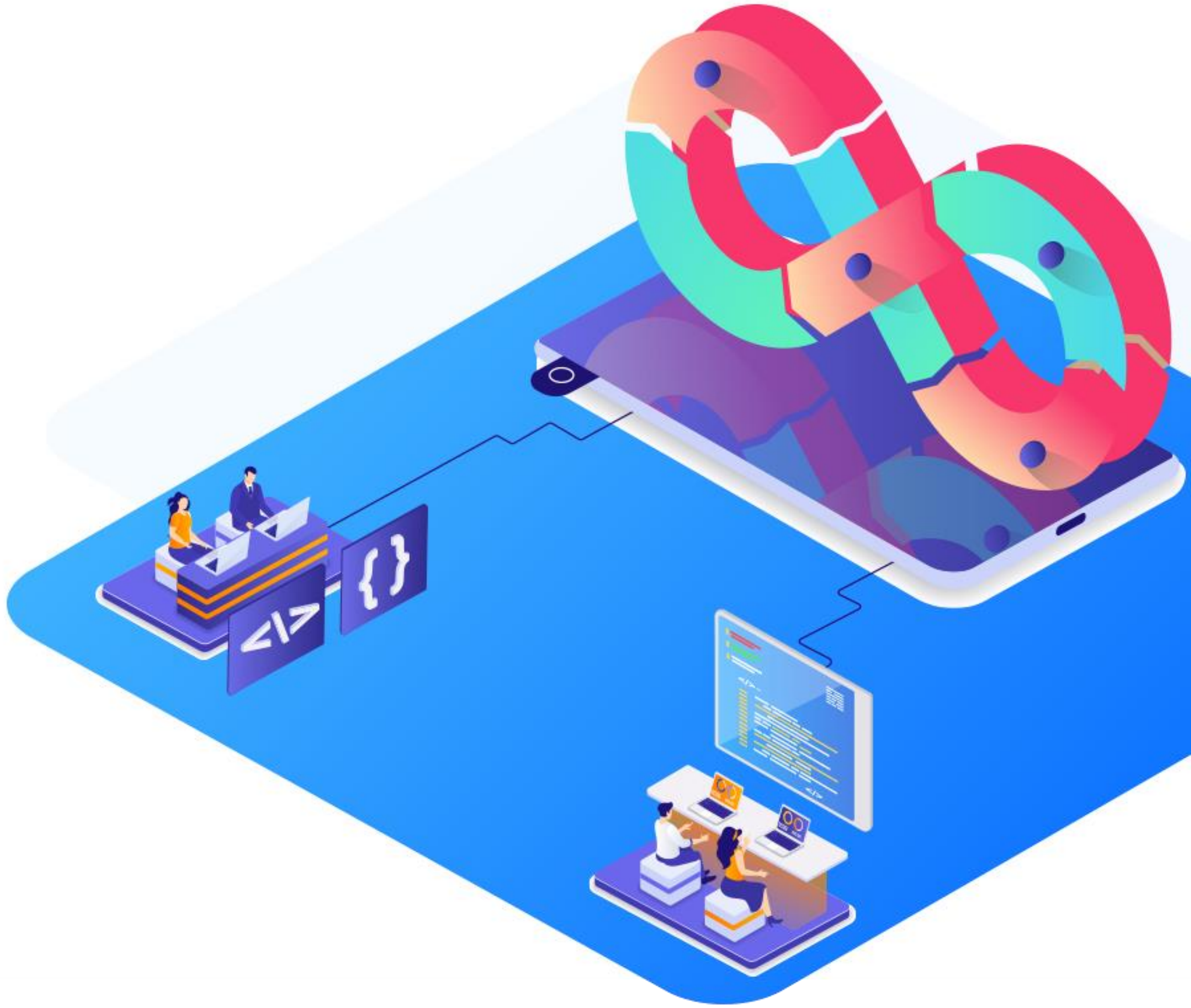
Monitoring and Logging in DevOps



Implementing Monitoring with Prometheus

Learning Objectives

By the end of this lesson, you will be able to:

- Identify the right metrics based on the application to effectively monitor and assess system performance
- Deploy various exporters to ensure comprehensive metric collection and integration with Prometheus
- Apply relabeling techniques for consistency in the metrics and optimization of data querying
- Utilize PromQL functions to identify data trends and calculate rates of change in time series data
- Utilize matchers and selectors to improve query execution time and resource utilization

Metrics and Exporters

What Are Metrics?

These are numerical data points collected over time to provide information about the performance and health of an infrastructure or application.

Some of the examples of metrics include:

- CPU utilization
- System error rate
- Memory usage
- Latency

Metrics are stored as time series data, with each tied to a specific time. They can also be labeled with additional contextual information, such as hostname or application name.

Uses of Metrics

Metrics are essential for the following purposes:

Measure

It evaluates server response times, memory usage, and error rates.

Monitor

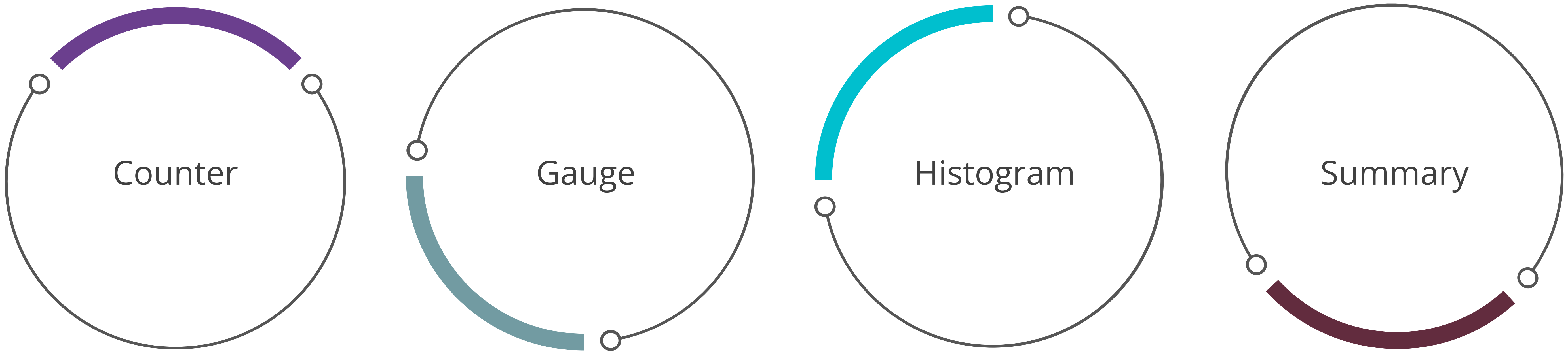
It tracks performance trends and identifies anomalies.

Compare

It assesses performance against historical data or other systems.

Types of Metrics

Metrics are classified into four major types based on their characteristics and the data they provide:



Each metric type serves a specific purpose and provides unique insights into various aspects of system behavior.

Counter Metrics

They track cumulative values that only increase over time and reset to zero when restarted.

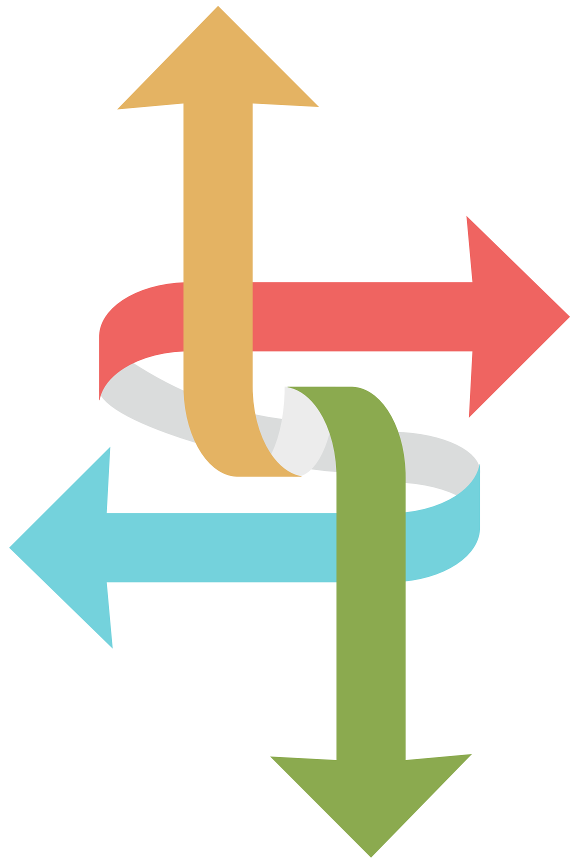
Example

```
# HELP http_requests_total Total number of http api requests
# TYPE http_requests_total counter
http_requests_total{api="add_product"} 4633433
```

This example illustrates how to monitor the total number of HTTP API requests for an **add product** function.

Counter Metrics: Uses

Records the total number of requests handled by a web server



Tracks the total number of errors encountered by an application

Counts the total number of jobs completed by a batch processing system

Measures the total bytes transferred over a network interface

Gauge Metrics

They represent values that can both increase and decrease, reflecting the current state.

