A REALISTIC AND RELIABLE APPROACH FOR REAL TIME MONITORING OF INFRASTRUCTURE

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Rakhen Garg

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

September 2020

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A REALISTIC AND RELIABLE APPROACH FOR REAL TIME MONITORING OF INFRASTRUCTURE

By

Rakhen Garg

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig
Chair

Dr. Pratap Kotala

Dr. Jacob Glower

Approved:

September 30, 2020	Dr. Simone Ludwig
Date	Department Chair

ABSTRACT

Rapid advancements in the field of site reliability engineering in the past few decades have made it an indispensable part of how humans create scalable and highly reliable software systems. The inherent need of a reliable site Infrastructure Monitoring System brings with it a great set of advantages in terms of monitoring availability, latency, performance and efficiency of the services.

The main objective of this work is to design and implement an intuitive Real-Time Infrastructure Monitoring System (RIMS) that acts as a one-stop shop for keeping track of overall system’s health, services, downtimes and availability. RIMS extract a wide variety of machine log data and transforms it into consistent and presentable format for the end-users. Data-driven business decisions using RIMS monitoring technique were reached and 98.46% availability of Site Systems was captured using all-time data. The learning curve of RIMS aims for higher SLOs over a period of its functioning.

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to my graduate advisor Dr. Simone Ludwig, Department Chair, Computer Science, North Dakota State University, Fargo, for her motivation, patience and immense knowledge. Specifically, I want to thank her for being available whenever I required her help and guidance. I feel extremely fortunate to have had the opportunity to work with her.

I am equally grateful to my supervisory committee members Dr. Pratap Kotala, Senior Lecturer, Department of Computer Science, North Dakota State University, Fargo and Dr. Jacob Glower, Professor, Department of Electrical and Computer Engineering, North Dakota State University, Fargo for serving on the committee. I would also like to thank Department of Computer Science at North Dakota State University of Agriculture and Applied Science for providing necessary tools and resources that helped me in pursuing my research interests.

I would like to express my gratitude to all my family members for their immense support, love and motivation.

DEDICATION

To my parents Raman Kumar Garg, Kiran Garg and lovely sister Sheena Garg

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

RIMS	Real-Time Infrastructure Monitoring System
YAML	Yet Another Markup Language
SPL	Search Processing Language
HTTP	Hypertext Transfer Protocol
SLO	Service Level Objective
PromQL	Prometheus Query Language
SLA	Service Level Availability
KPI	Key Performance Indicators
API	Application Programming Interface
TSDB	Time Series Database
SA	Site Availability
SS	Site Systems

1. INTRODUCTION

In this era of rapid changes in Software Development methodologies, an integrated approach of monitoring Infrastructure during DevOps software development life cycle remains a challenge. The paper has been focused towards design and implementation of a Real-Time Infrastructure Monitoring System (RIMS) which aims at easy and efficient monitoring of Infrastructure. The system helps track planned and unplanned downtimes of site clusters that are being used in DevOps software development by their geographical location.

The implementation of RIMS consists of a user interface which allows end-users to monitor the Infrastructure through Site Systems and Site Availability dashboards. The main component of RIMS is the log data extraction process. An integrated solution of Prometheus and Ansible on site systems scrapes data from the build systems through HTTP 500 requests and Ansible playbooks. The scrapped data is processed as logs to Splunk. Key Performance Indicators are studied and Service Level Objectives (SLO) are identified for determining the percentage values of Site Availability of the Site Systems.

The rest of the paper is organized as follows: Chapter 2 discusses about our objectives and gives an overview of earlier work that has been done in realms of Infrastructure Monitoring. Chapter 3 describes the architecture, key components, log data extraction process and the key challenges with data storage of RIMS. Chapter 4 discusses about monitoring and highlights key decisions that were made from Site Availability and Site Systems dashboards.

2. RELATED WORK

Adoption of DevOps as a software development methodology has its own set of challenges when it comes to understanding and monitoring the underlying infrastructure of the applications. One such challenge is the extensive possibility of having a visualized on-premise and cloud infrastructure and the other is enlarged usage of configuration management of systems for transparent development operations.

An integrated set of DevOps tools for monitoring has the power to improve visibility and productivity, achieve higher-performing systems, and establish cross-functional collaboration. An effective monitoring tools improves system performance and productivity and helps reduce or even eliminate unplanned downtimes.

Defining and calculating SLA and SLO are increasingly becoming the key criteria for service selection [1]. DeVSi [2] an Infrastructure Monitoring Solution can monitor a single cluster. It targets scalability, efficiency and reliability but not availability of Infrastructure systems for tracking planned and unplanned downtimes. The system lacks the capability to monitor and manage an environment with multiple clusters at different geographical locations. It does not define its own SLA instead it assigns a cost function to calculate SLA which further defines SLO.

2.1. Google Theory of Availability Levels

Google [4] defines SLA on an hourly, day, weekly, monthly, quarterly and yearly basis as a desired level of service availability Table 2.1. and this is usually expressed in terms of the number of "nines" 99.9%, 99.99%, 99.999% but that's not the ground reality.

Table 2.1. Google Site Availability Projections [4]

Availability Level	Allowed Unavailability Window					
	Per year	Per quarter	Per month	Per week	Per day	Per hour
90%	36.5 days	9 days	3 days	16.8 hours	2.4 hours	6 minutes
95%	18.25 days	4.5 days	1.5 days	8.4 hours	1.2 hours	3 minutes
99%	3.65 days	21.6 hours	7.2 hours	1.68 hours	14.4 minutes	36 seconds
99.5%	1.83 days	10.8 hours	3.6 hours	50.4 minutes	7.20 minutes	18 seconds
99.9%	8.76 hours	2.16 hours	43.2 minutes	10.1 minutes	1.44 minutes	3.6 seconds
99.95%	4.38 hours	1.08 hours	21.6 minutes	5.04 minutes	43.2 seconds	1.8 seconds
99.99%	52.6 minutes	12.96 minutes	4.32 minutes	60.5 seconds	8.64 seconds	0.36 seconds
99.999%	5.26 minutes	1.30 minutes	25.9 seconds	6.05 seconds	0.87 seconds	0.04 seconds

Defining and creating first own set of SLOs is important for development Infrastructure [5]. SLOs can be defined over various time intervals and can either use a rolling window.

2.2. Objective

Traditionally, monitoring has been a function performed by humans, but ideally, monitoring should be a function driven by machine learning and algorithms. Eliminating the need for human intervention and remediation, a continuous Infrastructure Monitoring Solution is the need of the hour. The objective is to design and implement an infrastructure monitoring mechanism that helps in understanding and identification of Key Performance Indicators from machine data for calculating the Infrastructure availability and defining SLO. The system is developed by writing up the code in Ansible Playbooks and YAML files, along with defining the rules in Prometheus files for scrapping machine metrics for calculating site uptime and downtimes. The integration of system is achieved by designing Splunk Processing Language (SPL) queries for understanding the logs from machine data and designing dashboards for monitoring the Infrastructure. Since monitoring could have also been done using Prometheus Query Language (PromQL) but Prometheus stores data only on local machines, which instead limits disk space. Due to long-term capacity planning and for calculating SA a longer retention period of data is desirable, I chose Splunk as the most effective solution for storing, searching and visualizing the data.

3. COMPONENTS AND IMPLEMENTATION OF RIMS

The technical structure of Real-Time Infrastructure Monitoring System (RIMS) can be classified into two segments. The first segment is comprised of a user interface and the second segment is the data extraction setup and its process for log monitoring. The user interface is a web-based medium designed on Splunk tool that lets user makes selection of time for displaying the site availability, downtimes and other health related statistics displayed in a readable and user-friendly format.

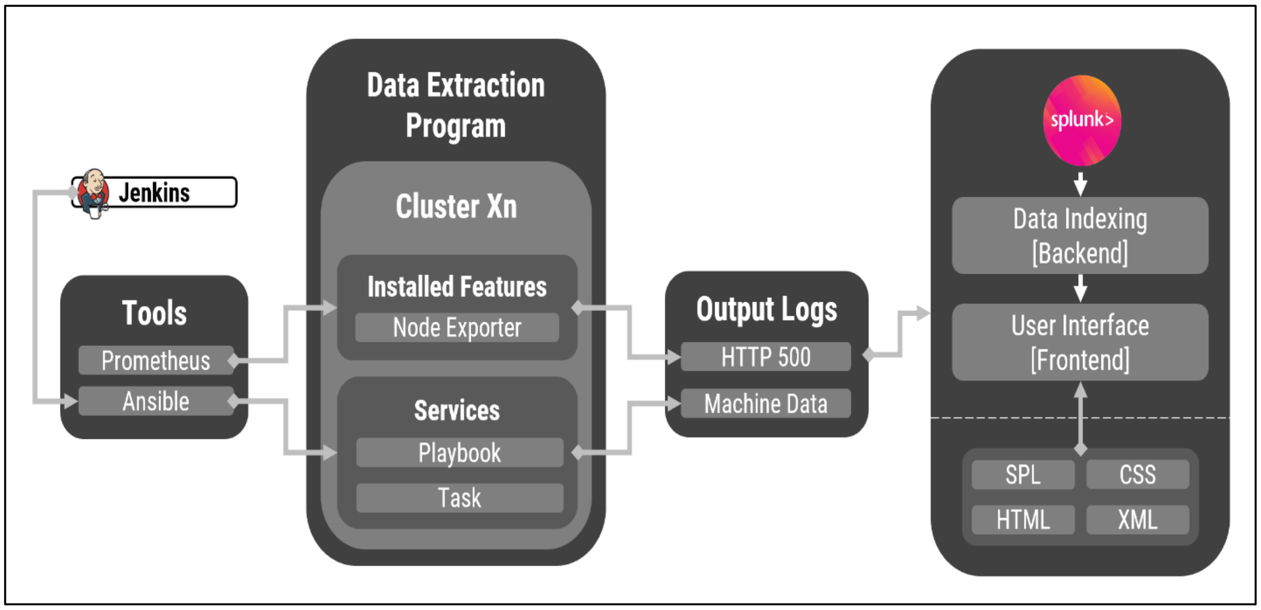


Figure 3.1. Architecture of RIMS

The data extraction program leverages the power of Ansible and Prometheus. Both are open source tools that have been configured for mining the data from logs generated from various build systems. Infrastructure systems composition data is gathered through configuring Ansible on site systems. Site availability log data that is extracted using Prometheus. Data is then indexed through Splunk into a time-series mechanism and then logs are analyzed and processed using Splunk's Search Processing Language (SPL) to design the user interface of RIMS.

