Observability and Monitoring

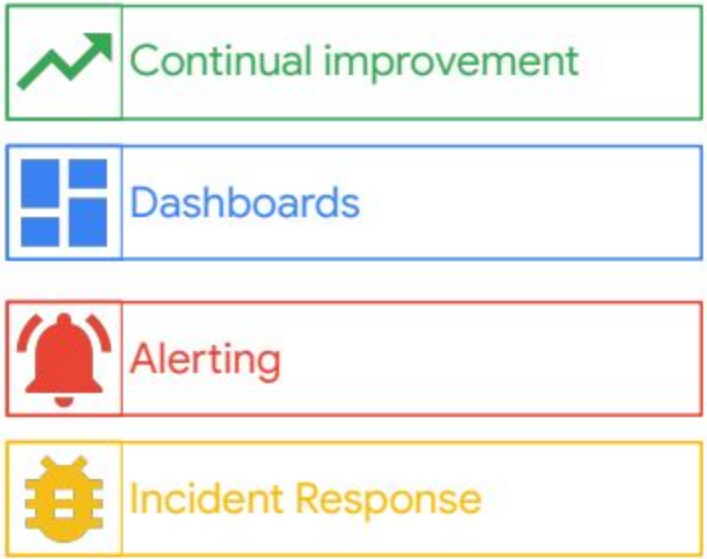
Observability and Monitoring

NAYA College

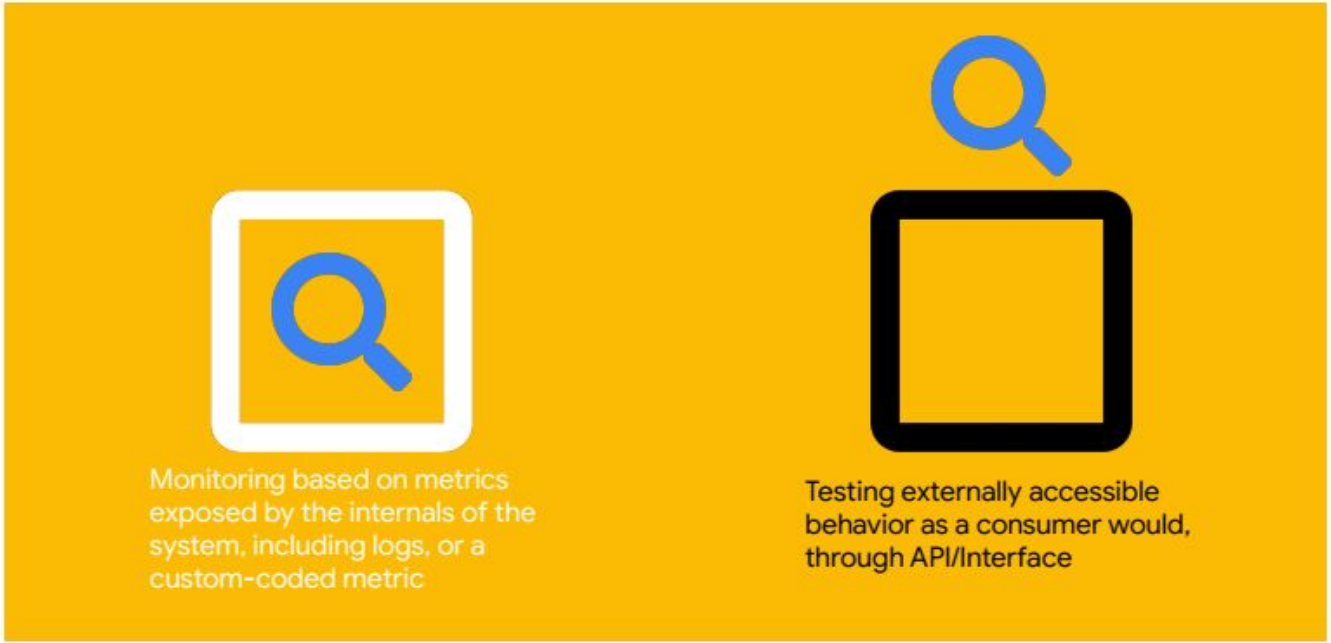
Monitoring

NAYA College

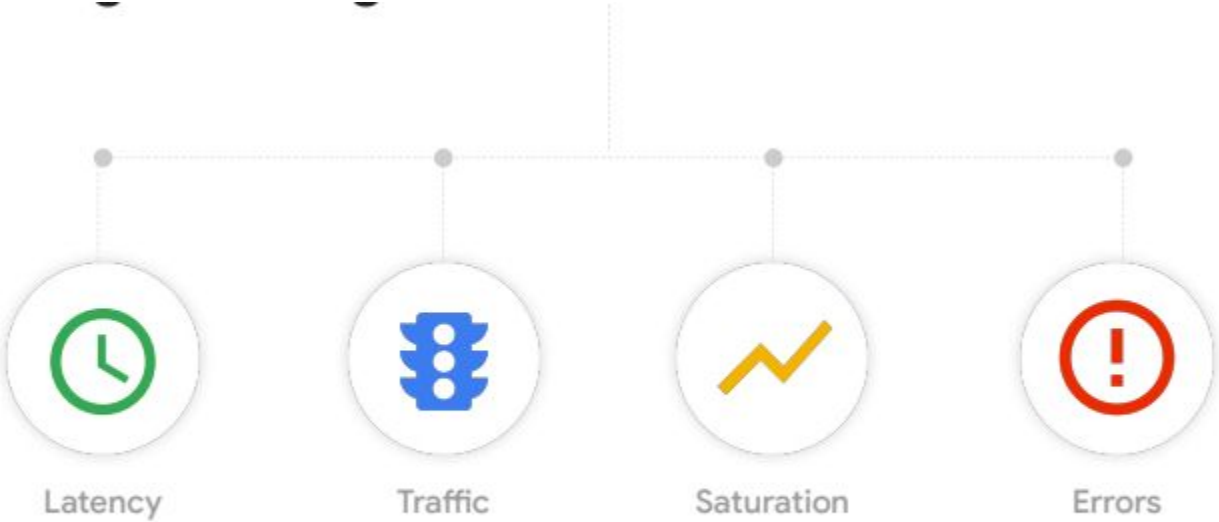
Why monitor


Continual improvement
Dashboards
Alerting
Incident Response

Clear box versus black box



The four golden signals



Latency

Important because

Impacts user experience.

Could indicate **emerging issues.**

May be tied to **capacity demands.**

May be used to show **improvements.**

Sample metrics

- **Page load latency**
- **Number of requests waiting for a thread**
- **Query duration**
- **Service response time**
- **Transaction duration**
- **Time until first response**
- **Time until complete data return**

NAYA College

Traffic

Important because

Indicates current **system demand.**

Historical trends are used for **capacity planning.**

Core to calculating **infrastructure spend.**

Sample metrics

- # HTTP requests per second
- # requests for static vs. dynamic content
- Network I/O
- # concurrent sessions
- # transactions per second
- # retrievals per second
- # active requests
- # write ops
- # read ops
- # active connections

NAYA College

Saturation

Important because

Indicates **how full the service is.**

Focuses on **most constrained resources.**

Frequently tied to **degrading performance.**

Sample metrics

- % memory utilization
- % thread pool utilization
- % cache utilization
- % disk utilization
- % CPU utilization
- Disk quota
- Memory quota
- # available connections
- # users on the system

NAYA College

Error

Important because

Indicates that **something is failing.**

May indicate **configuration or capacity issues.**

Can indicate **SLO violation.**

Time to **alert**?