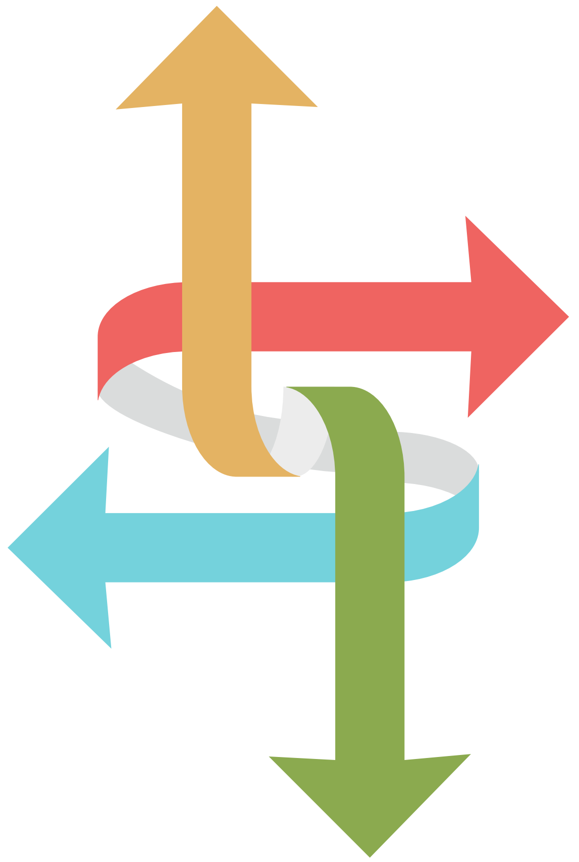
Example

```
# HELP node_memory_used_bytes Total memory used in the node in bytes
# TYPE node_memory_used_bytes gauge
node_memory_used_bytes{hostname="host1.domain.com"} 943348382
```

This example shows how to monitor the total memory used on a node, labeled as **node_memory_used_bytes** with a specific value for a given hostname.

Gauge Metrics: Uses

Assesses the current CPU load or temperature



Tracks the current memory consumption of a system or process

Reports the current length of a message queue or backlog

Records the number of active connections to a server or database

Histogram Metrics

They measure the distribution of values across predefined buckets and enable analysis of frequency and variance.

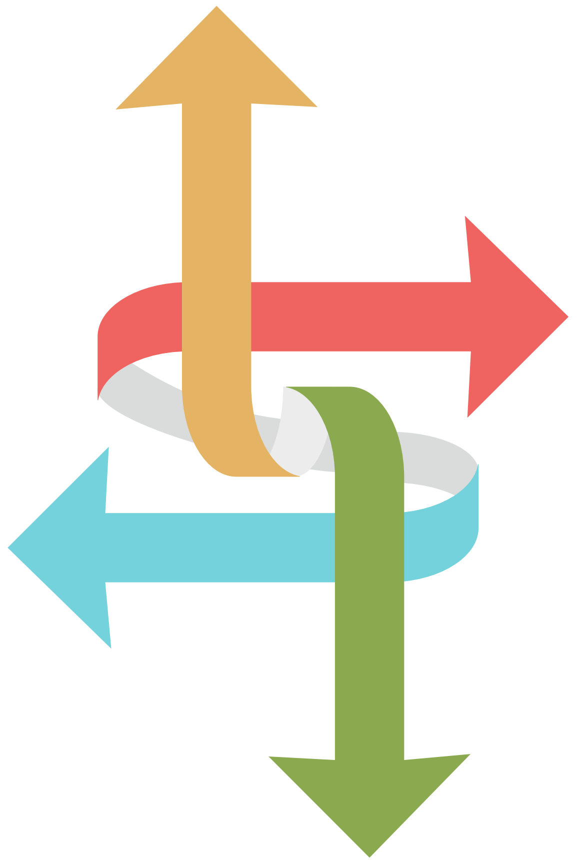
Example

```
# HELP http_request_duration_seconds A histogram of the request duration.
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.1"} 24054
http_request_duration_seconds_bucket{le="0.2"} 33444
http_request_duration_seconds_bucket{le="0.5"} 100392
http_request_duration_seconds_bucket{le="1"} 129389
http_request_duration_seconds_bucket{le="+Inf"} 144320
http_request_duration_seconds_sum 53423
http_request_duration_seconds_count 144320
```

This example demonstrates tracking HTTP request durations, offering insights into the distribution of request times.

Histogram Metrics: Uses

Monitors the range of file sizes or payloads in a file transfer system



Analyzes the distribution of request times or response durations in a web application

Tracks the distribution of network round-trip times or packet sizes

Measures the duration of batch jobs or data processing tasks

Histogram Metrics: Components

The following metric components work together in Prometheus histograms to give a detailed view of value distributions over time and support in-depth analysis and monitoring:

_count

Counts the total number of measurements

_sum

Adds up the total sum of all measured values

_bucket

Groups data points with upper limits set by the '**le**' label

Summary Metrics

They provide configurable quantile values, along with optional counts and sums, without exposing the entire distribution of observed values.

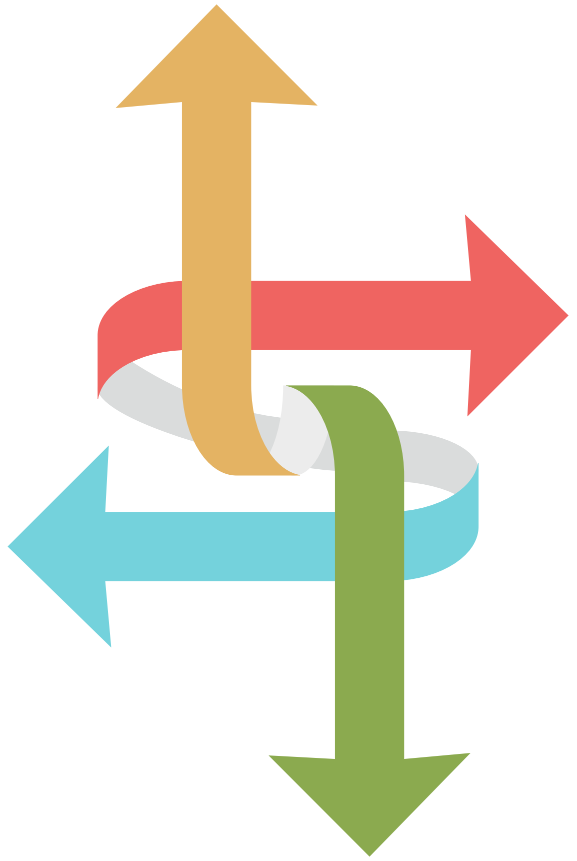
Example

```
# HELP rpc_durations_summary A summary of the RPC request durations.
# TYPE rpc_durations_summary summary
rpc_durations_summary{quantile="0.5"} 0.07
rpc_durations_summary{quantile="0.9"} 0.1
rpc_durations_summary{quantile="0.99"} 0.2
rpc_durations_summary_sum 1.7
rpc_durations_summary_count 20
```

This example tracks the duration of RPC requests, providing quantiles and total counts, which are useful for performance analysis.

Summary Metrics: Uses

Calculates quantiles, such as the 95th percentile, for request latencies in microservices



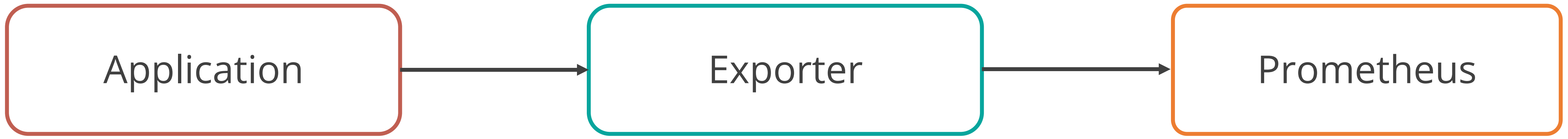
Tracks the range of response sizes in a content delivery network

Measures the processing times of events in a streaming data pipeline

Assesses the execution times of database queries

What Are Exporters?

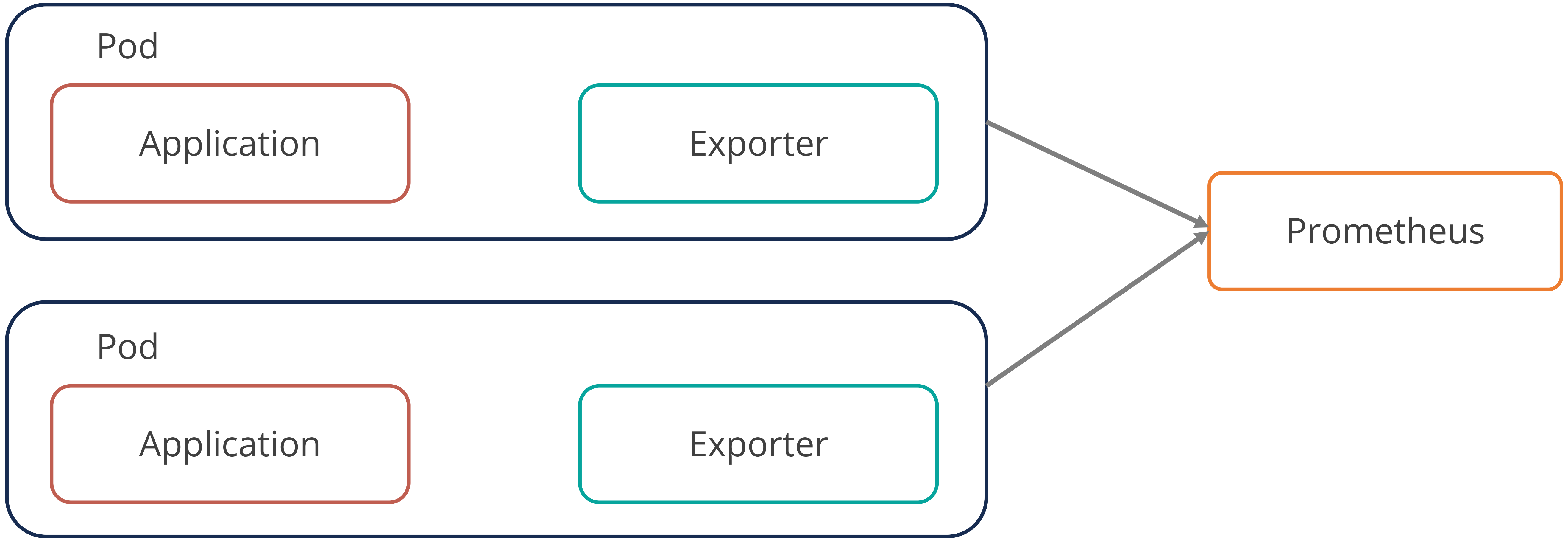
These are specialized components that collect metrics from different systems, services, or applications and format them so that monitoring tools can analyze and scrape them.