Section 3.1 briefs about the environment used to build the system. Section 3.2 discusses about the user interface and Section 3.3 discusses about the process of data extraction program for mining the logs that are of interest for determining the Key Performance Indicators required for monitoring the infrastructure. Section 3.4, 3.5 and 3.6 discusses about log data storage challenges, data scrapping intervals and alerting rules respectively. Figure 3.1 illustrates the proposed architecture of RIMS.

3.1. Environment Used to Build the Monitoring System

The environment used to develop the user interface for RIMS is Splunk. Splunk eases the process for searching, monitoring and analyzing the machine-generated big data via a Web-Style user friendly interface. The main reason for choosing Splunk is that it easily captures, indexes and corelates real-time data in a searchable repository from which it allows Machine Learning engineers to generate graphs, reports, alerts, dashboards and visualizations. Splunk helps in easy identification of data patterns, diagnosing problems, and providing business cum operational intelligence through web analytics.

The underlying design technique behind user interface is Splunk Processing Language (SPL), Extensible Marked Language (XML), Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript.

XML is a markup language that provides the capability of software and hardware independent way of storing, transporting and sharing the data which is both human-readable and machine-readable. XML is chosen because the underlying technique for data extraction is based on XML HTTP data requests sent and received for gathering data from build systems. Capabilities of HTTP API's via Prometheus are used as a source of data extraction process from build systems.

3.2. The User Interface

The User Interface provides user a cylindrical view of two buckets that lets user visualize Site Availability and Site Systems data. Figure 3.2 shows the entry point from where the user makes his selections for viewing either Availability or Systems dashboards. Interface is intended to be very simple and easy. Additional capabilities in dashboards provides users with various drilldowns. Home button, time-range selection are other such additional features.

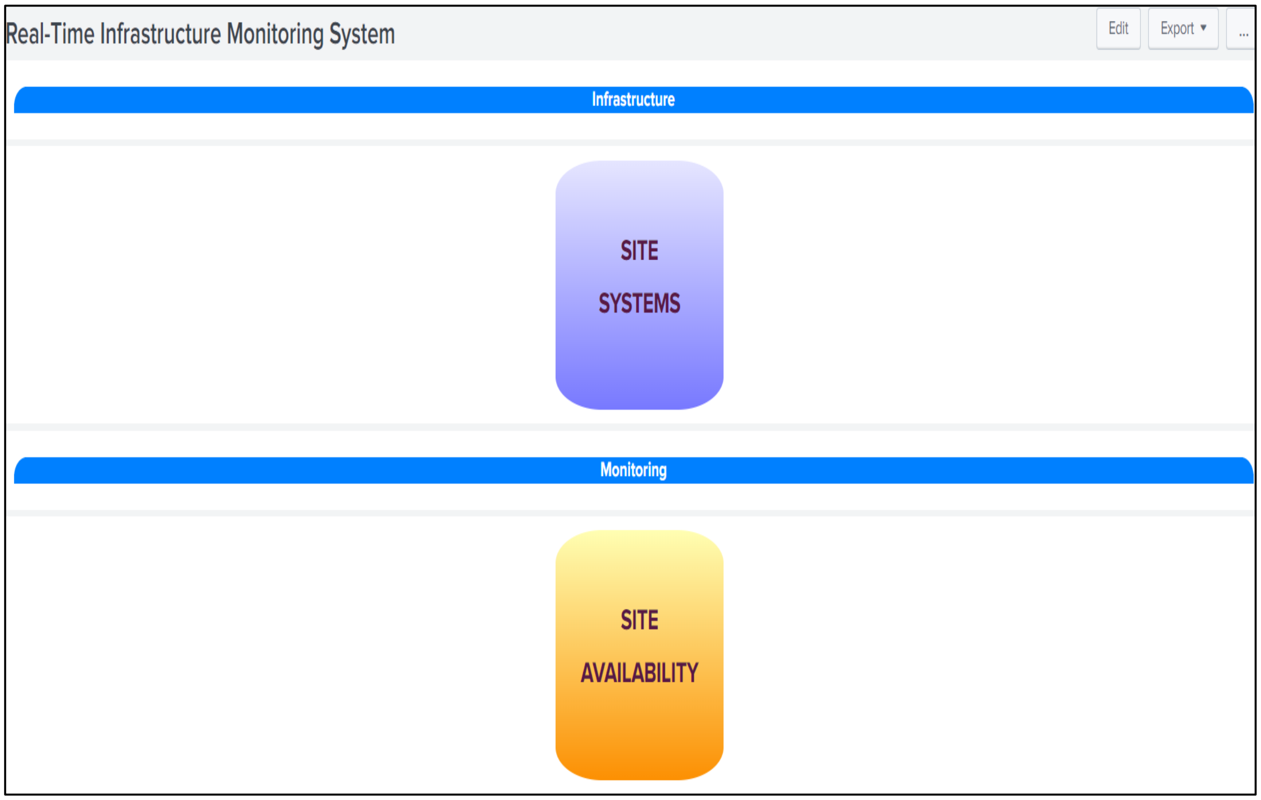


Figure 3.2. RIMS User Interface for Entry Point

Dropdown menus in Site Systems dashboard provides options for selecting site clusters that projects Key Performance Indicators of the systems in a cluster from all geographical locations. KPI’s includes Memory Used, Memory Available, Cluster Location, Hostname, IP Address, Processor Cores, CPUs, Threads Per Core. More details are discussed in Chapter 4.

Dropdown menus in Site Availability dashboard lists various options for selecting real-time SA data. For instance, a user can select data for last 7 days or any random date or time range and get meaningful insights from overall Key Performance Indicators that are being used for monitoring different services pertaining to infrastructure health, availability and glitches. Default time range for data is set to be as last 24 hours. Figure 3.3 shows the time selection input board for the last 7 days. Figure 3.4 shows the time selection input board for the specified date and time range.

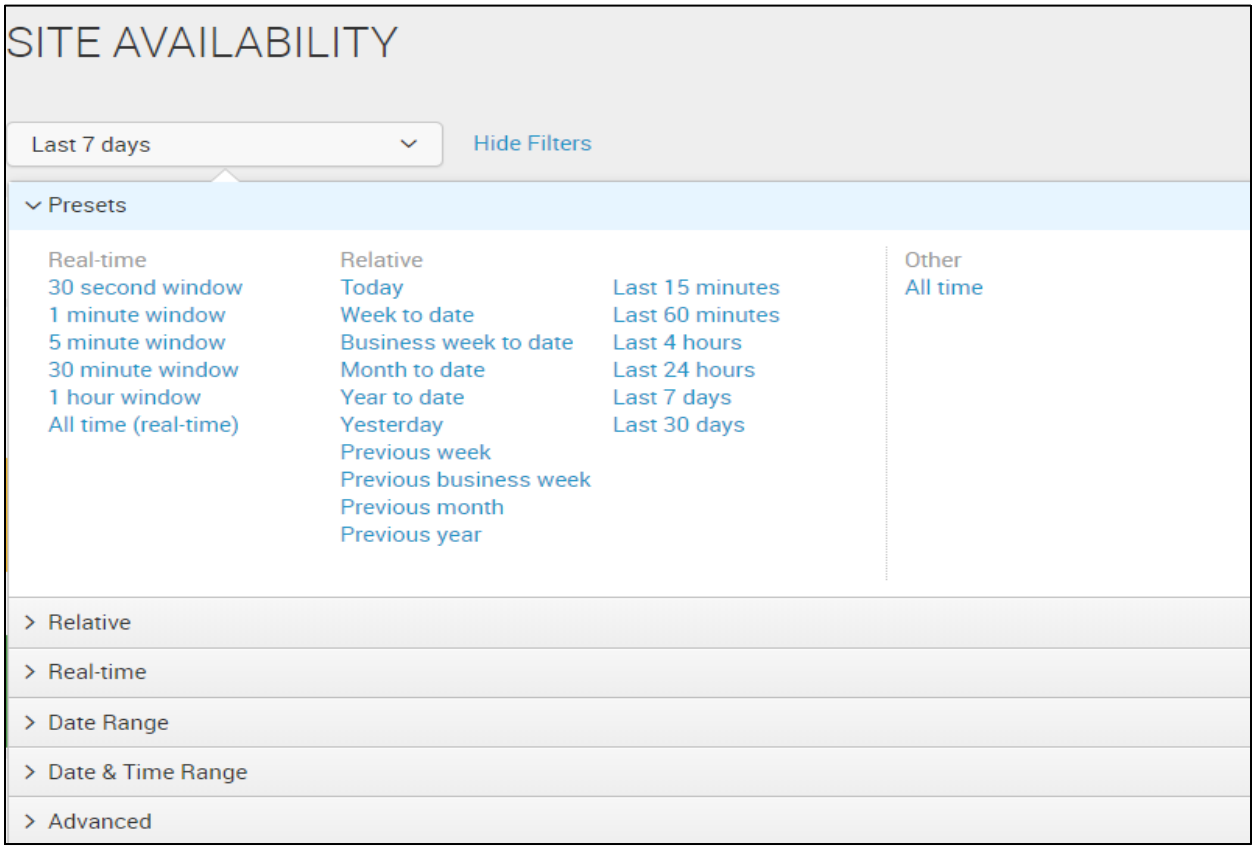


Figure 3.3. RIMS User Interface for Relative Time Selection

Once the user makes the selection of time from the dropdown menu and hits the submit button, user will be able to see the dashboard form that displays the percentage values and total instances of each cluster at a geographical location. Service Level Availability (SLA) value 100%

states that all the hosts within a build system are up and running without any glitches for the selected time range from the dropdown menu.

SLA values less than 100% and greater than or equal to 90% indicates there were downtimes. Any SLA value less than 90% RIMS defines it as a major failure. Downtimes can be planned or unplanned. RIMS cater to both types of downtimes.

HTTP 500 requests data is available for all the hosts within a build system. The user interface allows users to track unresponsive requests from hosts that were not serviced and helps users gain meaningful insights into data through graphical charts as a visualization strategy.

