Spring 2-5-2021

Hybrid Cloud Workload Monitoring as a Service

Shreya Kundu
San Jose State University

Hybrid Cloud Workload Monitoring as a Service

A Project Report

Presented to

The Faculty of the Department of Computer Science San José State University

In Partial Fulfillment

Of the Requirements for the Class CS 298

By

Shreya Kundu

Dec 2020

The Designated Project Committee Approves the Project Titled

Hybrid Cloud Workload Monitoring as a Service

by

Shreya Kundu

Approved For The Department Of Computer Science

San José State University

Dec 2020

ABSTRACT

Cloud computing and cloud-based hosting has become embedded in our daily lives. It is imperative for cloud providers to make sure all services used by both enterprises and consumers have high availability and elasticity to prevent any downtime, which impacts negatively for any business. To ensure cloud infrastructures are working reliably, cloud monitoring becomes an essential need for both businesses, the provider and the consumer. This thesis project reports on the need of efficient scalable monitoring, enumerating the necessary types of metrics of interest to be collected. Current understanding of various architectures designed to collect, store and process monitoring data to provide useful insight is surveyed. The pros and cons of each architecture and when such architecture should be used, based on deployment style and strategy, is also reported in the survey. Finally, the essential characteristics of a cloud monitoring system, primarily the features they host to operationalize an efficient monitoring framework, are provided as part of this review. While its apparent that embedded and decentralized architectures are the current favorite in the industry, service-oriented architectures are gaining traction. This project aims to build a light-weight, scalable, embedded monitoring tool which collects metrics at different layers of the cloud stack and aims at achieving correlation in resource-consumption between layers. Future research can be conducted on efficient machine learning models used on the monitoring data to predict resource usage spikes pre-emptively.

***Index Terms* - Cloud computing, cloud monitoring, cloud metrics, container runtime (CRI), guest operating system (OS), hypervisor, infrastructure as a service (IaaS), key performance index (KPI), multi-tenancy, operating system (OS), platform as a service (PaaS), software as a service (SaaS), system information gatherer and reporter (SIGAR), Time-Series Database (TSDB), virtual machine**

ACKNOWLEDGEMENTS

I want to thank my mentor Dr. Robert Chun for his continuous mentorship throughout this project and for motivating me to deliver my best. I would also like to thank my committee members, Dr. Thomas Austin and Mr. Amit Dutta for taking interest in my project and providing valuable feedbacks.

I thank my parents for their encouragement and support. Lastly, I thank the God almighty, for giving me this privilege and opportunity to pursue my dreams.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The National Institute of Standards and Technology's (NIST) definition of *cloud computing* includes certain characteristics [1], one of which is *Measured Service*. It is important to understand what measured service is and why it is needed. The services to be measured need metrics to be defined for that purpose. NIST defines a metric as "knowledge about a cloud property through both its definition (e.g., expression, unit, rules) and the values resulting from the measurement of the property" [4].

As enterprises and consumers use the cloud more and more [2], cloud providers need to provide certain Service Level Agreements (SLA) to their customers. These SLAs can be bound legally if an enterprise is purchasing the service. To provide SLA, the cloud service provider must define the metric, such as uptime of a service and make sure the uptime does not fall below a certain threshold [3]. However, the cloud is a complex environment with multiple layers, and customers can have various requirements for monitoring their services. A customer, running serverless workloads in the cloud would be interested in response times, service uptimes and connections being made to their applications, whereas a customer using Infrastructure-as-a-service (IaaS) would be interested in monitoring workloads, CPU, memory and storage utilizations. As a cloud service provider, this becomes an increasingly complex problem on how to provide monitoring information in multiple levels. In spite of having multiple stand-alone monitoring tools built by several cloud providers in the market, there is the lack of a single service-based monitoring tool in the market which can integrate with the middle-stack of multiple cloud providers. Also, in spite of large data provided by the monitoring tools, we do not see any predictive machine learning models implemented which can predict usage or billing which can be immensely helpful for businesses that have high activity in seasonal times, for example, Christmas, or Thanksgiving. The information gathered from monitoring is extremely important and forms the basis for multiple decisions in terms of operational cost planning and preemptive billing forecasting for a cloud service provider, as well as, consumer. [5]

Padhy et al. states that current monitoring systems for the cloud are not resilient and do not have a trust boundary for data access. Assuming Byzantine failure model, the paper discusses on how state model replication will allow the system to endure arbitrary faults, yet recover and continue working. The paper proposes a publish-subscribe mechanism for event handling which can be trustworthy. “Trust” has been covered in two ways: primarily authenticated information, and secondly, reliable information. [16] Authenticated information alludes to the fact that, data coming from the IaaS layer is indeed correct and is coming from the correct reporter. The reliability aspect is more specific to the data integrity itself and that the data is correctly marked and tenanted. However, if a major failure does happen at the IaaS layer, Veeraraghavan et al. proposes a system to mitigate such failures through traffic management. This is one of the ways a service provider can act if their data centers hosting cloud infrastructure are failing. The paper describes how continuous verification and dependency management handles such when the failures occur. [17] The impact of such failures on highly available and distributed systems running in scaled data centers are explored in the paper presented by Yuan et al. Such services are most commonly exposed at the PaaS layer and is an integral requirement for cloud architects to define as they are designing their environments. [18]

This literature survey focuses on exploring the categorizations of monitoring metrics in Section II, popular architectures pertaining to monitoring solutions in Section III, inferring essential characteristics for any cloud monitoring solution in Section IV, and thereby determining the final conclusion from the literature survey in Section V. The literature survey answers these questions: How different are monitoring metrics of specific importance for different cloud consumers and providers? What are the different monitoring architectures and the trade-offs between them? What are the essential features of cloud monitoring systems? This survey uses references from published papers and conference proceedings. Fig. 1 shows the organization of the literature survey.

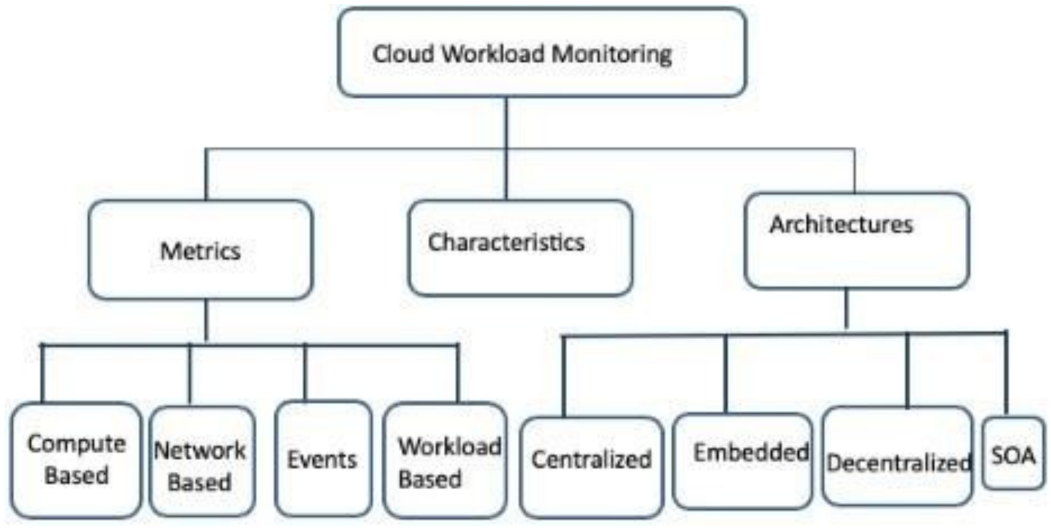


Fig. 1. Organization of literature survey

II. METRICS

Before we start monitoring, it is imperative to understand what to monitor. As mentioned previously, monitoring happens at multiple layers of the cloud model which can be application, network, middle-ware or physical. The literatures surveyed mention monitoring done at different layers with metrics that can be broadly classified as:

A. **Workload-based**: These KPIs (Key Performance Index) measure cloud workloads based on application level metrics. Shao et al. [6], Tovarňák et al. [8] both developed agent-based systems to monitor at the Software-as-a-Service (SaaS) layer where they mention KPIs such as response time between services, Inter Process Communication (IPC) response time, number of thread counts, number of I/O (Input/Output) computations, etc. Moses et al. proposes a new kind of abstracted metric by utilizing the basic IPC metric and assigning weights to it for each VM or workload for which the IPC is monitored. To analyze the success or failure of VM migration due to shared resource contention, Moses et al. incorporates the QoS (Quality of Service) value of the ith workload/VM and the corresponding IPC, and measures the Qos-Weighted throughput performance metric. [15]

B. **Compute-Based**: Tovarňák et al. [8], Alhamazani et al. [2], Rodrigues et al. [3] focus on monitoring at IaaS/ Platform-as-a-Service (PaaS) layer mentioning metrics as system uptime, disk throughput, storage I/O, CPU usage, CPU allocations, memory usage, memory allocations, virtual machine creation and release times.

C. **Network-Based**: Dhingra et al. [9], Tovarňák et al. [8], Alhamazani et al. [2], Rodrigues et al. [3] mention the metrics to measure the status of network connectivity which focus on packet counts, link throughput, Network Interface Cards (NIC)/vNIC interface statuses.

D. **Events**: Dhingra et al. [9], Calero et al [11], Rodrigues et al. [3] mention events sent by cloud components, which primarily are asynchronous changes that take place at IaaS/PaaS layers. The authors use these events to trigger workflows or make certain decisions based on correlation of the data being sent by the events and the data being collected by metric monitoring

III. ARCHITECTURES

Once the KPIs or metrics are defined on what to monitor, an architecture is required in how the monitoring is to be achieved. NIST has provided a standard literature around the "Cloud Services Metric Model" or "CSM", which provides the definitions and descriptions for how metrics should be and how their values can be determined. However, to determine the value, a metric need to be collected, stored and calculated upon to provide such information. Collection of monitoring data takes a few specific architectures:

A. **Centralized**: Shao et. al [6] developed a centralized architecture by employing a monitoring tool in each VM which sends the runtime information to a central database and monitoring agent. Shao et al. employed a server-agent style architecture where a monitoring agent is deployed on each virtual machine which is equipped with different monitoring facilities like Hyperic's System Information Gatherer and Reporter (SIGAR) cross-platform API (Application Programming Interface), JVM (Java Virtual Machine) agent, Filters, and JMX interface. SIGAR API is used to provide runtime information about the infrastructure regardless of their platforms, JVM agent monitors the health of JVM, Filters intercept messages passed to and from the monitoring device and thus help to understand the interacting behavior between users and services. The technique of service probing where the monitoring code is embedded with target code is also employed. This information is then instantiated to an abstraction called the runtime model for cloud monitoring (RMCM) which is then validated, and if flagged, an alarm is sent to the central monitoring center. Huang et. al [7] employed a push and pull algorithm for a centralized hub-spoke model where change of status between "producers" and "consumers" are being tracked continuously. For the push phase, the producer is the initiator and it sends out status information when it detects a change indegree greater than the threshold called User Tolerant Degree (UTD). For the pull-phase, the consumer is the initiator and requests the producer for a status update. The abstraction service that provides a control-plane to program and manage the producers and consumers based on programmer's intent is centralized. Both architectures assume a single point of aggregation of all data, which is simple to implement and provides data across the different layers of the cloud model. Moses et al. focuses on monitoring shared resource contention, especially Last-Level Cache (LLC), and aims at improving the overall datacenter throughput via VM migration while maintaining the SLAs. The approach proposed is called MIMe (Monitor Identify and Migrate), in which VMs suffering due to resource contention are identified and prioritized for migration to achieve improved weighted throughput. Moses et al. employs a centralized architecture where a centralized policy server collects the cache occupancy of each VM and acts as a scheduler to identify candidates for migration. [15] Centralized architectures can take advantage of existing methods, like polling of collection of data and storing them easily in a singular place. The literatures also point out that a major disadvantage of a centralized paradigm is scalability. As more cloud services are spun up or horizontally scaled, the amount of data generated can be quite overwhelming. It also introduces a single point of failure, wherein if the centralized monitoring service fails, then the monitoring for the whole cloud is stopped until the service is back up again.