Many applications and infrastructure components do not natively expose Prometheus-compatible metrics, so exporters collect and translate these metrics into the required format.

Exporters for Efficient Monitoring

Prometheus can monitor multiple applications; however, it is recommended to use pods where each pod contains a single application and its corresponding exporter.

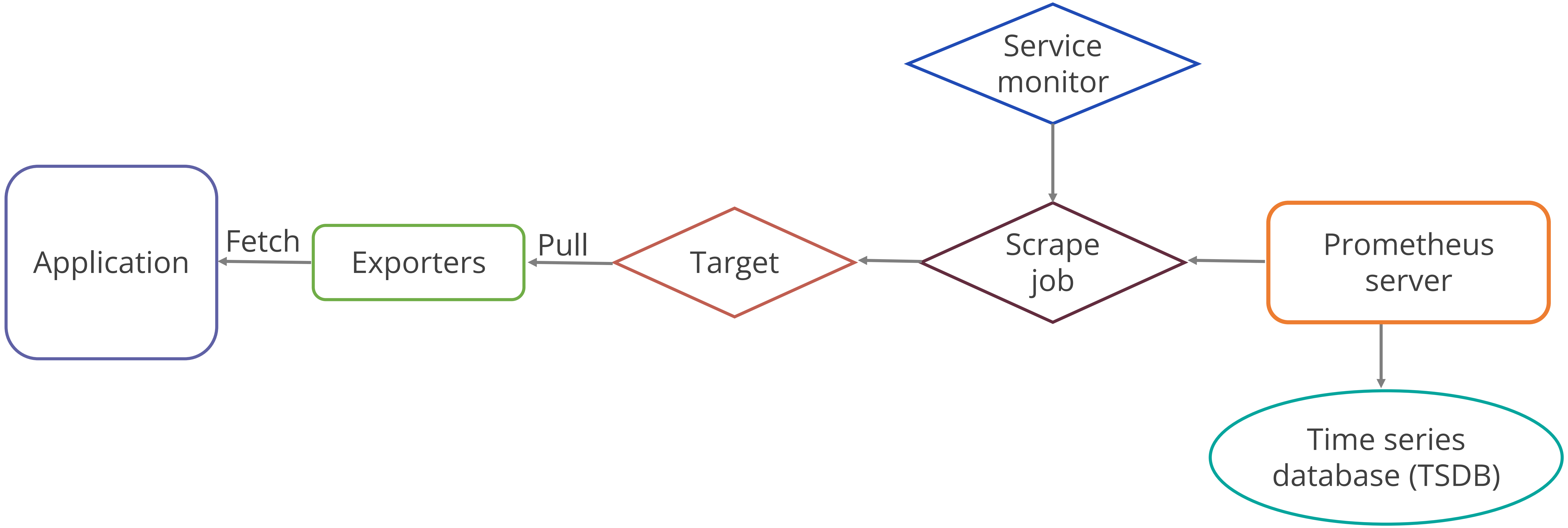


What Exporters Do?

- Collect a wide range of metrics, including CPU, memory, disk I/O, and network statistics
- Support third-party integrations or custom tools for specific services
- Monitor system components, providing insights into system functionality and health
- Help Prometheus gather time-series data for creating dashboards, alerts, and graphs

How Exporters Work?

The below diagram illustrates this process:



The diagram shows how application data is fetched by exporters, pulled by targets, and then scraped by a service monitor into the Prometheus server, which stores the data in a TSDB.

How Exporters Work?

Application: The process starts with an application or system that generates metrics.

Fetch: Exporters fetch these metrics from the application.

Exporters: The exporters then process and format the metrics.

Pull action: Prometheus pulls the metrics from the exporters.

Target: Exporters prepare formatted metrics for Prometheus to scrape.

How Exporters Work?

Scrape job: Prometheus scrapes metrics based on configuration settings and applies relabeling.

Service monitor: It manages Prometheus configuration to scrape metrics from specified targets.

Prometheus server: The collected metrics are stored in the Prometheus server.

TSDB: The time-series data is stored in the time series database (TSDB) for further analysis and visualization.

Essential Prometheus Exporters for System Monitoring

The Prometheus exporters to enhance system monitoring efficiency include:

Node exporter	Provides system metrics from Unix based systems
MySQL exporter	Collects performance metrics from MySQL databases
PostgreSQL exporter	Collects metrics from PostgreSQL databases
JMX exporter	Retrieves metrics from Java applications via JMX (Java Management Extensions)
Blackbox exporter	Performs black-box probing of endpoints using different protocols

Importance of Exporters

These are vital in Prometheus for monitoring diverse components and offering comprehensive observability through the following key aspects:

Integrate non-instrumented system

Enables Prometheus to collect metrics from systems that do not natively expose them and ensures seamless monitoring integration

Monitor heterogeneous environment

Allows collection of metrics from diverse technologies and ensures comprehensive monitoring coverage

Importance of Exporters

These are vital in Prometheus for monitoring diverse components and offering comprehensive observability through the following key aspects:

Consistent metric collection

Standardizes metrics collection to align with Prometheus format and simplifies the analysis process

Community-driven ecosystem

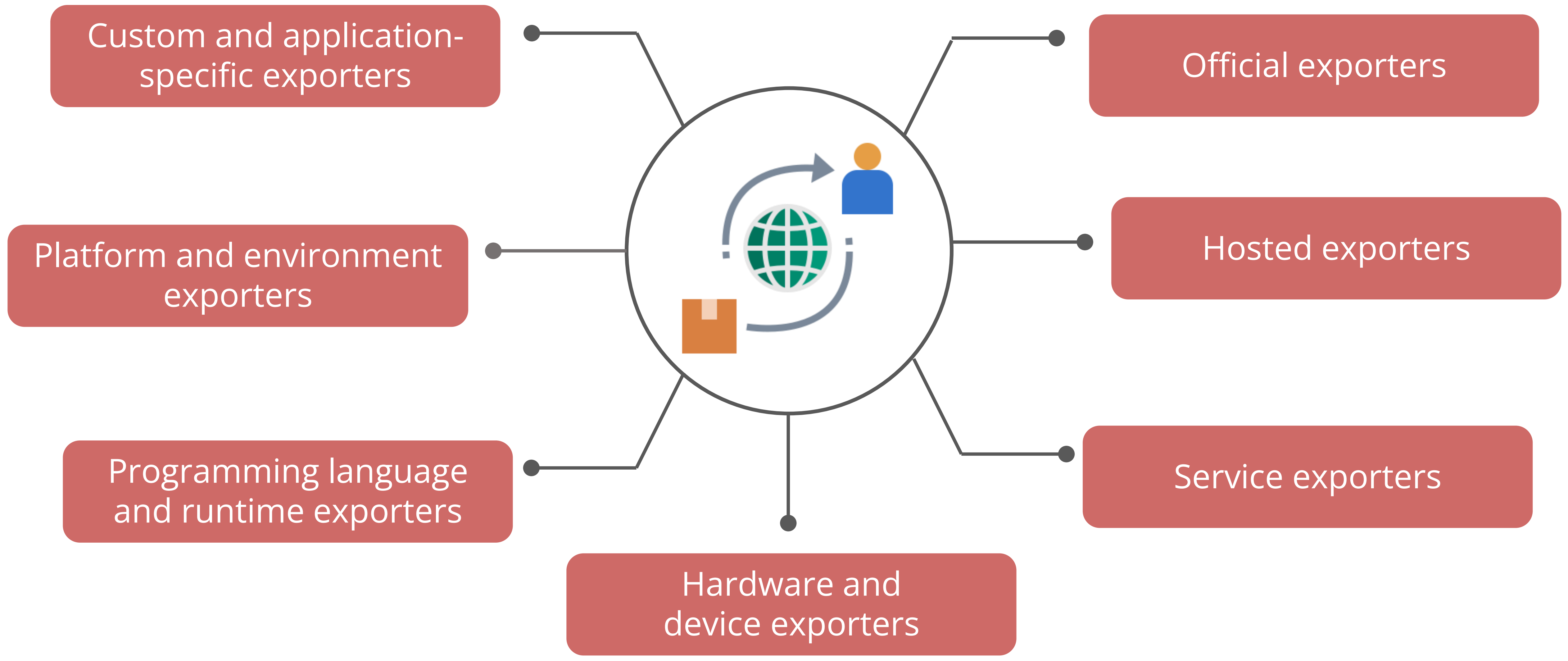
Promotes community growth by encouraging adoption, contribution, and innovation within Prometheus

Scalability and performance

Supports scalable deployment and improves resource management and efficiency in large environments

Categories of Exporters

Prometheus provides various exporters for different systems, services, and technologies. These exporters are classified into the following categories:



Custom and application-specific exporters
Official exporters
Platform and environment exporters
Hosted exporters
Programming language and runtime exporters
Service exporters
Hardware and device exporters

Categories of Exporters

Exporters are classified into the following categories:

Official exporters

They are developed and maintained by the Prometheus project itself to ensure compatibility and reliability.

Hosted exporters

They are third-party exporters hosted and maintained by their respective organizations or communities.

Service exporters

They are designed to collect metrics from specific services or technologies such as databases, message queues, and web servers.

Hardware and device exporters

They gather metrics from various platforms, environments, and infrastructure components like cloud providers.

Categories of Exporters

Exporters are classified into the following categories:

Programming language and runtime exporters

They collect metrics specific to programming languages, runtimes, and their associated libraries or frameworks.

Platform and environment exporters

They are designed to collect metrics from hardware devices, sensors, and other components.