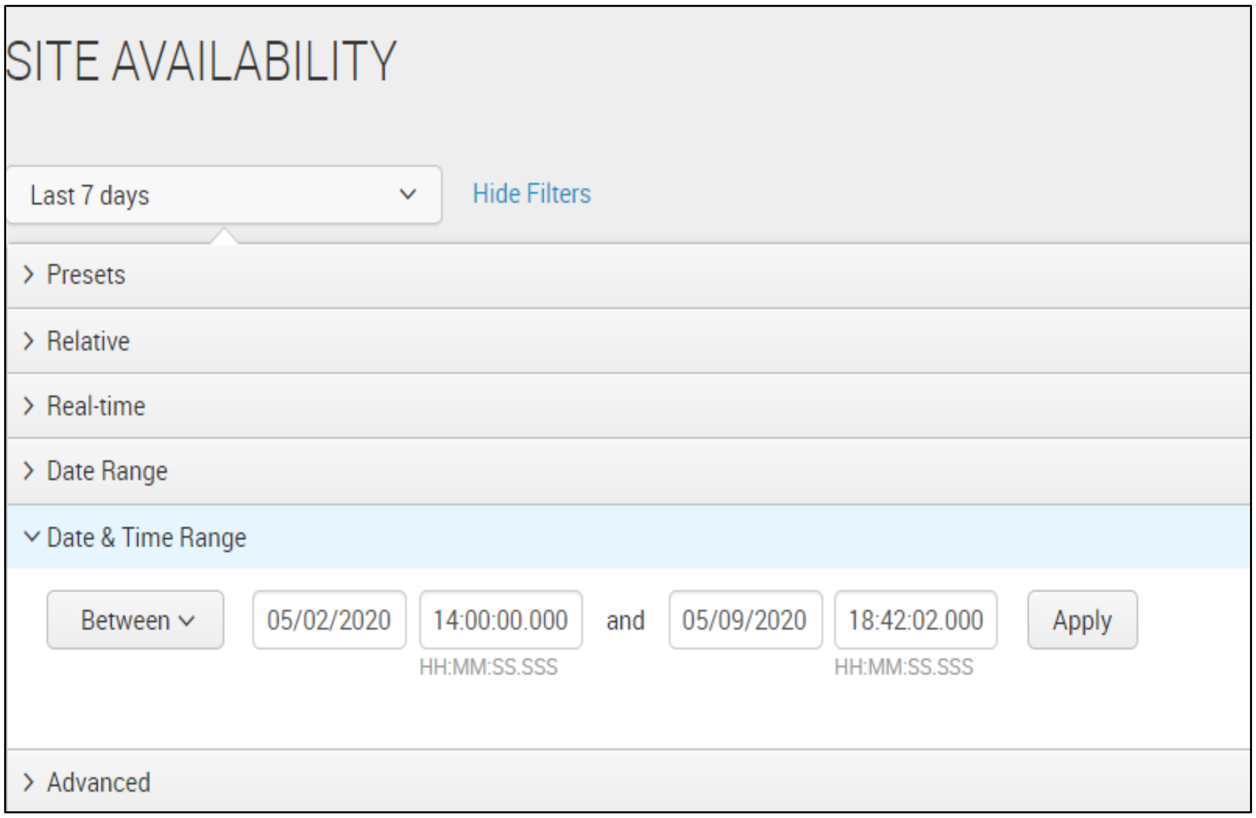


Figure 3.4. RIMS User Interface for Date & Time Range Selection

The user interface gives flexibility to the users to re-arrange default displayed panels and add new visualizations for tracking meticulous glitches in real-time or historical data. Figure 3.5–3.6 shows the downtimes that were tracked on a site system in the month of April and June.

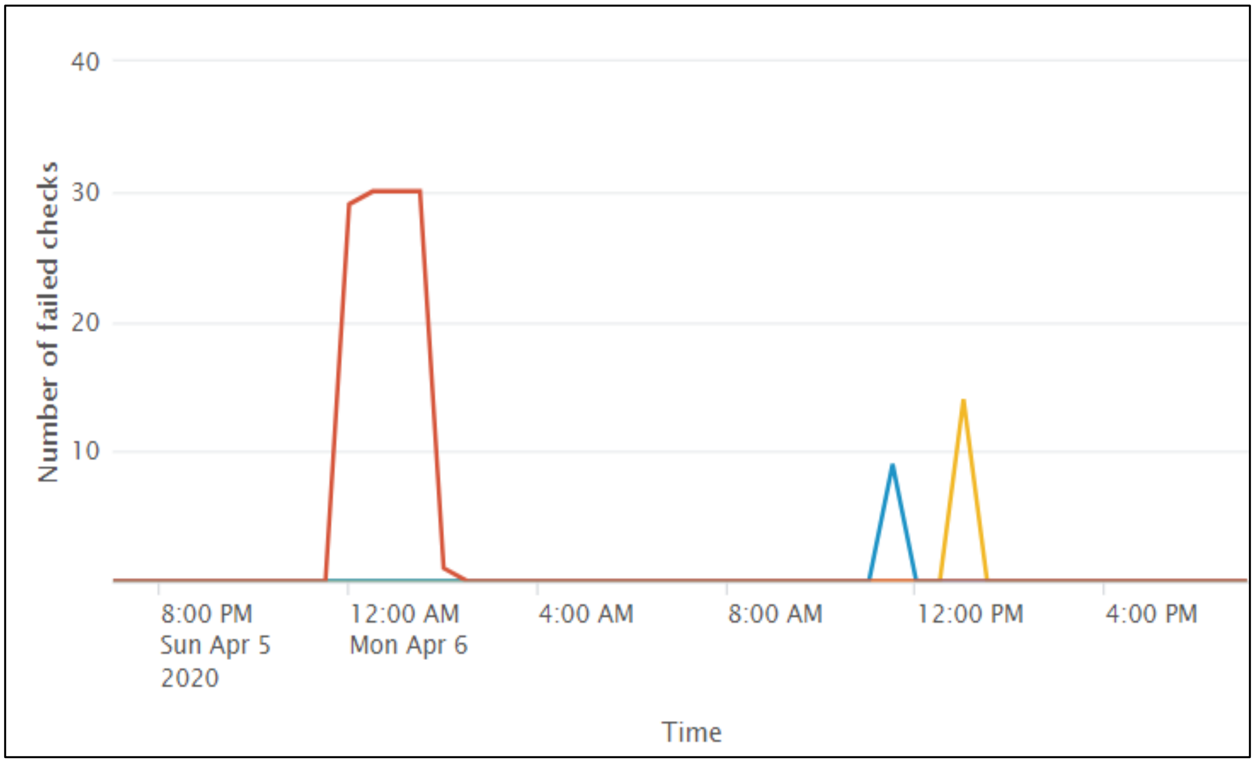


Figure 3.5. RIMS Line Chart for Downtimes in April

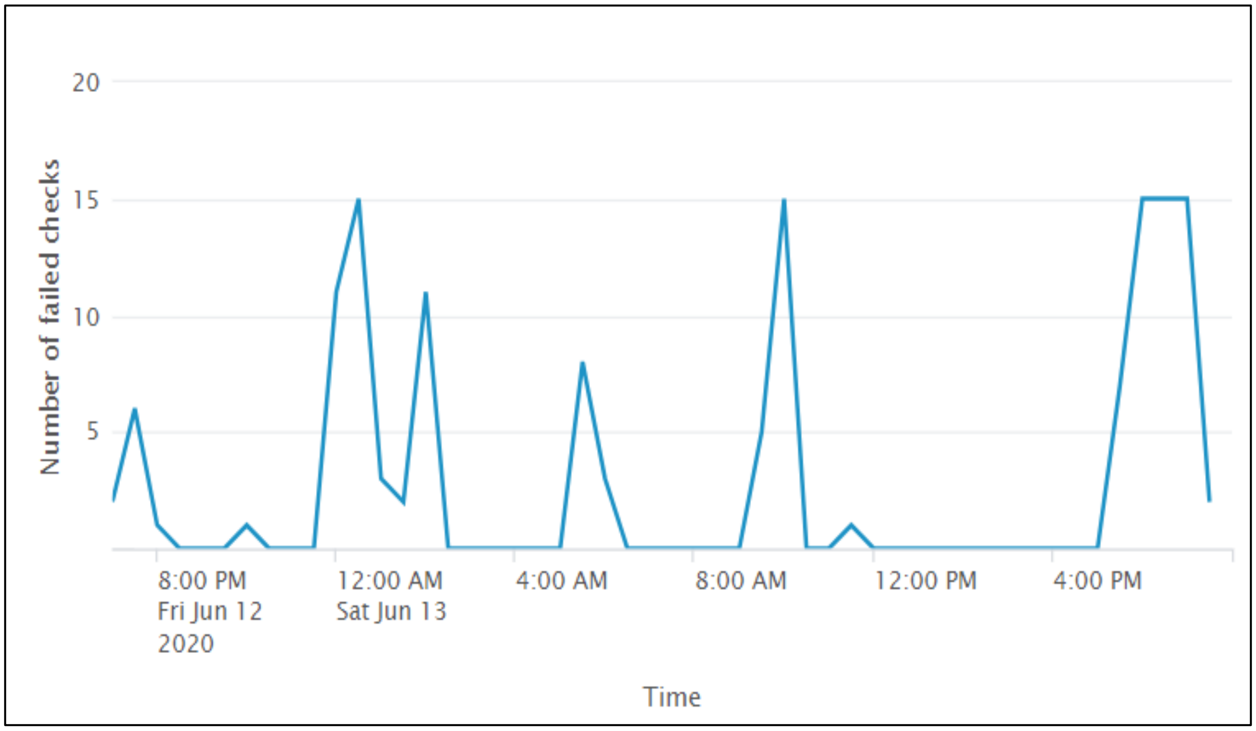


Figure 3.6. RIMS Line Chart for Downtimes in June

3.3. Implementation of Log Data Extraction Workflow

The log data extraction workflow forms the core part that drives the functioning of RIMS. Enabling Prometheus and Ansible on remote site systems serves to be as the main components for scraping the data. Configuration of Prometheus on remote systems helps in scrapping of HTTP 500 requests data whereas system metrics such as Memory Used, Memory Available, Cluster Location, Hostname, IP Address, Processor Cores, CPUs, Threads Per Core are scrapped using Ansible and fed through a timely Jenkins job as logs to the Splunk.

