Intent-Driven Multi-Engine Observability Dataflows For Heterogeneous Geo-Distributed Clouds

Aishwariya Chakraborty*, Anand Eswaran*, Pankaj Thorat*, Mudit Verma*,
Pranjal Gupta, Praveen Jayachandran
IBM Research, India
aishwariya.chakraborty1@ibm.com, anand.eswaran@ibm.com, pankaj.thorat@ibm.com, mudiverm@in.ibm.com,
pranjal.gupta2@ibm.com, praveen.j@in.ibm.com

Abstract*—With the growth of multi-cloud computing across a heterogeneous substrate of public cloud, edge, and on-premise sites, observability has been gaining importance in comprehending the state of availability and performance of large-scale geo-distributed systems. Collecting, processing and analyzing observability data from multiple geo-distributed clouds can be naturally modelled as dataflows comprising chained functions. These observability flows pose a unique set of challenges including (a) keeping cost budgets, resource overheads, network bandwidth consumed and latency low, (b) scaling to a large number of clusters, (c) adapting the volume of observability data to satisfy resource constraints and service level objectives (d) supporting diverse engines per dataflow depending on each processing function of the flow and (e) automating and optimizing placement of observability processing functions including closed-loop orchestration. Towards this end, we propose Octopus, a multi-cloud multi-engine observability processing framework. In Octopus, declarative observability dataflows (DODs) serve as an intent-driven abstraction for site reliability engineers (SREs) to specify self-driven observability dataflows. A dataflow engine in Octopus, then orchestrates these DODs to automatically deploy and self-manage observability dataflows over large fabrics spanning multiple clouds and clusters. Octopus supports a mix of streaming and batch functions and supports pluggable runtime engines, thereby enabling flexible composition of multi-engine observability flows. Our early deployment experience with Octopus is promising. We have successfully deployed production-grade metrics analysis and log processing data flows in an objective-optimized fashion across 1 cloud and 10 edge clusters spanning continents. Our results indicate data volume and WAN bandwidth savings of 2.3x and 56%, respectively, the ability to support auto-scaling of DODs as input load varies, and the ability to flexibly relocate functions across clusters without hurting latency targets.

***Index Terms*—Observability, multi-cloud, dataflow, intent**

I. INTRODUCTION

As cloud computing becomes deeply entrenched as the de-facto approach for delivering enterprise services, there is a natural proliferation of these complex services into geo-distributed multi-cloud settings. Examples of such applications include web services, machine learning training and fine-tuning pipelines, and data integration workflows, where both applications and data span multiple clusters hosted on public, private, and edge clouds. Such geo-distributed multi-cloud applications typically comprise many independently deployable and elastic microservices, making it complicated for a Site Reliability Engineer (SRE) to build a consolidated view of the application health and performance by monitoring these disparate microservices and shared infrastructure services.

*Equal contributors, alphabetically sorted

Observability focuses on providing better visibility and insights into the complex behavior of applications using telemetry data collected at run-time. Observability processing must operate in real-time because the resilience and failure recovery (automated or with manual interventions) of monitored systems rely entirely on its responsiveness to prevent downtime or service degradation. Extending observability to a multi-cloud geo-distributed context necessitates consolidating views across the full stack, namely application, middleware (arising from data and ML frameworks) and infrastructure (arising from software-defined infrastructure), and across multiple clouds. Given the massive volume of observability data comprising of metrics, events, logs, and traces (MELT), the need for quick and continuous insights, fast responsiveness to rapidly changing ambient context, and the inherent diversity in the data types (e.g. time-series data, semi-structured log data, unstructured trace data and events) makes observability processing in multi-cloud settings at scale a challenging big data processing problem.

Observability processing is naturally modelled as a dataflow pipeline, with each transformation step implemented using elementary data transformation functions, which we call *atoms* (described in more detail in Section III). Transforms could be arbitrarily complex and some of them require to run on specialized frameworks depending on their complexity. OpenTelemetry processors are a good example of such functions [1]. Such observability processing poses some unique challenges in a geo-distributed multi-cloud setting.

- *Cost:* Naively shipping raw MELT data across the wide area network (WAN) to a consolidated analysis engine in a central "hub" cluster is prohibitively expensive from data movement cost and WAN bandwidth perspectives.
- *Automation at Geo-Scale:* Deploying and managing the observability processing pipelines across a large and changing number of edge, on-premise, and cloud sites is a first-class problem for SREs.
- *Flexible Adaptation:* Observability data is typically aggregated at source clusters and summaries are periodically shipped to a central location. Unfortunately, this approach is inflexible to dynamic adaptations under re-

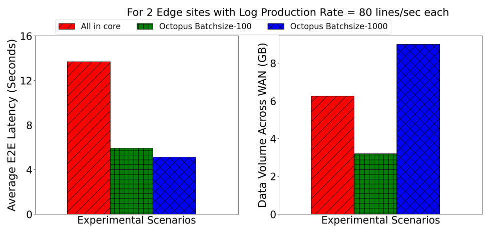

Fig. 1: Naive vs SLO-Aware Placement of Log Analysis Dataflow

source constraints and service level objectives (SLOs).

- *Objective-Driven Placement Optimization at Scale:* It would be beneficial to opportunistically place observability functions factoring in target response deadlines for closed loop dataflows and the need to utilize resources across multiple clusters frugally.

Figure 1 demonstrates how addressing these challenges can significantly reduce WAN data volume (translates into the cost for egress bandwidth) as well as end-to-end latency, in contrast to naively shipping all telemetry data to a centralized site ('All in Core') for a production log analysis dataflow (explained in Section V-A). This dataflow is deployed in a multi-cloud topology spanning many edge clusters, where logs and metrics are continuously collected, and a central cloud cluster, where the results of log and metric analysis from all edge clusters are available for analysis by SREs. Using techniques such as intelligent relocation of functions and transparent injection of lossless data compression and decompression stages at cross-WAN segments, we achieve a compression ratio of 2.3x and bandwidth cost savings of 56.8% without violating log delivery latency objectives (set to 6 seconds in our experiment). In fact, the suitable batch size selection for compression reduces the end-to-end latency by a factor of 2, with a small compute overhead at the edge. We also show the results for a larger batch size of 1000 where the latency objective is violated, highlighting the need for intelligent control and orchestration of the dataflow for optimal results.

Considering the above requirements, in this paper, we present Octopus, a multi-cloud multi-engine observability processing framework with the following novel elements.

1) We introduce the abstraction of *declarative observability dataflows (DODs)*, which allows SREs to define intent-driven specifications of observability processing dataflows in a multi-cloud setting. DOD specifies the observability functions or *atoms* (e.g., metrics filtering), their runtime requirements, and logical dependencies (e.g., metrics filtering must precede metrics aggregation) encoded as a directed acyclic graph *DAG*. While the observability functions may be arbitrarily complex and remain a black-box, having a specification of it as an atom permits us to orchestrate these functions, and place them at edge or core clusters, significantly reducing the burden on SREs to manage observability dataflows at scale. Besides functional specifications, the intent declaration also supports the specification of non-functional concerns such as objectives (e.g., minimize WAN bandwidth consumed) and constraints (e.g., do not utilize more than 5% memory on the source cluster) associated with the DOD.
2) We design and implement a multi-cloud observability dataflow engine that is responsible for realizing functioning dataflows from a *DOD* specification. Given a DOD and a set of source clusters from where observability data needs to be collected, Octopus our dataflow engine, is responsible for planning and placing the functions factoring in the specified runtime requirements, objectives, and constraints. It subsequently deploys the plan across a geo-distributed substrate of clusters, enabling automated observability processing and collection. Finally, Octopus takes responsibility for continuous monitoring and closed-loop orchestration of the deployed functions.