Sample metrics

- **Wrong answer/content**
- **# 400/500 HTTP codes**
- **# failed requests**
- **# exceptions**
- **# stack traces**
- **Server fails liveness check**
- **# dropped connections**

NAYA College

The **most important feature** of any system is its **reliability**.

1

NAYA College

Reliability

Reliability is defined as the probability that a product, system, or service will perform its intended function adequately for a specified period of time, or will operate in a defined environment without failure.

Service Level Indicator

A **quantifiable** measure of service **reliability**

SLI Menu

Request/Response	Availability Latency Quality
Data Processing	Coverage Correctness Freshness Throughput
Storage	Throughput Latency

Service Level Objective

A **reliability target** for an SLI

Services *need* SLOs

Don’t believe us?

“Since introducing SLOs, the **relationship** between our operations and development teams has **subtly but markedly improved**.”

— *Ben McCormack, Evernote;* The Site Reliability Workbook, Chapter 3

“... it is difficult to *do your job well* without clearly defining *well*. SLOs **provide the language** *we need to* **define *well***.”

— *Theo Schlossnagle, Circonus;* Seeking SRE, Chapter 21


Developers
Agility
How do you
incentivize
reliability?
Operators
Stability

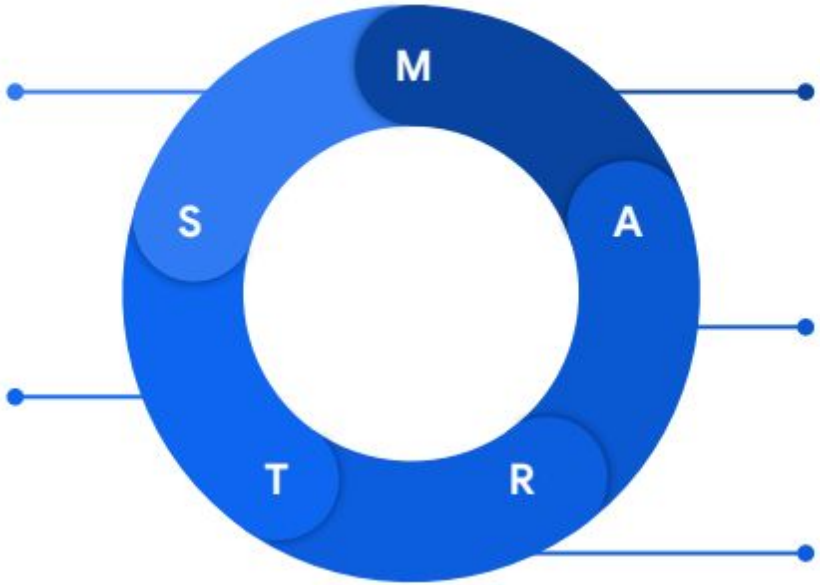
SLO

A **principled** way
to agree on the
desired reliability
of a service

To be effective, SLOs must be SMART

Specific
"Fast" is not as specific as "Results in 100ms."

Time bound
99% available: Per year? Per month? Per day? If we don't know, how can we measure?



Measurable
A number, a delta, something we can measure and place in a mathematical equation.

Achievable
"100% Availability" might sound good, but it's not typically possible to maintain.

Relevant
Does it matter to the user? Will it help achieve application-related goals?

Error budgets

An SLO implies an **acceptable level** of unreliability.

*This is a **budget** that can be **allocated**.*

100% - SLO = Error Budget

Implementation mechanics

Evaluate SLO **performance** over a set **window**, e.g., 28 days.
Remaining budget (100%-SLO) **drives prioritization**
of engineering effort.

What should we **spend**
our error budget on?

NAYA College

Error budgets can accommodate:

- New **feature releases**
- Expected system **changes**
- Inevitable **failure** in hardware, networks, etc.
- Planned **downtime**
- Risky **experiments**

Prometheus

What is Prometheus

Prometheus is a high-scalable open-source monitoring framework. It provides out-of-the-box monitoring capabilities for the Kubernetes **container orchestration platform**. Also, In the observability space, it is gaining huge popularity as it helps with metrics and alerts.

NAYA College

Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit with the following main features:

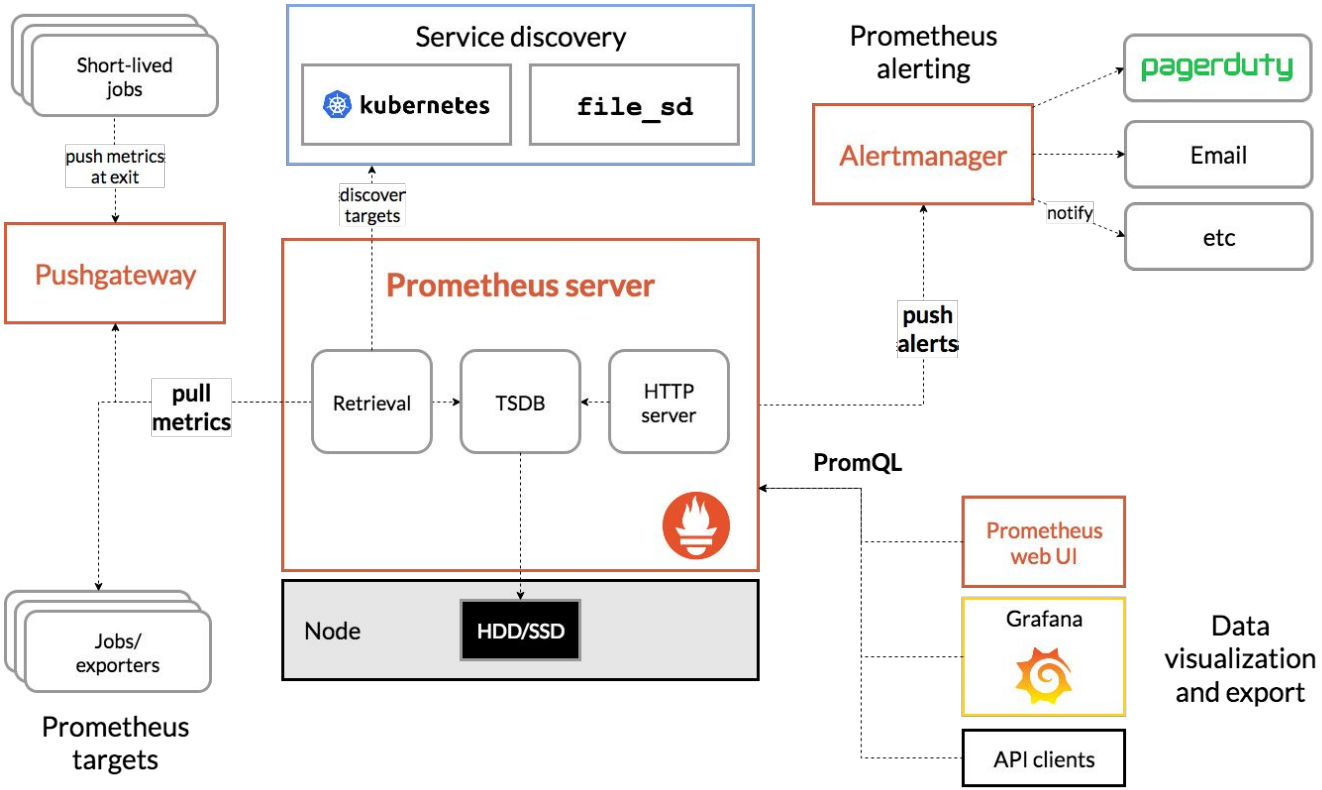
- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

What are metrics ?

