

[14] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2828–2837.

[15] P. Filonov, A. Lavrentyev, and A. Vorontsov, "Multivariate industrial time series with cyber-attack simulation: Fault detection using an lstm-based predictive data model," *arXiv preprint arXiv:1612.06676*, 2016.

[16] L. M. Candanedo and V. Feldheim, "Accurate occupancy detection of an office room from light, temperature, humidity and co2 measurements using statistical learning models," *Energy and Buildings*, vol. 112, pp. 28–39, 2016.

[17] Z. Wang, J. Xue, and Z. Shao, "Heracles: an efficient storage model and data flushing for performance monitoring timeseries," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1080–1092, 2021.

[18] Z. Li, Y. Zhao, N. Botta, C. Ionescu, and X. Hu, "Copod: copula-based outlier detection," in *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 1118–1123.

[19] Y. Zhao, Z. Nasrullah, and Z. Li, "Pyod: A python toolbox for scalable outlier detection," *Journal of Machine Learning Research*, vol. 20, no. 96, pp. 1–7, 2019. [Online]. Available: http://jmlr.org/papers/v20/19-011.html

[20] D. Giouroukis, A. Dadiani, J. Traub, S. Zeuch, and V. Markl, "A survey of adaptive sampling and filtering algorithms for the internet of things," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 27–38. [Online]. Available: https://doi.org/10.1145/3401025.3403777

[21] G. M. Dias, B. Bellalta, and S. Oechsner, "A survey about prediction-based data reduction in wireless sensor networks," *ACM Comput. Surv.*, vol. 49, no. 3, nov 2016. [Online]. Available: https://doi.org/10.1145/2996356

[22] N. Yaseen, B. Arzani, K. Chintalapudi, V. Ranganathan, F. Frujeri, K. Hsieh, D. S. Berger, V. Liu, and S. Kandula, "Towards a cost vs. quality sweet spot for monitoring networks," in *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, 2021, pp. 38–44.

[23] "OpenTelemetry." https://opentelemetry.io/, Jan. 2024, [Online; accessed 23. Jan. 2024].

[24] Prometheus, "Overview | Prometheus," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://prometheus.io/docs/introduction/overview

[25] "Thanos," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://thanos.io

[26] "Cortex," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://cortexmetrics.io

[27] "Grafana: The open observability platform | Grafana Labs," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://grafana.com

[28] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19.

[29] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming*

Languages and Operating Systems, ser. ASPLOS '19.

[30] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19.

[31] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16.

[32] S. Meng, A. K. Iyengar, I. M. Rouvellou, and L. Liu, "Volley: Violation likelihood based state monitoring for datacenters," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*, 2013.

[33] T. Mastelic and I. Brandic, "Data velocity scaling via dynamic monitoring frequency on ultrascale infrastructures," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[34] Y. Meng, Z. Luan, and D. Qian, "Differentiating data collection for cloud environment monitoring," *China Communications*, 2014.

[35] S. Talla, P. Ghare, and K. Singh, "Tbdrs: Threshold based data reduction system for data transmission and computation reduction in wsns," *IEEE Sensors Journal*, vol. 22, no. 11, pp. 10 880–10 889, 2022.

[36] J. Haghighat and W. Hamouda, "A power-efficient scheme for wireless sensor networks based on transmission of good bits and threshold optimization," *IEEE Transactions on Communications*, vol. 64, no. 8, pp. 3520–3533, 2016.

[37] B. Gedik, L. Liu, and P. S. Yu, "Asap: An adaptive sampling approach to data collection in sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 12, pp. 1766–1783, 2007.

[38] H. Harb, A. Makhoul, A. Jaber, R. Tawil, and O. Bazzi, "Adaptive data collection approach based on sets similarity function for saving energy in periodic sensor networks," *Int. J. Inf. Technol. Manage.*, vol. 15, no. 4, p. 346–363, jan 2016. [Online]. Available: https://doi.org/10.1504/IJITM.2016.079603

[39] S. Goel and T. Imielinski, "Prediction-based monitoring in sensor networks: taking lessons from mpeg," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 5, p. 82–98, oct 2001. [Online]. Available: https://doi.org/10.1145/1037107.1037117

[40] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco, "Practical data prediction for real-world wireless sensor networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 8, pp. 2231–2244, 2015.

[41] R.-S. Liu, K.-W. Fan, Z. Zheng, and P. Sinha, "Perpetual and fair data collection for environmental energy harvesting sensor networks," *IEEE/ACM Transactions on Networking*, vol. 19, no. 4, pp. 947–960, 2011.

[42] Y. Zhang, S. He, J. Chen, Y. Sun, and X. S. Shen, "Distributed sampling rate control for rechargeable sensor nodes with limited battery capacity," *IEEE Transactions on Wireless Communications*, vol. 12, no. 6, pp. 3096–3106, 2013.

[43] U. Kulau, J. van Balen, S. Schildt, F. Büsching, and L. Wolf, "Dynamic sample rate adaptation for long-term iot sensing applications," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 271–276.

[44] G. LLC, "Protocol Buffers," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://protobuf.dev/

[45] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, 1989.

[46] T. K. Authors, "Component Specification," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/v1/reference/component-spec

[47] OpenTelemetry, "Transform Processor," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/transformprocessor

[48] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[49] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com

[50] SQLite, "SQLite3 Release," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://www.sqlite.org/releaselog/3_31_1.html

[51] prometheus, "Prometheus Authors," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://prometheus.io/download/

[52] C. Authors, "Cortex," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://github.com/cortexproject/cortex/releases

[53] T. O. Authors, "Install the Collector," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://opentelemetry.io/docs/collector/installation/

[54] G. OPTIMIZATION, "Gurobi Optimization," Jan. 2024, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://www.gurobi.com/downloads/older-versions-gurobi-software/

[55] A. Valialkin, "Measuring vertical scalability for time series databases in Google Cloud," Jan. 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://valyala.medium.com/measuring-vertical-scalability-for-time-series-databases-in-google-cloud-92550d78d8ae

[56] X. Shi, Z. Feng, K. Li, Y. Zhou, H. Jin, Y. Jiang, B. He, Z. Ling, and X. Li, "Byteseries: an in-memory time series database for large-scale monitoring systems," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, 2020.

[57] C. Shen, Q. Ouyang, F. Li, Z. Liu, L. Zhu, Y. Zou, Q. Su, T. Yu, Y. Yi, J. Hu, C. Zheng, B. Wen, H. Zheng, L. Xu,

S. Pan, B. Wu, X. He, Y. Li, J. Tan, S. Wang, D. Pei, W. Zhang, and F. Li, "Lindorm tsdb: A cloud-native time-series database for large-scale monitoring systems," *Proc. VLDB Endow.*, p. 3715–3727, aug 2023.

[58] B. C. Arnold, "Pareto distribution," *Wiley StatsRef: Statistics Reference Online*, pp. 1–10, 2014.

[59] N. Megiddo *et al.*, *On the complexity of linear programming*, 1986.