Custom and application-specific exporters

They are custom-built by developers to expose application-specific metrics for tailored monitoring.

Relabeling

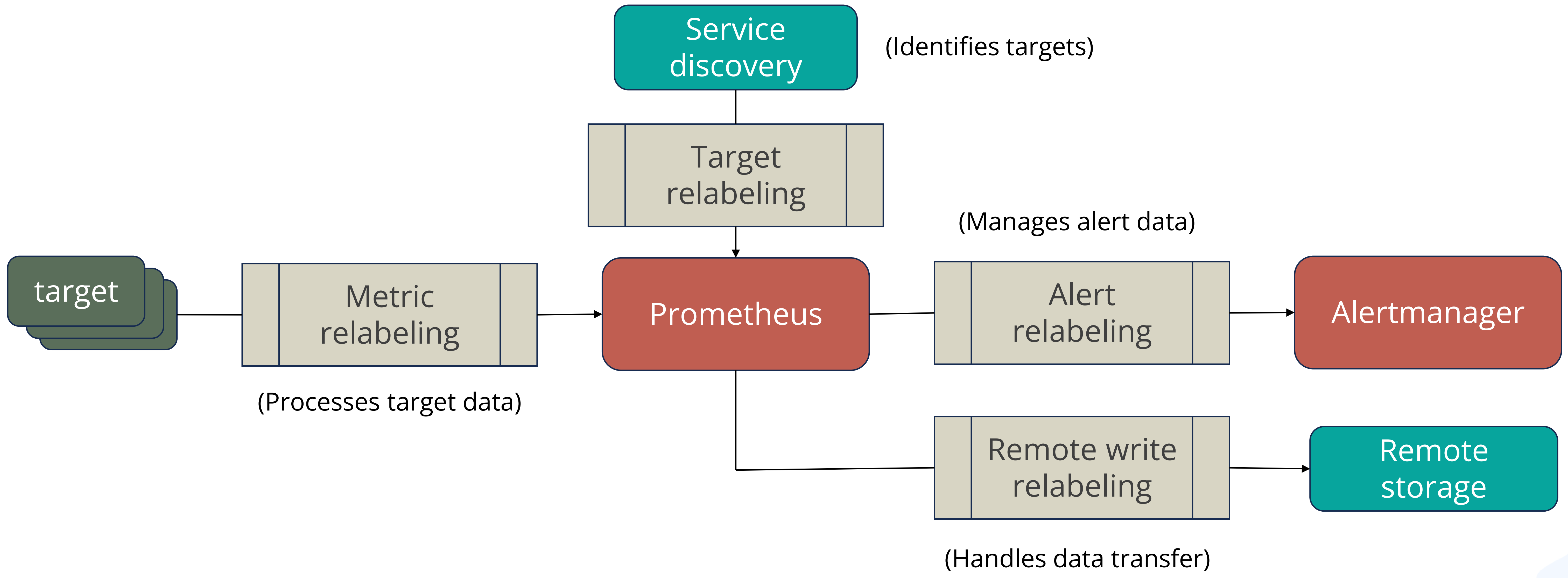
It is a feature in Prometheus that allows manipulation and transformation of metric labels before ingestion. It is used for:

- Label renaming
- Label filtering
- Label addition
- Label manipulation

It is performed during the scrape configuration and is essential for normalizing, filtering, and routing metrics according to specific criteria.

Relabeling Pipeline in Prometheus Ecosystem

The below diagram illustrates the various relabeling processes within Prometheus:



Relabeling Pipeline in Prometheus Ecosystem

The various relabeling processes within Prometheus are:

Target relabeling	This stage adjusts target labels during service discovery to ensure Prometheus scrapes the correct data from each source.
Metric relabeling	This stage modifies or removes metric labels after scraping to refine the data that is stored and queried in Prometheus.
Alert relabeling	This stage adjusts alert labels or removes unnecessary alerts to manage and route them effectively to Alertmanager.
Remote write relabeling	This stage filters and formats metrics data to ensure only relevant information is stored in remote storage for long-term use.

Relabeling Rules: Renaming a Label

This rule involves changing the name of an existing label to a new one.

Example

```
— source_labels: [_name_]
target_label: _metric_name_
```

The source label __*name*__ is renamed to a target label __*metric_name*__ during the relabeling process in Prometheus.

Relabeling Rules: Filtering Metrics

This rule allows for selective retention of metrics based on specific criteria.

Example

```
— source_labels: [_name_]
regex: 'go_.*'
action: keep
```

This filters metrics by keeping only those that match a specific pattern ***go_**** using a regex applied to the source label __***name***__.

Relabeling Rules: Manipulating Label Values

This rule modifies the value of a label based on a regex match.

Example

```
— source_labels: [_address_]
target_label: _instance_
regex: '(.*):(\d+)'
replacement: '$1'
```

Relabeling rule takes an address with a port and matches it with the regex to separate the host from the port, and then sets the instance label to just the host name.

Relabeling Rules: Dropping Labels

This rule removes metrics that match a certain label value.

Example

```
— source_labels: [job]
regex: 'node—exporter’
action: drop
```

Metrics with a **[job]** label matching **node-exporter** are discarded to exclude irrelevant data.

Relabeling Rules: Keeping Labels

This rule retains only the metrics that match a certain label value.

Example

```
— source_labels: [job]
regex: 'prometheus'
action: keep
```

Metrics with a **[job]** label matching **prometheus** are kept while all others are filtered out.

Advanced Exporter Configurations

Prometheus exporters provide advanced configuration options to customize monitoring for specific requirements. These options help to:

1. Change metric exposure path
2. Enable or disable metric collection
3. Adjust scrape interval and timeout
4. Apply relabeling and metric transformation
5. Configure target and service discovery

Change Metric Exposure Path

Prometheus exporters can be configured to expose metrics on a specific path to meet integration requirements.

Example

```
--web.telemetry-path="/metrics"
```

web.telemetry-path="/metrics" sets the exporter to serve metrics at */metrics*, ensuring monitoring tools find the data at the expected location.

Enable or Disable Metric Collection

Prometheus exporters can be configured to selectively enable or disable the collection of specific metrics based on monitoring needs.

Example

```
--collect.info_schema.userstats (enable),
--collect.binlog_size=false (disable)
```

The example demonstrates how to enable collection for user statistics with --*collect.info_schema.userstats* and disable collection for binary log size with *--collect.binlog_size=false*.

Adjust Scrape Interval and Timeout

Prometheus exporters allow setting specific intervals and timeouts for data scraping to optimize performance and reliability.

Example

```
--jmx.interval=30s --jmx.timeout=10s
```

The example sets --*scrape.interval=30s* to collect data every 30 seconds and --*scrape.timeout=10s* to ensure each scrape completes within 10 seconds.

Apply Relabeling and Metric Transformation

Prometheus exporters support relabeling and metric transformation to modify metric labels and refine data collection.

Example

```
--relabel-configs="--relabel-
configs=<relabel_config>"
```

The example uses *--relabel-configs=<relabel_config>* to apply specific relabeling rules and transformations. This configuration allows altering metric labels or dropping unnecessary metrics.

Configure Target and Service Discovery

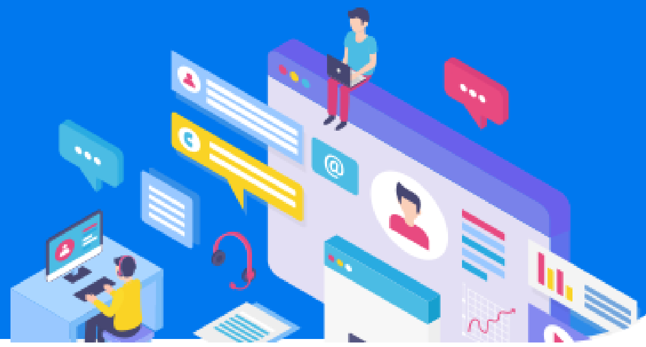
Prometheus exporters support dynamic target and service discovery to automatically detect and monitor services across different environments.

Example

```
--aws.ecs-cluster=my-cluster --aws.ecs-service-
name=my-service
```

--aws.ecs-cluster=my-cluster and *--aws.ecs-service-name=my-service* are used to specify the AWS ECS cluster and service to monitor. It enables automatic detection of services within the specified cluster.

Assisted Practice



Setting up and Monitoring with Node Exporter **Duration: 10 Min.**

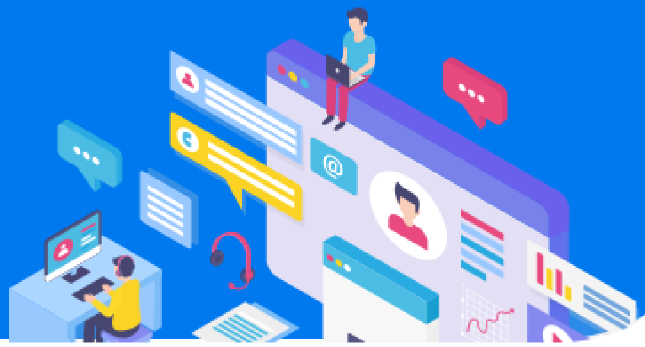
Problem statement:
You have been assigned a task to set up and monitor Node Exporter, which exposes system metrics from a Linux host, making it a valuable tool for monitoring hosts within a Prometheus ecosystem.

Outcome:

By the end of this demo, you will be able to deploy and use Node Exporter to gather system metrics on a Linux host for monitoring in a Prometheus ecosystem.

Note: Refer to the demo document for detailed steps:
01_Setting_up_and_Monitoring_with_Node_Exporter

Assisted Practice: Guidelines



Steps to be followed:

1. Download and set up Node Exporter on localhost
2. Set up a Prometheus server on localhost to scrape Node Exporter metrics
3. Access Node Exporter metrics using Prometheus UI

Quick Check

A system administrator is setting up a Prometheus exporter to monitor a MySQL database and wants to ensure that only the metrics relevant to query performance are collected. Which advanced exporter configuration would be useful?