We evaluate the efficacy of Octopus using two real-world geo-distributed observability scenarios — log analysis and metrics processing, which use different run-time engines in a multi-cloud context. Our results demonstrate the ability to automatically deploy DODs across a large number of clouds, optimize pipelines for target objectives, support quality-of-service (QoS) differentiation, and provide adaptive closed-loop control at scale.

II. Background and Use-cases

Observability in large-scale systems focuses on providing comprehensive visibility and understanding into the system's internal dynamics and behavior. This extends beyond visibility (monitoring), to extract meaningful insights from collected data and enabling the early detection of problems thereby averting service outages, SLO violations, behavioral anomalies and security attacks. The primary observability data types are:

- ***Metrics*** provide numerical measurements of system performance, such as CPU usage, memory consumption, response time, throughput, and other user-defined KPIs.
- ***Events*** are real-time triggers generated when predefined thresholds are breached and/or when specific conditions are met, ensuring timely response to critical incidents.
- ***Logs*** offer detailed records of the execution of processes, along with actions performed, playing a pivotal role in root cause analysis, troubleshooting, and debugging.
- ***Traces*** capture the control flow of requests and interactions across multiple components of a system, aiding in the comprehension of end-to-end trajectory of requests.

In today's cloud service landscape, a wide array of site-scoped monitoring tools encompassing data collection, storage, querying, and visualization, are available, which when coupled with analysis provide observability. Examples include *Fluentd, ElasticSearch, Kibana, Prometheus, Thanos, Vector, Loki, Grafana* and *Jaeger*.

A. Observability in Multi-Cloud Settings

In recent years, the landscape of enterprise software architectures has been significantly reshaped by proliferation of microservice-based designs that span across multi-cloud and multi-cluster environments. Observability dataflows in such heterogenous settings result in varied needs such as type of telemetry data processed, type of dataflow (batch vs streaming) and reaction time for closed-loop remediation. Further, unlike observability in large datacenter cloud settings where near-infinite bisection bandwidth, CPU and storage resources are generally taken for granted, multi-cluster/edge settings may be constrained. This is exacerbated by the fact that observability flows are not the primary workload and hence function placement of these flows must favor solutions that are both (resource) interference-free and economical. Thus, observability dataflows need to dynamically balance the need for in-situ processing vs remote data processing.

B. Observability Use-Case Domains

Following are some of the domains where multi-cloud observability dataflows are gaining prominence.

IT and Telecom Edge Clouds: As edge computing takes off across sectors such as factories, hospitals, retail, automobiles, and drones, SREs aim for single-pane-of-glass visibility into all clusters' health and performance. This involves analytical processing of MELT data for shipping events periodically to a central hub, which then aggregates, analyzes and mines observability data across clusters in the same region or with similar functions.

Data-preprocessing for AI workloads: A global enterprise often needs to collate training/fine-tuning data from diverse sources for AI model customization. Data scientists spend significant effort curating this data, often near the source, before aggregating it centrally for training. Here observability flows need to monitor not only the underlying infrastructure and storage but also data and model-related metrics from AI frameworks across the data science lifecycle.

BizOps Systems: BizOps refers to a suite of workflows designed to derive insights from both IT systems as well as business process-related metrics, such as number of orders processed per hour. Observability dataflows in this setting involve correlating application health and performance metrics with business KPIs, via data collection, enrichment, business metric calculation, visualization, and alerting.

III. Design

We start by outlining the concepts and abstractions in Octopus in Section III-A, that enable SREs to specify the observability dataflows using a declarative specification. In Section III-B, we discuss the workflow for transforming the intent dataflow as it is optimized (in a query optimization sense) for stated non-functional objectives, partitioned, and placed across the geo-distributed substrate without violating stated constraints (e.g. geography, function-specific capability requirements). Finally in Section III-C, we discuss the overall architecture of the Octopus dataflow engine and how the Octopus components are organized. Henceforth, we use the term *cluster* to denote any of a wide range of compute environments such as a Kubernetes cluster, a VPC in a cloud, a virtual machine, or a bare metal server in any public, private or edge cloud.

A. Concepts and Abstractions

Declarative Observability Dataflow (DOD) is the key concept within Octopus. A DOD is an intent object that allows SREs to declaratively specify the observability dataflow as an intent-driven DAG of functions akin to a database logical query plan spanning geo-distributed scope. It specifies the following aspects:

- *Atoms:* The set of explicitly-specified logical stages that constitute vertices of the DOD, wherein each stage is an observability data processing function (such as filtering, deduplication, aggregation, and enrichment). Each stage may be executed in parallel across multiple clusters to process input data streams or to process data persisted locally at that cluster. An atom explicitly specifies whether it runs on *all* clusters that match (or do not match) a given label or on *any* one of them. Each atom may map to a different run-time engine depending on its functionality. For example, some atoms can map to Kubernetes microservices (e.g., filtering, aggregation stages) and yet others could be Spark or Ray jobs (e.g., data-parallel analysis functions), all within the same dataflow. Atoms may also run on FaaS frameworks such as OpenFaaS.
- *Edges:* Edges denote the dependencies between atoms and depict the producer-consumer relationship between them. Edges can be of type *local* or *global*, depending on whether producers ship data only to consumers within the local cluster or not. Each edge has a *mode* parameter that denotes the mechanism used for transferring data from the source atom to the destination atom along that edge. For instance, one edge might stream observability data over Kafka, whereas another edge might invoke an API call to directly push data in micro-batches. Octopus seamlessly maps edges to appropriate data transfer mechanisms to automate the orchestration of dataflows across clusters.
- *Constraints:* Constraints are specified as hard or soft label affinities or anti-affinities on atoms. These are typically used to specify capability requirements such as COS (cloud object store) storage, or geographical constraints related to data compliance. Since labels correspond to arbitrary opaque (key, values) and support composability via conjunctive, disjunctive or negation predicates, they can be combined expressively to specify complex constraints for the placement of atoms.
- *Objectives:* Objectives are used to specify end-to-end goals related to performance and cost. Examples of objectives could be maximization of aggregated dataflow throughput across consumer sites or minimization of total \$ cost of all consumed resources or a (weighted) combination of both. Objectives are used as the optimization criteria by the planning phase of Octopus.