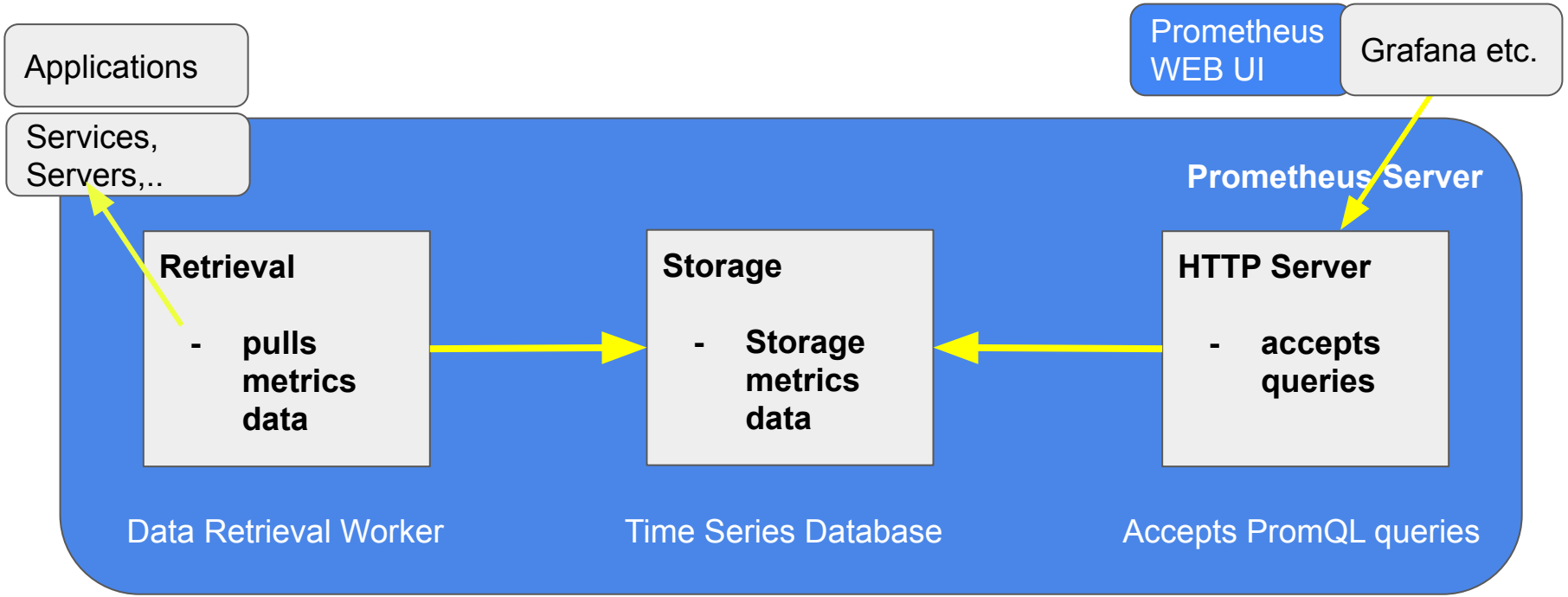
In layperson terms, metrics are numeric measurements, time series mean that changes are recorded over time. What users want to measure differs from application to application. For a web server it might be request times, for a database it might be number of active connections or number of active queries etc.

Metrics play an important role in understanding why your application is working in a certain way. Let's assume you are running a web application and find that the application is slow. You will need some information to find out what is happening with your application. For example the application can become slow when the number of requests are high. If you have the request count metric you can spot the reason and increase the number of servers to handle the load.

Prometheus architecture



Main Component: Prometheus Server



NAYA College

Prometheus key points

- **Metric Collection**: Prometheus uses the pull model to retrieve metrics over HTTP. There is an option to push metrics to Prometheus using **Pushgateway** for use cases where Prometheus cannot Scrape the metrics.
- **Metric Endpoint**: The systems that you want to monitor using Prometheus should expose the metrics on an **/metrics** endpoint. Prometheus uses this endpoint to pull the metrics in regular intervals.
- **PromQL**: Prometheus comes with PromQL, a very flexible query language that can be used to query the metrics in the Prometheus dashboard. Also, the PromQL query will be used by Prometheus UI and Grafana to visualize metrics.

Prometheus key points

- **Prometheus Exporters**: Exporters are libraries that convert existing metrics from third-party apps to Prometheus metrics format. There are many official and community Prometheus exporters. One example is, the Prometheus node exporter. It exposes all Linux system-level metrics in Prometheus format.
- **TSDB** (time-series database): Prometheus uses TSDB for storing all the data efficiently. By default, all the data gets stored locally. However, to avoid a single point of failure, there are options to integrate remote storage for Prometheus TSDB.

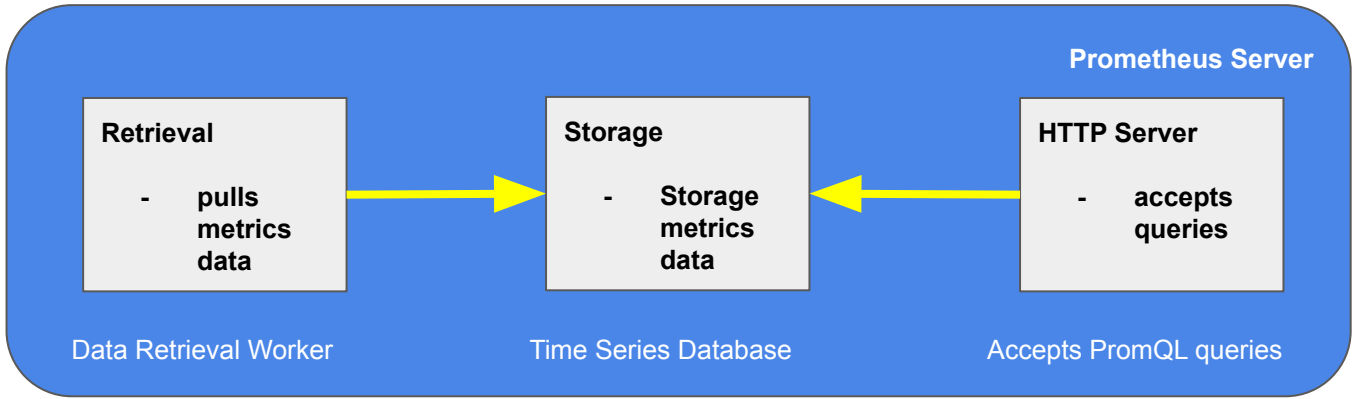
Targets and Metrics

What does Prometheus monitor?

- Linux/Windows Server
- Single Application
- Apache Server
- Service, like Database

Which units are monitored of those targets?