A. Adjust scrape interval and timeout

B. Enable or disable metrics collection

C. Change metric exposure path

D. Configure target discovery

PromQL: Prometheus Query Language

What Is PromQL?

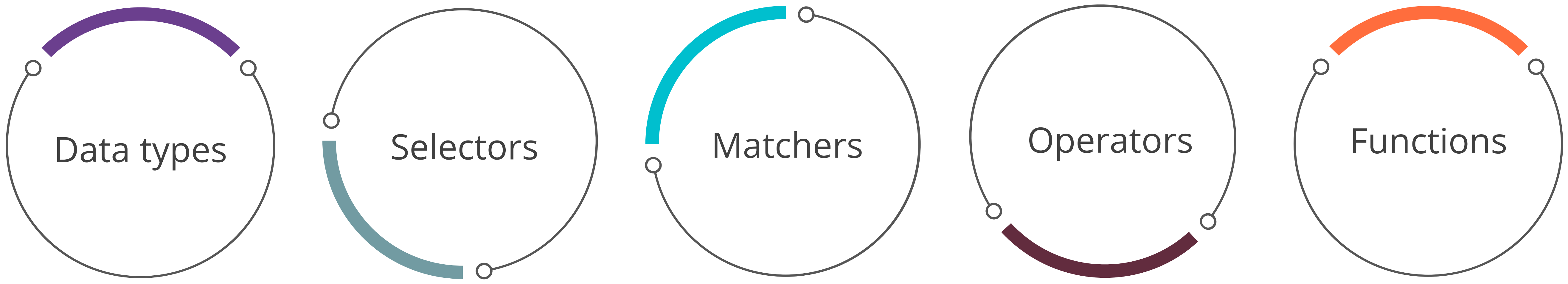
It is a powerful and flexible language used to query and retrieve data stored in Prometheus.

PromQL

It is an open-source systems monitoring and alerting toolkit.

Elements in PromQL

Core elements in PromQL for efficient metric querying and analysis include:



Data Types in PromQL

The following are used to evaluate an expression or sub-expression in PromQL:

String

Represents a simple text value

Instant vector

Represents a set of time series containing a single sample for each time series

Scalar

Represents a numerical floating-point value without any time series context

Range vector

Represents a set of time series data points over a specified time range

Selectors

It specifies a set of time series based on their labels, including an optional metric name and a set of label matchers enclosed in curly braces {}. The below example illustrates this:

```
node_cpu_seconds_total{job="node"}
```

This example retrieves time series data for **node_cpu_seconds_total** where the **job** label is equal to **node**, allowing for targeted analysis of CPU metrics specific to that job.

Matchers

They filter metrics by applying specific conditions related to labels or timestamps.

The below example shows the usage of matchers:

```
instance=~"web-server-[0-9]+"
```

Here, **instance=~"web-server-[0-9]+"** filters metrics to include only those with an instance label that matches the pattern **web-server-** followed by one or more digits.

Types of Matchers

They are categorized into the following types:

Equality matcher

Filters labels that are exactly equal to the specified value or string

Syntax:

```
label_name="label_value"
```

Not equality matcher

Filters labels that are not equal to the provided value or string

Syntax:

```
label_name!="label_value"
```

Types of Matchers

They are categorized into the following types:

Regular expression matcher

Filters labels that match a given regular expression pattern

Syntax:

```
label_name=~"regex_pattern"
```

Negative regular expression matcher

Filters out labels that do not match the specified regular expression pattern

Syntax:

```
label_name!~"regex_pattern"
```

Combining Multiple Matchers in a Selector

Selectors and matchers can be used together to form sophisticated queries that accurately identify the desired metrics for analysis.

The below example illustrates this:

```
node_cpu_seconds_total{job="node",
mode=~"(user|system)"}
```

This example filters user and system CPU modes from the ***node_cpu_seconds_total metric***, enabling analysis of other modes like **idle** for the node job.

Binary Operators

It allows operations on two instant vectors or scalar values.

The following are the commonly used binary operators:

Arithmetic operators

These are used to perform basic mathematical calculations between two instant vectors or scalar values in PromQL.

Comparison operators

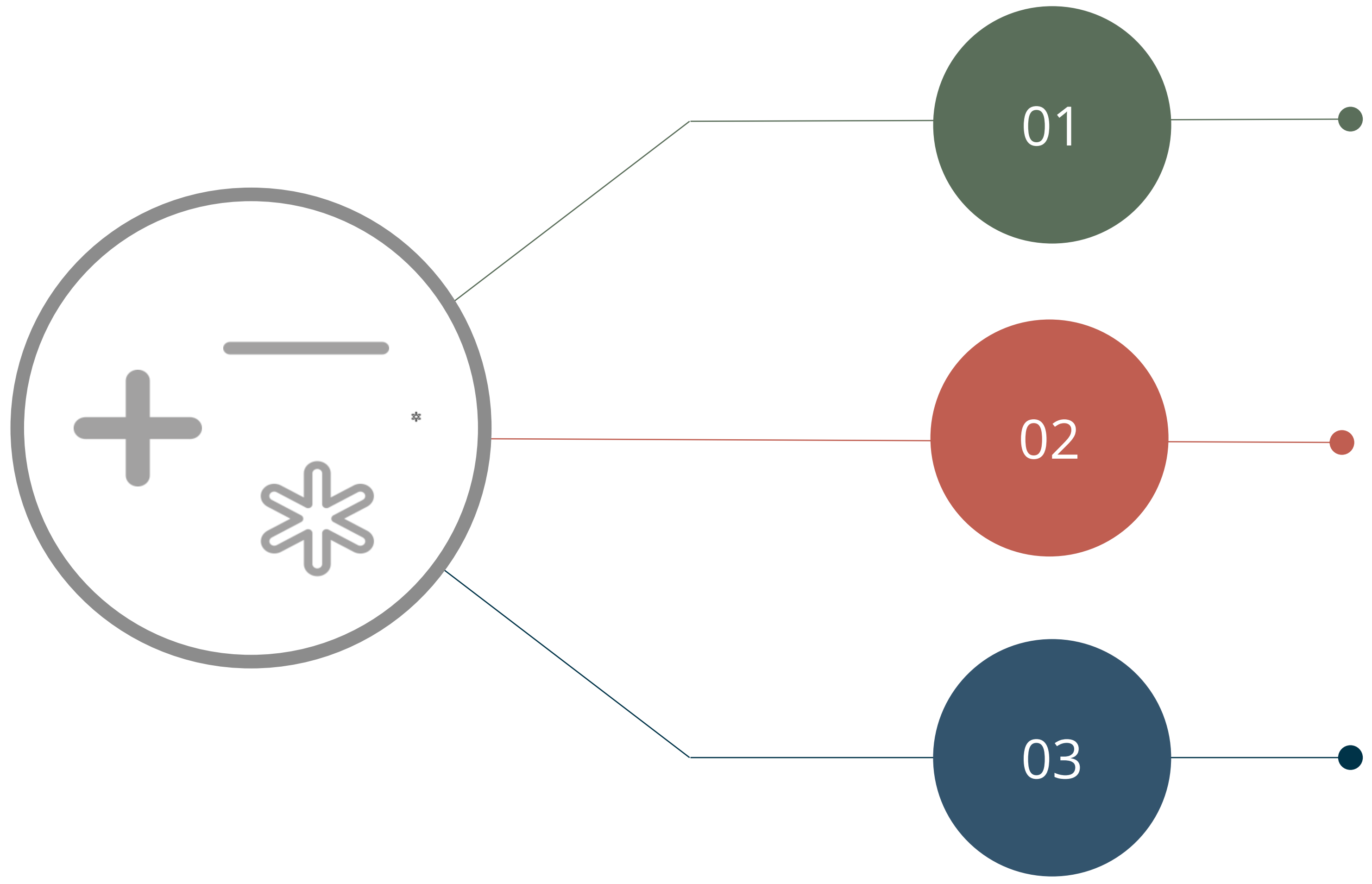
These are used to compare two instant vectors or scalar values, returning a Boolean result based on the comparison.

Logical or set operators

These are used to combine, filter, or modify sets of time series data based on logical conditions.

Arithmetic Operators

The following operators are used for performing mathematical calculations:



+ (addition)
Adds two values

- (subtraction)
Subtracts one value from another

*** (multiplication)**
Multiplies two values

Arithmetic Operators

The following operators are used for performing mathematical calculations:

04

/ (division)
Divides one value by another

05

% (modulus)
Returns the remainder of the division

06

^ (power or exponentiation)
Raises one value to the power of another

Arithmetic Operators

The following are examples of these operators:

Convert bytes to megabytes

```
node_memory_MemAvailable_bytes / 1024
/ 1024
```

Convert bytes to bits

```
rate(node_network_receive_bytes_total
[5m]) * 8
```

These operators are used to normalize and convert data units for easier interpretation and analysis.