Note that the DOD only represents a logical dataflow and thus is independent of the actual multi-cluster environment where observability data is collected and processed with labels

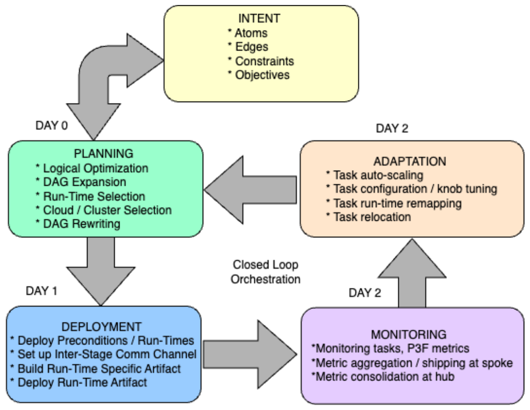
INTENT
* Atoms
* Edges
* Constraints
* Objectives
DAY 0
PLANNING
* Logical Optimization
* DAG Expansion
* Run-Time Selection
* Cloud / Cluster Selection
* DAG Rewriting
DAY 2
ADAPTATION
* Task auto-scaling
* Task configuration / knob tuning
* Task run-time remapping
* Task relocation
Closed Loop Orchestration
DAY 1
DAY 2
DEPLOYMENT
* Deploy Preconditions / Run-Times
* Set up Inter-Stage Comm Channel
* Build Run-Time Specific Artifact
* Deploy Run-Time Artifact
MONITORING
*Monitoring tasks, P3F metrics
*Metric aggregation / shipping at spoke
*Metric consolidation at hub

Fig. 2: Octopus Closed-Loop Orchestration

serving as the mechanism to signal geo-location affinities for atoms. Based on the specification, Octopus's dataflow engine creates a physical plan DAG that includes the specific placement of atoms on the different clusters to realize the functionality of the logical dataflow. Octopus continuously reconciles the current dataflow system state with the intent specification to detect changes due to transient faults, lifecycle events, changing load, and competing dataflows using closed-loop monitoring and orchestration, as described in detail next.

B. Octopus Orchestration

Given a DOD intent specification, the Octopus dataflow engine is responsible for transforming it into a self-managed physical plan DAG spanning multiple geo-locations and clouds. This happens in stages as shown in Figure 2.

1) **Planning**: The planning phase is responsible for generating a physical plan DAG from the DOD specification. This involves the following steps: (i) Expanding the logical dataflow and converting it into a physical plan DAG, with an instance of an atom for each cluster, which we call a *task*. Tasks are elastic, i.e., could comprise of multiple replicas at a given cluster. Each stage of the physical plan DAG maps to the instantiation of an atom. Even if logical dataflows are themselves only linear chains, the combination of stages of type *all*, global edges, and multiple clusters expand to physical plans that are large and complex DAGs. This process is shown with an example in Figure 3 and we describe in further detail below; (ii) Selecting the best physical function(s) [algorithmic block, run-time] for implementing logical function; (iii) Creating a set of Partitioned Physical Plan Fragments (P3Fs) based on deciding how to group physical functions together and simultaneously deciding on which cloud or cluster to place each P3F in; and (iv) Modifying the P3Fs by injecting data copy or compression physical functions at the partition boundaries for efficient data transfer over WAN.
Consider the example in Figure 3. The input DOD specification and the multi-cluster topology inputs to Octopus are shown on the left. Function F1 needs to run on all edge clusters E1, E2 and E3 (denoted by a diamond). Function F2, F3 and F4 need to run on all clusters where F1 is also deployed (denoted by 'All:Prev' for brevity). Function F5 needs to run on any one of the cloud clusters C1 or C2 (denoted by a triangle). The expanded physical plan DAG is shown with task instances created for each atom or function in the DOD. Finally, the P3Fs pertaining to each cluster is shown and this is handed to the controller in each cluster to deploy and orchestrate. This includes the decision of placing F5 in C1 and injecting a data compression stage at each of the edge clusters. We have omitted to show all the global edges in the P3Fs to keep the picture simple.
2) **Orchestration**: The generated P3Fs are at the cluster-scope granularity. They are submitted to each target cluster to realize a portion of the end-to-end geo-distributed dataflow. Each target cluster has an Octopus component that receives the P3F plan object and stand up a functioning streaming pipeline using a workflow engine. This workflow engine is run-time aware and is thereby able to instantiate each stage in a run-time specific manner, factoring in engine-specific dependencies before launching a stage. This includes coalescing / fusing functions of successive stages that map to the same run-time into a run-time specific job, e.g. a Spark job. Buffered communication channels between successive stages are also automatically synthesized if producer and consumer functions span run-time boundaries[1].
3) **Monitoring**: The deployed dataflow is subsequently monitored at multiple granularities - at the task level, at the atom level, at the run-time coalesced task-group level, and at the P3F level. The monitored metrics include resource consumption statistics (CPU, memory, storage, network) as well as application SLO metrics, e.g., tuples processed per second. This information is aggregated at the Octopus control plane and used for adaptation and closed loop orchestration.
4) **Adaptation**: The monitoring statistics collected are used for actions such as (i) elastic scaling of the task replicas to cope with load, (ii) switching physical function implementations when measured run-time profiles vary from planning time assumptions, and (iii) relocating functions across clusters (say from edge to cloud or vice versa) without violating the pipeline dependency structure to trade-off network bandwidth for CPU resources. This closed loop assurance runs continuously over the lifetime of the DOD intent specification.

C. High Level Architecture

Octopus is itself organized in a hub and spoke architecture. Broadly, the Octopus hub engine, running at a centralized cloud location, is responsible for the planning steps and for inter-site adaptation actions (e.g., cross-cluster function

[1]Reliable ordered communication between successive tasks within the same run-time could either be the responsibility of that run-time or a preference can be explicitly specified in the DOD specification.

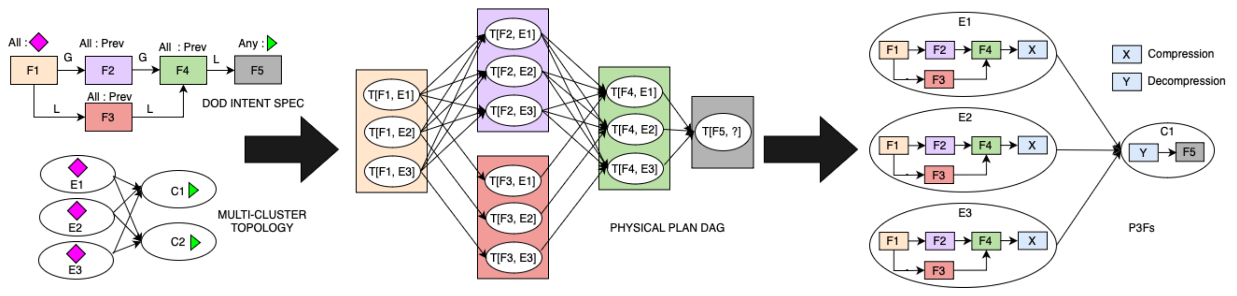

Fig. 3: Converting a DOD Specification to a Physical Plan DAG and then to P3Fs

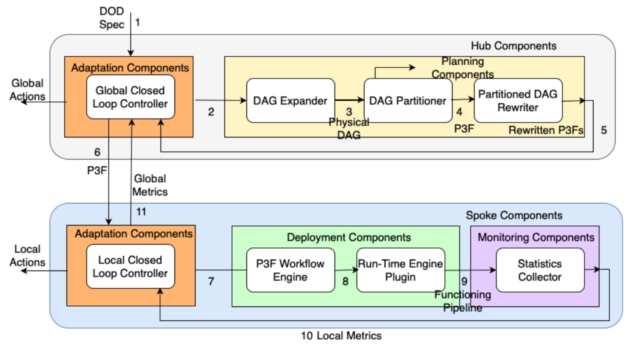

Fig. 4: Octopus Componentry

relocation) listed above. In contrast, the Octopus spoke engine, running at each cluster, is responsible for the deployment of P3Fs, local monitoring, and local adaptation actions (e.g., load-aware scaling). Below we discuss the hub and spoke componentry of Octopus. P3Fs are the primary objects of interchange between the hub and spoke engines. A consolidated view of the Octopus componentry is depicted in Figure 4.