- CPU Status
- Memory/Disk Space Usage
- Request Count
- Exceptions Count
- Request Duration

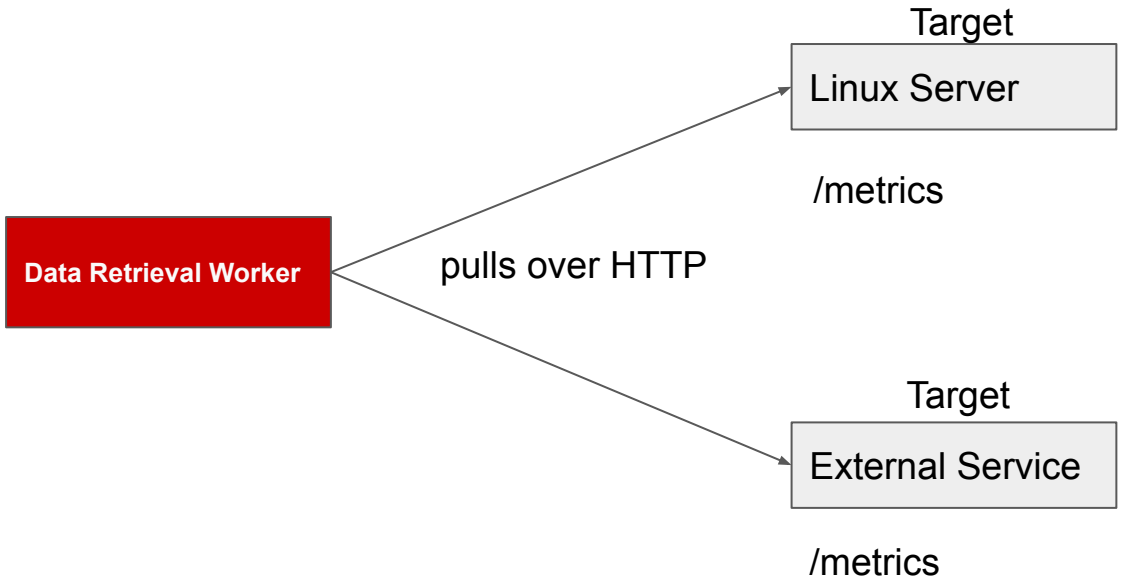


NAYA College

Prometheus helm chart Installation

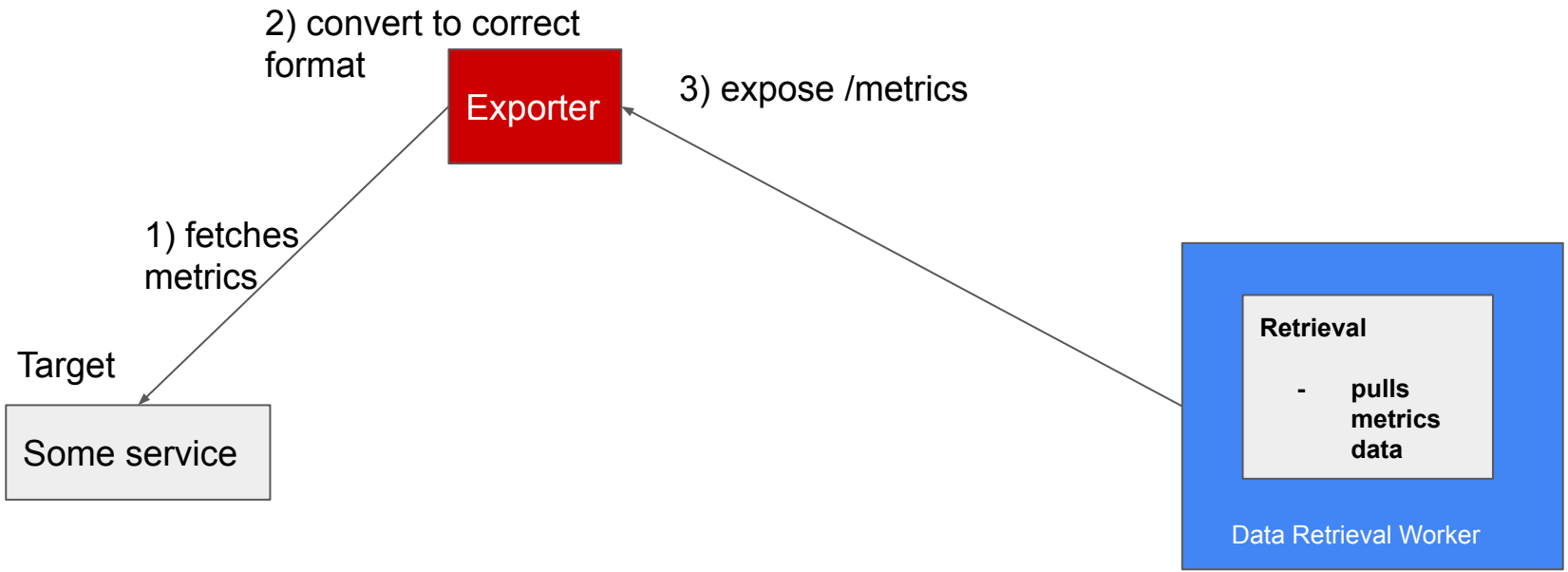
https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

Collecting Metrics Data from Targets

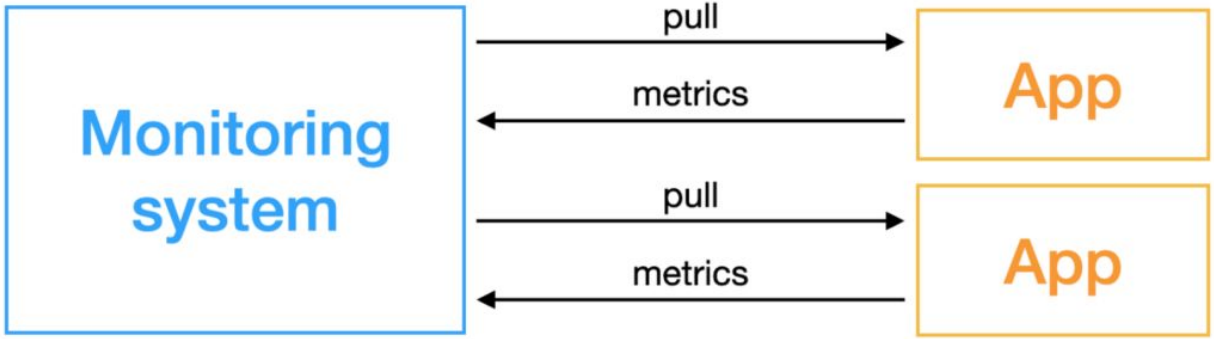


- Pulls from HTTP endpoints
- hostaddress/metrics
- must be in correct format

Target Endpoints and Exporters

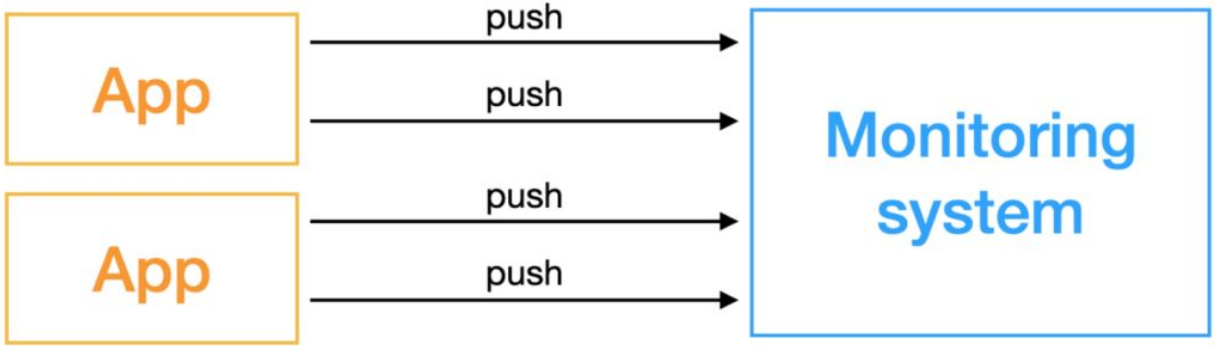


Pull-based system



- multiple Prometheus instances can pull metrics data
- better detection/insight if service is up and running

Push-based monitoring system



- high load of network traffic
- monitoring can become your bottleneck
- install additional software or tool to push metrics

EXPORTERS AND INTEGRATIONS

There are a number of libraries and servers which help in exporting existing metrics from third-party systems as Prometheus metrics. This is useful for cases where it is not feasible to instrument a given system with Prometheus metrics directly (for example, HAProxy or Linux system stats).