B. **Embedded:** Tovarňák et. al [8] developed an embedded model by employing an event- based daemon called Ngmon at the core of the guest operating system of each VM which listens on an UNIX domain socket and collects data across all layers. Dhingra et. al [9] developed an architecture with one Dom0 agent per physical host, and a VM agent per VM both communicating with the metric collector. Both of these models do not assume any aggregation point. Instead, it is a direct generation of the monitoring data. The central principle of both of these architectures is agents or scripts within a code which sends data back. The "agents" provide real time information as code or services are executed and provides immensely granular data with details. The major disadvantage is that as agents reside inside the guest OS (Operating System) or hypervisor, there are many compatibility and portability considerations to take care of. The survey finds that the Dom0 agent [9] can work only with Xen hypervisor, and Ngmon agent [8] can work only in Linux based environments. In spite of this disadvantage, the embedded monitoring is one of the best ways to monitor services which are multi-tenanted, since both the Dom0 agent and Ngmon can determine the per VM or tenant effort. Long Zhang et al. propose a resilient architecture and solution for observability within applications itself which can be monitored through a framework. They aim to automatically improve exception handling in an already running or executed code. They propose a system called TRIPLEAGENT, which embeds a component which allows for monitoring, fault injection and validation. The paper goes on to elaborate on the design and deployment of the agent. Even though an embedded architecture, the design does heavily borrow from centralized command and control paradigms to control fault injection and analysis. The monitoring agent is used to collect dynamic data such as stack distance (method reporting the fault vs the method generating the fault), number of method exceptions etc. The monitoring agent provides a report to the developer when experiments are conducted. In a distributed cloud environment, and particularly at the SaaS and PaaS layer, where hardware is completely abstracted, having a tool to determine the effect of problems such as memory out of bound, or Disk IO failures causing operational issues in code execution helps in maintaining uptime and SLA for that service. [19]

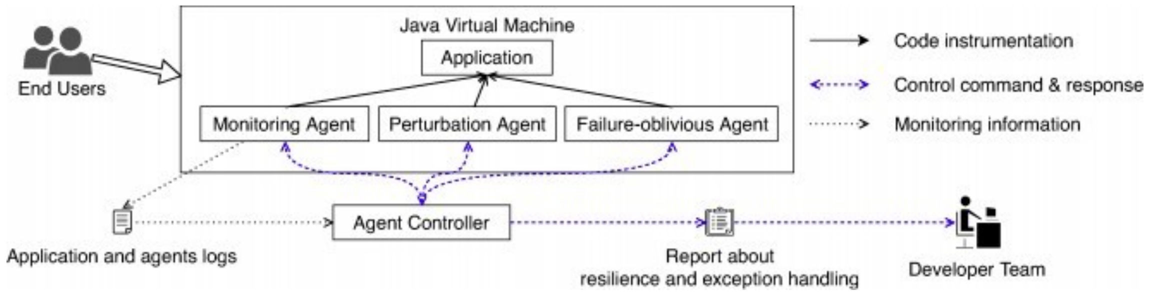


Fig. 2. Triple Agent Architecture [19]

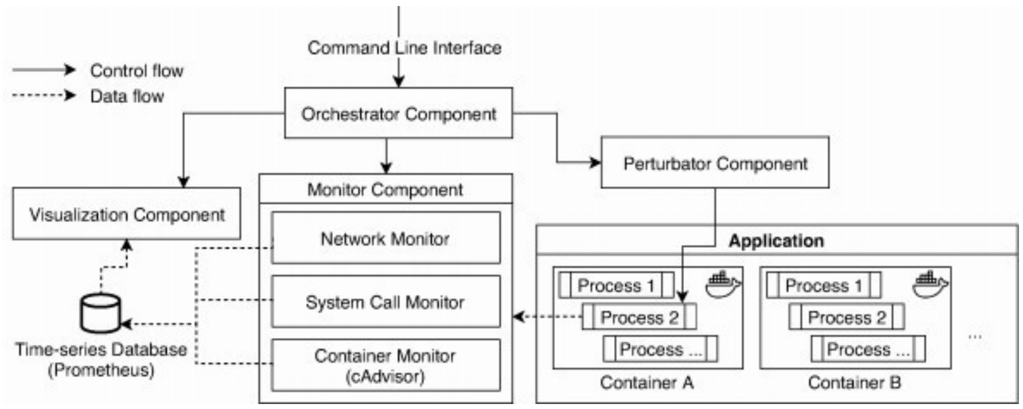
C. **Decentralized**: Anderolini et. al [10] designed a decentralized model by developing a "probe process" which collects performance and utilization indexes on each hardware and software resource on each monitored node and thereby the information is received by the collection agent. The collection agent validates the metric data, compresses it and send it to a dedicated collector node, which can then plot it in real time or sent it to a distributed analyzer with a set of analyzer nodes for map-reduce processing. These collector and analyzer nodes indicate that there is no single point of aggregation, but multiple services running in tandem collecting certain parts of data. Skvortsov et al. [12] compared the decentralized model of EXCESS and ECO_2 monitoring. The EXCESS model comprises of ATOM (neAr-real Time Monitoring fraMework) which includes a monitoring server called MONITOR and multiple light-weight collector agents called ACTORS. In the ATOM architecture, the ACTORS continuously sample node and application specific data, and send that to the

MONITOR. In the ECO_2 model, multiple Zabbix agents are employed on each physical node and VM that collect the metric data and send it to a Zabbix server running on a dedicated VM. All of these architectures achieve excellent scalability by increasing the number of collector nodes [10], ACTORS or Zabbix agents [12]. Yuan et al. thoroughly investigated distributed failures in large scale cloud environments running PaaS services which are used heavily. Hadoop, Mongo, Cassandra, DynamoDB, Kafka etc. are services which are used by cloud architects to create system designs and workflow for their applications. Region and geo-redundancy are generally assumed in a cloud environment, in which case, all PaaS layer components are, by extension, distributed. Monitoring in such an environment is challenging and costly, since at PaaS, the infrastructure has been already reserved and every vCPU thread and memory are being billed. So, knowing what to monitor for is extremely important to make sure applications are not starved to execute due to heavy monitoring burdens. Yuan et al. focus primarily on understanding the sequence of failures that manifests due to one or more temporal errors. Generally, these errors propagate and culminate in component failures which reaches the user. The entire manifestation is less understood; however, individual failures can be isolated and has been studied in great detail, including categorization of root causes and symptoms. The manifestation of the failure tends to be very complex, even though the cause could be very simple. As per the paper "almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software." The paper goes on to report that the complexity of a failure requires more than one input to manifest, and there is specificity to these inputs. It also concludes that a lot of the failures stem from daily operational tasks such as adding/removing nodes (assuming service is configured for auto-scaling), configuration changes made to the service and network partitioning. [18]

D. **Service Oriented Architecture (SOA)**: SOA monitoring is a relatively new paradigm of monitoring where the monitoring of workloads or services itself is a service at the PaaS layer. This architecture is gaining traction in recent times and not well-explored, hence the survey did not find multiple references to report. In this case, the monitoring is handled through a dedicated VM or a dedicated service which integrates at the middleware. Calero et. al [11] developed this new plug-in service dedicated for monitoring called "MonPAAS" (Monitoring-Platform-As-A-Service). MonPAAS application integrates with the infrastructure layer implemented with Openstack and the platform layer coupled with the monitoring software NAGIOS. The MonPAAS service is responsible for managing MVMs (monitoring VM) per tenant. Topology changes in either the physical or virtual layer are intercepted by

MonPAAS. It is configured as a cloud consumer for security and isolation purposes. Since the monitoring architecture is service oriented, it provides unprecedented flexibility in “what” to be monitored and “how” to report, based on “rules” set by the administrator. As monitoring is itself a service, this architecture also provides unprecedented scalability. However, it does so at the cost of being too resource heavy as dedicated VMs are allotted for the purpose of running the service. Simonsson et al. proposed the emerging concept of “Observability” which extends the traditional idea of monitoring to help understand and correlate the internals of applications and infrastructures. This is essential as microservice based cloud applications are becoming more and more ubiquitous. Analyzing microservice performance is now an absolutely essential part of monitoring application health. Observability leverages structured event logs, multiple metrics and tracing to correlate distributed but related events, which is very common in microservice based architecture, for example, a user request that spans through multiple microservices. To evaluate resiliency in production environment, and face real-world uncertainties, Simonsson et al. mentions the concept of “Chaos Engineering”. In this concept, first a steady state of the application is hypothesized based on monitorable metrics. Then a real-world failure simulation is injected into the production environment such as disk full or unavailable third-party services. These experiments are done after deployment, and continuously to build confidence in the production system. Automation and minimizing the blast-radius (impact of the chaos engineering experiment) are the final steps. To achieve observability and perform chaos engineering experiments in containerized applications, Simonsson et al. proposed CHAOSORCA which is comprised of a monitor, a perturbator, and an orchestrator. The collection of system information at runtime is the responsibility of the monitor component, and it helps to attain observability and connect system level failures to application-level behavior by providing KPI information at the container level, operating system level, and application level. The perturbator injects system failures at runtime which is defined as "<*s, e, d*> where *s* is the system call, *e* is the error code and *d* is the delay before the call is invoked". The orchestrator acts as an interface between these two and helps to generate reports and conduct chaos experiments. [14]

Fig. 3. CHAOSORCA Architecture [14]



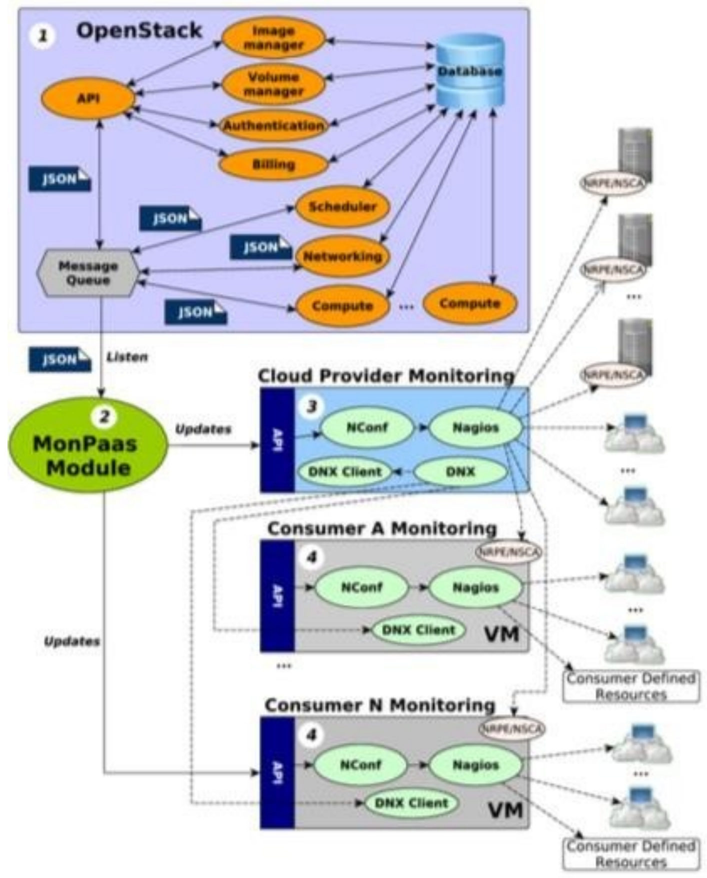


Fig. 4. MonPAAS Architecture [11]

IV. CHARACTERISTICS OF CLOUD MONITORING

Cloud monitoring systems need to have certain characteristics to effectively monitor cloud workloads. Survey on popular cloud monitoring tools such as NAGIOS, Cloud Watch, OpenNebula, MonPAAS etc., ascertained certain generic characteristics which have been corroborated extensively by the literatures reviewed. These characteristics can be generically detailed as:

A. **Monitoring at scale**: A cloud monitoring solution should be able to monitor massively distributed workloads and provide information with relatively low margin of error. Anderolini et. al [10] achieved scale by increasing the number of collector and analyzer nodes with increment in input data stream. Calero et. al [11] on the other hand achieved scalability by creating more MVMs on demand. Veeraraghavan et al. proposes "Maelstorm" to drain traffic in case of a failure and redirect the information to a set of services which are not in failed state. Nicolas et al, discusses an observability framework for monitoring large scale workloads executing in micro-services. [20]