1) Hub Componentry: The hub components below run only at the hub or control cluster.

- **Global Closed Loop Controller:** It is responsible for all lifecycle events related to the creation, modification and deletion of a DOD by the user, invoking the planning components, cross-site adaptation actions, as well as closed-loop assurance actions to meet end goals.
- **DAG Expander:** It is responsible for accepting the logical dataflow and expanding it into a physical plan DAG based on the multi-cloud cluster topology while factoring in [any/all] atom constraints and [local/global] edge constraints.
- **DAG Partitioner:** It is responsible for decomposing the physical DAG into P3Fs. We use a deterministic "prune and score" heuristic that prunes clusters or clouds that do not match constraints and then scores the survivors based on successive objective criteria such as WAN movement cost and load imbalance. Multiple objectives can be combined flexibly in the scoring function. The pseudo-code for the approach is provided in Algorithm 1. Besides new workload allocation, the placement heuristic is also automatically triggered during large workload shifts, cluster outages etc, thereby supporting geo-distributed dynamic adaptation.
- **Partitioned DAG Rewriter:** It is responsible for modifying each P3F by transparently injecting functions responsible for modality-specific data compression and/or efficient data copy across clusters.

Algorithm 1 Prune and Score Pseudocode

```
Input:
$candidateClusters$        ▷ List of candidate clusters
$clusterResources$         ▷ List of resources of candidate clusters
$taskList$                 ▷ List of tasks
$taskRequirements$         ▷ List of requirements of the tasks
Output:
$taskAssignments$          ▷ Assignment of Clusters to Tasks
 1: procedure
 2:     $taskAssignments = []$
 3:     $eligibleClusters = []$
 4:     Sort $taskList$ by size based on optimization criteria
 5:     for all $task$ in $taskList$ do
 6:         $t = taskResourceRequirement[task]$
 7:         for all $cluster$ in $candidateClusters$ do
 8:             if $cluster.satisfiesHardConstraints()$ and $clusterResources[cluster] \geq t$ then
 9:                 $eligibleClusters.Add(cluster)$
10:         if $isEmpty(eligibleClusters)$ then
11:             break                      ▷ No cluster has sufficient resources
12:         $c = highestRanked(eligibleClusters, task)$   ▷ Placement criteria
13:         Assign $task$ to $c$, i.e., $taskAssignments[task] = c$
14:         $clusterResources[c] = clusterResources[c] - t$
15:     Return $taskAssignments$
```

2) Spoke Componentry: The spoke components run at each workload cluster that makes up the multi-cloud substrate.

- **Local Closed Loop Controller:** It is responsible for invoking the deployment components to deploy a P3F and for the enforcement of cluster-local closed-loop control. Supported task adaptation actions currently include auto-scaling and dynamic tuning of task configuration knobs.
- **P3F Workflow Engine:** It is responsible for receiving and parsing the P3F intent objects. Each stage in the P3F is encoded with run-time engine-specific information. The P3F engine uses the run-time specific plugin to (i) ensure that the run-time engine and associated dependencies are launched and ready, and (ii) create a deployment artifact corresponding to a task of that stage which can then be launched locally in the workload cluster.
- **Run-Time Engine Plugin:** A P3F comprises stages that could map to different run-times. Octopus supports a modular run-time plugin framework within the P3F work-

flow engine as a way for creating a wide and extensible range of deployment artifacts.

- **Statistics Collector:** Once a P3F is deployed, metrics for DAG monitoring are continuously collected. These statistics are normalized, aggregated, and shipped to the hub for global adaptation.

IV. Implementation

We implement all Octopus components in a cloud-native fashion leveraging the artifacts from the Kubernetes ecosystem. In this section, we describe the key building blocks and technology choices for the system.

A. KubeStellar

KubeStellar [2] is an open-source multi-cluster configuration management system that can manage and synchronize Kubernetes objects between different clusters. KubeStellar is organized as a set of workspaces/namespaces responsible for state-keeping of resources such as workloads and inventory, along with a set of related controllers that are maintained at the hub database. Syncers run on target (workload) clusters and are responsible for synchronizing objects from the hub workspaces to namespaces in target clusters, while also handling conflicts arising from concurrent updates in different clusters. Within Octopus, intent at various levels of abstraction is modeled using Kubernetes custom resource definitions (CRDs), which we describe next. Kubestellar acts as the geo-distributed control plane within Octopus to synchronize custom resource objects across hub and spoke locations.

B. DOD Custom Resource Definition

The *DOD* CRD encapsulates the functional specification of the observability dataflow, its constraints and objectives. This CRD defines the DAG as shown in Figure 5, that outlines the data flow of processing stages. It specifies the name, functionality, runtime, type of job, and artifact of each atom and the dependencies or edges between the atoms. The *inputKVs* and *outputKVs* offer a mechanism for passing in and out, respectively, arbitrary key-value pairs such as environmental variables, data locations, protocols, and streaming schemas. The field *constraints* specifies restrictions on placement of the atom at clusters such as *candidateSites* for specifying eligible candidate clusters and *labelContraints* to specify affinity or anti-affinity associations between functions. The *objectives* element quantifies performance objectives such as throughput in streaming scenarios or job execution time in batch settings, and cost goals such as budgetary restrictions. *compoundObjectives* enable combining performance and cost objectives in richer ways. Finally, the *status* field allows the furnishing of profiling statistics which provide insights into function execution and resource utilization. These fields are constantly reconciled to reflect the current state and run-time profile of the execution.

C. Controllers

The global closed loop controller is implemented using the Kubernetes Operator framework in Golang comprising of

```
apiVersion: edge.operator.com/v1alpha1
kind: Atomgraph
metadata:
 name: # pipeline name
 atoms:
 - function: # function class (filter, etc.)
   name: # instance name
   runtime: # runtime needed (k8s, kubeflow,  etc.)
   root: # if first node in pipeline
   jobType: # stream or batch job
   encoderCompatible: # compression eligible
   decoderCompatible: # decompression eligible
   artifact: # image or binary loc
   inputKVs: # List of runtime {k,v} pairs
   outputKVs: # List of runtime {k,v} pairs
   constraints:
    candidateSites: # locations this should run on
   objectives:
     performanceObjectives: # e.g. desired
    throughput
     costObjectives: # e.g. desired cost
     compoundObjectives:
   command: # runtime execution command
     ..
 edges:
 - name: # edge name
   sourceAtom: # name of the source atom
   destAtom: # name of the dest atom
   mode: # how data flows (Kafka, API)
   ..
 status:
   resourceConsumtion: # resources used
   kpis: # throughput, bandwidth, cost
```

Fig. 5: DoD Custom Resource (YAML format)

roughly 5900 lines of code. It parses the CR submitted to Kubestellar and executes the planning stages in order (also written in Golang in 1600 lines of code), sequentially invoking the *DAG Expander*, *DAG Partitioner*, and *Partitioned DAG Rewriter*. The local closed loop controller is also implemented as a Python-based component comprising 2566 lines of code. It further partitions the P3F CR received from the global controller into sets of contiguous tasks mapping to the same run-time (called *islands*) by using a simple breath-first-search traversal. Thereafter, it launches each island pipeline on the worker site by submitting the artifact to the chosen run-time. If the P3F CR is deleted, the controller performs the necessary cleanup by deleting the pipeline and associated artifacts. The controllers use the status field of the CR object to obtain information regarding the pipeline's health and performance and take requisite actions when significant events occur. The overall workflow of Octopus is depicted in Figure 6.