Third-party exporters

Some of these exporters are maintained as part of the official Prometheus GitHub organization, those are marked as *official*, others are externally contributed and maintained.

We encourage the creation of more exporters but cannot vet all of them for best practices. Commonly, those exporters are hosted outside of the Prometheus GitHub organization.

The exporter default port wiki page has become another catalog of exporters, and may include exporters not listed here due to overlapping functionality or still being in development.

The JMX exporter can export from a wide variety of JVM-based applications, for example Kafka and Cassandra.

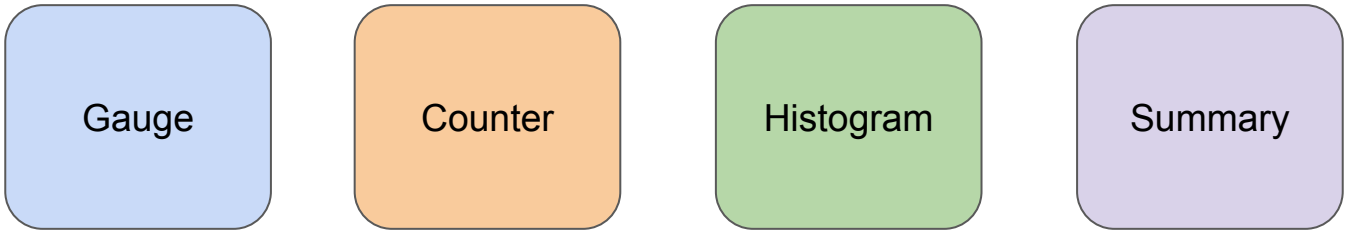
How Does Prometheus Integrate With Your Workloads?

When using client libraries, you get a lot of default metrics from your application. For example, in Go, you get the number of bytes allocated, number of bytes used by the GC, and a lot more. See the below

```
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.3"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 626792
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 626792
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.442982e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 124
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.240512e+06
```

NAYA College

Prometheus Metric Types



A Time Series

Monotonically Increasing

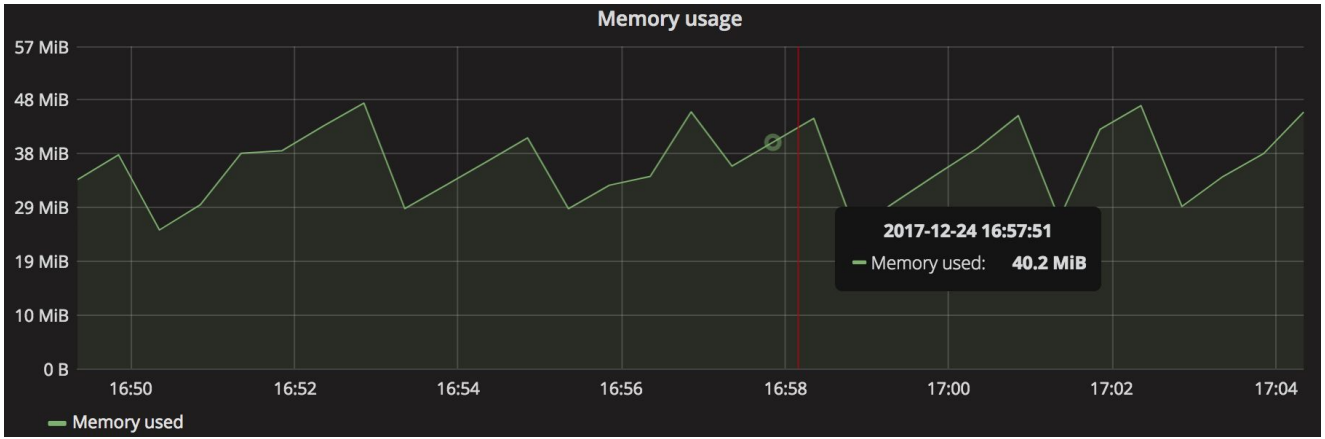
Cumulative Histogram of Values

Snapshot of Values in a Time Windows

NAYA College

Gauge

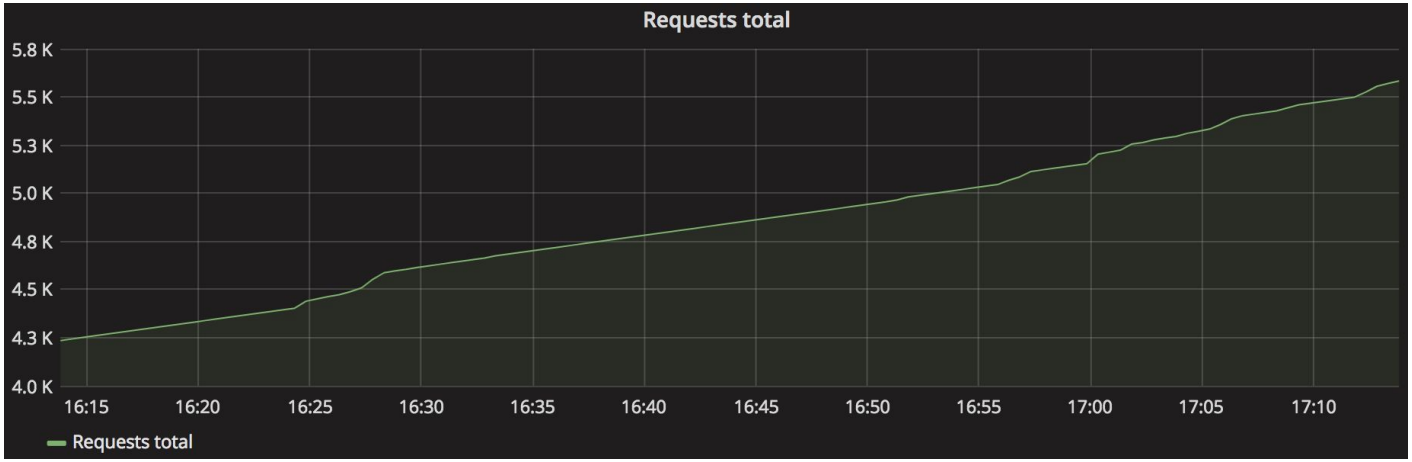
```
# Amount of memory currently used
memory_bytes_used

# Number of jobs currently in queue
batch_jobs_in_queue{job_type="hourly-cleanup"}
```