B. **Monitoring at different layers**: A cloud monitoring solution should be able to monitor at multiple layers within the cloud model. Tovarňák et. al [8] achieved this by deploying Ngmon (event-based daemon) listening to an UNIX socket that collects data across all layers. Calero et. al [11] achieved monitoring capability at different layers by integrating the MonPAAS with the message queue from the infrastructure layer and with agents in the platform layer through

APIs. Long et al. focuses specifically on microservices running on docker containers. They propose ChaosOrca, a chaos engineering tool for fault-injection, monitoring, reporting and analysis at the micro-service layer. [19] Nicolas et al. discuss more generically of the framework and proposes designs on auto-scaling of the observability framework itself using IaaS monitoring as an input. [20]

C. **Interoperability**: A cloud monitoring solution must be agnostic to the cloud model as they are implemented by different cloud providers. Shao et. al [6] achieved this by instantiating RMCM (Runtime Model for Cloud Monitoring) from the raw data which essentially hides the underlying heterogeneity of the cloud platform. Tovarňák et. al [8] on the other hand designed an event-based object from the performance and utilization metrics collected by the embedded agent. Marcio et al. proposes a paradigm of "Monitoring Slice" which is per tenant in a cloud hosted environment. Each slice is monitored using multiple tools at multiple layers, controlled though a centralized policy engine called "FlexACMS". The tool is specifically to create and manage monitoring slices irrespective of the monitoring solutions being employed. [21]

D. **Shared services**: A cloud monitoring solution should be able to monitor services which are shared across multiple resources such as multi-tenant systems and should be non-intrusive to the core workload. Calero et. al [11] achieved resource monitoring in a multi-tenant system by employing a dedicated monitoring VM (MVM) per tenant. Moses et al. describe shared resource monitoring mostly at the PaaS and hypervisor layers controlling VM lifecycle. They create a novel approach on how to identify VMs based on their behavior for migration, achieving determinism in auto-scaling and aggregation. Since all resources in cloud are generally shared and tenanted, using a technique to aggregate loads in predictable patterns helps in troubleshooting when issues do occur. Tovarňák et. al [8] achieved this property by deploying the Ngmon to be the core service of the guest OS, and to monitor resource usage per VM, i.e. tenant on it. Dhingra et. al [9] also achieved this property by employing the Dom0 agent at the hypervisor level which can determine per VM usage.

V. PROPOSED ARCHITECTURE

There are multiple monitoring solutions in the cloud and infrastructure space and each one of them, have their own advantages and disadvantages. Generally, the monitoring solution architectures have three major components, a timeseries database, a collector agent framework and a visualization dashboard. This project implements the same tuple: influx for timeseries, self-developed collectors and self-developed visualization system.

To monitor resource usage, most open source cloud monitoring software (Prometheus, Graphite, Sysdig) depend heavily on the orchestration layer (openStack, Kubernetes, AWS). It is rare for the same systems to collect details from the IaaS layer below. The IaaS layer monitoring is generally left to more commercial solutions such as NAGIOS or CACTI. In this project the monitoring methods implemented for PODs or the applications layer is extended to the IaaS layer (BareMetal and VMs) as well. The rationale behind this approach is to collect the VM usage metrics as reported by the VM host rather than the hypervisor. Hypervisor level metrics collection can report burst usage due to CPU steal cycles. The project VMs all reside in shared services and do not have dedicated resources assigned, hence dynamic CPU contention needs to be addressed. If CPU steal cycles are incorporated in the calculations, then results can be erroneous.

To monitor resource usage of the PODs or the containers, most open source cloud monitoring software gather data from an orchestration system perspective, i.e. the metrics of resource usage by PODs are reported by the orchestration layer instead of an application within the POD. This project not only collects POD utilization statistics, but the same stats are reported directly by the application itself. This makes monitoring agent scalable along with the application without complex auto-scale logic and enables close monitoring of the application health.

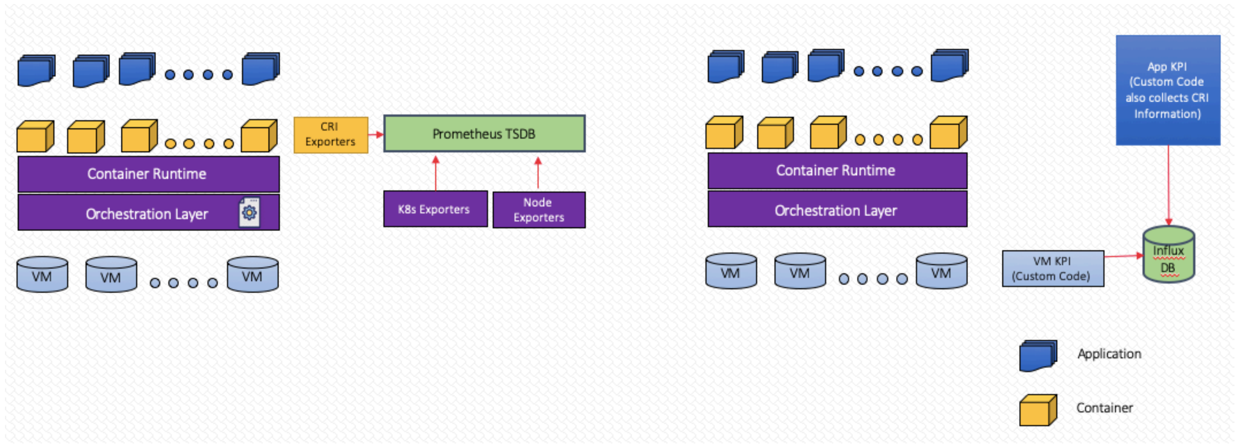


Fig. 5. Generic Orchestration Layer Collection vs Collection w/ Embedded Custom Code at Multiple Layers

All monitoring solutions provide alerts and error management but co-relation of any type is either commercialized such as Sysdig and InfluxDB enterprise, or generally ignored. Some basic co-relations such as VM affinity of PODs, or evictions due to resource shortages which are incredibly useful for operators could easily be added and this project aims to do so.

The open source monitoring solutions are created by adding multiple different types of software together. Generally, the setup of such systems is fairly easy, but the rules to define and collect metrics has a steep learning curve. Prometheus, for example, provides an excellent enterprise level collection framework, but the config files can become very large and increasingly complex if very granular metrics are targeted. This becomes cumbersome in future stages to manage and control. In this project, programmatic methods are employed to collect the metrics which is much simpler and deeply integrated with application code. Since this project targets a very specific use case, unlike Prometheus which is more generically built for enterprise level use cases, the framework is lightweight and collects only specific data points and generates lower monitoring overhead.

The approach adopted for cloud monitoring is such that multiple best practices are combined to achieve monitoring at different layers, which facilitates correlation of application level resource utilization to platform level resource utilization, and allows us to gain insights about the containerized cloud application as a whole.

Firstly, the code responsible for gathering metrics is coupled with the code generating workload, in a containerized environment. This approach ensures that the monitoring agent is embedded with application logic, and thus KPI metrics are collected at application layer (PaaS). An agent is also employed at the IaaS layer by running a code natively on each VM of the Kubernetes cluster which collects the same metrics at VM level. Three levels of workload are generated (high, medium, low) and are replicated at factor of three to run on the Kubernetes cluster. The KPI data generated at the VM level is compared with KPI data generated at the application layer to infer the number and types of workload running on each VM. This continuous monitoring at different layers of key parameters gives granular information about the application's internal state, and thus helps in determining the source of failure in case of performance bottleneck, in spite of the cloud system's inherent complexity. Thus, the proposed architecture aims at achieving observability.

The KPI metrics are collected at the IaaS layer by the APIs provided by the SIGAR (System Gatherer and Reporter) library which gives low-level operating-system and hardware level information and is ported to various OS environments. This simple API driven metric collection ensures the monitoring mechanism is non-intrusive to the core-workload. In the PaaS layer, a bash script is employed to continuously monitor CPU and memory percentages of processes running within the container via the Linux Top command.

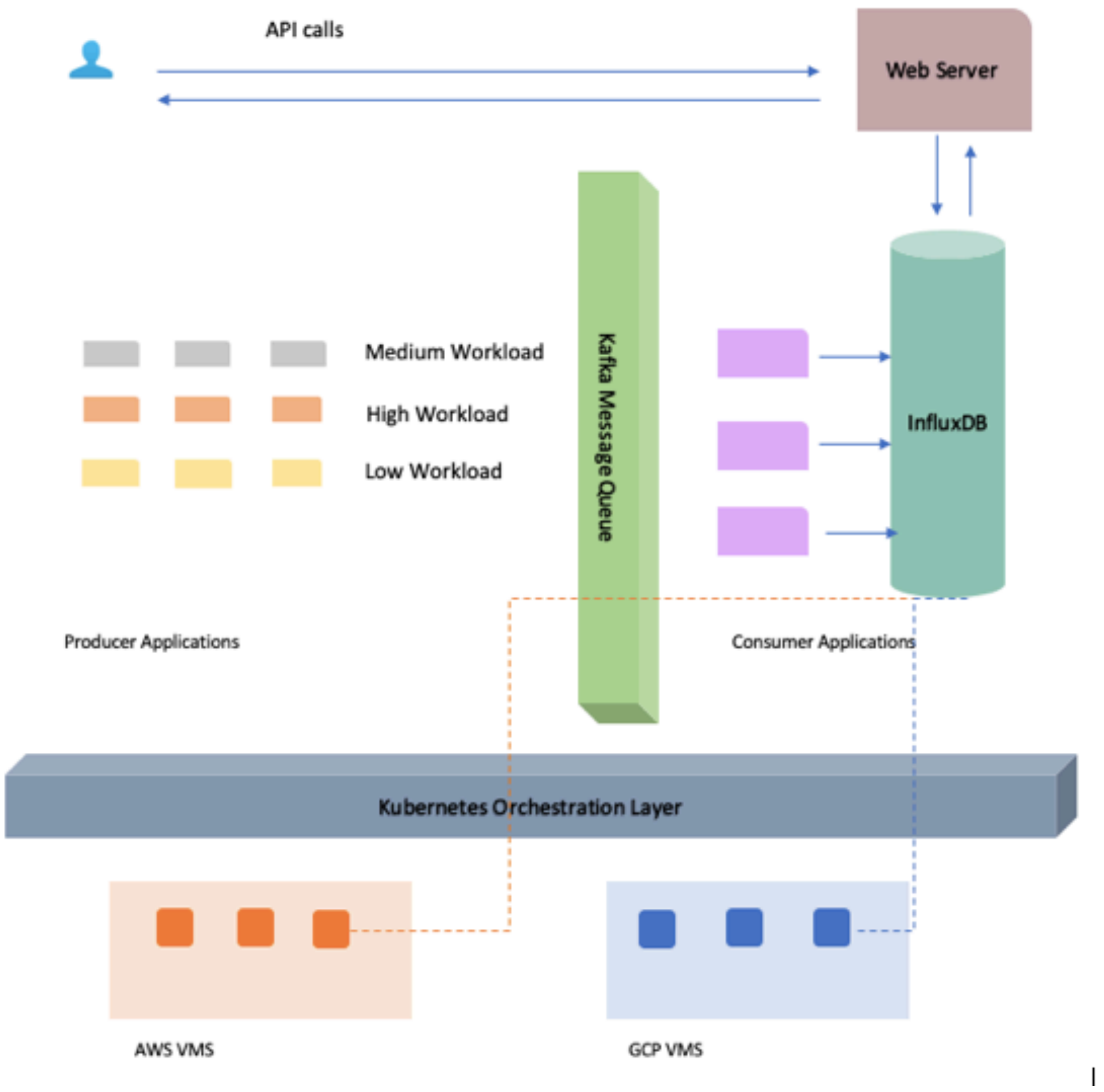


Fig. 6. Project Framework Architecture

VI. IMPLEMENTATION

The architecture displayed in Fig. 6 is designed for the process-flow shown below.