V. Evaluation

In this section, we evaluate Octopus's performance in two production-grade multi-cloud observability dataflows: one for log analysis and another for metrics processing. Both pipelines support operator relocation. Additionally, the log analysis pipeline supports operator injection, while the metric processing pipeline supports QoS differentiation. Further details are provided in the following subsections.

Our experimental setup is comprised of 10 geographically distributed 3 nodes Kubernetes clusters, 1 cloud cluster with 6

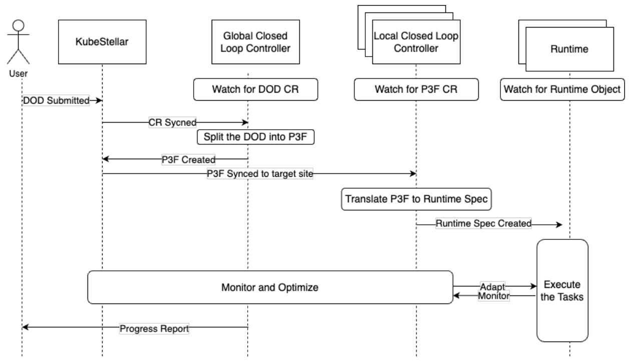

Fig. 6: End-to-end workflow of an observability dataflow execution

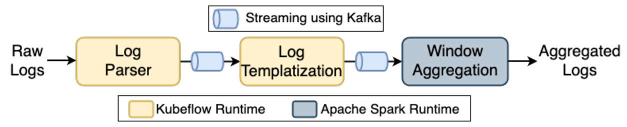

Fig. 7: Logical Dataflow for a Log Analysis Pipeline

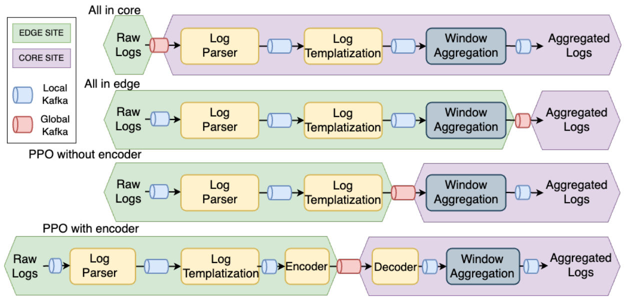

Fig. 8: Deployment Scenarios of Log Analysis Pipeline

nodes and 1 control cluster, each having 16 cores and 32 GB of memory. All workloads are containerized and run within these Kubernetes environments. The control cluster, located in San Jose, acts as the hub cluster where the hub components of Octopus execute. 10 edge clusters, 5 in London and 5 in Tokyo, have the sources of observability data, for both logs and metrics. The cluster located in Dallas acts as the cloud (core) cluster where an SRE expects the results of log / metric analysis to be available. The edge clusters and the cloud cluster together comprise the workload clusters for our experiments. Each workload cluster has appropriate runtime engines (Apache Spark, Kubeflow etc) preinstalled under Octopus local controllers at each location.

A. *Log Analysis Dataflow*

Figure 7 depicts the logical dataflow for the log analysis pipeline, which processes the logs into a structured representation for downstream use cases such as log anomaly detection. This dataflow consists of the following stages: (a) *Log Parser*, which parses and identifies the incoming logs and converts them into a key-value structured format, (b) *Log Templatization*, which extracts dynamic (*variable*) and constant (*template*) part of log lines, which are not parsed by the Log Parser, and (c) *Window Aggregation* which computes a count-based distribution of log templates over a time window. These template-count distributions are then used to determine the occurrence of an anomaly in the given time window.

In our experimental setup, logs are produced at each edge cluster at the rate of 80 log lines per second for a duration of 60 hours. The first two stages of the cluster local P3F run as a Kubeflow pipeline where stages are run as pods, and the last stage runs on Apache Spark. Between each pair of consecutive stages, data communication takes place using Kafka. Additionally, to support the dynamic adaptation of the pipeline based on the load, local closed loop controller uses Keda with Apache Kafka scaler [3] that scales individual stages of the pipeline based on the offset lag between producers and consumers. For our experiments, we select the range of the number of replicas of each stage as $[1, 5]$, a polling interval of 30 seconds, and the threshold value of offset lag used for scaling as 100 log lines. These parameters are tunable by the local closed loop controller. We consider the following deployment scenarios by Octopus as shown in Figure 8.

1) *All in cloud* - Raw logs from the edge clusters are shipped to the cloud cluster with all processing stages being scheduled in the cloud.
2) *All in edge* - All stages of log processing are scheduled at the edge clusters, from where aggregated information in a structured format is shipped to the cloud cluster.
3) *Octopus Optimized Plan (OOP)* - We consider two variants of this scenario:
   (a) *OOP without encoding* - Log parser and templatization stages are scheduled in the edge clusters, while the window aggregation stage is located in the cloud cluster.
   (b) *OOP with encoding* - Same as (a) with encoder operator (parameterized by batch size) injected dynamically by Octopus after log templatization in each edge cluster, and corresponding decoder operator injected before window aggregation in the cloud cluster. For evaluation, we consider encoder batch sizes of 100, 500, and 1000 log lines per batch.

Octopus has the flexibility to execute all stages at the edge clusters or all stages at the cloud cluster, if the constraints and edge resources allow it. In the log analysis pipeline, the window-aggregator stage is expected to aggregate logs collected from multiple edge sites, hence necessitating that the window aggregator must be at a cloud. Therefore, we do not henceforth consider the ’All in Edge’ scenario in our experiments. Additionally, we identify the 3.a and 3.b scenarios with Octopus, as they exemplify the benefits of our design in reducing the volume of data transferred over WAN.

In our experiments, we measure the volume of log data shipped across the WAN (i.e., from the edge clusters to the cloud cluster) and the average end-to-end latency of processing a log line in case of each of the aforementioned three (1, 3.a and 3.b) scenarios. We also study the effect of the variation of load on the performance of the pipeline managed using

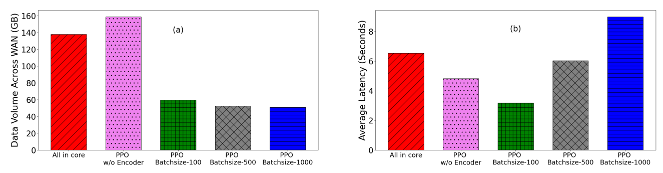

Fig. 9: (a) Volume of data shipped over WAN, (b) End-to-end processing latency per log line

Octopus. We discuss our findings in the following subsections.