NAYA College

Counter

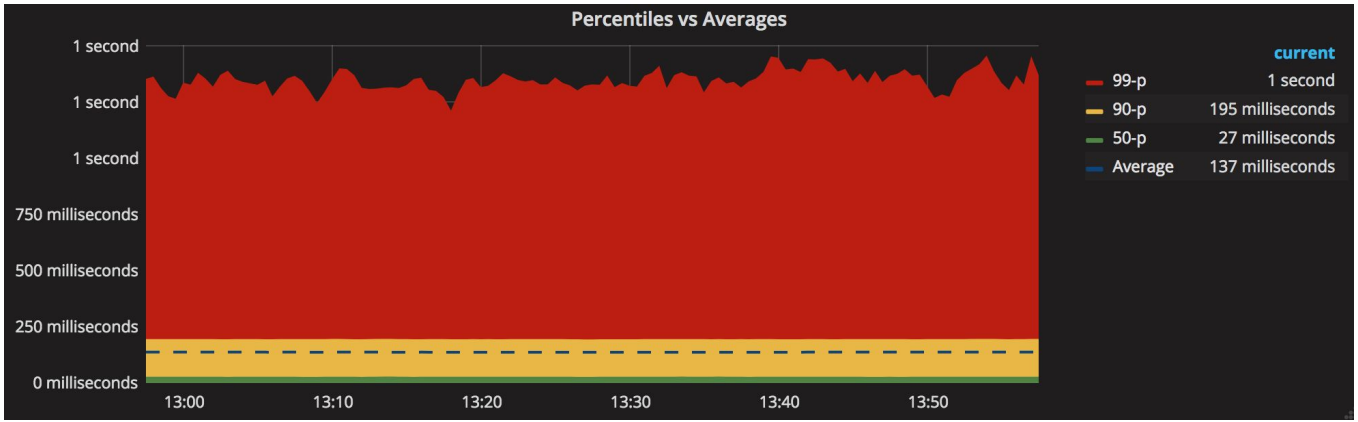
http_requests_total



NAYA College

Histogram

```
# Request duration 90th percentile
histogram_quantile(0.9, rate(http_request_duration_milliseconds_bucket[5m]))
```



NAYA College

Data Model

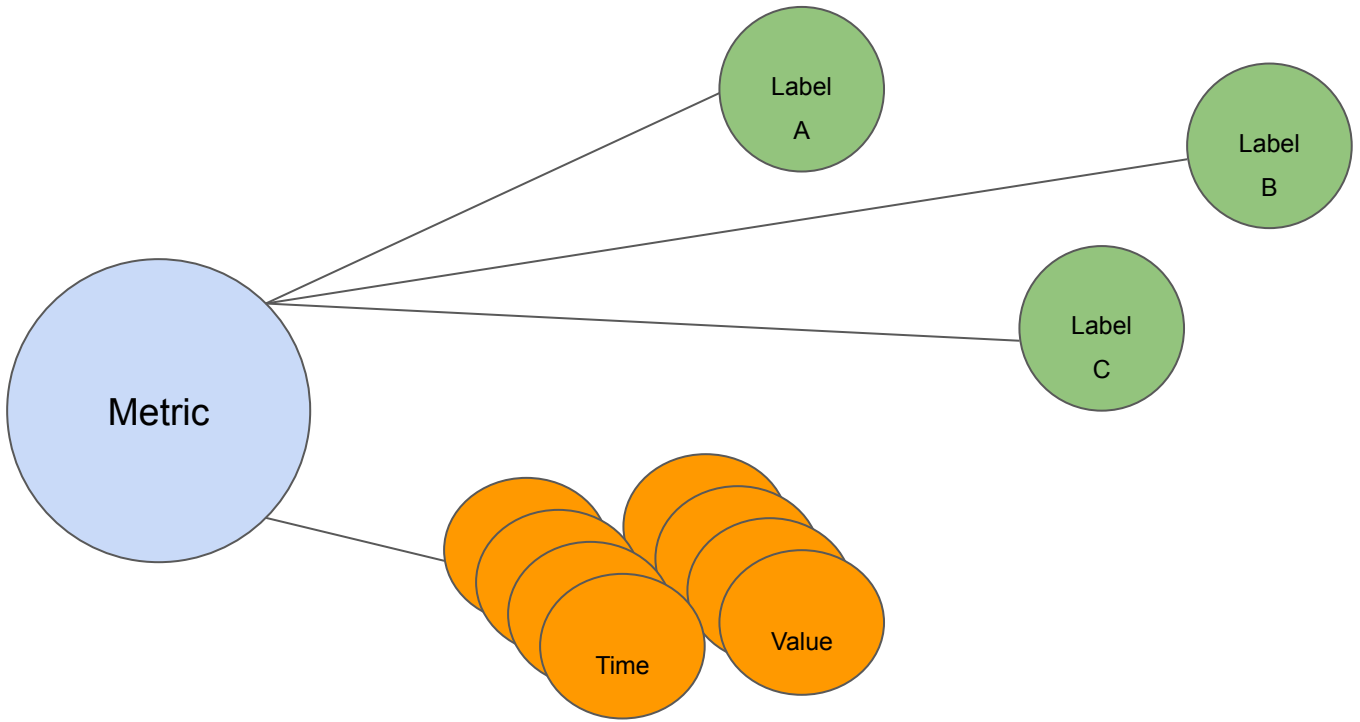
Prometheus fundamentally stores all data as time series: streams of timestamped values belonging to the same metric and the same set of labeled dimensions.

Metric names and labels

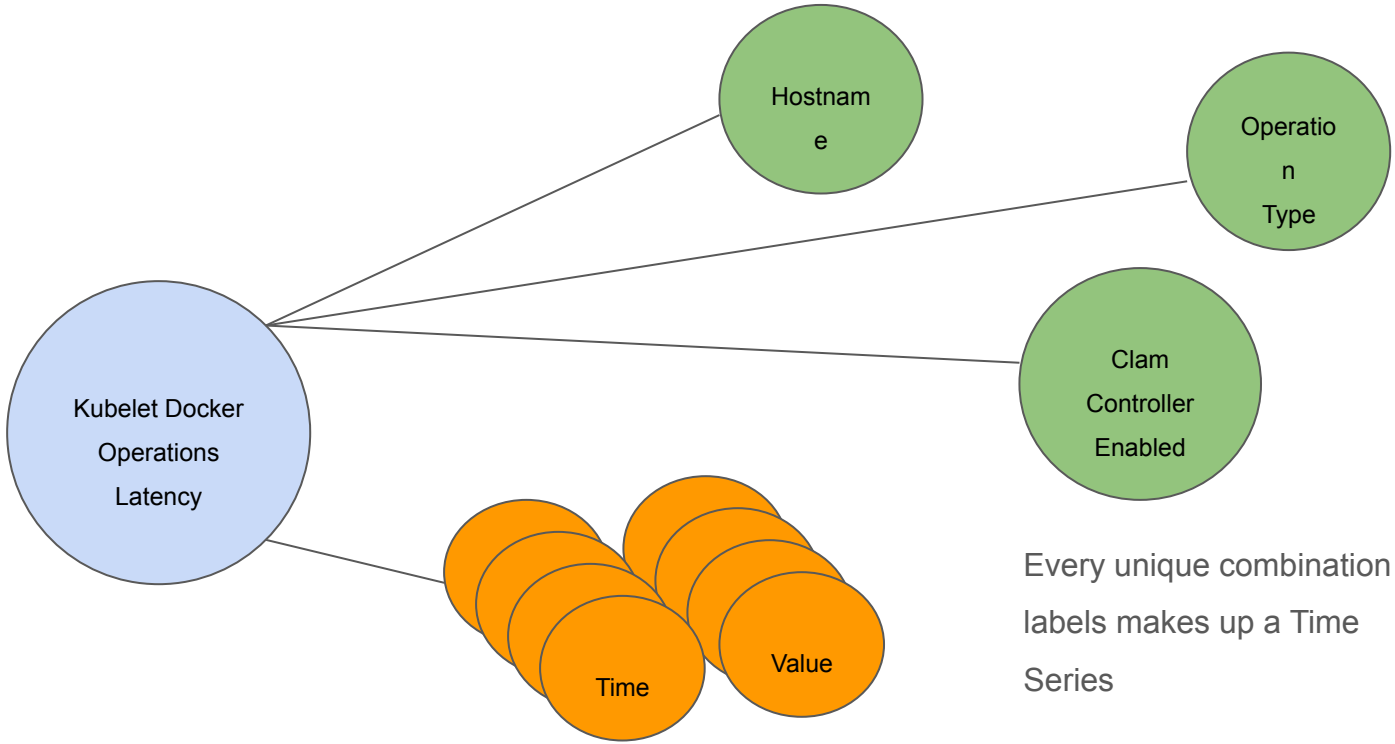
Every time series is uniquely identified by its metric name and optional key-value pairs called labels.