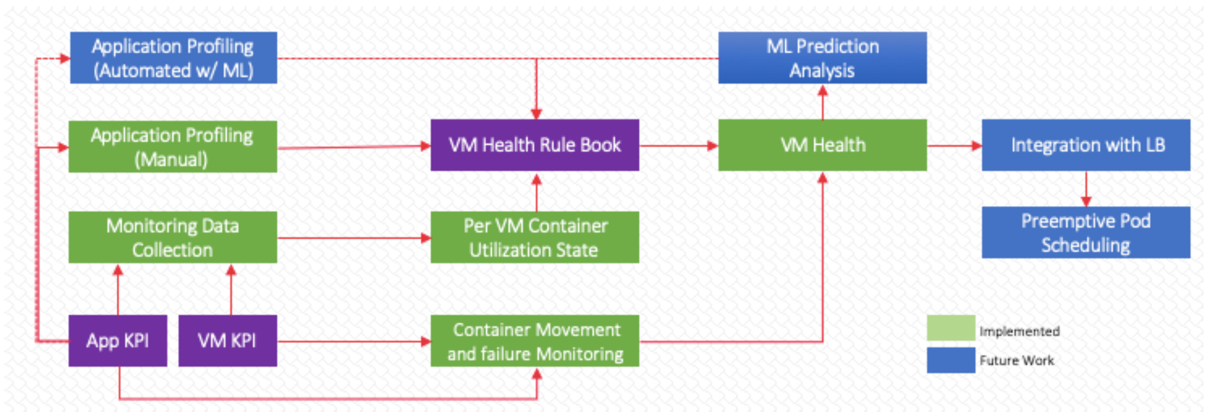

Application Profiling (Automated w/ ML)
ML Prediction Analysis
Application Profiling (Manual)
VM Health Rule Book
VM Health
Integration with LB
Monitoring Data Collection
Per VM Container Utilization State
Preemptive Pod Scheduling
App KPI
VM KPI
Container Movement and failure Monitoring
Implemented
Future Work

Fig. 7. Process Flow Diagram

The environment for the project is created to showcase multi-cloud deployment and collection for workloads running across them. The cloud providers used for the environment are the AWS EC2 and the GCP Compute engine. Hosted services provided by these service providers were not used. Instead the clusters and all corresponding services are run natively in the VMs to mimic an IaaS deployment. Two independent VMs apart from the cluster VMs are used. One VM hosts the timeseries database and the analyzer for aggregation of the raw data & correlation. The other VM hosts the webserver for visualization. The operating system used is Ubuntu 18.04 LTS. The VM specifications are 2vCPU, 2GB RAM, 10GB Volume.

The orchestration layer used for this project is Kubernetes. The k8s cluster consists of one master node and seven worker-nodes. Five of the worker-nodes are hosted in GCP. The master node, and two worker-nodes reside in AWS. The connection between GCP and AWS is achieved via a classic VPN tunnel. The tunnel is created by first reserving a static IP address on GCP, and then creating a customer service gateway attached to a VPN service gateway deployed through site-to-site VPN on AWS. Both IP ranges, on AWS and GCP, are redistributed through static routes and the tunnel is created using the IKEv1 pre-shared key method. Information provided by the AWS VPN endpoint is used to create the two tunnels on GCP. Once the above steps are completed, the tunnel is negotiated and brought up. Firewall rules on both sides, AWS and GCP, need to be added to allow for the traffic from one VPC to come to the other.

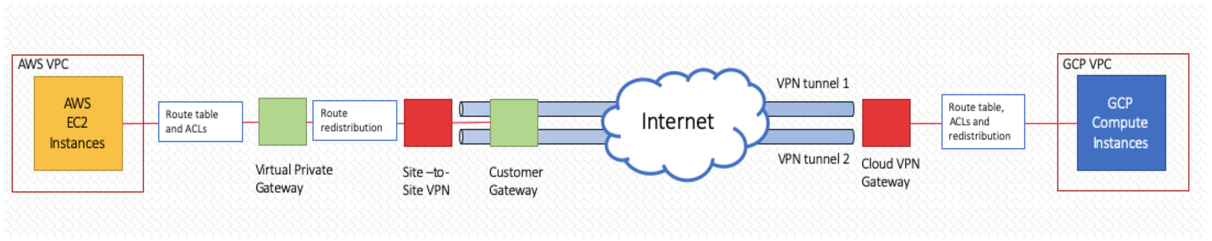


Fig. 8. VPN Tunnel Set-up

The orchestration service is then loaded with a messaging service for all the information to be passed from the containers to the database. To achieve this, the Kafka service is deployed on the k8s cluster to allow for inter-component information exchange. The database used is InfluxDB for its timeseries capability and storage for metrics data.

Spring boot applications are developed with Apache Kafka dependencies. These containerized applications have two shell-scripts mounted inside of it. The first script runs at entry-point to install and execute the Linux Stress-ng tool, which in turn simulates CPU, and Memory stressors. The second script is executed periodically by the application to fetch CPU and memory utilization percentages via the Linux Top command. These applications are tagged into three categories as "High load", "Low load", "Medium load", as per their respective stressors and are modeled as Kafka Producer applications which write the KPI metrics to the Kafka message queue under three different topics corresponding to their load generation. The table below shows the range of CPU % utilization generated by the stressors for each application type.

Table 1. Load Generation per Application Type

Application Type	% CPU Load generated
High Load	50 ~ 80
Medium Load	25 ~ 50
Low Load	10 ~ 20

Three spring boot applications are developed which are modeled as Kafka Consumers and are responsible for reading the messages on the queue per topic and writing them to the database.

Another spring boot application is developed which is also containerized and serves as the Webserver, and is responsible for reading the KPI metrics from the database and displaying information to the user in a graphical format. APIs are designed so that users can view workload per type of application (“High load”, “Low load”, “Medium load”) over time.

Finally, Java agents are run natively on each VM of the cluster which gathers the information about its resource utilization with SIGAR API and the number and types of containers running on the VM over time. Thus, KPI metrics are gathered at the PaaS, as well as, the IaaS layer. The table below shows all the databases and measurements and corresponding metric types.

Table 2. KPI Metrics Storage

Database	**Measurement**	**Description**
KPIDB-HIGH	Stats	KPI metrics for all applications tagged as High-Load
KPIDB-MED	Stats	KPI metrics for all applications tagged as Medium-Load
KPIDB-LOW	Stats	KPI metrics for all applications tagged as Low-Load
VMKPI	VmStats	KPI metrics for all VMs
PODCOUNT	ContainerCount	Number and type of applications that are running on each VM at each time-stamp

The data written to InfluxDB by the above applications is a time-series data which is essentially a sequence of data-points sorted over time, measuring the same metric. This data needs to be processed to gain insights from it. For this purpose, a new set of Spring-boot applications are employed which generates the mean of CPU consumption & memory consumption percentages over 1-minute time-windows. An Influx-Client is set up inside the Analyzer application which provides integration with the InfluxDB API to perform these analyses. In the diagram below, the process of aggregation by the analyzers is shown.

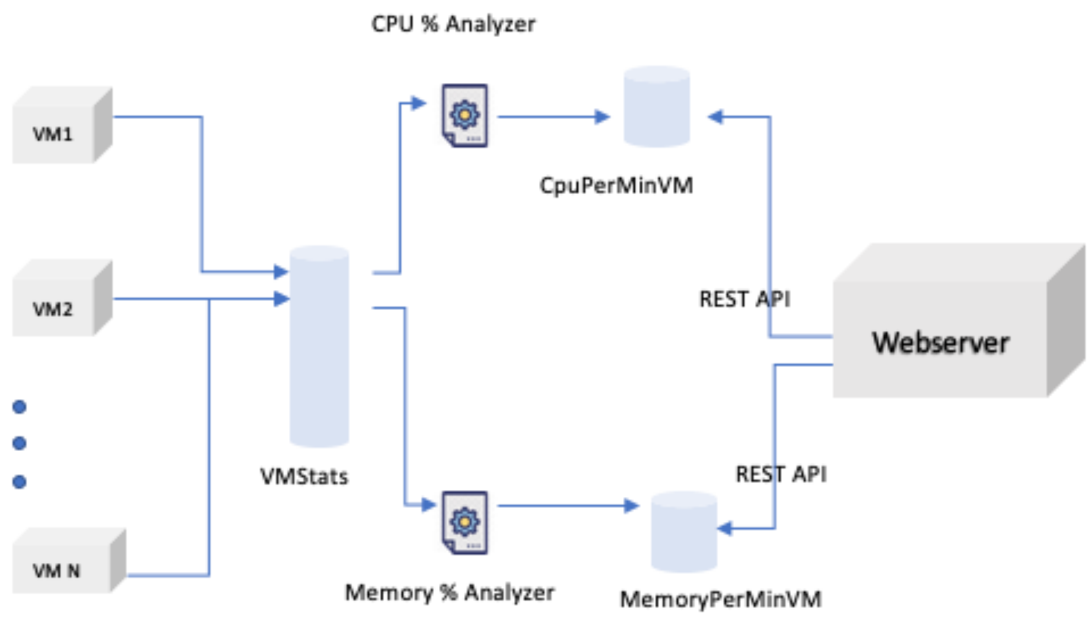


Fig. 9. Analyzer Work Process

VII. RESULTS & OBSERVATIONS

There were few exceptions observed which had to be remediated for proper running of the project. The first and foremost observation was when SIGAR libraries were used inside the container, the VM usage and container usage percentage were observed to be exactly the same. SIGAR worked well when running in VMs for system level monitoring since it directly queries the host CGROUP drivers and reports back the entire VM usage. But, when called from inside the container, SIGAR reported the entire VM usage since the container CGROUP is a child of the host CGROUP.

Because of this, the Linux top command is customized to determine the process utilization from inside the container. *Top -b -n 3 -d 0.1 -p <pid>* calculates KPI metrics per process in batch mode with three iterations and delay of 100 milliseconds between screen updates. Reading the usage from orchestration/k8s layer was avoided due to discrepancy issues as shown in Fig. 20.

The second finding of the experiment showed high CPU usage by the stress-ng memory stressor tool, as CPU was consumed by threads either performing read/write or stalled by other threads. Memory activity performed outside CPU cache resulted in longer execution time than CPU clock cycle, making the kernel scheduler to mark the threads busy, and as a result, the memory stressor consumed high scheduling time. This resulted in a resource bottleneck on the 2vCpu VMs. Hence, only one memory stressor was deployed, due to CPU shortage.

For the following experimentation, the cluster VMs were run for 7 hours and a workload was generated to collect KPI metrics. The table below shows the number of raw data points collected for each of the application types and the VM hosts.

Table 3. KPI Metric Collection

Metric Type	# Raw data points measuring KPI metrics
VmStats (IaaS) [CPU & memory utilization %]	18914
KPIDB-LOW [CPU & memory utilization %]	4356
KPIDB-MED [CPU & memory utilization %]	4278
KPIDB-HIGH [CPU & memory utilization %]	5636

A. **Identification of POD distribution across VMs**: The distribution of PODs and optimal VM resource utilization is a complex problem. From an orchestration layer point of view, the k8s load balancer assigns PODs to VMs based on current resource utilization. However, in many cases, future usage spikes can lead to load mismatches resulting in eviction of PODs and cluster failures. With VM resource-scaling, this can be minimized, but in resource strapped clusters, the situation can quickly become very grim with multiple containers restarting and evicting, causing working containers to fail as well.

The first result deals with matching the load characteristics of the VM to that of the applications inside of PODs over time. If one application in a POD is using the VM resources at 80%, and another POD is scheduled on the same VM which requires a similar amount of CPU, the failure can cascade. The applications within the PODs have random sleep times in between load generations to randomize the usage spikes. This simulates usage spikes of an actual production environment. A cloud-admin operator can pre-emptively move PODs from a heavily loaded VM to an idle VM by profiling applications based on their generic resource utilization characteristics over-time and closely identifying applications running on a VM and thereby predicting a future usage spike. Thus, POD eviction scenarios can be avoided altogether.

The cluster VM with hostname cworker6 is chosen to visualize and identify POD distribution in it.