1) Data Movement Reduction: Figure 9(a) depicts that the volume of data shipped across the WAN increases by 13.1% using 'OOP without encoder', compared to the 'All in core' scenario. This is due to the fact that the templatized logs add additional metadata to the raw logs, increasing the data volume further, compared to raw logs shipped across the WAN in the 'All in core' scenario. However, Octopus has the ability to transparently inject lossless log encoder and decoder stages at the edge-to-cloud boundary, enabling a reduction in the volume of data shipped across the WAN by 56.7% compared to the 'All in core' scenario. Additionally, Figure 9(a) shows that, with increasing batch size of compression, the volume of shipped data reduces even further. This, however, has an effect of increasing the end-to-end latency, which we discuss next.

2) End-to-End Latency: In Figure 9(b), we observe that the average end-to-end latency of processing each log line reduces by 26.3% in the 'OOP without encoder' scenario, compared to the 'All in core' scenario. In case of the former, the pipeline runs partially in parallel in the edge clusters which reduces the latency, whereas, in the latter, the raw logs generated in the edge clusters are shipped to and processed in the cloud cluster, which in turn increases the processing latency. However, the reduction in latency in case of 'OOP without encoder' is limited since it is overpowered by the latency of shipping a high volume of data across the WAN. This data transfer latency is reduced considerably in case of 'OOP with encoder' for smaller batch sizes, as evident in Figure 9(b). With increasing batch sizes, the encoder waits to receive a higher number of log lines per batch before compressing the data, which causes a significant increase in the end-to-end latency. It is noteworthy that, in the case of 'OOP with encoder', the increase in batch size results in the reduction in data volume shipped across the WAN, with a corresponding increase in the end-to-end latency. Hence, the batch size needs to be chosen judiciously such that the data savings are maximized while adhering to the end-to-end latency targets of the pipeline.

3) Scalability: We evaluate the performance of load-based auto-scaling implemented using the Octopus in the log analysis pipeline. Each enterprise application replica for this study produces approximately 20 log lines per second. We increase the number of instances of the application on each edge site from 1 to 4 and reduce it back to 1 (shown as the log production rate in the 1st plot of Figure 10) and study its impact on the geo-distributed pipeline deployment and the end-to-end log processing latency. We observe that the increase

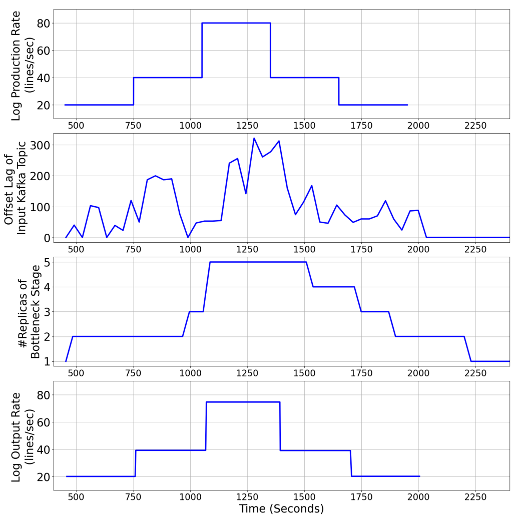

Fig. 10: Auto-Scaling of log analysis pipeline using Octopus

in log production rate results in increasing offset lag (due to queue build-up) in the input Kafka topic of the log parser stage which is identified as the bottleneck stage in the pipeline (2nd plot). Due to the increase in the load, the local closed loop controller spawns replicas of the log parser stage (3rd plot) to reduce the offset lag, which in turn eliminates queue build-up (4th plot). When the load is reduced, there is a commensurate decrease in the number of replicas of the parser, which in turn brings down the output rate of the pipeline. Auto-scaling proactively scales to eliminate bottlenecks of the end-to-end pipeline no matter which cluster the bottleneck stage(s) run on. This demonstrates the ability of Octopus to scale the log analysis pipeline with varying loads at the edge.

B. Metrics Processing Dataflow

Figure 11 illustrates the metric processing which gathers metrics from various applications on the edge clusters, feeds the data through multiple functions that perform filtering, enrichment, and aggregation, and then transports transformed data to the cloud cluster for further processing and storage. These functions are processors provided by OpenTelemetry [1] and multiple functions can be deployed within a single Collector pod that acts as their run-time. Octopus deploys the necessary functions in a cluster (a P3F) using a config map to run them within OpenTelemetry's Collector pod. The Collector functions as a containerized, vendor-neutral proxy, adept at receiving, processing, and exporting metric data to an HTTP endpoint.

To demonstrate ability to support QoS differentiation, the applications generating metrics are classified into three service classes: *gold*, *silver*, and *bronze*. The gold services have the most stringent metric collection needs and aren't amenable to adaptation when compared to silver and bronze services, with the bronze services exhibiting the greatest flexibility for metric dataflow adaptation. We utilize the following functions in our metric dataflow: (a) *Filtering* selects pertinent metrics for the class of service, discarding superfluous metrics. It can operate

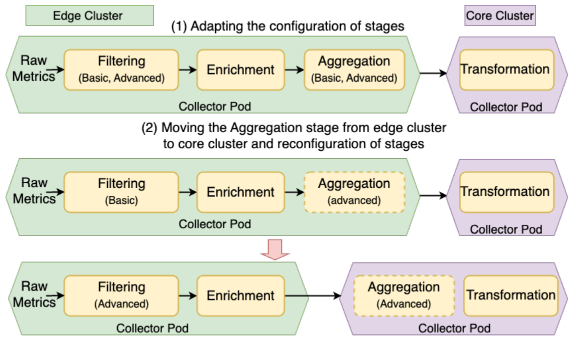

Fig. 11: Deployment Scenarios of Metric Processing Pipeline

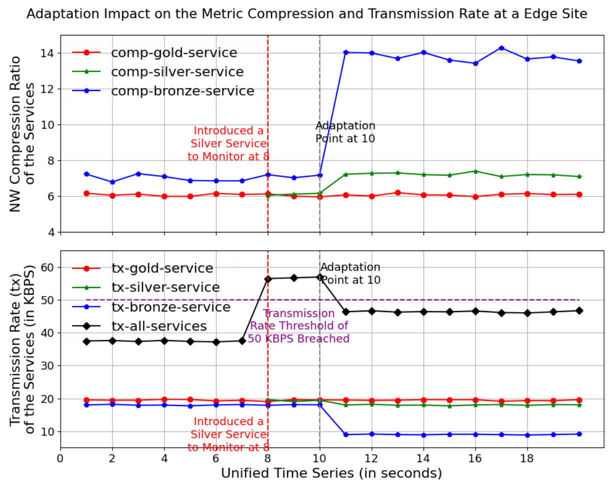

Fig. 12: Adaptation Impact on the Compression Ratio and Transmission Rate

in basic or advanced mode. In basic mode, only metrics pertaining to an application of the class are permitted and all infrastructure metrics are omitted. In advanced mode, metrics pertaining to containers that aren't in 'running' state are also omitted; (b) *Enrichment* augments metrics with metadata pertaining to the edge cluster; (c) *Aggregation* computes the sum or mean of metrics based on designated labels such as namespace, pod, and cluster location; and (d) *Transformation* modifies the metric's value, typically for normalization or comprehension purposes. In our example, it translates metric units from seconds to milliseconds.