Comparison Operators

The following operators are used for comparing values:

01

== (equal)

Checks if two values are equal

02

!= (not-equal)

Checks if two values are not equal

03

> (greater-than)

Checks if one value is greater than another

Comparison Operators

The following operators are used for comparing values:

04

< (less-than)

Checks if one value is less than another

05

>= (greater-than-or-equal-to)

Checks if one value is greater than or equal to another

06

<= (less-than-or-equal-to)

Checks if one value is less than or equal to another

Comparison Operators

The following are examples of these operators:

Filters for high CPU usage

```
node_cpu_seconds_total > 0.9
```

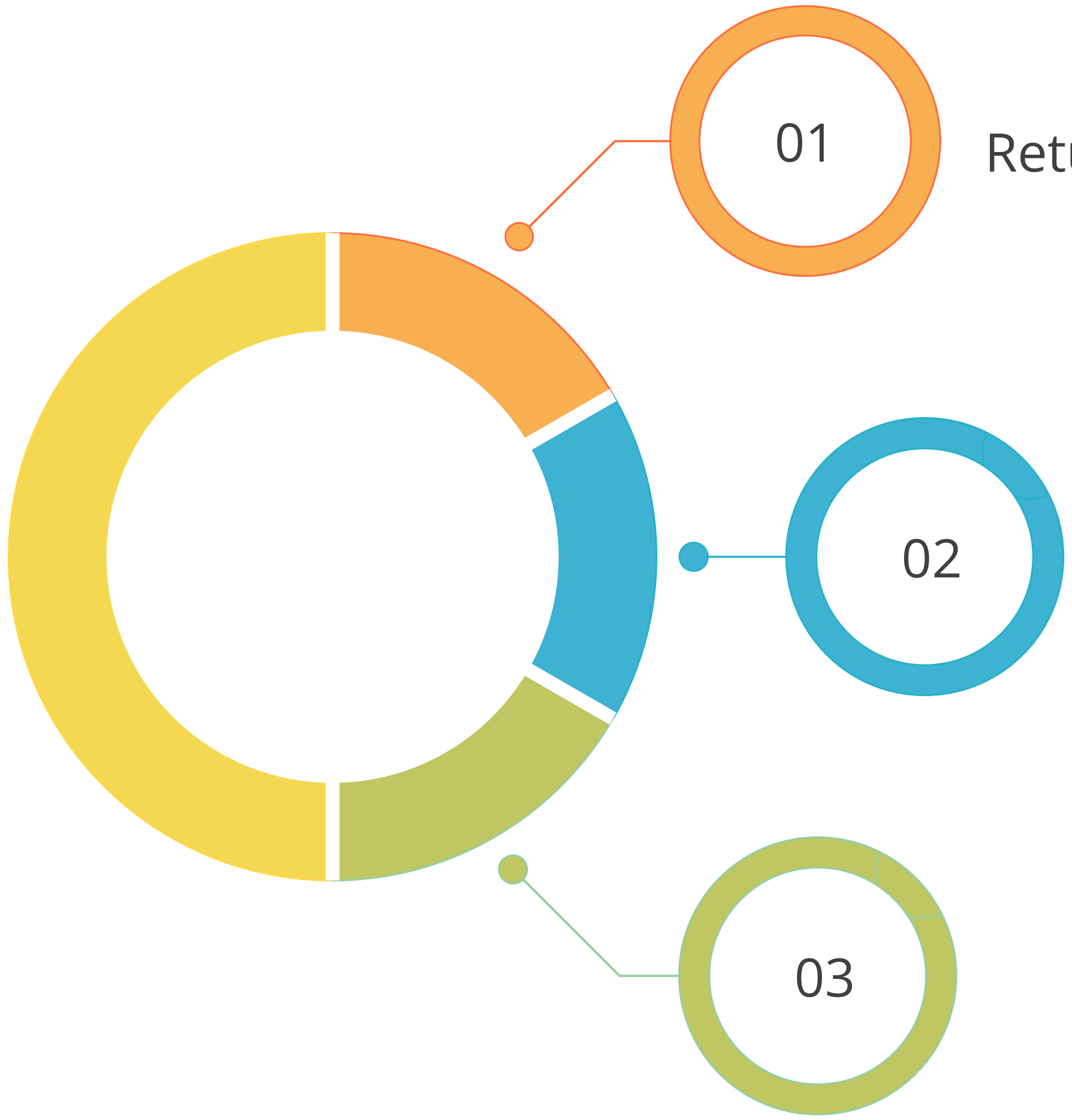
Filters for low memory availability

```
node_memory_MemAvailable_bytes /
node_memory_MemTotal_bytes < 0.2
```

These operators help to identify nodes with critical resource usage levels for performance monitoring.

Logical and Set Operators

The following operators are used for combining or filtering sets of time series data:



and (intersection)
Returns the intersection of two sets

or (union)
Combines two sets into one

unless (complement)
Excludes elements in one set from another

Logical and Set Operators

The following are examples of these operators:

Combines user and system CPU time series

```
node_cpu_seconds_total{mode="user"} or
node_cpu_seconds_total{mode="system"}
```

Excludes root filesystem from the result

```
node_filesystem_free_bytes{mountpoint="/
root"} unless
node_filesystem_free_bytes{mountpoint="/
"}
```

These operators are used to aggregate relevant data while excluding non-essential metrics for a more focused analysis.

Filtering Labels with *ignoring* Keyword

It excludes specified labels from vector matching, enabling operations on metrics without considering the excluded labels.

Example

```
node_memory_MemFree_bytes + ignoring(instance, job)
node_memory_Cached_bytes
```

This example adds the *node_memory_MemFree_bytes* and *node_memory_Cached_bytes* metrics while ignoring the instance and job labels, allowing the operation to focus on other matching labels.

Filtering Labels with *on* Keyword

It focuses operations on specific labels, ensuring that only those labels are used for vector matching.

The below example illustrates the use of this keyword:

Example

```
node_memory_MemFree_bytes + on(instance, job)
node_memory_Cached_bytes
```

This example adds the *MemFree* and *Cached memory* metrics while using the instance and job labels to ensure the operation is matched based on these labels.

Aggregation Operators

It combines multiple time series into a single time series or aggregates data across specified dimensions. The key operators are:

sum

- Computes the total of all elements in the vector

Example: sum(node_cpu_seconds_total)

min

- Finds the smallest value among the elements in the vector

Example: min(node_memory_MemAvailable_bytes)

Aggregation Operators

It combines multiple time series into a single time series or aggregates data across specified dimensions. The key operators are:

max

- Identifies the largest value across elements in the vector

Example: max(node_network_receive_bytes_total)

avg

- Calculates the average value across elements in the vector

Example: avg(node_disk_io_time_seconds_total)

Aggregation Operators

It combines multiple time series into a single time series or aggregates data across specified dimensions. The key operators are:

count

- Counts the number of elements in the vector

Example: count(node_cpu_seconds_total)

quantile

- Determines the φ-quantile (where 0 ≤ φ ≤ 1) of the elements in the vector

Example: quantile(0.95,rate(node_network_receive_bytes_total[5m]))

Aggregation Operators

It combines multiple time series into a single time series or aggregates data across specified dimensions. The key operators are:

stddev

- Computes the standard deviation of elements in the vector

Example: stddev(node_cpu_seconds_total)

stdvar

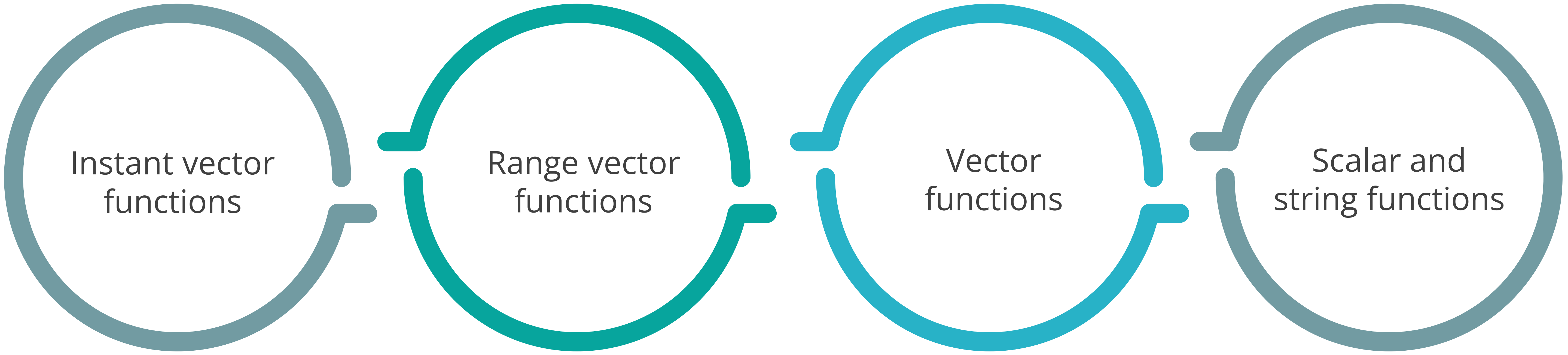
- Calculates the standard variance across elements in the vector

Example: stdvar(node_memory_MemAvailable_bytes)

Functions

These are built-in operations that can manipulate time series data to perform specific calculations.

The following are some commonly used functions:



Functions in Prometheus analyze, transform, and aggregate time series data to extract insights and monitor system performance.

Instant Vector Functions

These evaluate a single data point from each time series at a specific time.

Here are a few examples of the instant vector functions:

rate()

- Computes the average per-second rate of a counter's increase over a specified time range

Example: rate(node_network_receive_bytes_total[5m])

delta()

- Calculates the difference between the last two samples in a range vector

Example: delta(node_memory_MemAvailable_bytes[5m])

Instant Vector Functions

These evaluate a single data point from each time series at a specific time.

Here are a few examples of the instant vector functions:

irate()

- Determines the instantaneous per-second rate of a counter's increase

Example: irate(node_cpu_seconds_total[1m])

increase()

- Measures the cumulative increase in a counter over a specified time range

Example: increase(http_requests_total{job="api-server"}[5m])

predict_linear()

- Forecasts the future value of a time series based on its recent trend

Example: predict_linear(node_filesystem_free_bytes[1h], 2*3600)

Range Vector Functions

These aggregate multiple data points over a defined time range for deeper insights. They include the following functions:

avg_over_time()

- Computes the average value of a range vector over a specified time range

Example: avg_over_time(node_cpu_seconds_total[5m])

min_over_time()

- Finds the minimum value of a range vector over a specified time range

Example: min_over_time(node_memory_MemAvailable_bytes[1h])

sum_over_time()

- Calculates the total sum of values in a range vector over a specified time range

Example: sum_over_time(node_disk_io_time_seconds_total[1d])

Range Vector Functions

These aggregate multiple data points over a defined time range for deeper insights. They include the following functions:

quantile_over_time()

- Determines the φ-quantile of a range vector over a specified time range

Example:
quantile_over_time(0.95, node_network_transmit_bytes_total[10m])

max_over_time()

- Identifies the maximum value of a range vector over a specified time range

Example: max_over_time(node_network_receive_bytes_total[30m])

Vector Functions

These functions manipulate and organize vectors by sorting, filtering, or selecting elements.