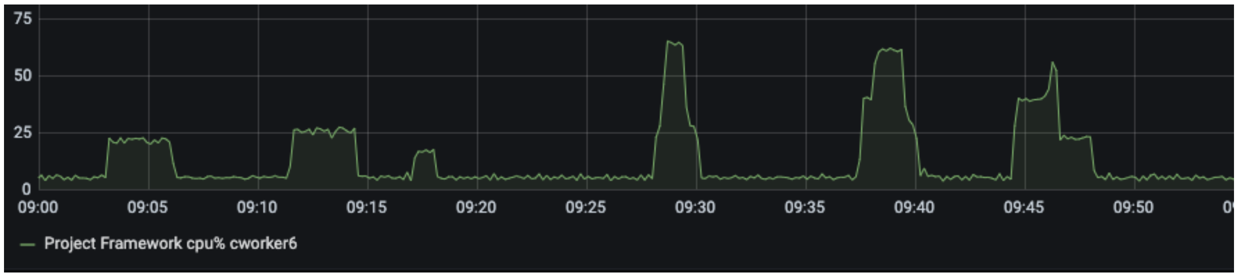
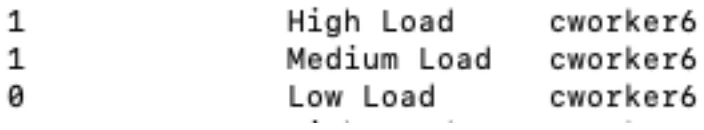


Fig. 10. cworker6 [CPU utilization %]



```
1          High Load     cworker6
1          Medium Load   cworker6
0          Low Load      cworker6
```

Fig. 11. cworker6 [Number and type of Containers]

As it is observed the VM cworker6 has one instance of “Medium Load” and one instance of “High Load” application running on it, all the “High Load” and “Medium Load” applications’ CPU utilization plots are overlaid with that of cworker6 to identify the ones are running on the VM.

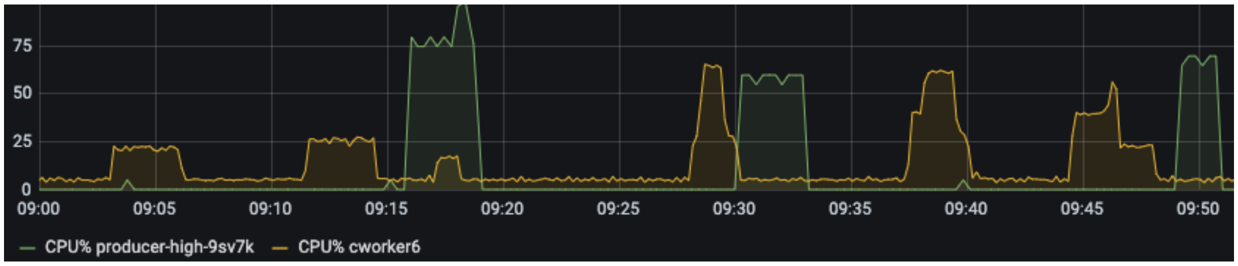


Fig. 12. producer-high-9sv7k vs cworker6 [CPU utilization %]

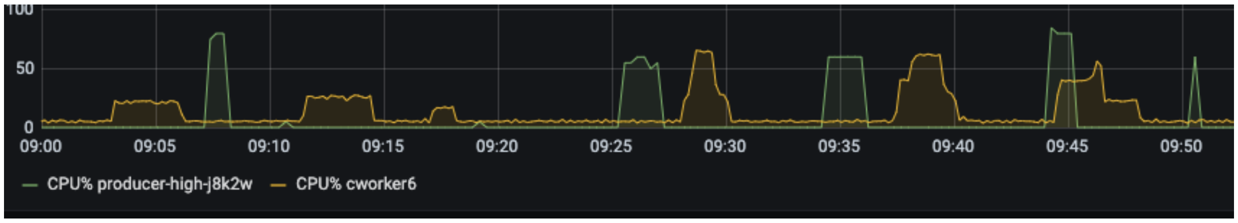


Fig. 13. producer-high-j8k2w vs cworker6 [CPU utilization %]

It is observed that PODs producer-high-9sv7k and producer-high-j8k2w show CPU utilizations that does not match with cworker6, i.e. they show CPU utilization in time-frames where VM cworker6 is idle. Hence, they can be eliminated from consideration.

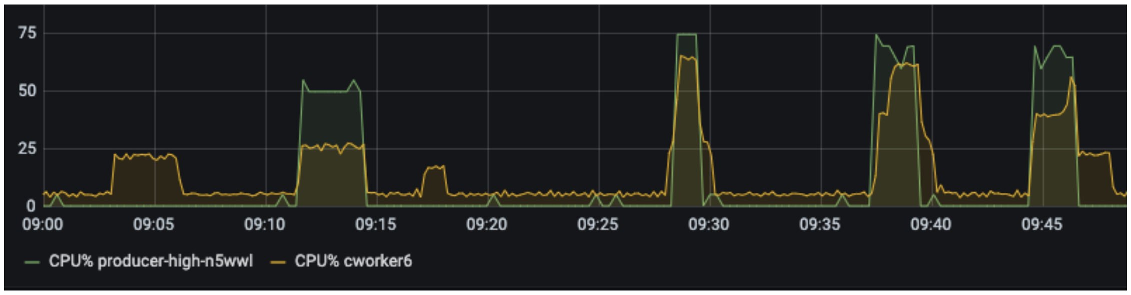


Fig. 14. producer-high-n5wwl vs cworker6 [CPU utilization %]

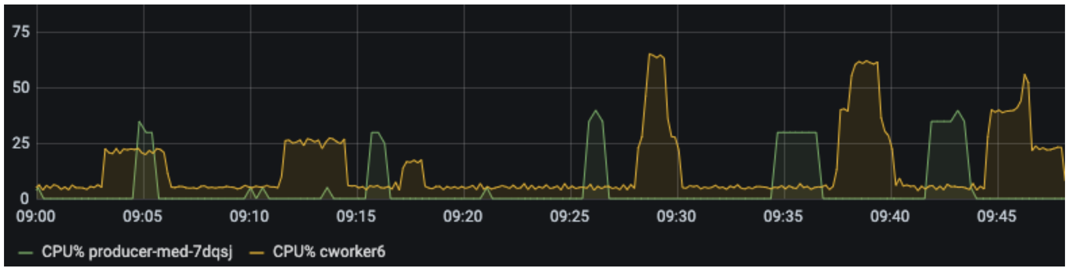


Fig. 15. producer-med-7dqsj vs cworker6 [CPU utilization %]

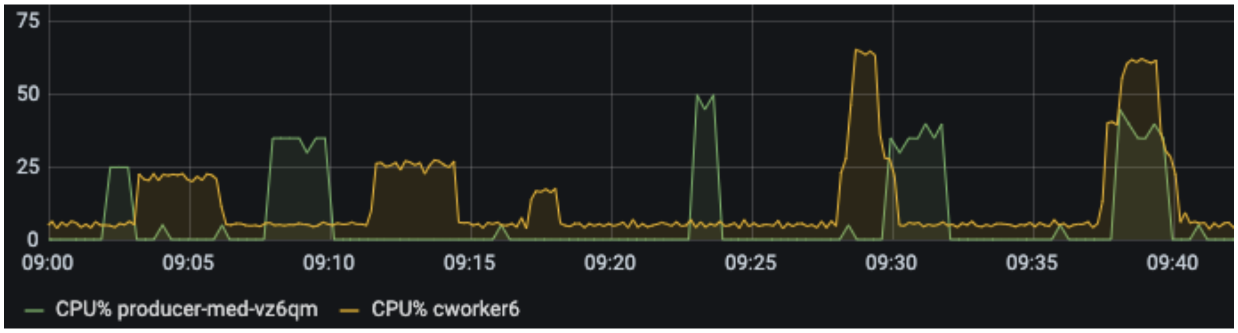


Fig. 16. producer-med-vz6qm vs cworker6 [CPU utilization %]

Similarly, PODs producer-med-7dqsj and producer-med-vz6qm can be eliminated since they show CPU utilization in time-frames where VM cworker6 is idle.

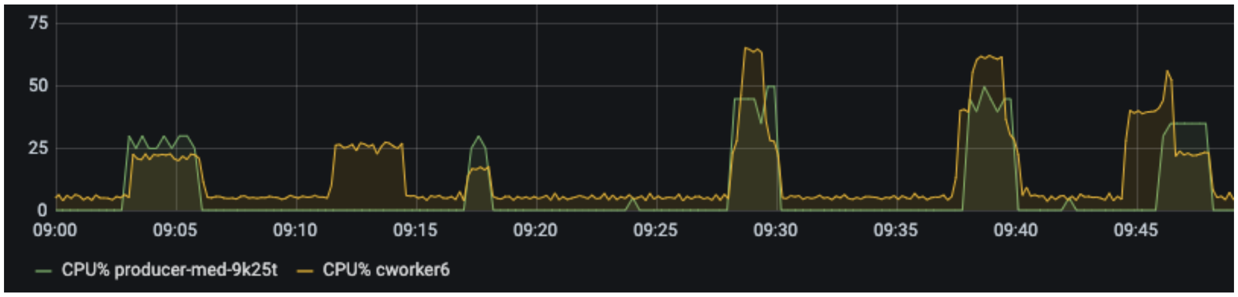


Fig. 17. producer-med-9k25t vs cworker6 [CPU utilization %]

Hence, the VM cworker6, has POD producer-high-n5wwl and producer-med-9k25t running on it. This kind of information is extremely useful from the cloud admin point of view to predict future resource usage patterns and possible resource bottlenecks. This correlation can be further automated and improved by employing machine-learning algorithms to enable the cloud-admin to take pre-emptive actions.

An experimentation was performed using the widely popular open source Prometheus collector & associated Grafana dashboard alongside this project framework tool to perform comparative analysis on the same KPI data. Prometheus is essentially a TSDB (Time Series Database) and has a custom query language. It is purposefully built for numeric time series and supports exporters to collect detailed data from multiple data sources.

In context of the use case to match VM and POD load characteristics,

first and foremost, advantage of this project framework is that it does application profiling (tagging applications as per their load characteristics) and storing that information into the InfluxDB as illustrated in Fig 8. Whereas, Prometheus allows us to get information about number of the pods running on each node, but there is no built-in application profiling involved.

Prometheus exporters infer POD usage as reported by the kubelet cadvisor in the k8s orchestration layer. However, in this project framework, the PaaS level KPI metrics are reported as seen by the application running inside the POD, which has a distinct advantage in inferring actual application health, rather than overall POD usage. The following graphs show an overlaid comparison between CPU utilization reported by the project framework tool to the same reported by Prometheus.

The POD utilization reported by Prometheus and the application utilization reported by the project framework vary by a small percentage as seen in the graphs below, since POD utilization takes all processes into account including k8s infra services, instead of just the profiled application.

VM cworker6 with PODs producer-high-n5wwl and producer-med-9k25t is chosen to perform the overlaid comparison of CPU metrics reported by Prometheus and the project framework tool.

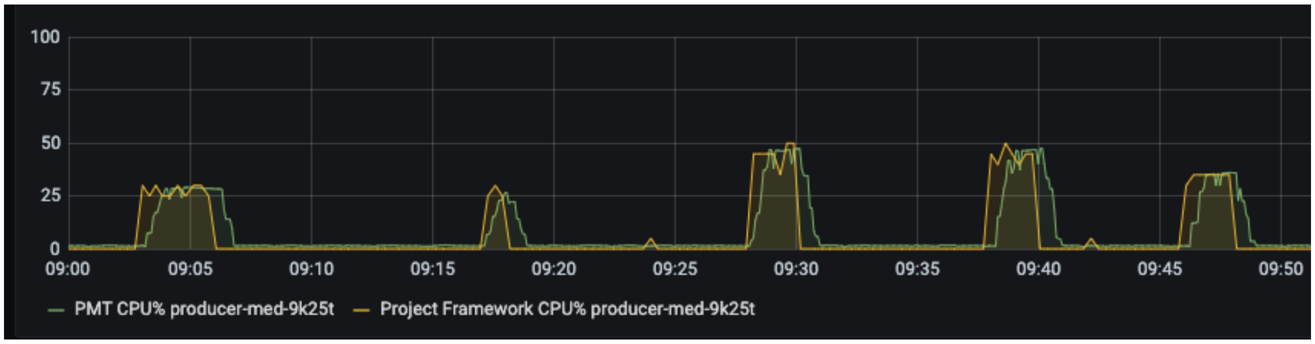


Fig. 18. producer-med-9k25t [CPU utilization % Prometheus vs Project Framework]

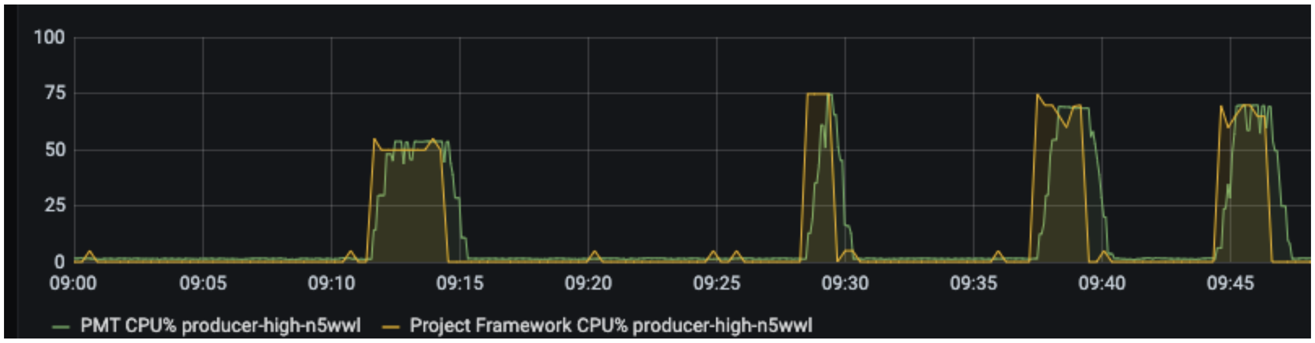


Fig. 19. producer-high-n5wwl [CPU utilization % Prometheus vs Project Framework]

A snapshot of CPU% metric and corresponding timestamp reported by Prometheus and Project Framework is shown in the table below to identify the deviation in metric percentage and timestamp reported.