Dynamic analysis of logs is done in Splunk and meaningful information is concatenated and captured in the form of queries using Splunk Processing Language (SPL) for Site Systems and Site Availability dashboards.

Section 3.3.1 gives details on Prometheus features, components and configuration. Section 3.3.2 gives details on core Ansible features, components and configuration. Figure 3.7 illustrates the architecture of Prometheus; Figure 3.8 illustrates the architecture of Ansible and gives us a detailed flow of events on how metrics are scrapped from remote systems.

3.3.1. Prometheus Log Data Extraction Strategy for Site Availability

The log data extraction strategy using Prometheus for RIMS can be categorized into four segments which are:

1) Prometheus Features and Components
2) Architecture of Prometheus
3) Node Exporter Metrics
4) Prometheus Configuration

3.3.1.1. Prometheus Features and Components

Prometheus is an open-source system monitoring and alerting tool. The main features of Prometheus are:

1) Prometheus offers a multi-dimensional data model with time series data identified by metric name and key-value pairs
2) Prometheus does not rely on distributed storage
3) Single server nodes are autonomous
4) Pushing time series is supported via an intermediary gateway

Prometheus ecosystem consists of multiple components, many of which are optional. The components are:

1) The main Prometheus server scraps and stores time series data
2) Provision of client libraries for instrumenting application code
3) A push gateway for supporting short-lived jobs

3.3.1.2. Architecture of Prometheus

Prometheus scrapes data from instrumented jobs either directly or via an intermediary push gateway. It stores all scraped samples locally and runs rules over this data to either aggregate or record new time series from existing data.

Prometheus works well for recording numeric time series. It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures. In a world of microservices, its support for multi-dimensional data collection and querying is a strength.

Figure 3.7 illustrates the architecture of Prometheus and its ecosystem components.

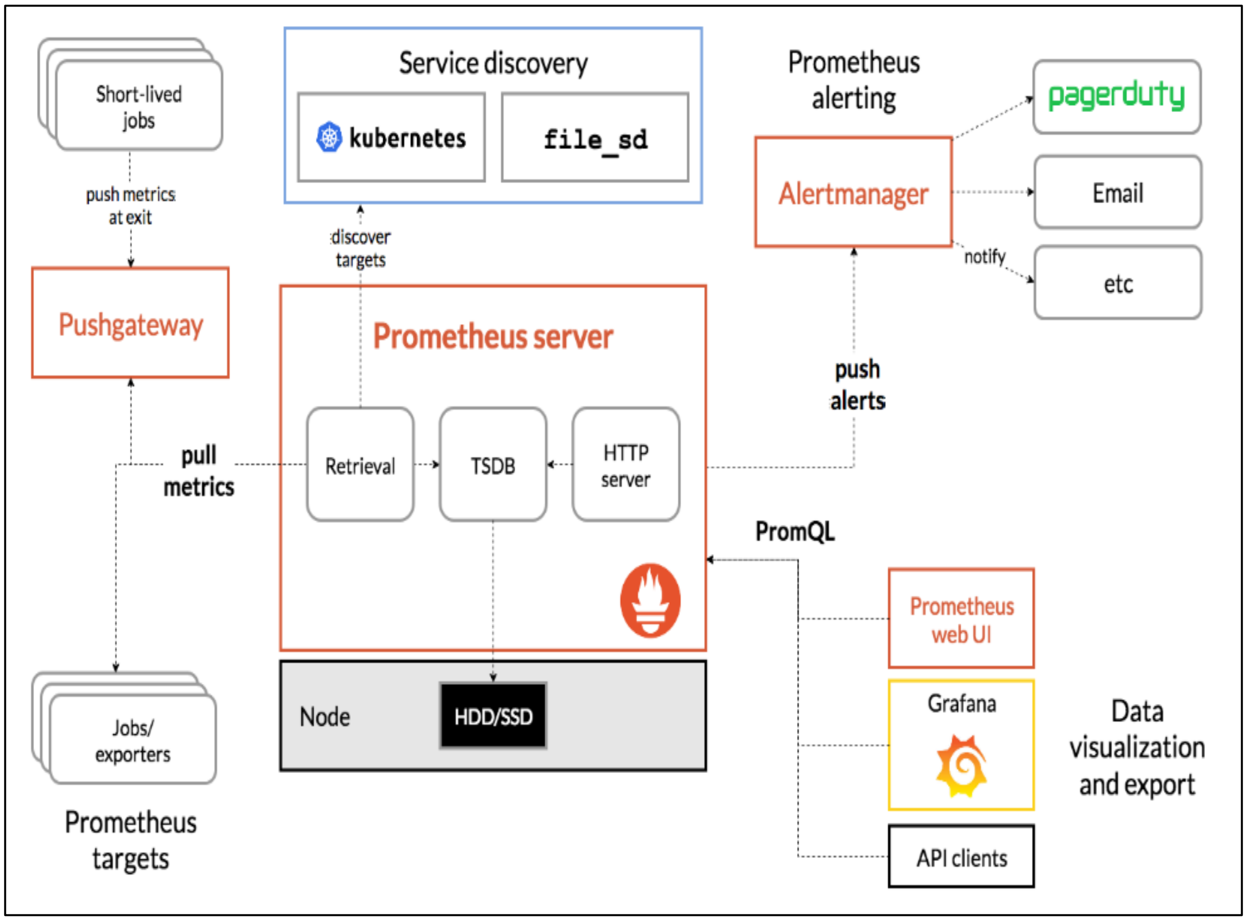


Figure 3.7. Architecture of Prometheus [6]

The design of Prometheus is highly focused towards achieving reliability. Prometheus allows end-users to quickly track outages and diagnose infrastructure problems. Irrespective of network storage or other remote services each Prometheus server is standalone. At times when some parts of the infrastructure are non-functional, Prometheus is a highly reliable service. Also, there is not a need to set up extensive infrastructure for using it.

3.3.1.3. Node Exporter Metrics

The Prometheus Node Exporter is installed on each of RIMS cluster Linux hosts through which a wide variety of hardware and kernel-related metrics are exposed, which further are visualized through Splunk. The first step that draws our attention is installing and running Node Exporter. Below are the steps for its installation.

Step1: Extracting and running Node Exporter Package on a host.

```
wget https://github.com/prometheus/node_exporter/releases/download/v*/node_exporter-*.*-
amd64.tar.gz

tar xvfz node_exporter-*.*-amd64.tar.gz

cd node_exporter-*.*-amd64

./node_exporter
```

Output indicating that the Node Exporter is running and exposing metrics on port 9100

```
INFO[0000] Starting node_exporter (version=0.16.0, branch=HEAD,
revision=d42bd70f4363dced6b77d8fc311ea57b63387e4f)  source="node_exporter.go:82"

INFO[0000]    Build    context    (go=go1.9.6,    user=root,    date=20200515-15:53:28)
source="node_exporter.go:83"cd node_exporter-*.*-amd64

INFO[0000] Enabled collectors:                    source="node_exporter.go:90"

INFO[0000]  - boottime                            source="node_exporter.go:97"

INFO[0000] Listening on :9100                     source="node_exporter.go:111"
```

Step2: Once the Node Exporter is installed and running you can verify the metrics that are being exported by curling the metrics endpoint.

```
curl http://localhost:9100/metrics
```

Expected output of curl command

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations

go_gc_duration_seconds{quantile="0"} 3.8996e-05

go_gc_duration_seconds{quantile="0.25"} 4.5926e-05

go_gc_duration_seconds{quantile="0.5"} 5.846e-05
```

Evident from the above output Node Exporter is now exposing metrics that Prometheus can scrape, including a wide variety of system metrics further down in the output (prefixed with node_). The same can be viewed using the below command.

```
curl http://localhost:9100/metrics | grep "node_"s
```

3.3.1.4. Prometheus Configuration

Prometheus instances needs to be properly configured in order to access node exporter metrics. The first step towards achieving this is Prometheus installation and starting it up followed by configuring the .yml file.

Step1: Prometheus Installation

```
wget https://github.com/prometheus/releases/download/v*/prometheus-*.*-amd64.tar.gz

tar xvf prometheus-*.*-amd64.tar.gz

cd Prometheus-*.*s
```

Step2: Prometheus Startup

```
./prometheus –config.file=./prometheus.yml
```

Step3: The following scrape_config block in a Prometheus.yml configuration file will tell the Prometheus instance to scrape metrics from the node exporter via localhost:9100.