The metric dataflow spans many edge clusters and a cloud clusters. We evaluate Octopus under two scenarios as depicted in Figure 11. In the first scenario, we evaluate how the local closed loop controller detects and adapts to a violation of a constraint on the total observability data transmission rate between edge and cloud clusters. In the second scenario, we assess how the global closed loop controller avoids interference of observability workloads in resource-constrained edge environments by relocating stages from the edge to the cloud cluster, upon detecting CPU utilization at the edge exceeding a threshold. In both experiments, the x-axis (Unified Time Series) represents a consistent timeline, providing a chronological sequence of events or data.

1) Local Closed Loop Controller Adaptation: In this experiment, we consider a constraint of 50 kbps on the total observability data transmitted between edge and cloud clusters. At the start of the experiment, one gold and one bronze class application is running in the edge cluster. At time=8, a silver class application is launched, which causes the overall data bandwidth to violate the constraint as shown in Figure 12.

Initially, Octopus deploys two dataflows, for the gold and bronze class applications respectively. The dataflow for the gold application includes basic filtering, enrichment and aggregation functions at the edge cluster and a transformation function at the cloud cluster. The dataflow for the bronze application comprises advanced filtering, enrichment, and basic aggregation stages at the edge cluster and transformation stage at the cloud cluster. Upon introducing a silver class application, a metric dataflow with basic filtering and enrichment is deployed. This causes a violation of the overall transmission rate constraint, prompting Octopus's local closed loop controller to lower the transmission rate below the 50 kbps threshold. Based on the knowledge of the pipeline stage and its impact on the transmission rate, the controller modifies the silver pipeline's filtering stage from basic to advanced and incorporates an additional basic aggregation stage between the enrichment and transformation stages. Likewise, the aggregation phase for the bronze service transitions from basic to advanced. Note that the data transmission rate of the bronze application is reduced the most, while there is no impact on the data collection and processing of the gold application. We also plot the compression ratio of the collector pod for each of the applications, defined as the ratio of the output rate to the input rate of that pod. The compression ratio for the bronze application increases from 7x to 13.8x, and for the silver application, it goes from 6x to 7.2x. A marked alteration in the compression ratio showcases the adaptive responsiveness of Octopus's local closed loop controller when faced with a load-induced constraint violation.

2) Global Closed Loop Adaptation: In this scenario, we deploy metric dataflows (shown in part (2) of Figure 11) originating at the different edge clusters and exporting metrics to a common cloud cluster. We consider a single CPU core utilization constraint of 15% for the functions executing at the edge cluster. To emulate higher CPU load, we progressively reduce the scraping interval of metrics from the 20s to 10s, 5s, and 2s at times t=1,2,3, and 4, respectively as shown in Figure 13. This increases the rate of metric data produced and also increases the processing needed at the edge cluster. At t=4, CPU utilization of the functions deployed on the edge cluster exceeds the constraint of 15% as visible with the green line. Octopus's global closed loop controller relocates the CPU-intensive aggregation phase from the edge to the cloud cluster, effectively reducing the CPU utilization of the edge cluster's collector pod to below the 15% constraint as indicated by the green line at t=6. These experiments demonstrate how, under dynamic conditions, Octopus is able to make both local

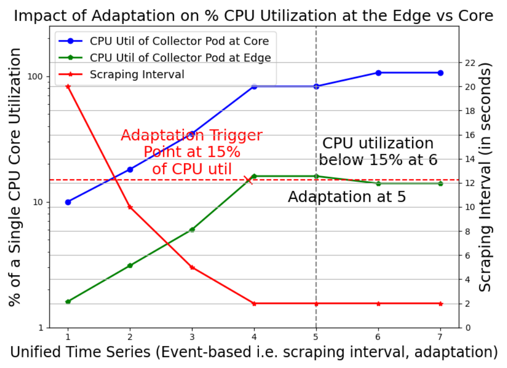

Fig. 13: Impact of Adaptation on % CPU Utilization

TABLE I: Resource Overheads of Octopus

Component	CPU (millicores)	Memory (MB)
Control Plane	1364	536
Data Plane	1	79

adaptation decisions at the edge cluster as well as global adaptation decisions by relocating entire stages across clusters.

C. Framework Scalability and Overheads

Here we outline the scalability and overheads of the Octopus control plane. Since Octopus is dependent on KubeStellar for remote orchestration of dataflows, the number of spoke clusters supported by Octopus is limited by the capacity of KubeStellar, which supports 1000s of sites [4]. In terms of overheads, Octopus has minimal contribution on the overall latency of pipeline execution and resource consumption. The planning overhead taken by the hub to process a logical plan, create the physical plan DAG, split it into P3Fs, and distribute them to corresponding spoke clusters in Octopus is 1 second. Thereafter, the deployment overhead or the time taken by the spoke to start run-time specific atoms from received P3Fs is 2.2 seconds. The CPU and memory overheads of the control and data planes of Octopus are negligible as summarised in Table I.

VI. Related Work

Cloud computing providers expose monitoring tools for recording metric data from infrastructure, hosted services and applications e.g. Amazon CloudWatch [5], Azure Monitor [6], and Google Cloud operation suite [7] that help in the visualization and processing of metric data for real-time or post-facto analysis. In academic literature, approaches have been proposed to scale metric processing based on window tuning [8], clustering [9] and in settings where timeliness is critical [10]. Reducing the size and dimensionality of the bulk of metrics data has been proposed based on sampling [11]–[13] and clustering [14], [15] approaches. Finally, anomaly analysis of metrics based on predefined relationships [16], application-specific techniques [17] and generic call-graph based techniques [18] have been proposed. These works constitute potential Octopus atoms and our main contributions focuses on orthogonal aspects of automation, control and adaptation of observability dataflows in a multi-cloud environment.

Frameworks like Loki [19] and Elastic Search [20] are popular for scalable log aggregation. Several approaches have been proposed in the literature for problem identification or root cause analysis based on machine learning [21], [22], static analysis [23], [24], causal analysis [25]–[27], and extracting execution models [28], [29]. Similarly, distributed tracing frameworks have been proposed both in academia [25], [30]–[32] and in industry [33]–[37]. These frameworks have been used for anomaly detection [38], [39], problem diagnosis [34], [40] and resource usage [32], [41] in offline and recently online [42], [43] settings. Instead of specializing frameworks for specific MELT modalities / point problems, our approach enables best-in-class multi-modal observability functions to be assembled across diverse use cases. Further, our control plane supports automation and closed loop control in geo-distributed/multi-cloud settings.

In contrast to observability dataflows which focus on in-flight data processing, observability data stores focus on querying observability data spanning MELT data that has been collected apriori. [44] outlines one such polystore architecture for querying multi-modal observability data. Scalable, flexible and efficient monitoring expressed using declarative query language in a single site context is explored in [45], [46].