Table 4. PMT vs Project Framework [Metric & Timestamp]

Project Framework CPU% metric & Timestamp	PMT CPU % metric & Timestamp	Δ Time	Δ CPU % Metric
30% 09.03.03	25% 09.03.45	+42s	+5%
25% 09.03.24	24% 09.03.56	+32s	+1%
30% 09.03.36	28% 09.03.57	+21s	+2%
25% 09.03.55	24% 09.04.26	+31s	+1%
30% 09.04.30	29% 09.04.42	+12s	+1%
25% 09.04.50	24% 09.05.10	+30s	+1%
30% 09.05.11	29% 09.05.25	+14s	+1%
30% 09.05.29	28% 09.06.10	+41s	+2%
25% 09.05.47	23% 09.06.22	+35s	+2%
45% 09.28.16	39% 09.28.47	+31s	+6%
45% 09.28.33	47% 09.29.01	+28s	-2%
46% 09.29.05	46% 09.29.15	+10s	0%
35% 09.29.22	38% 09.29.47	+25s	-3%
50% 09.29.39	47% 09.29.51	+12s	+3%
50% 09.29.54	47% 09.30.12	+18s	+2%

Another observation is that for overall VM utilization Prometheus reports a CPU utilization burst whereas SIGAR (used in the IaaS layer monitoring in this project) does not report so. It is suspected that, since kubernetes uses cgroupfs driver for resource management, the cgroup hierarchy have caused Prometheus to report the burst. Since SIGAR queries the linux kernel directly the utilizations reported is as seen by the linux kernel itself. If actual application utilization per VM needs to monitored this CPU burst can be misleading.

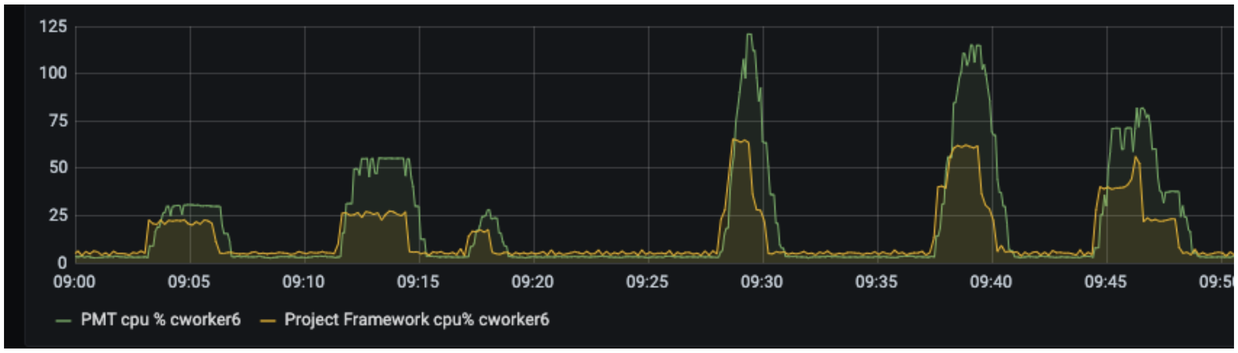


Fig. 20. cworker6 [CPU utilization % Prometheus vs Project Framework]

The burst usage reported from the Hypervisor perspective is illustrated in the diagram below. The hypervisor allocates CPU cycles on demand to VMs and reserves the rest (for instances to be scheduled in future, or probable future CPU cycle demands of the existing instances). Generally, a hypervisor allocates a specific amount of CPU cycles to each shared VM, for example a T3.small instance is entitled to 1.0 GHz clock speed (40%) of the AWS XEN hypervisor with 2.5 GHz clock speed. If the VM consumes 40%, then as per the hypervisor the VM is utilizing 100% of its allocated CPU resources. If the VM requires more clock speed e.g. 50% then it is shown as a burst and the hypervisor reports >100% utilization. This is a source of contention for tools using cgroupfs to monitor the CPU cycle usage since the hypervisor is stealing resource from any arbitrary VM in its shared resource pool to provide the resource to the VM asking for the burst.

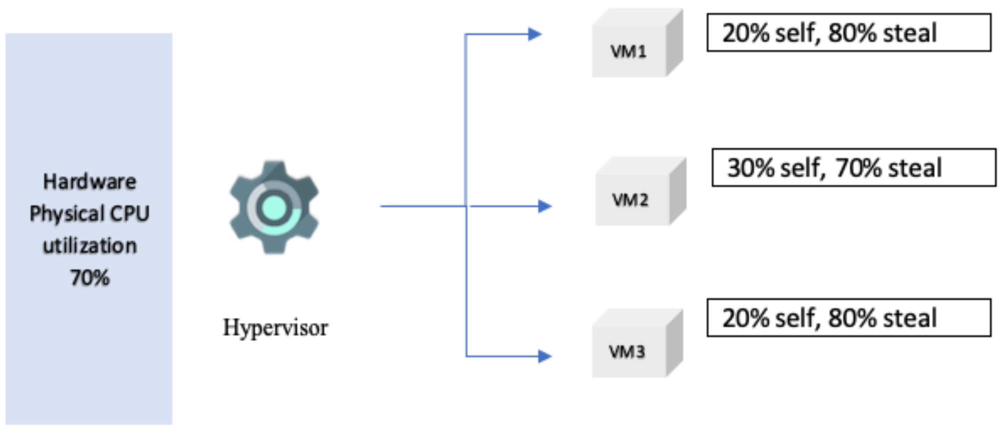


Fig. 21. Representation of CPU Cycle Steal

Other comparison factors were:

1. Deviation in metric and time: Metrics reported by both frameworks show a deviation of -5% ~ +5% as observed in Table 4. However, since the project framework tool is embedded while Prometheus collects metrics via endpoint scraping, there was a 30 seconds delay on average in the collection timestamp by Prometheus in comparison to embedded collection agent implemented in this project. This is a significant advantage of push model followed by this project in comparison to pull model followed by Prometheus.
2. Ease of setup: Prometheus is enterprise grade hence setup is complicated. ConfigMaps and exporters needs to be setup for data collection. The project implements a simpler setup (minutes compared to hours) since its geared for a specific use case.

3. Monitoring overhead: The monitoring agent in the framework implemented in the project is deeply coupled with the application code, hence it does not require additional resources to run, and can be scaled in and out as the system evolves. Whereas for the Prometheus deployment had to be set up on a separate node to run it as a service, and it incurred additional resource cost on the cluster. An experimentation was performed with the cluster was horizontally scaled from 8 to 20 VMs. For each cluster state, Prometheus and Project Framework Tool were used as collector for 2 hours, to compare their overhead.

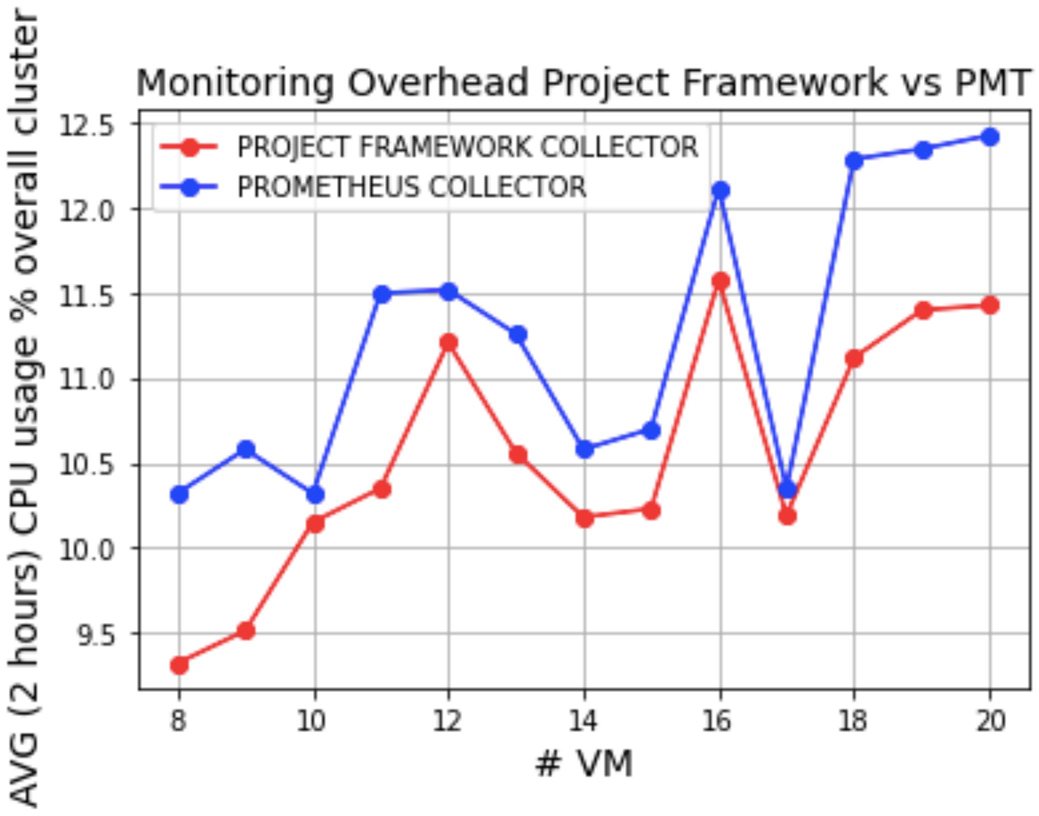


Fig. 22. Average CPU utilization % of Cluster [PMT vs Project Framework]

B. **Preemptive identification of POD evictions in a resource-strapped cluster:**

In a resource-strapped cluster, a resource bottleneck can occur easily if the applications are replicated at a higher factor and the application usage exceeds physical capacity of the cluster. As a result, multiple pods get evicted and the k8s replication controller tries to bring them back up. This results in a cascading eviction scenario and ultimately results in cluster failure. The k8s orchestration layer itself does not report any problem in this scenario. The tool implemented in this project reports PODs started and evicted per VM in every 15minutes and current VM health, to indicate possible cluster failure.

A rule-book is designed in the agents running on the VMs as depicted in the following table. This helps to anticipate cluster failure and evictions pre-emptively.

Table 5. VM Health Rule Book

#High Load Apps	#Medium Load Apps	#Low Load Apps	VM Health
>=2	0 or more	0 or more	Red
1	1	1	Red
0 or more	>=3	0 or more	Red
0 or more	0 or more	>=5	Red
1	>=2	0 or more	Red
1	0 or more	>=2	Red
0	2	>2 & <5	Red
1	0	1	Yellow
1	1	0	Yellow
0	2	<=2	Yellow
0	1	>=3 & <5	Yellow
0	0	>=4 & <5	Yellow
0	1	<3	Green
0	0	<4	Green
1	0	0	Green

For this experimentation after initial cluster setup of each application type of replication factor three which runs in stable state, a cluster health state and PODs started per VM snapshot is taken.