```
Scrape_configs:
- job_name: 'RIMS-probes'
  metrics_path:/probe
  params:
    module: [http_500]
   static_configs:
    -targets:
      - 10.200.20.01
      - 10.200.20.02
      - 10.200.20.03
      - 10.200.20.04
      - 10.200.20.05
      # The above IP address are just for demo
labels:
       group: 'RIMS-probes'
```

3.3.2. Ansible Log Data Extraction Strategy for Site Systems

The log data extraction strategy using Ansible for RIMS can be categorized into two segments which are:

1) Ansible Features

2) Ansible Structure Components and Configuration

3.3.2.1. Ansible Features

Ansible is an open-source software provisioning, configuration management and application deployment tool that enables system administrators to manage and control multiple servers from one central location. Ansible runs on many Unix-like systems and can configure both Unix-like systems as well as Microsoft Windows. Ansible has its own declarative language for describing system configuration i.e.: YAML (Yet Another Markup Language). Ansible is an ideal choice when multiple and repetitive tasks are to be performed on multiple systems. The main features of Ansible are:

1) Ansible is agentless i.e.: Ansible does not deploy agents to nodes. This feature eliminates the need for having an additional agent pre-installed.
2) Ansible uses SSH as a transport medium that helps in remote connection via SSH allowing remote PowerShell execution to perform its tasks.
3) Eliminates the need for logging into each of the remote systems to carry out the tasks.
4) Ansible is Python based and very easily extendable.

3.3.2.2. Ansible Structure Components and Configuration

Figure 3.8 illustrates the architecture of Ansible and its connection with Splunk ecosystem components.

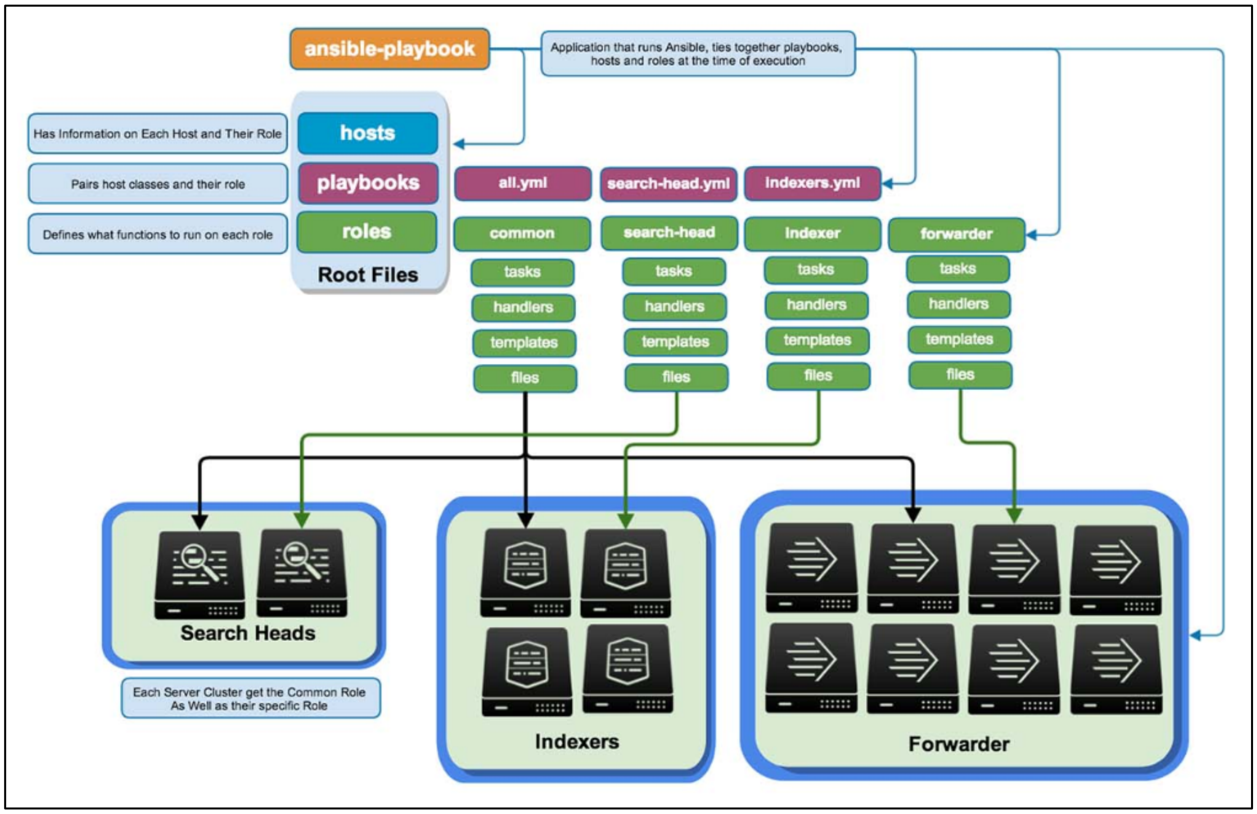

ansible-playbook
Application that runs Ansible, ties together playbooks, hosts and roles at the time of execution
Has Information on Each Host and Their Role
hosts
Pairs host classes and their role
playbooks
Defines what functions to run on each role
roles
Root Files
all.yml
search-head.yml
indexers.yml
common
search-head
indexer
forwarder
tasks
handlers
templates
files
Search Heads
Each Server Cluster get the Common Role As Well as their specific Role
Indexers
Forwarder

Figure 3.8. Architecture of Ansible [7]

Ansible ecosystem consists of multiple components. The components which are of interest for RIMS are:

Inventory: An inventory is a text file that contains a list of servers or nodes that system administrators intends to manage or configure. Default location for inventory file is /etc/ansible/hosts. An inventory file consists of IP addresses of remote systems. Below is an example:

```
11.200.20.01
```

Metrics identification of RIMS Site Systems has been accomplished by placing servers in a group named as RIMSSYSTEMS. Grouping makes the task easier as we can access remote systems using their group name not their IP addresses. This further simplifies the operation processes. RIMS cater to multiple groups with multiple servers clustered under specific group.

```
[RIMSSYSTEMS]

# The site systems are for  Sydney cluster

# The above IP address are just for demo

10.200.20.01

10.200.20.02

10.200.20.03

10.200.20.04

10.200.20.05
```

Playbook: A playbook is a set of configuration management scripts that define how tasks are to be executed on remote hosts or a group of host machines. The scripts or instructions are written in YAML format. Each playbook is composed of one or more ‘plays’ in a list. The goal of play is to map a group of hosts to some well-defined roles, represented by things ansible calls tasks. For a basic level understanding task is nothing more than a call to ansible module. Composing a playbook of multiple ‘plays’, it is possible to orchestrate multi-machine deployments, running certain steps on all machines in RIMSSYSTEMS group. Below steps gives more insights into creating a playbook and defining its tasks.

The below playbook ssh into remote systems defined as RIMSSYSTEMS in inventory file.

Step1: Creating a Playbook

```
$ touch RIMS_TOOL.yml
```

Step2: Structure of Playbook

```
---
- hosts: RIMSSYSTEMS
  remote_user: root
  gather_facts: True

  tasks:
  - name: ssh
    ping:
    remote_user: yourname
```

Hosts and Users: Each play in a playbook defines the machines in the infrastructure to target. The hosts line is a list of one or more groups or host patterns separated by colons. The remote_user is just the name of the user account. Facts from remote systems are gathered through gather_facts.

```
---
- hosts: RIMSSYSTEMS

  remote_user: root

  gather_facts: True
```

Modules: Modules are discrete units of code used in playbooks for executing commands on remote hosts or servers. Each module is followed by an argument. The basic format of module is key: value. Below represents YAML code snippet where name and ping are modules.

```
- name: ssh
  ping:
```

Task List: Each play contains a list of tasks that are executed in order, one at a time against all machines matched by the host pattern before moving onto the next task. If a task fails modification of playbook can be done by taking out the failed tasks from playbook and re-running it. The structure of task includes a name which gets included in the output from running the playbook. The output is human readable, so it is useful to provide a good description of each task. Below is an example demonstrating how a basic task looks like.

```
tasks:
- name: ssh
  ping
  remote_user: yourname
```

The end goal of each task is to execute a module with specific arguments. Variables can also be passed as an argument to modules. Below is an example demonstrating a variable vhost in the vars section and is passed as an argument to module.

```
tasks:
- name: ssh for {{ vhost }}
   ping:
   remote_user: yourname
```

Follwing the above hierarchy of steps I was able to successfully scrape mertics from all the cluster hosts from all the geographical locations. Challenges of log data read and write are discussed in next sections. Complete panel details of SA and SS are discussed in Chapter 4.

3.4. Remote End Points and Log Data Storage

The HTTP 500 requests data from site systems of our Infrastructure forms to be a core component of RIMS Site Availability dashboard. Prometheus shares the ability to directly interact with its time-series database storage using the remote API. These API's allows third party systems to interact with metrics data through two methods. The methods are discussed in Section 3.4.1 Fixing data scrape interval for RIMS is discussed in Section 3.6.

3.4.1. Remote Read and Write Data to RIMS

The remote write and remote read features of Prometheus allows seamlessly and transparently sending and receiving the samples of data. This is primarily intended for long term storage. Let us understand how remote API's data synchronization strategy works through an architectural diagram.

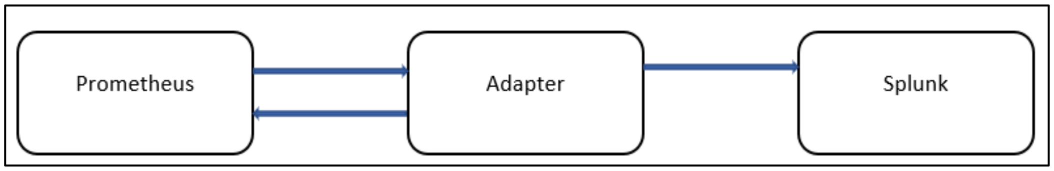


Figure 3.9. RIMS Remote Read Write Mechanism

- Write – receives samples pushed by Prometheus
- Read – pull samples from Prometheus

3.4.1.1. Remote Write to RIMS

In the growing world of Prometheus remote write is the most popular feature which is used to replicate Prometheus data into Splunk. A Prometheus Metrics for Splunk app has been installed on a forwarder to receive HTTP 500 requests log data. In remote write mode Prometheus streams samples of data and periodically sends a batch of samples to the given endpoints.

In RIMS Prometheus is sending a lot of performance, system and service information data which is further analyzed and visualized using the capabilities of Splunk for creating dashboards. Site Systems dashboard is developed by considering various Key Performance Indicators and the same has been discussed in Section 4.1.

Limiting the type and amount of data that is being sent through Prometheus write_relabel_configs have been configured to drop metrics that are not deemed necessary. Below are two such metrics node_boot_time_seconds, probe_ip_protocol that have been dropped.

```
remote_write:
  - url: “<Node URL>”
    bearer_token_file: /var/run/secrets/splunk-token/remote-rewrite-token
    write_relabel_configs:
      - source_labels: [__name__]
        regex: node_boot_time_seconds
        action: drop
    write_relabel_configs:
      - source_labels: [__name__]
        regex: probe_ip_protocol
        action: drop
```

3.4.1.2. Exclusion of Remote Read for RIMS

Remote read is a less common method. The key strategy behind remote read is that it allows querying Prometheus storage Time Series Database (TSDB) directly without Prometheus Query Language (PromQL) evaluation.

The remote read API exposes a simple HTTP endpoint that expects following protobuf payload:

```
message ReadRequest {
    repeated Query queries = 1;
}

Message Query {
    int64 start_timestamp_ms = 1;
    int64 end_timestamp_ms = 2;
    repeated prometheus.LabelMatcher matchers = 3;
    prometheus.Read hints = 4;
}
```

With this payload, the client can request certain series matching given matchers and time range with end and start. Below is the response.