Some examples of the vector functions are as follows:

sort()

- Arranges a vector by a specified label in ascending order

Example: sort(rate(http_requests_total[5m]))

sort_desc()

- Arranges a vector by a specified label in descending order

Example: sort_desc(cpu_usage)

Vector Functions

These functions manipulate and organize vectors by sorting, filtering, or selecting elements.

Some examples of the vector functions are as follows:

topk()

- Retrieves the top k elements with the largest values in a vector

Example: topk(5, rate(node_network_receive_bytes_total[1h]))

bottomk()

- Retrieves the bottom k elements with the smallest values in a vector

Example: bottomk(5, disk_free_bytes)

Scalar and String Functions

These process individual numeric values or text for mathematical operations or formatting.

Here are a few examples of the scalar and string functions:

abs()

- Computes the absolute value of a scalar

Example: abs(sum_over_time(rate(errors_total[5m])))

ceil()

- Rounds a scalar value up to the nearest integer

Example: ceil(avg_over_time(node_cpu_seconds_total{mode="user"}[1h]) * 100)

floor()

- Rounds a scalar value down to the nearest integer

Example: floor(memory_usage_bytes / 1024 / 1024)

Scalar and String Functions

These process individual numeric values or text for mathematical operations or formatting.

Here are a few examples of the scalar and string functions:

round()

- Rounds a scalar value to the nearest integer

Example: round(avg_over_time(rate(http_requests_total[5m])) * 100)

exp()

- Computes the exponential value of a scalar

Example: exp(sum_over_time(rate(logins_total[5m])))

ln()

- Determines the natural logarithm of a scalar value

Example: ln(rate(http_requests_total[5m]))

Aggregation Functions

These aggregate each series within a range vector over time and return an instant vector with the aggregated results for each series.

Here are a few examples of the aggregation function:

count_over_time()

- Counts the number of data points within the specified time range for each series

Example: count_over_time(errors_total[10m])

stddev_over_time()

- Calculates the standard deviation of the data points within the specified time range for each series

Example: stddev_over_time(node_memory_active_bytes[1h])

Aggregation Functions

These aggregate each series within a range vector over time and return an instant vector with the aggregated results for each series.

Here are a few examples of the aggregation function:

last_over_time()

- Retrieves the most recent data point within the specified time range for each series

Example: last_over_time(up[10m])

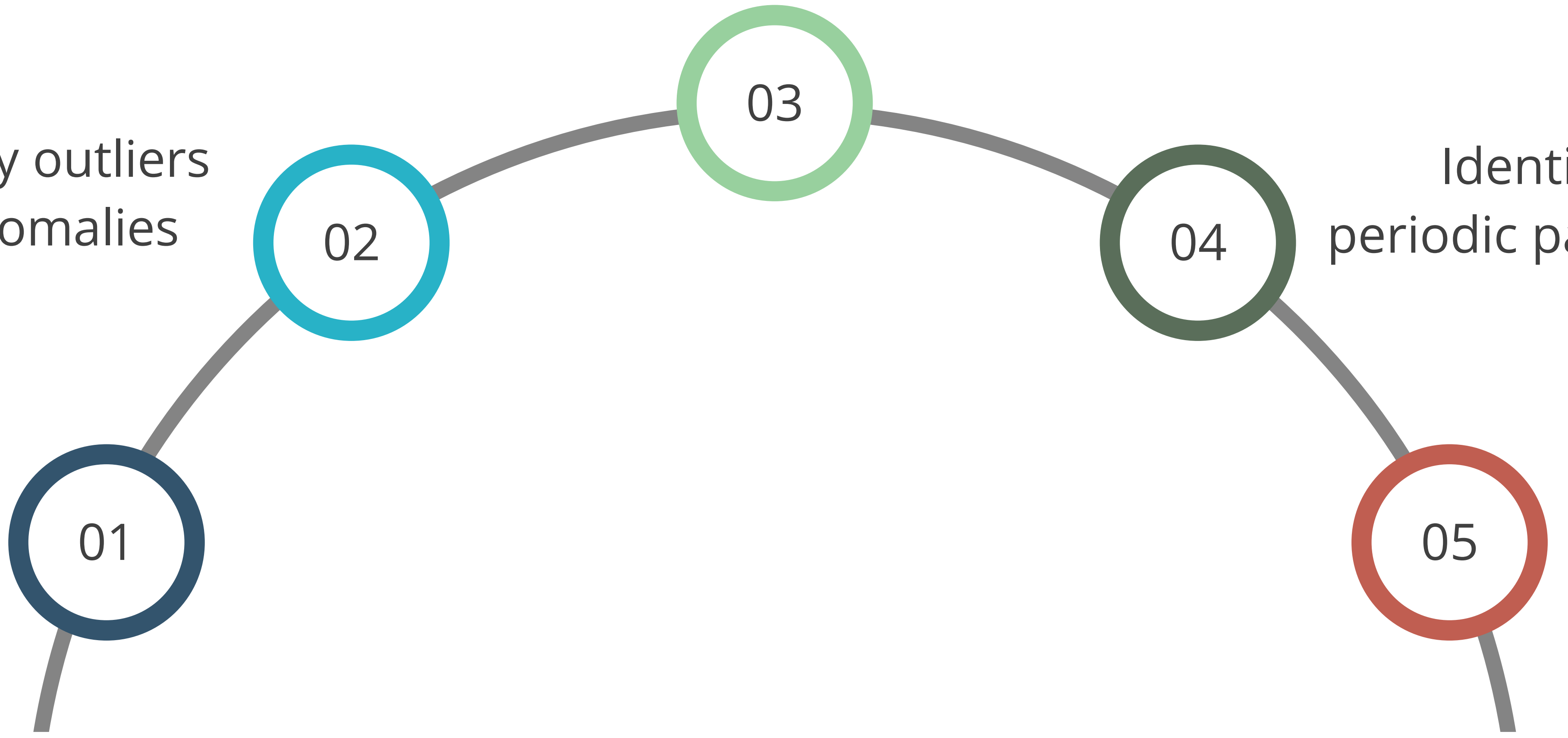
present_over_time()

- Returns a value of 1 for any series present within the specified time range, indicating its presence during that interval

Example: present_over_time(cpu_idle[5m])

Querying for Patterns and Anomalies

PromQL enables querying time series data to identify patterns and anomalies using various functions and operators. Common techniques include:



01 Detect spikes or sudden increases

02 Identify outliers or anomalies

03 Detect the absence of data or missing values

04 Identify periodic patterns

05 Combine multiple conditions

Importance of Querying for Patterns and Anomalies

Querying for patterns and anomalies in data keeps systems secure, efficient, and reliable. It allows for faster responses, better performance management, and early detection of risks for timely action.

Below are the key reasons why querying for patterns and anomalies is important:

Performance optimization	Detecting spikes or unusual data helps optimize resource usage, ensuring systems run efficiently and preventing slowdowns.
Proactive issue resolution	Recognizing anomalies helps identify issues early, fix problems quickly, reduce downtime, and improve user satisfaction.
Security enhancements	Detecting unusual patterns, like traffic spikes, signals security threats and enables quick responses.

Importance of Querying for Patterns and Anomalies

Data driven decisions

Recognizing data patterns provides insights for smarter decisions on capacity planning, scaling, and system adjustments.

Regulatory compliance

Monitoring anomalies ensures compliance with standards and helps pass audits smoothly.

Detect Spikes or Sudden Increases

The following approach identifies significant data spikes by measuring differences between consecutive samples and sorting them:

```
topk(5, sort_desc(delta(node_network_receive_bytes_total[5m])))
```

This query retrieves the top 5 network instances with the largest increase in network bytes received over the last 5 minutes.

Identify Outliers or Anomalies

The following approach flags data points that deviate significantly from the average by using statistical calculations like average and standard deviation:

```
node_cpu_seconds_total > (avg_over_time(node_cpu_seconds_total[1h]) +
3 * stddev_over_time(node_cpu_seconds_total[1h]))
```

This query highlights instances where CPU usage is more than 3 standard deviations above the average, suggesting anomalies.

Detect the Absence of Data or Missing Values

The following approach detects missing data by identifying periods where no change is recorded in the data:

```
delta(node_cpu_seconds_total[5m]) == 0
```

This query finds instances where CPU usage has not changed in the last 5 minutes, indicating possible data absence.

Identify Periodic Patterns

The following approach uncovers recurring patterns by calculating the rate of increase for a counter metric over time:

```
rate(node_network_receive_bytes_total[1h])
```

This query determines the per-second rate of network bytes received over the past hour, aiding in the detection of periodic traffic patterns.

Combine Multiple Conditions

The following approach uses logical operators to merge multiple conditions for complex pattern detection:

```
(node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes < 0.2)
and (delta(node_memory_MemAvailable_bytes[5m]) < 0)
```

This query detects situations where the available memory is less than 20% and is continually decreasing, indicating possible memory-related problems.