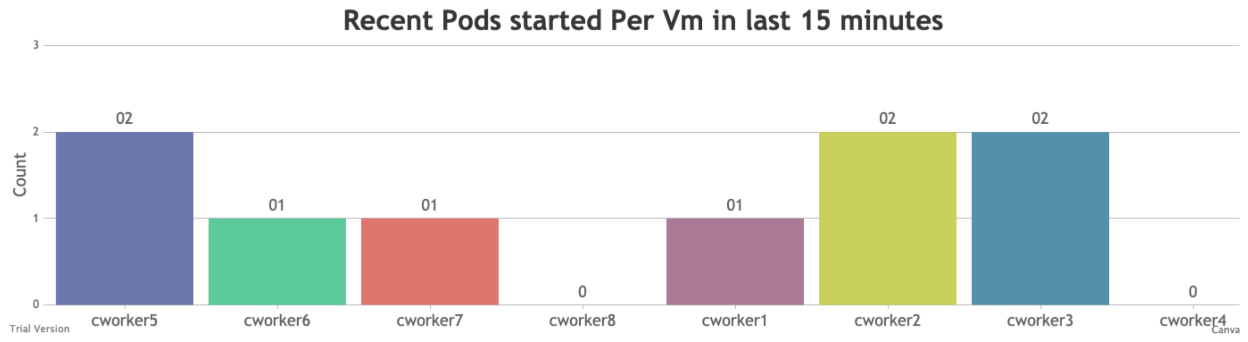
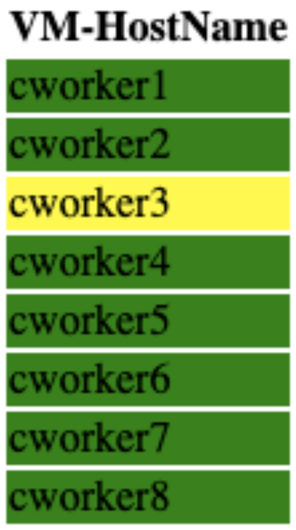


Fig. 23. PODs started per VM after initial set up



VM-HostName

cworker1
cworker2
cworker3
cworker4
cworker5
cworker6
cworker7
cworker8

Fig. 24. VM Health after initial set up

Table 6. Applications per VM after initial set-up

cworker1	cworker2	cworker3	cworker4	cworker5	cworker6	cworker7	cworker8
1 High	1Medium 1 Low	1 High 1 Low	0	1Medium 1 Low	1High	1Medium	0

Next application type High Load was scaled by three more instances.

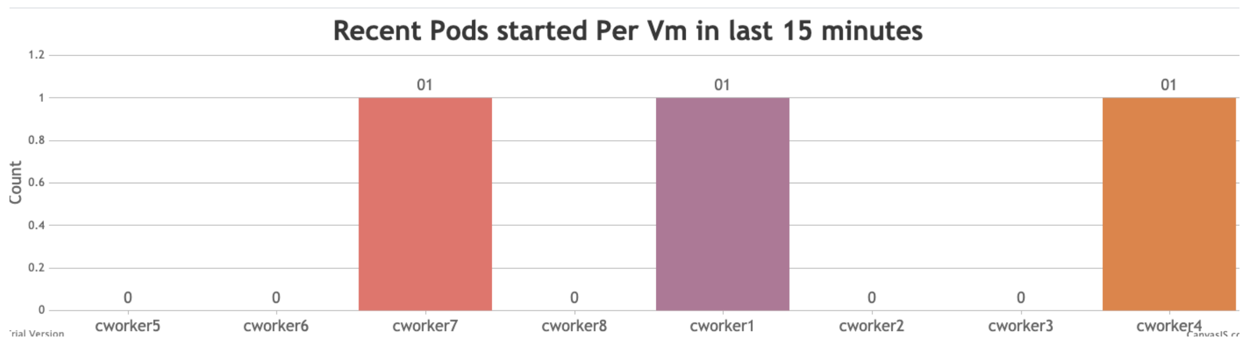
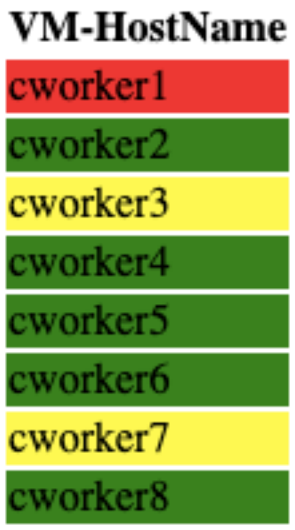


Fig. 25. PODs started per VM after first scale



VM-HostName
🟥 cworker1
🟩 cworker2
🟨 cworker3
🟩 cworker4
🟩 cworker5
🟩 cworker6
🟨 cworker7
🟩 cworker8

Fig. 26. VM Health after first scale

Table 7. Applications per VM after first scale

cworker1	cworker2	cworker3	cworker4	cworker5	cworker6	cworker7	cworker8
2 High	1Medium 1 Low	1 High 1 Low	1 High	1Medium 1 Low	1High	1Medium 1High	0

Next application type Medium Load was scaled by three more instances.

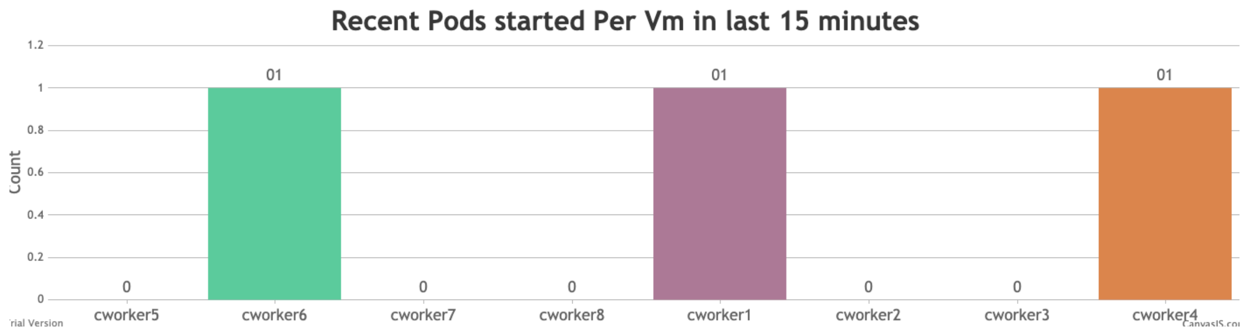
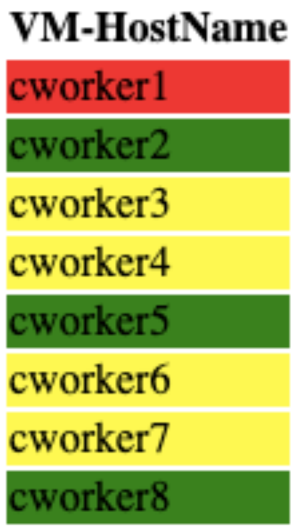


Fig. 27. PODs started per VM after second scale



VM-HostName
🟥 cworker1
🟩 cworker2
🟨 cworker3
🟨 cworker4
🟩 cworker5
🟨 cworker6
🟨 cworker7
🟩 cworker8

Fig. 28. VM Health after second scale

Table 8. Applications per VM after second scale

cworker1	cworker2	cworker3	cworker4	cworker5	cworker6	cworker7	cworker8
2 High 1Medium	1Medium 1 Low	1 High 1 Low	1 High 1Medium	1Medium 1 Low	1High 1Medium	1Medium 1High	0

Next application type High Load was scaled by three more instances.

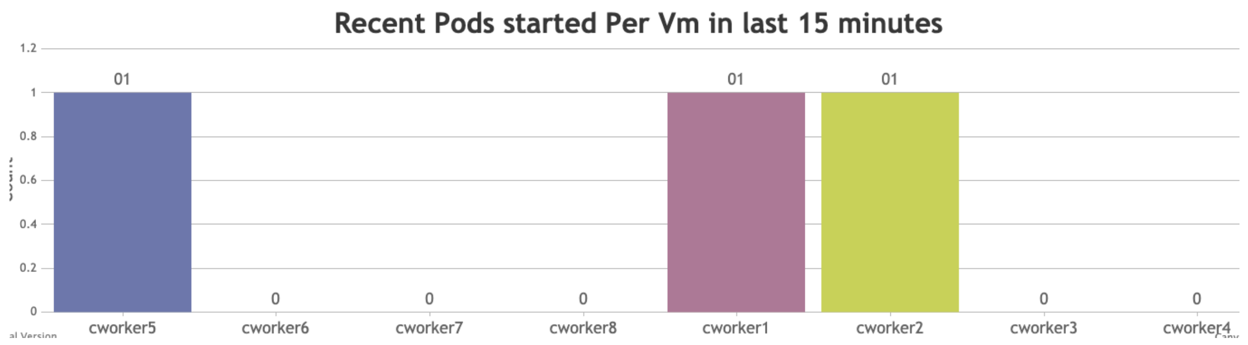
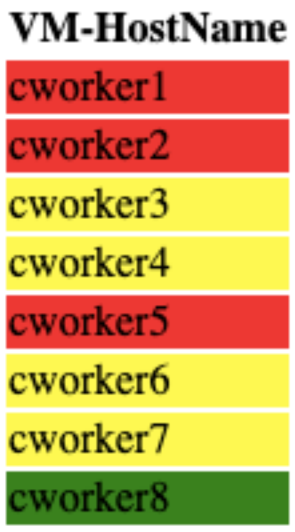


Fig. 29. PODs started per VM after third scale



VM-HostName
🟥 cworker1
🟥 cworker2
🟨 cworker3
🟨 cworker4
🟥 cworker5
🟨 cworker6
🟨 cworker7
🟩 cworker8

Fig. 30. VM Health after third scale

Table 9. Applications per VM after third scale

cworker1	cworker2	cworker3	cworker4	cworker5	cworker6	cworker7	cworker8
3 High 1Medium	1Medium 1 Low 1 High	1 High 1 Low	1 High 1Medium	1Medium 1 Low 1 High	1High 1Medium	1Medium 1High	0

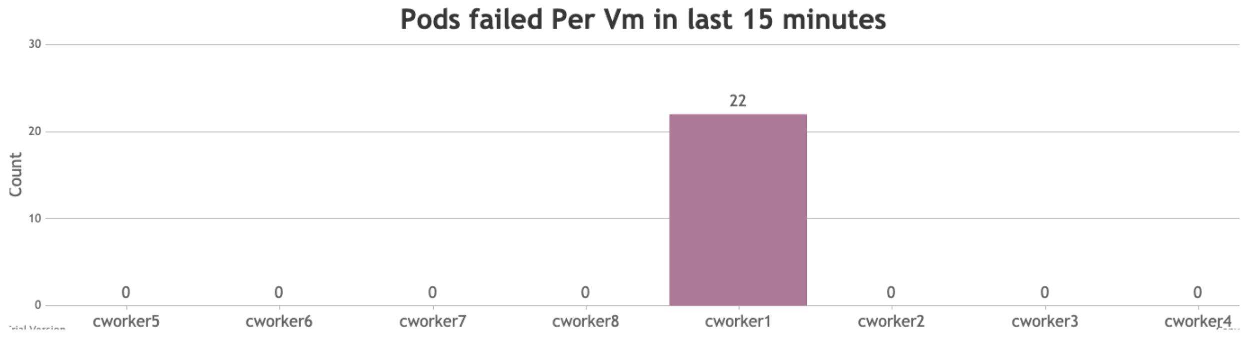


Fig. 31. Evictions after third scale

It is seen despite of cworker1 in "Red" health state PODs getting scheduled on it, which resulted in multiple evictions on it. PODs got repeatedly scheduled and evicted on it resulting in cascading evictions.

Whereas, in absence of such simple API-driven health check, cloud-admin user has to either perform command-line query on the k8s master to get such information, or customize PromQL query to monitor kubelet-evictions metrics for the namespace required.