```
message ReadResponse {
    repeated Query queries = 1;
}

message Sample {
    double value = 1;
    int64 timestamp = 2;
}

message TimeSeries {
    repeated Label labels = 1;
    repeated Sample samples = 2;
}

message QueryResult {
    repeated prometheus.TimeSeries timeseries = 1;
}
```

There are two key problems associated with remote read. Though remote read is easy to understand but there were no streaming capabilities within the single HTTP request for the protobuf format. Secondly the HTTP 500 request response included raw samples (float64 value and int64 timestamp) instead of an encoded, compressed batch of samples called “chunks” that are used to store metrics inside Time Series Database.

The server algorithm for remote read without streaming can be categorized into six unique steps:

1) Parse HTTP 500 request
2) Select metrics from Time Series Database
3) For all decoded series for all samples add to response protobuf
4) Marshal response
5) Snappy compress
6) Send back the HTTP 500 response

Now the whole response of the remote read had to be buffered in a raw, uncompressed format in order to marshal it in a potentially huge protobuf message before sending it to the client. The whole response message needs to be completely buffered in the client again to be able to unmarshal it from the received protobuf. Following this step, the client was able to use raw samples. Excessive memory utilization and time consumption limitations led us to choose remote write as a novel technique for scrapping the HTTP 500 data for RIMS.

3.5. Understanding Log Data Scrape Intervals for RIMS Site Availability

By default, Prometheus scrapes data from targets every minute. Considering one such example of long-distance targets in North America and Asia scrapping data from different geographical locations in a minute time frame is not a viable option for the functioning of RIMS.

As RIMS is scrapping logs from distant targets the Prometheus instance has been configured with a longer time interval and a higher timeout to avoid dropping scrapes from long distance targets. The section to change the intervals can be found in the values.yml under ‘server:’, then ‘global:’. The table below shows the default setting. Scrape interval, timeout and evaluation intervals are changed in later Section 3.6. for RIMS.

```
global:

  ## How frequently to scrape targets by default
  ##
  scrape_interval: 1m

  ## How long until a scrape request times out
  ##
  scrape_timeout: 30s

  ## How frequently to evaluate rules
  ##
  evaluation_interval: 1ms
```

3.6. RIMS Alerting Rules for Site Systems Availability

Operational intelligence through data-driven business decisions are important for the success of RIMS. The key feature for reaching this stand-off point is by defining alerting rules for the alert manager. Alerting rules allows us to define expressions using Prometheus expression language. On the occasion when an alert condition defined in the expression is met, an alert is recorded and send to the alert managers for tracking downtimes of RIMS. Referring Prometheus documentation below is the syntax for alerting rules.

```
alert: <string>

# The PromQL expression to evaluate. Every evaluation cycle this is
# evaluated at the current time, and all resultant time series become
# pending/firing alerts.
```

```
expr: <string>

# Alerts are considered firing once they have been returned for this long.
# Alerts which have not yet fired for long enough are considered pending.
[ for: <duration> | default = 0s ]

# Labels to add or overwrite for each alert.
labels:
  [ <labelname>: <tmpl_string> ]

# Annotations to add to each alert.
annotations:
   [ <labelname>: <tmpl_string>]
```

Alerting Rules for RIMS Site Availability