Big data processing of geo-distributed data sets based on Hadoop/Spark-like frameworks has been proposed in the multi-cloud context [47]–[49] and the edge-cloud context [50]–[52]. Further, geo-distributed workflow orchestration engines in multi-cloud and edge settings have also been proposed [53]–[55]. In contrast to these works, our thrust is on unified stream + batch observability processing requiring continuous adaptation and fine-grained closed loop control at geo-scale. Further, we require support for a rich set of pluggable ”observability operators” spanning heterogeneous run-times.

VII. Conclusion and Future Work

In this paper, we presented Octopus, a declarative multi-cloud engine for end-to-end management of observability dataflows composed of atoms spanning heterogeneous run-times. Octopus enables optimized placement of these dataflows across geodistributed topologies while respecting dataflow constraints and service-level objectives such as performance and cost. The engine architecture supports pluggable run-times and thereby a large diversity of functions. Closed-loop control enables Octopus to continuously adapt the dataflow resource requirements to match measured performance indicators. We have successfully implemented two production quality IT observability pipelines for metrics and log processing using Octopus. Our evaluations indicate ability of Octopus to support dataflow customization and dynamic adaptation at geo-scale.

In future, we plan to enhance coverage of engine capabilities to AI/ML frameworks to support AI-capable observability atoms. We also plan to combine observability dataflows with observability query processing of stored MELT data in a unified geo-distributed framework.

References

[1] Opentelemetry. https://opentelemetry.io/.

[2] Kubestellar, 2023, https://github.com/kubestellar/kubestellar.

[3] "Keda — kubernetes event-driven autoscaling," https://keda.sh/, (Accessed on 10/21/2023).

[4] How to experiment with 1 million+ edge devices. retrieved oct 2023 from https://medium.com/@brauliodumba/how-to-experiment-with-1-million-edge-devices-fa2ad491527d.

[5] Amazon cloud watch. https://aws.amazon.com/cloudwatch/.

[6] Azure monitor. https://learn.microsoft.com/en-us/azure/azure-monitor/overview.

[7] Google cloud operation suite. https://cloud.google.com/products/operations.

[8] S. Meng and L. Liu, "Enhanced monitoring-as-a-service for effective cloud management," *Computers, IEEE Transactions on*, vol. 62, pp. 1705–1720, 09 2013.

[9] C. Canali and R. Lancellotti, "An adaptive technique to model virtual machine behavior for scalable cloud monitoring," 06 2014.

[10] G. Rodrigues, R. Calheiros, M. De Carvalho, C. R. Paula dos Santos, L. Granville, L. Tarouco, and R. Buyya, "The interplay between timeliness and scalability in cloud monitoring systems," 07 2015, pp. 776–781.

[11] D. Krishnan, D. Le, P. Bhatotia, C. Fetzer, and R. Rodrigues, "Incapprox: A data analytics system for incremental approximate computing," 2016.

[12] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe, "Privapprox: Privacy-preserving stream analytics," ser. USENIX ATC '17. USENIX Association, 2017.

[13] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe, "Streamapprox: approximate computing for stream analytics," *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13362737

[14] R. T. Ng and J. Han, "Efficient and effective clustering methods for spatial data mining," in *Very Large Data Bases Conference*, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:13136810

[15] R. Ding, Q. Wang, Y. Dang, Q. Fu, H. Zhang, and D. Zhang, "Yading: Fast clustering of large-scale time series data," 08 2015.

[16] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," ser. SIGMETRICS '13. New York, NY, USA: Association for Computing Machinery, 2013.

[17] A. Goel, S. Kalra, and M. Dhawan, "Gretel: Lightweight fault localization for openstack," ser. CoNEXT '16, 2016.

[18] 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.

[19] Loki. 2023. https://grafana.com/oss/loki/.

[20] Elastic search, 2023, https://github.com/elastic/elasticsearch.

[21] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009.

[22] S. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco: Statistical diagnosis of chronic problems in large distributed systems," 06 2012, pp. 1–12.

[23] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Oct. 2014.

[24] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems."

[25] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'07. USA: USENIX Association, 2007, p. 20.

[26] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[27] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016.

[28] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2568225.2568246

[29] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 217–231. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[30] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," ser. EuroSys '07. New York, NY, USA: Association for Computing Machinery, 2007.

[31] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Symposium on Networked Systems Design and Implementation*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:11701030

[32] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: Tracking activity in a distributed storage system," in *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06)*. Saint-Malo, France: ACM, Jun 2006.

[33] Apache. 2023. htrace. retrieved aug 2023 from http://htrace.incubator.apache.org/.

[34] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[35] Dynatrace. 2023. dynatrace application monitoring. retrieved aug 2023 from http://www.dynatrace.com.

[36] Solarwinds. 2023. traceview. retrieved july 2023 from https://traceview.solarwinds.com/.

[37] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," ser. SOSP '17, 2017.

[38] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," vol. 37, 12 2003, pp. 74–89.

[39] M. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," 01 2004, pp. 309–322.

[40] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, p. 17–30, mar 2007. [Online]. Available: https://doi.org/10.1145/1272998.1273001

[41] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *6th Symposium on Operating Systems Design and Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling

[42] H. Mi, H. Wang, Z. Chen, and Y. Zhou, "Automatic detecting performance bugs in cloud computing systems via learning latency specification model," ser. SOSE '14. USA: IEEE Computer Society, 2014.

[43] J. Zhou, Z. Chen, H. Mi, and J. Wang, "Mtracer: A trace-oriented monitoring framework for medium-scale distributed systems," ser. SOSE '14. USA: IEEE Computer Society, 2014.

[44] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards observability data management at scale," *SIGMOD Rec.*, 2021.

[45] S. Y. Ko, P. Yalagandula, I. Gupta, V. Talwar, D. Milojicic, and S. Iyer, "Moara: Flexible and scalable group-based querying system," in *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, ser. Middleware '08. Springer-Verlag, 2008.

[46] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining." New York, NY, USA: Association for Computing Machinery, 2003.

[47] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[48] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, “Wanalytics: Geo-distributed analytics for a data intensive world,” ser. SIGMOD ’15, 2015.

[49] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, “Low latency geo-distributed data analytics,” ser. SIGCOMM ’15, 2015.

[50] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, “Edgewise: a better stream processing engine for the edge,” ser. USENIX ATC ’19. USENIX Association, 2019.

[51] A. Sandur, C. Park, S. Volos, G. Agha, and M. Jeon, “Jarvis: Large-scale server monitoring with adaptive near-data processing,” in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022.

[52] B. Ramprasad, P. Mishra, M. Thiessen, H. Chen, A. da Silva Veith, M. Gabel, O. Balmau, A. Chow, and E. de Lara, “Shepherd: Seamless stream processing on the edge,” pp. 40–53, 2022.

[53] M. Pérez and A. Sanchez, “Fmone: A flexible monitoring solution at the edge,” *Wireless Communications and Mobile Computing*, vol. 2018, 11 2018.

[54] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, “Geo-distributed efficient deployment of containers with kubernetes,” *Computer Communications*, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419317931

[55] G. Bartolomeo, S. Bäurle, N. Mohan, and J. Ott, “Oakestra: A lightweight hierarchical orchestration framework for edge computing,” *USENIX ATC*, 2023.