Optimizing Queries for Performance

Enhance the performance of PromQL queries on large datasets and complex scenarios with these methods:

Use efficient matchers and selectors

- Avoid using unnecessary matchers to speed up queries
- Prefer equality matchers over regular expressions whenever possible

Optimize range vector selectors

- Apply the most specific matchers first to enhance efficiency
- Divide complex queries into smaller time ranges and combine results

Optimizing Queries for Performance

Enhance the performance of PromQL queries on large datasets and complex scenarios with these methods:

Avoid unnecessary calculations

- Reduce the use of expensive functions like rate() and increase()
- Use count_scalar() when the requirement is to count the series

Leverage caching

- Use caching to improve the performance for repeated queries
- Modify queries minimally to take advantage of cached results

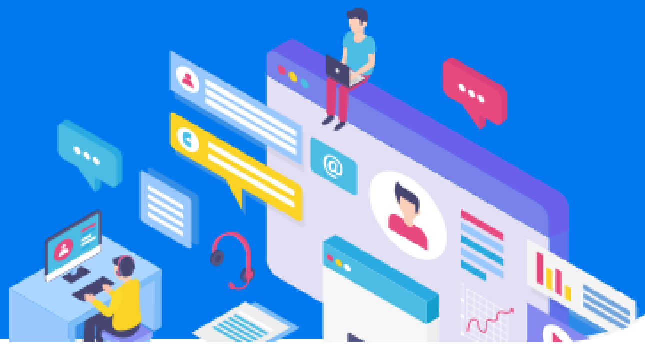
Optimizing Queries for Performance

Enhance the performance of PromQL queries on large datasets and complex scenarios with these methods:

Use subqueries and evaluate time ranges

- Divide complex queries into subqueries and combine them
- Choose appropriate time ranges and use the offset parameter if needed

Assisted Practice



Writing Basic Queries in PromQL **Duration: 10 Min.**

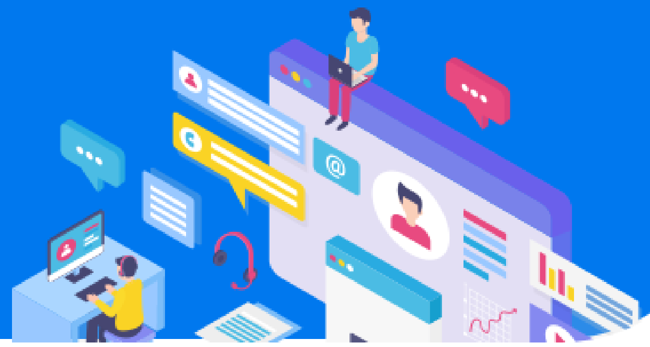
Problem statement:
You have been assigned a task to query and analyze monitoring data using Prometheus Query Language (PromQL) for effective monitoring of system performance and health.

Outcome:

By the end of this demo, you will be able to construct and execute basic queries in Prometheus Query Language (PromQL) for efficient system performance and health monitoring.

Note: Refer to the demo document for detailed steps:
02_Writing_Basic_Queries_in_PromQL

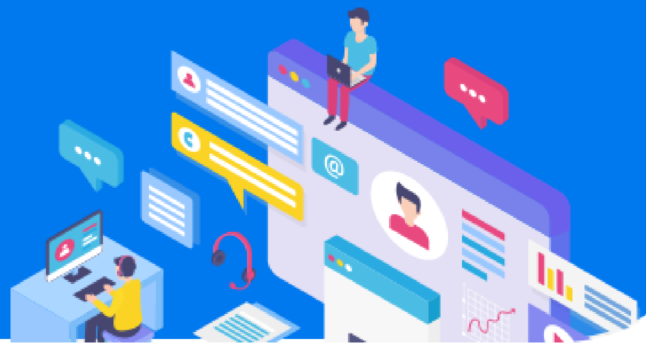
Assisted Practice: Guidelines



Steps to be followed:

1. Query to retrieve a single metric
2. Filter by label
3. Aggregate data with the sum() function
4. Query data using an arithmetic operation
5. Calculate a metric using the rate() function

Assisted Practice



Writing Advanced Queries for Real-Life Use Cases **Duration: 10 Min.**

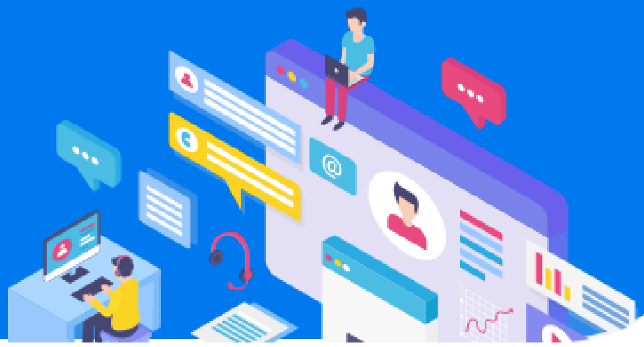
Problem statement:
You have been assigned a task to write complex PromQL queries for detecting performance issues, monitoring resource utilization, and setting up alerts

Outcome:

By the end of this demo, you will be able to formulate advanced PromQL queries to efficiently detect performance issues, monitor resource utilization, and configure alerts for real-time incident management.

Note: Refer to the demo document for detailed steps:
03_Writing_Advanced_Queries_for_Real_Life_Use_Cases

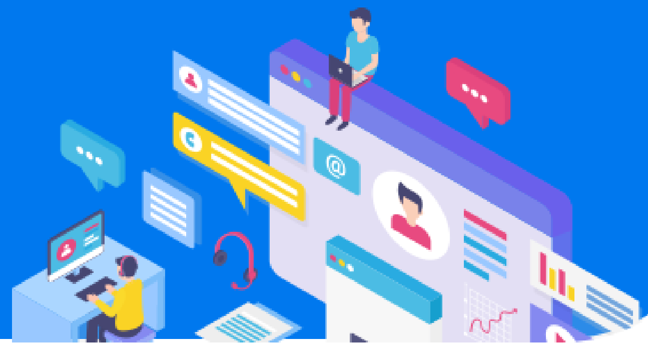
Assisted Practice: Guidelines



Steps to be followed:

1. Set alerts for high memory usage
2. Analyze disk I/O patterns for performance issues
3. Track node uptime for maintenance alerts
4. Monitor network traffic patterns to identify bottlenecks

Assisted Practice



Writing PromQL Queries to Extract Specific Metrics from a Sample Dataset **Duration: 10 Min.**

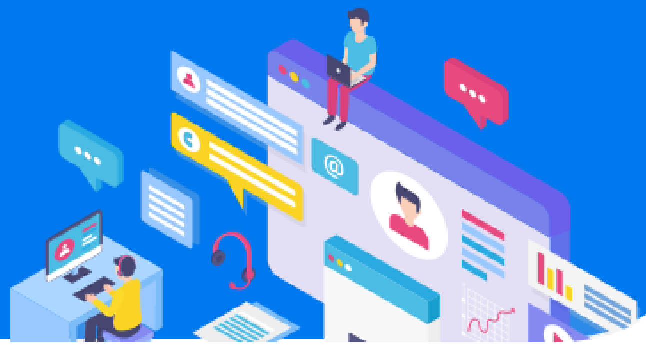
Problem statement:
You have been assigned a task to demonstrate the process of writing PromQL queries to extract and analyze specific metrics from a Node Exporter dataset using the Prometheus UI.

Outcome:

By the end of this demo, you will be able to craft PromQL queries to precisely extract and analyze specific metrics from a Node Exporter dataset using the Prometheus UI.

Note: Refer to the demo document for detailed steps:
04_Writing_PromQL_Queries_to_Extract_Specific_Metrics_from_a_Sample_Dataset

Assisted Practice: Guidelines



Steps to be followed:

1. Set up Prometheus and Node Exporter using Docker
2. Use the Prometheus UI to query Node Exporter metrics

Quick Check

As a DevOps engineer, you need to calculate the total available memory across all servers by summing node_memory_MemFree_bytes and node_memory_Cached_bytes while ignoring varying instance and job labels. Which PromQL expression should you use to ensure accurate results?

A. sum(node_memory_MemFree_bytes + ignoring(instance, job) node_memory_Cached_bytes)

B. sum(node_memory_MemFree_bytes + on(instance, job) node_memory_Cached_bytes)

C. sum(node_memory_MemFree_bytes) + ignoring(instance) sum(node_memory_Cached_bytes)

D. node_memory_MemFree_bytes + ignoring(instance, job) sum(node_memory_Cached_bytes)

Key Takeaways

- Metrics provide numeric data on system performance, collected over time, and are essential for monitoring and analysis.
- Exporters collect and expose metrics from diverse systems, bridging the gap between non-instrumented applications and Prometheus.
- Relabeling in Prometheus allows for manipulation and transformation of metric labels to ensure consistent data and optimized querying.
- PromQL enables advanced metric analysis through functions, aggregation operators, and label management to derive insights and optimize query performance.
- Binary operators, along with selectors and matchers, provide a powerful way to manipulate and analyze time series data in PromQL.

Monitoring Apache Server Metrics Using Prometheus

Duration: 25 Min.

Project agenda: To set up Prometheus to monitor an Apache web server, collect metrics using Apache Exporter, and analyze performance using PromQL queries

Description: As a DevOps engineer at a company running a high-traffic e-commerce website with Apache web servers, your team has identified performance issues during peak hours. Your task is to implement a monitoring solution to identify potential bottlenecks and ensure optimal performance. This involves configuring Prometheus to monitor the Apache web servers and collect relevant metrics. You are responsible for setting up Apache Exporter and Prometheus and using PromQL queries to track metrics. This setup will help proactively identify and address performance issues.

LESSON-END PROJECT

Monitoring Apache Server Metrics Using Prometheus

Duration: 25 Min.

Perform the following:

1. Install and configure the Apache web server
2. Install and configure the Apache Exporter for Prometheus
3. Configure Prometheus to scrape metrics from the Apache Exporter
4. Run PromQL queries in Prometheus UI to analyze Apache web server metrics

Expected deliverables: A fully configured Prometheus and Apache Exporter setup to collect and visualize Apache web server metrics, accessible through a web interface for real-time monitoring and performance analysis using custom PromQL queries

Thank You