```
rules:
  groups:
     - name: RIMSSYSTEMS
  rules:
     - alert: InstanceDown
  expr: up == 0
  for: 5m
  labels:
    severity: critical
  annotations:
      description: '{{ $labels.instance }} of job {{ $labels.job }} has been down
  for more than 5 minutes.'
       summary: Instance {{ $labels.instance }} down
```

The current alert setup of RIMS shows that if the 'expr: up' does not equals to zero for five minutes an alert will be fired. As defined in Section 3.5. log data scrape intervals in Prometheus has been configured to check once a minute for changes. If a site system does not report as being 'up' and fails all the five checks, then alerts are sent to the alert managers. A reference table is present under Site Systems Dashboard section in Chapter 4.

4. RIMS AVAILABILITY AND SITE SYSTEMS DASHBOARDS

Determining a logistical and reliable solution for replicating the site systems data into Splunk for calculating Site Availability has been a major challenge for developing an integrated RIMS system. The integration of data with third party systems was possible with the help of remote API's.

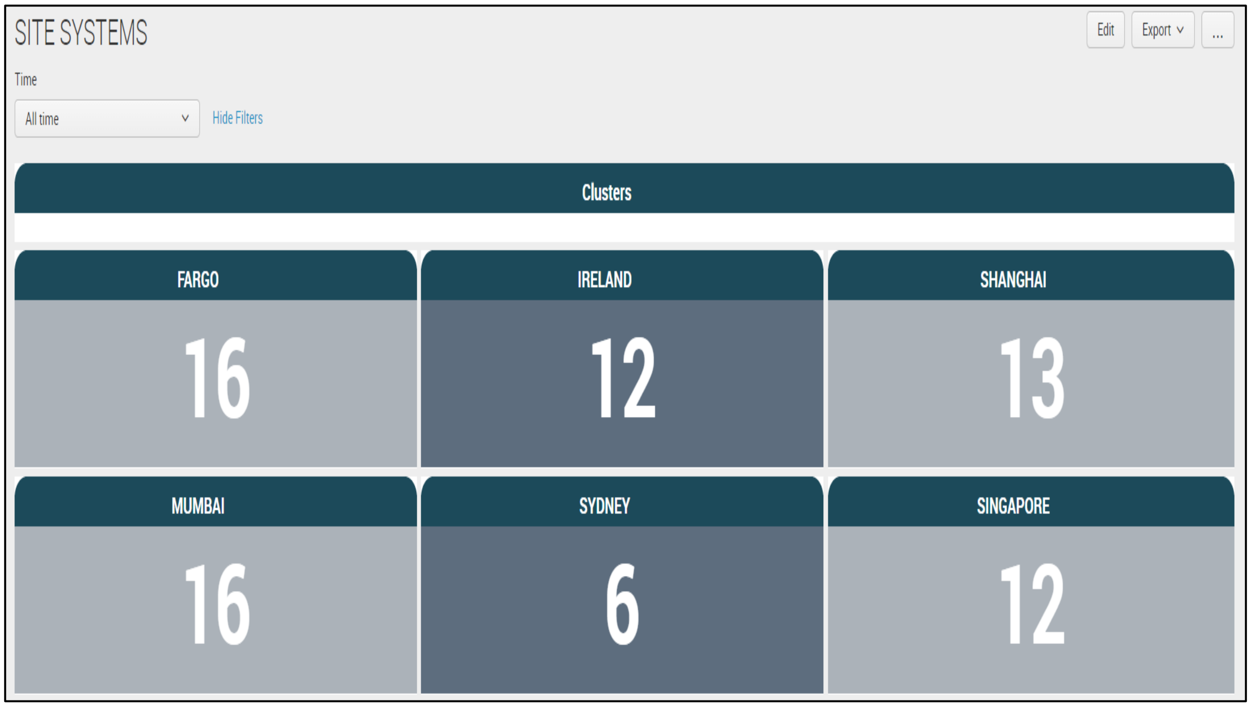
4.1. RIMS Site Systems Dashboard

Section 3.2 walks us through the user interface of RIMS. Rightful selection from cylindrical view takes user to the Site Systems dashboard. RIMS categorize Site Systems dashboard into four unique panels. Each panel gives end user a variety of machine related information pertaining to all the clustered Site Systems. The panels are:

1) Clusters
2) Cluster Details
3) Host Details
4) Systems by Profile and Location

Clusters: RIMS Site Systems dashboard scrapes machine related data from multiple infrastructure sites. Systems at each of the sites have been grouped into six different clusters at their respective geographical location.

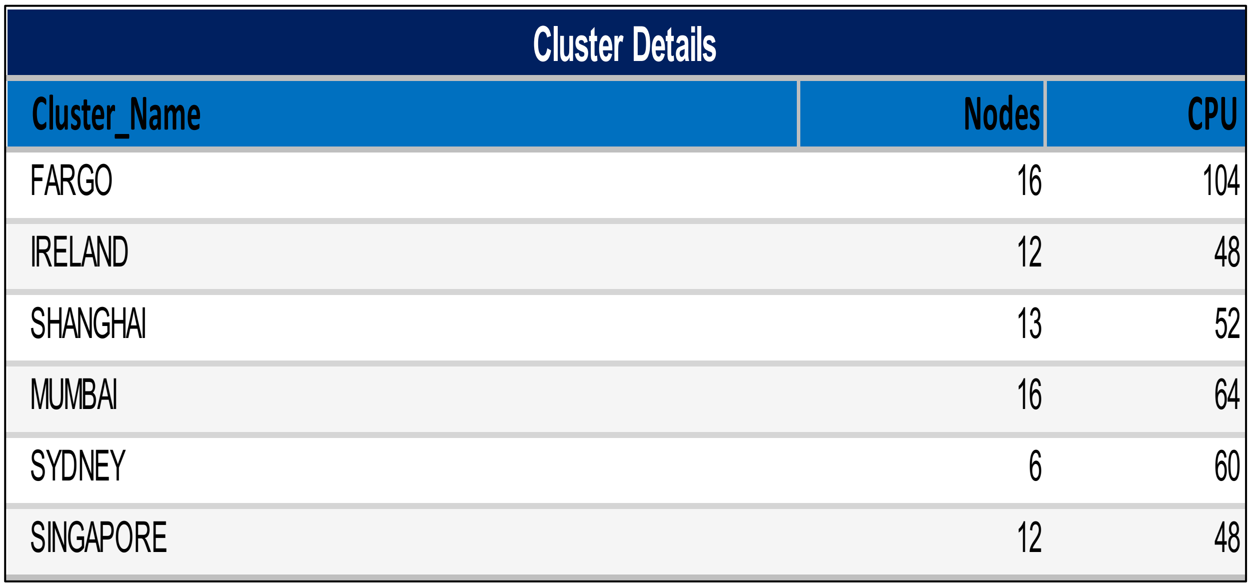
Figure 4.1 gives us the cluster details such as cluster name and the host systems that are present within a cluster. Addition or removal of any host from a cluster can be monitored through real time data. Trends from past data can also be analyzed for determining yearly trends of clusters. The count of each of the cluster has been calculated by uniquely identifying and understanding the scrapped logs from site systems and writing a SPL query.



FARGO	IRELAND	SHANGHAI
16	12	13
MUMBAI	**SYDNEY**	**SINGAPORE**
16	6	12

Figure 4.1. Site Systems Clusters

Clusters Details: Figure 4.2 gives us the overall cluster level details of RIMS not only limiting to the location but also details about the total number of nodes that have been provisioned for a cluster and the total number of CPU's that are available for the task allocation.

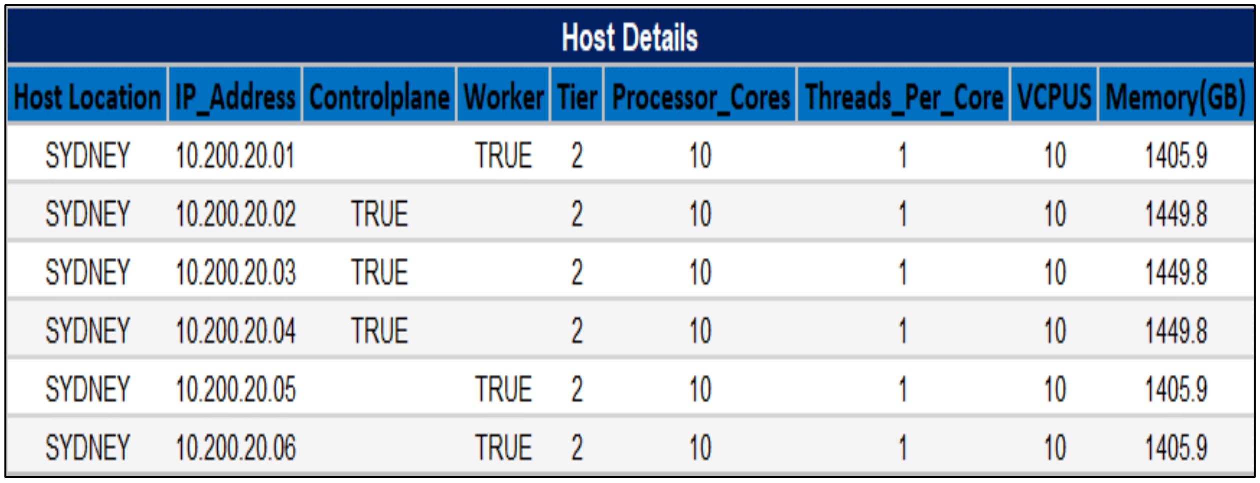


Cluster Details

Cluster_Name	Nodes	CPU
FARGO	16	104
IRELAND	12	48
SHANGHAI	13	52
MUMBAI	16	64
SYDNEY	6	60
SINGAPORE	12	48

Figure 4.2. Cluster Details

Host Details: Host details panel gives in depth information about all the hosts. Hosts within a cluster can be identified using the IP address and a further distinction can be done whether a host is a control plane or worker node. Hardware relevant information such as processor cores, threads per core, vcpus and memory gives key information about the host systems of RIMS.

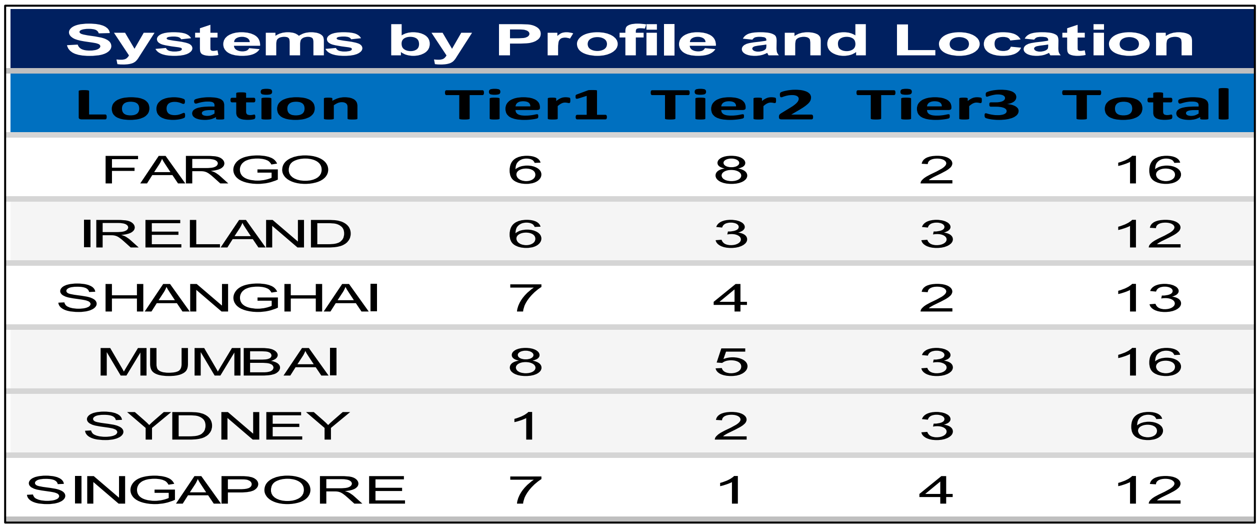


Host Details

Host Location	IP_Address	Controlplane	Worker	Tier	Processor_Cores	Threads_Per_Core	VCPUS	Memory(GB)
SYDNEY	10.200.20.01		TRUE	2	10	1	10	1405.9
SYDNEY	10.200.20.02	TRUE		2	10	1	10	1449.8
SYDNEY	10.200.20.03	TRUE		2	10	1	10	1449.8
SYDNEY	10.200.20.04	TRUE		2	10	1	10	1449.8
SYDNEY	10.200.20.05		TRUE	2	10	1	10	1405.9
SYDNEY	10.200.20.06		TRUE	2	10	1	10	1405.9

Figure 4.3. Host Level Details of a Cluster

Systems by Profile and Location: Profiling of systems is also an important feature of Site Systems dashboard of RIMS. Profiling of systems by placing them in correct tiers helps Site Reliability Engineers with easy path finding of the hosts within a given cluster.



Systems by Profile and Location

Location	Tier1	Tier2	Tier3	Total
FARGO	6	8	2	16
IRELAND	6	3	3	12
SHANGHAI	7	4	2	13
MUMBAI	8	5	3	16
SYDNEY	1	2	3	6
SINGAPORE	7	1	4	12

Figure 4.4. Profiling of Cluster Systems

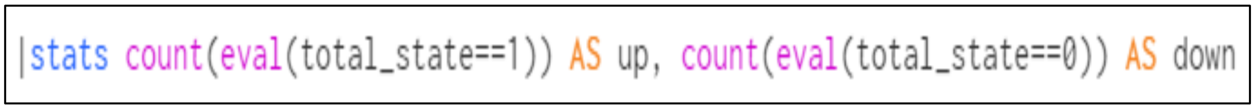
4.2. RIMS Site Availability Dashboard

RIMS use some instances of five clusters and some instances of other clusters are kept for backup. The image shown in Figure 4.6 shows us five clusters at different geographical location for calculation of SA using last 30 days of data. Similarly Figure 4.7 shows us the SA for all-time data since the system has been enabled. HTTP 500 requests data is stored internally in a Time-series based database as shown in Table 4.1. and count of total state of system responses for determining uptime and downtime is done using the logic that we have embedded in the SPL query in Figure 4.5.

Table 4.1. Time Series Data for Sydney Cluster

Instance/Frame	1min	2min	3min	4min	5min
Instance 1	1	1	1	0	1
Instance 2	1	1	1	0	1
Instance 3	1	1	1	1	1
Instance 4	1	0	0	0	0
Instance 5	1	1	1	1	1

SPL logic for calculating the uptime and downtime state of Sydney Instances.



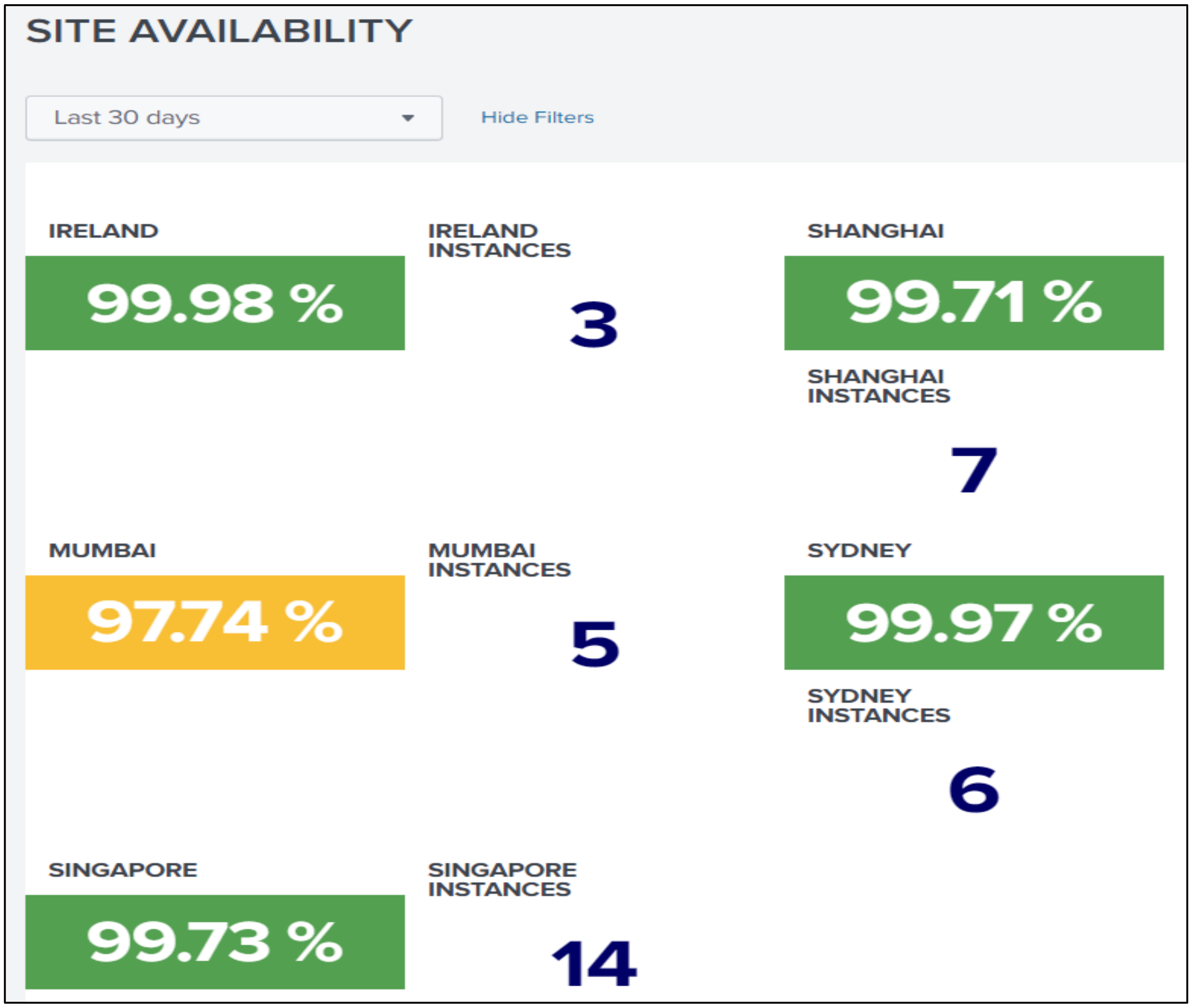
```
|stats count(eval(total_state==1)) AS up, count(eval(total_state==0)) AS down
```

Figure 4.5. SPL Query for Total State Calculation

Using the total state data Site Availability of a cluster at a specific geographical location is calculated using the formula.

$$\text{Site Availability} = \frac{Uptime}{Uptime + Downtime} * 100$$