C. **Identification of VM affinity for Applications:** An experimentation was performed where PODs of each application type were deleted repeatedly over 100 iterations, and the VM hosts on which they were brought back up by the k8s replication controller was tracked. Even though affinity was not set for PODs as part of initialization, it was observed that k8s load balancing algorithm tends to schedule specific applications/PODs in specific VMs. There were two motivations to conduct this experiment:

1. The first one is to highlight, that if the application failure occurs i.e. the POD is not evicted due to resource bottleneck, k8s load balancer does not consider it to be a failure scenario and tends to schedule the PODs on the same VMs over time.
2. The second is to identify, if the same application or similar applications are failing over time, where are they re-manifesting. The experiment follows the PODs over multiple failures deployed as part of a replication controller to identify affinity to any particular VM hosts.

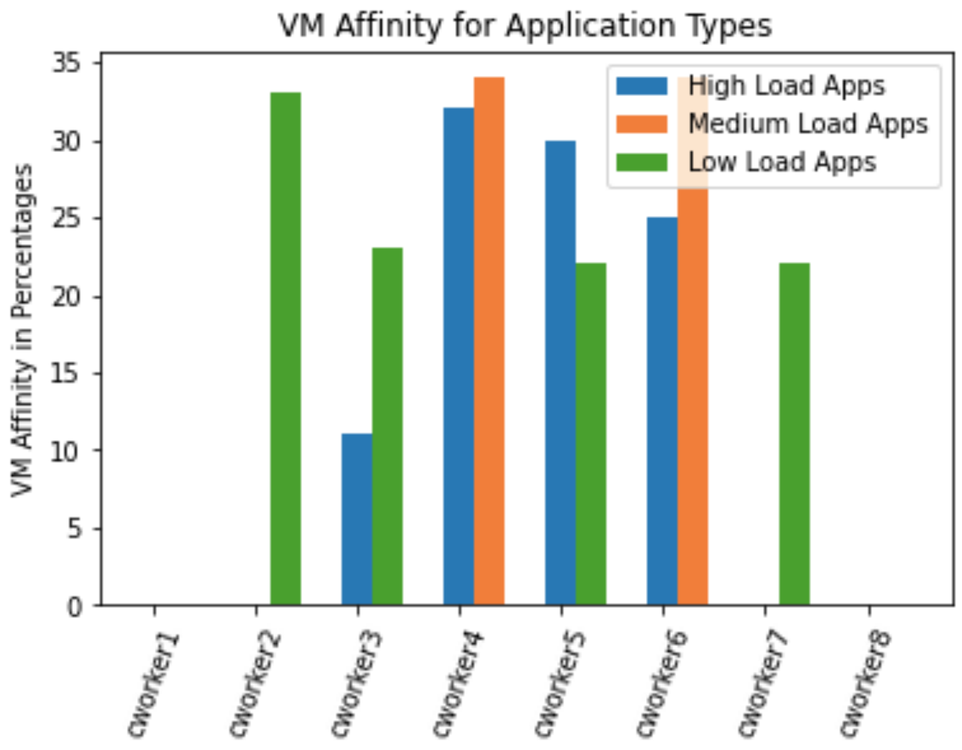


Fig. 32. VM affinity per Application Type

VIII. CONCLUSION & FUTURE WORK

In this paper, the existing and well-researched cloud monitoring architectures are surveyed and the use-cases each are targeted for are reported. The essential characteristics that any cloud monitoring solution should offer are discussed. With the survey it is inferred that containerized, service-oriented monitoring solutions which are efficient from a usability point-of-view, and offers "observability" are gaining traction in recent times. This project builds such a tool which provides containerized monitoring solution and combines multiple best practices such as embedded agents, scalable and service-oriented framework.

The framework implemented successfully eliminates resource utilization bursts as reported by the hypervisor by monitoring at different layers and provides consistent data for the use-case of actual process load monitoring on the host VMs. The tool also employs a unique solution to monitor host VM health by tagging applications as per their load, and monitoring the number and type of applications running on each VM. This health monitoring indicates possible cluster failure pre-emptively. However, as the monitoring framework is deeply integrated with application, any additional update in the monitoring metrics requires rolling upgrade of the application images.

The tool implemented is compared with widely popular Prometheus/Grafana monitoring stack to derive a comparative performance evaluation, and as expected the metrics reported by both vary by a small percentage, since the tool monitors application health from inside the container while Prometheus collects data about the application from the orchestration layer.

As future work, the data collected by the monitoring tools can be analyzed by machine-learning algorithms and trained models can be used to predict the VM health dynamically. The VM health logic can be incorporated with the Kubernetes load balancing algorithm or any other orchestration engine algorithms to enhance the POD-scheduling mechanism. For example, dynamic "taints" can be used to stop scheduling PODs on high usage VMs. Also, the VM affinity checker can be coupled with the Kubernetes load balancing algorithm to dynamically shift PODs to different VM host which is in a stable state, after a certain threshold of affinity for the current host has been reached. Additionally, developing a user-management system is another notable future work that can be incorporated as part of the service-oriented architecture, to notify admin users in case of deteriorating cluster health.

REFERENCES

[1] NIST Cloud Computing Standards Roadmap, NIST Special Publication 500-291, Version 2, July 2013, National Institute of Standards and Technology, U. S. Department of Commerce [Online] Available: https://www.nist.gov/sites/default/files/documents/itl/cloud/NIST_SP-500-291_Version-2_2013_June18_FINAL.pdf

[2] K. Alhamazani *et. al,* "An overview of the commercial cloud monitoring tools: Research Dimensions, Design Issues, and State-of-the-Art", *J. Computing,* vol. 97, 2014, DOI: 10.1007/s00607-014-0398-5,[Online] Available:https://www.researchgate.net/publication/265053448_An_Overview_of_the_Commercial_Cloud_Monitoring_Tools_Research_Dimensions_Design_Issues_and_State-of-the-Art

[3] G. Rodrigues *et. al,* "Monitoring of cloud computing environments: concepts, solutions, trends, and future directions" presented at 31st Annual ACM Symp. April,2016 pp. 378-383, DOI: 10.1145/2851613.2851619,[Online]Available: : https://www.researchgate.net/publication/303773193_Monitoring_of_cloud_computing_environments_concepts_solutions_trends_and_future_directions

[4] Cloud Computing Service Metrics Description, NIST Special Publication 500-307, April 2018, NIST Cloud Computing Program, National Institute of Standards and Technology, U.S. Department of Commerce , [Online] Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-307.pdf

[5] Q. Zhang, L. Cheng, R. Boutaba, "Cloud computing: state-of-the-art and research challenges", *J. Internet Serv. Appl.,*2010, vol.1, pp. 7-18, DOI : 10.1007/s13174-010-0007-6, [Online] Available: https://www.researchgate.net/publication/225252747_Cloud_Computing_State-of-the-art_and_Research_Challenges

[6] J. Shao *et. al,* "A runtime model based monitoring approach for cloud" in *2010 IEEE 3rd Int. Conf. on Cloud Computing, CLOUD 2010*, pp. 313-320, Miami, FL, USA, 2010 DOI: 10.1109/CLOUD.2010.31,[Online]Available: https://www.researchgate.net/publication/221399872_A_Runtime_Model_Based_Monitoring_Approach_for_Cloud

[7] H. Huang, L. Wang, "P&P: A combined push-pull model for resource monitoring in cloud computing environment" in *2010 IEEE 3rd Int. Conf. on Cloud Computing, CLOUD 2010,* pp. 260-267, DOI:10.1109/CLOUD.2010.85, [Online] Available: https://www.researchgate.net/publication/221399942_PP_A_Combined_Push-Pull_Model_for_Resource_Monitoring_in_Cloud_Computing_Environment

[8] D. Tovarňák, T. Pitner, "Towards multi-tenant and interoperable monitoring of virtualmachines in cloud", presented at 14th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012 pp. 436-442, 2012, DOI: 10.1109/synasc.2012.55[Online] Available: https://ieeexplore.ieee.org/abstract/document/6481063

[9] M.Dhingra, J. Lakshmi, S.Nandy, "Resource usage monitoring in clouds" in 2012 ACM/IEEE 13th Int. Conf. (GRID) , 2012, pp. 184-191. DOI: 10.1109/Grid.2012.10, [Online] Available:https://www.researchgate.net/publication/261088030_Resource_Usage_Monitoring_in_Clouds

[10] M. Andreolini, M. Pietri, M. Colajanni, "A scalable architecture for real-time monitoring of large information systems" in *IEEE 2nd Symp. on Network Cloud Computing and Appl., NCCA 2012*, pp. 143-150, DOI: 10.1109/NCCA.2012.24, [Online] Available: https://www.researchgate.net/publication/261277642_A_Scalable_Architecture_for_Real-Time_Monitoring_of_Large_Information_Systems

[11] J. Calero, J. Aguado, "MonPaaS: An adaptive monitoring platform as a service for cloud computing infrastructures and services" , *J. IEEE Trans. on Services Computing*, vol. 8, pp 1-1, DOI:10.1109/TSC.2014.2302810[Online]Available: https://www.researchgate.net/publication/270791480_MonPaaS_An_Adaptive_Monitoring_Platform_as_a_Service_for_Cloud_Computing_Infrastructures_and_Services

[12] P. Skvortsov *et. al* "Monitoring in the clouds: comparison of ECO2Clouds and EXCESS monitoring approaches", 2016, [Online] Available: https://www.researchgate.net/publication/301874389_Monitoring_in_the_Clouds_Comparison_of_ECO2Clouds_and_EXCESS_Monitoring_Approaches

[13] K. Fatema *et. al* “A survey of cloud monitoring tools: taxonomy, capabilities and objectives”, *J. of Parallel and Distributed Computing,* 2014, vol. 74, DOI: 10.1016/j.jpdc.2014.06.007, [Online] Available: https://www.researchgate.net/publication/263774524_A_survey_of_Cloud_monitoring_tools_Taxonomy_capabilities_and_objective

[14] J. Simonsson, L. Zhang, B. Morin et. al “Observability and Chaos Engineering on System Calls for Containerized Applications in Docker”, 2019, KTH Royal Institute of Technology, Stockholm, Sweden.

[15] Moses et. al, “Shared Resource Monitoring and Throughput Optimization in Cloud Computing data centers”, in *IEEE 25th International Parallel and Distributed Processing Symp.,*2011, pp.1024-1033,DOI:10.1109/IPDPS.2011.98[Online]Available: https://www.researchgate.net/publication/224257776_Shared_Resource_Monitoring_and_Throughput_Optimization_in_Cloud-Computing_Datacenters

[16] S. Padhy et. al, “Trustworthy and resilient monitoring system for cloud infrastructures”, in *Proceedings of the Workshop on Posters and Demos Track,* 2011, pp 3.1-3.2 , DOI: 10.1145/2088960.2088963 [Online] Available: https://www.researchgate.net/publication/236158667_Trustworthy_and_resilient_monitoring_system_for_cloud_infrastructures

[17] Veeraraghavan et. al, “Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently”, in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’18), 2018, Carlsbad, CA, USA, [Online] Available: https://www.usenix.org/conference/osdi18/presentation/veeraraghavan

[18] D. Yuan, Y. Luo, X. Zhuang et. al, “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems”, in 11th USENIX Symposium on Operating Systems Design and Implementation, 2014, Broomfield, CO, USA, [Online] Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[19] L. Zhang et. al, “TripleAgent: Monitoring, Perturbation and Failure-Obliviousness for Automated Resilience Improvement in Java Applications”, *in IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 116-127, DOI: 10.1109/ISSRE.2019.00021 [Online] Available: https://www.researchgate.net/publication/339174418_TripleAgent_Monitoring_Perturbation_and_Failure-Obliviousness_for_Automated_Resilience_Improvement_in_Java_Applications

[20] N. Magdelaine, T. Ahmed, G. Amato, "Demonstration of an Observability Framework for Cloud Native Microservices", in *IFIP/IEEE Symposium on Integrated Network and Service Management(IM)*,2019,pp.722–724[Online]Available: https://ieeexplore.ieee.org/document/8717923

[21] M. De Carvalho et. al, "A Cloud Monitoring Framework for Self-Configured Monitoring Slices Based on Multiple Tools", in Proceedings of *9th International Conference on Network and Service Management (CNSM),* 2013, pp. 180-184, DOI: 10.1109/CNSM.2013.6727833 [Online] Available: https://www.researchgate.net/publication/260002372_A_Cloud_Monitoring_Framework_for_Self-Configured_Monitoring_Slices_Based_on_Multiple_Tools