The metric name specifies the general feature of a system that is measured (e.g. *http_requests_total* - the total number of HTTP requests received). It may contain ASCII letters and digits, as well as underscores and colons. It must match the regex *[a-zA-Z_:][a-zA-Z0-9_:]*.*

Anatomy of metric



docker_latency



Every unique combination of labels makes up a Time Series

Metrics

- Format: **Human-readable** text-based
- Metrics entries: TYPE and HELP attributes

```
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.3"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 626792
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 626792
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.442982e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 124
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.240512e+06
```

NAYA College

Demo/Labs

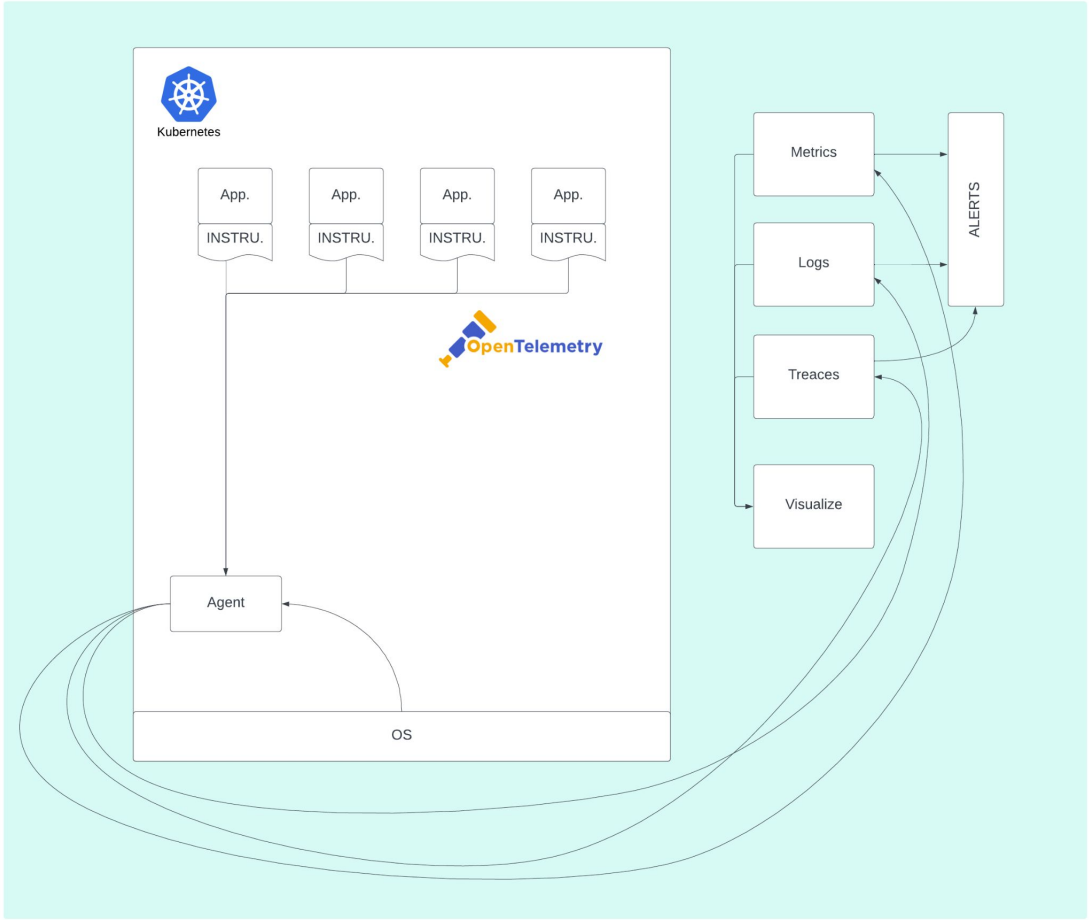
K8S Prometheus:

In our repository

$ cd cognyte-devops-32/k8s-3/prometheus-tutorial/

Follow README.md

Observability


Kubernetes
App.
INSTRU.
App.
INSTRU.
App.
INSTRU.
App.
INSTRU.
OpenTelemetry
Agent
OS
Metrics
Logs
Treaces
Visualize
ALERTS

Instrumentation

Process of *adding code to your application* to collect and emit **telemetry data**. Telemetry data includes information about the **performance**, **behavior**, and **state of an application**.

The primary goal of *instrumentation* is to gain insights into how an application is running and performing in real-world scenarios.

Instrumentation

1. Visibility into Application Behavior
2. Monitoring and Observability
3. Distributed Tracing
4. Performance Profiling
5. Troubleshooting and Debugging
6. Adaptation to Cloud-Native Environments

Jaeger

Jaeger is an open-source, end-to-end distributed tracing system designed to monitor and troubleshoot complex microservices architectures. It provides visibility into the flow of requests and responses across various services in a distributed system, helping developers and operators understand the performance, latency, and dependencies between different components.

Demo/Labs

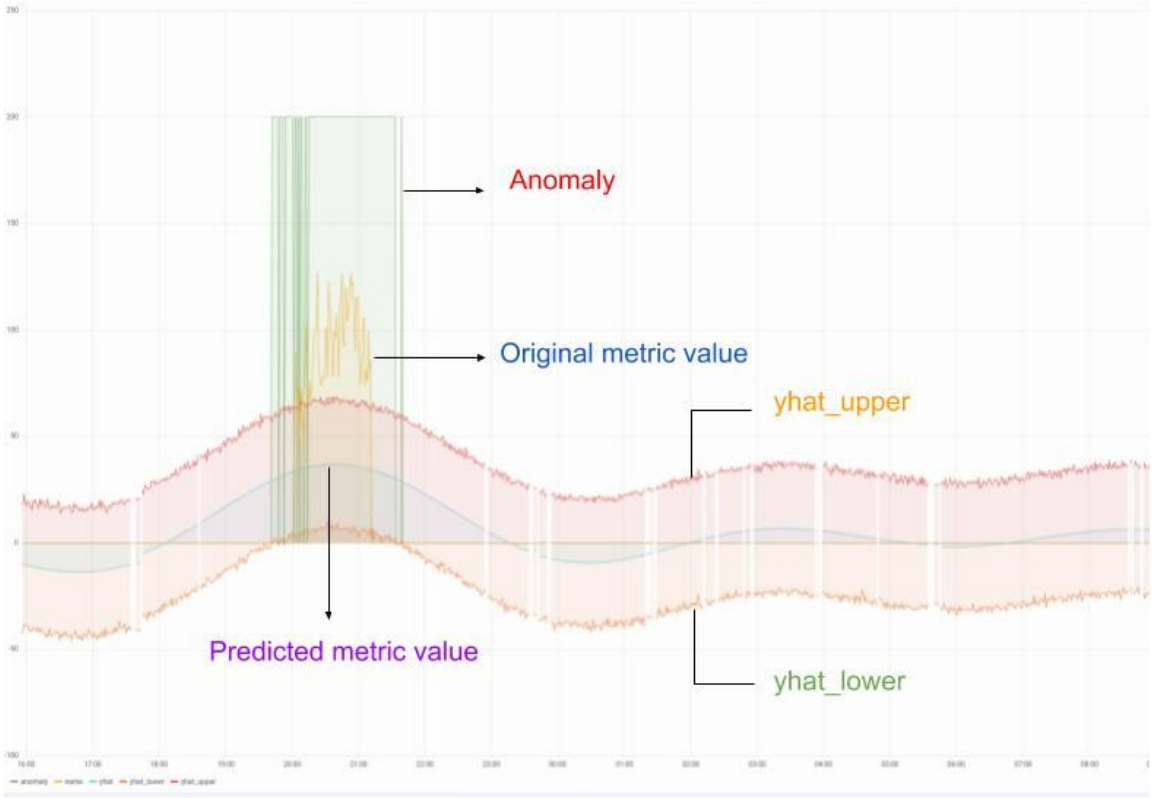
K8S Prometheus:

In our repository

$ cd cognyte-devops-32/k8s-3/OpenTelemetry/

Follow README.md

Anomaly detection


Anomaly
Original metric value
yhat_upper
Predicted metric value
yhat_lower

Anomaly detection strategy

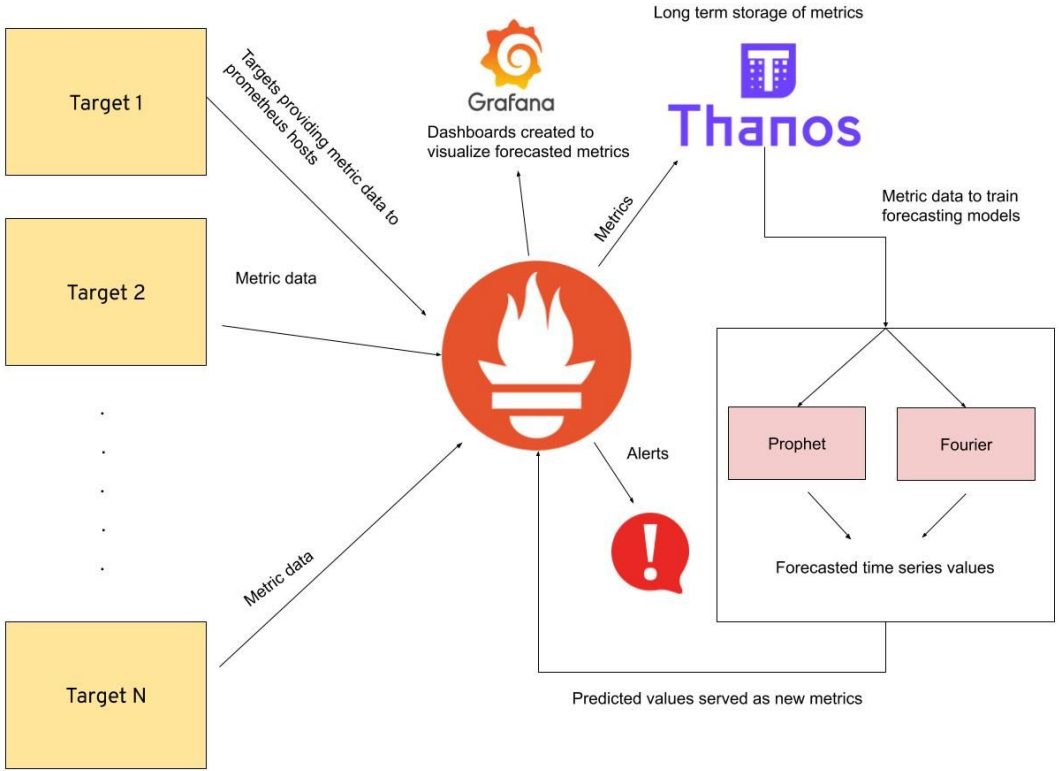
- Define Objectives and Use Cases
- Choose Appropriate Data Sources
- Instrumentation with OpenTelemetry
- Collecting Metrics and Logs
- Set Up Prometheus Alerts
- Implement Machine Learning Models
- Deploying Anomaly Detection Solutions
- Integrating with Grafana
- Implementing Auto-Scaling Strategies
- Implementing AIOps Practices

NAYA College

Objectives and Use Cases

- Ensure High Availability
- Optimize Resource Utilization
- Reduce Downtime
- Enhance Scalability
- Improve Application Performance
- Cost Optimization
- Enhance Security

Prometheus anomaly detector



Prometheus anomaly detector

https://github.com/AICoE/prometheus-anomaly-detector

Optimizing Application Performance

Performance

Observability

What is granularity of observability?

- Trade-off between accurate information and overhead.

Additional Operational info:

- Quarkus Micrometer
- Spring Actuator
- Liberty MicroProfile
- Node.js prom-client

NAYA College

Don’t Forget The Hardware

BIOS

- CPU Power and Performance Policy: <Performance>

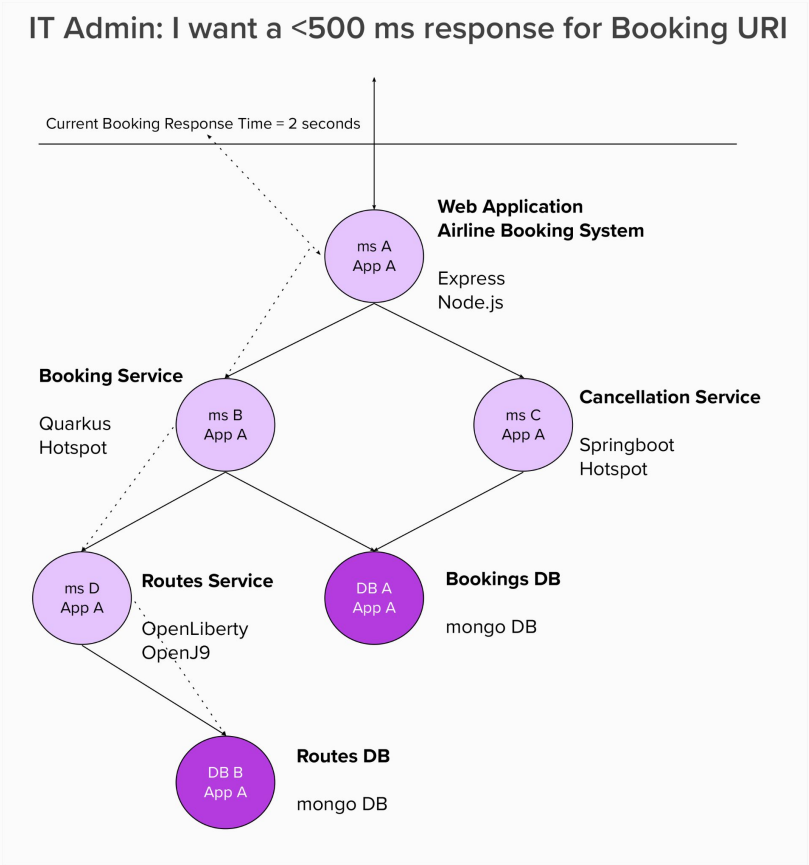
OS/Hypervisor

- CPU Scaling governor: <Performance>

Hyperthreading

- Do not count hyperthreading while capacity planning

Lower My Response Time



Lower My Response Time

- Node Affinity / Pod Affinity
- CPU Request / Limit
- Memory Request/Limit
- Java Heap Size/Ratio
- VPA
- HPA
- CA

Kruize Autotune

Kruize Autotune - Autonomous Performance Tuning for Kubernetes !

https://github.com/kruize/autotune

NAYA College