The above calculations are shown just for one such cluster instances. The rest of the used instances within clusters follows the same logic for calculating the total sate of systems and similar mathematical hierarchy is followed for calculating the Site Availability for each of the clusters.


SITE AVAILABILITY

Last 30 days

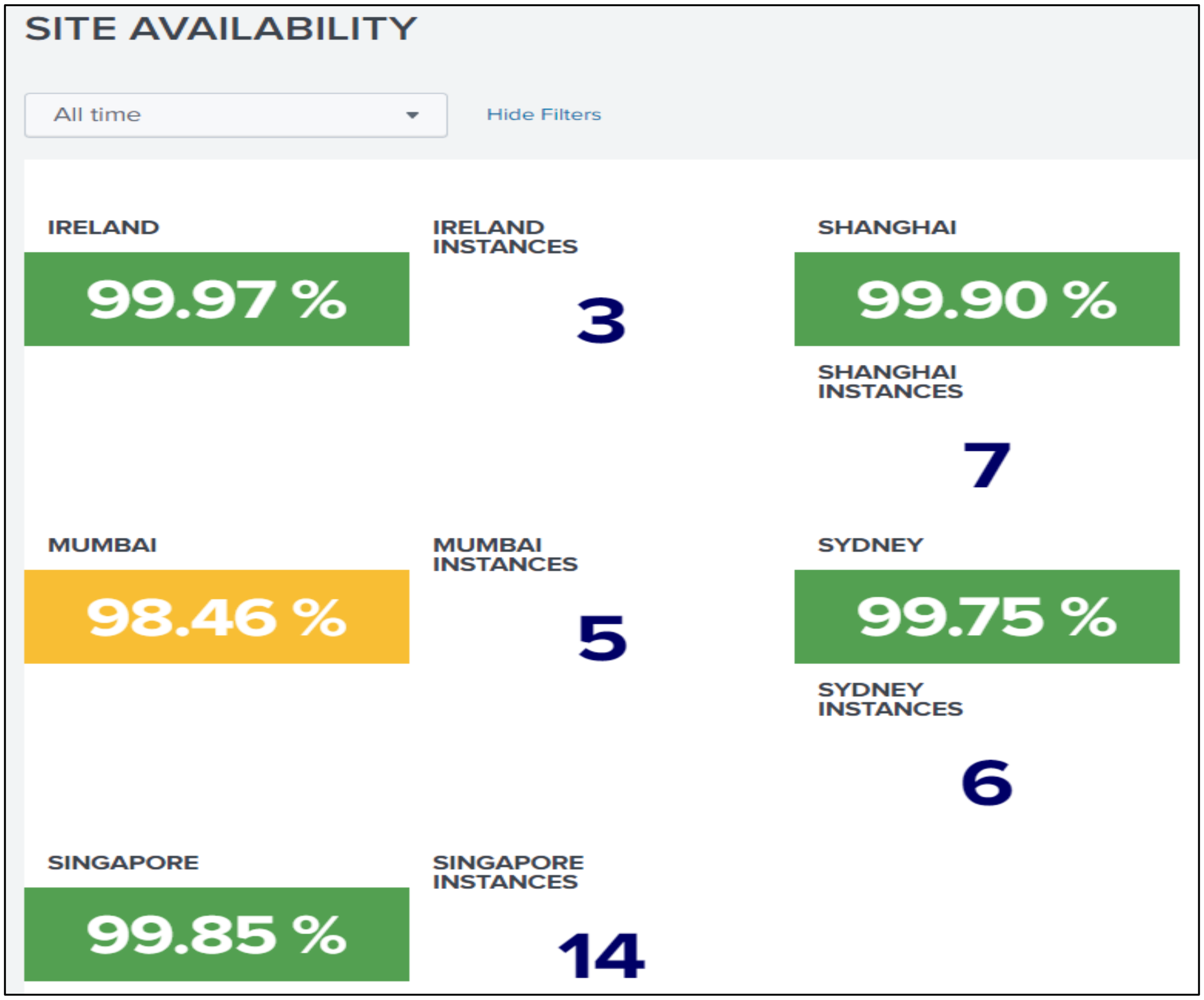
Hide Filters

IRELAND	IRELAND INSTANCES	SHANGHAI
99.98 %	3	99.71 %
		SHANGHAI INSTANCES
		7
MUMBAI	MUMBAI INSTANCES	SYDNEY
97.74 %	5	99.97 %
		SYDNEY INSTANCES
		6
SINGAPORE	SINGAPORE INSTANCES	
99.73 %	14	

Figure 4.6. 30-Days Site Availability

As our objective is to define SLO using the Site Availability data. SLOs are defined using the all-time data. Evident from Figure 4.7, we have calculated individual availabilities of each of the clusters using the above formula. The Net Availability or Reliance of entire Infrastructure is defined as the minimum value of Site Availability amongst clusters. So, the Net Availability that we were able to trace through RIMS is 98.46%.

Referring to the Google's unavailability definition Table 2.1., we will calculate allowed unavailability window for a given year, meaning the total time within a year when systems within Infrastructure were not functioning due to either of planned or unplanned downtimes.



IRELAND 99.97 %
IRELAND INSTANCES 3
SHANGHAI 99.90 %
SHANGHAI INSTANCES 7
MUMBAI 98.46 %
MUMBAI INSTANCES 5
SYDNEY 99.75 %
SYDNEY INSTANCES 6
SINGAPORE 99.85 %
SINGAPORE INSTANCES 14

Figure 4.7. All-Time Site Availability

The acceptable number of downtimes to reach a given number of nines of availability can be concluded from the table below for RIMS.

Table 4.2. RIMS Allowed Unavailability Window

Availability Level	RIMS Allowed Unavailability Window					
	Per year	Per quarter	Per month	Per week	Per day	Per hour
98.46%	3.63 days	21.4 hours	7.16 hours	1.67 hours	14.32 minutes	35.8 seconds

Through this we define Service Level Objective as 98.46% meaning out of 365 days, for 3.63 days Infrastructure services were not available.

5. CONCLUSION AND FUTURE WORK

This work has presented us with a technique for building a Site Monitoring and Alerting system that facilitates the job of Site Reliability Engineers and helps them with efficient management of the Infrastructure. Existing Site Monitoring Systems are internal to the organization as each organization sets its unique set of standards and requirements for meeting their Infrastructure needs. Defining Service Level Objectives (SLO) are incremental to the success of RIMS. SLOs are key to making data-driven decisions about reliability. RIMS lay a common platform so that organizations can quickly adopt this mechanism of monitoring and consider pre-defined SLOs for their Infrastructure at a reduced cost. Also, enabling RIMS will help Infrastructure teams reduce manual engineering effort that Site Reliability Engineers had to spend for tracking unplanned downtimes of the site systems.

In a world of rapid changing technology, as new demands and complexity grows with insane amount of data traffic, Site Engineering requires even much more attention. My future work lies in design and development of a more sophisticated system that not only limits to Site Availability but also caters to Site Reliability and Data Durability. Long-term data protection for avoiding degradation or corruption has been a major challenge for decades. In future, I would also work towards addressing this challenge of data protection and design much more intuitive and efficient Real-time Infrastructure Site Monitoring Systems. Redefining SLOs for achieving higher site availability will also be considered.

REFERENCES

[1] Blasi, L., Brataas, G., Boniface, M., Butler, J., Dandria, F., Drescher, M., Jimenez, R., Krogmann, K., Kousiouris, G., Koller, B., Landi, G., Matera, F., Menychtas, A., Oberle, K., Phillips, S., Rea, L., Romano, P., Symonds, M., & Ziegler, W. (2013). Cloud computing service level agreements. Retrieved from http://ec.europa.eu/information society/newsroom/cf/dae/document.cfm?doc id =2496.

[2] Emeakaroha, V.C., Netto, M.A.S., Calheiros, R.N., Brandic, I., Buyya, R., & Rose, C.A.F. (2012, July). Towards autonomic detection of SLA violations in Cloud Infrastructures. Future Generation Computer Systems, 8(7), 1017-1029.

[3] Hernantes, J., Gallardo, G., & Serrano, N. (2015, April). IT Infrastructure-Monitoring Tools. IEEE Software. 32(4), 88–93.

[4] Beyer, B., Jones, C., Petoff, J., Murphy, N.R. (2016) Site Reliability Engineering. California: O'Reilly Media

[5] Beyer, B., Murphy, N.R., Rensin, D.K., Kawahara, K., & Thorne, S. (2018) The Site Reliability Workbook.
California: O'Reilly Media

[6] Prometheus Documentation. (2020). Configuration and feeding data to Splunk. Retrieved from https://prometheus.io/docs/introduction/overview/

[7] Ansible Documentation. (2020). Configuration and feeding data to Splunk. Retrieved from https://docs.ansible.com/ansible/latest/plugins/callback/splunk.html

[8] Ansible Documentation. (2020). Ansible Components. Retrieved from https://www.tecmint.com/understand-core-components-of-ansible/#coreansible

[9] Splunk Documentation. (2020). Splunk Processing Language Syntax. Retrieved from https://docs.splunk.com/Documentation/Splunk/8.0.4/SearchReference/UnderstandingSPLsyntax

[10] Splunk Documentation. (2020). Dashboards and Panels. Retrieved from https://docs.splunk.com/Documentation/Splunk/8.0.4/SearchTutorial/Aboutdashboards