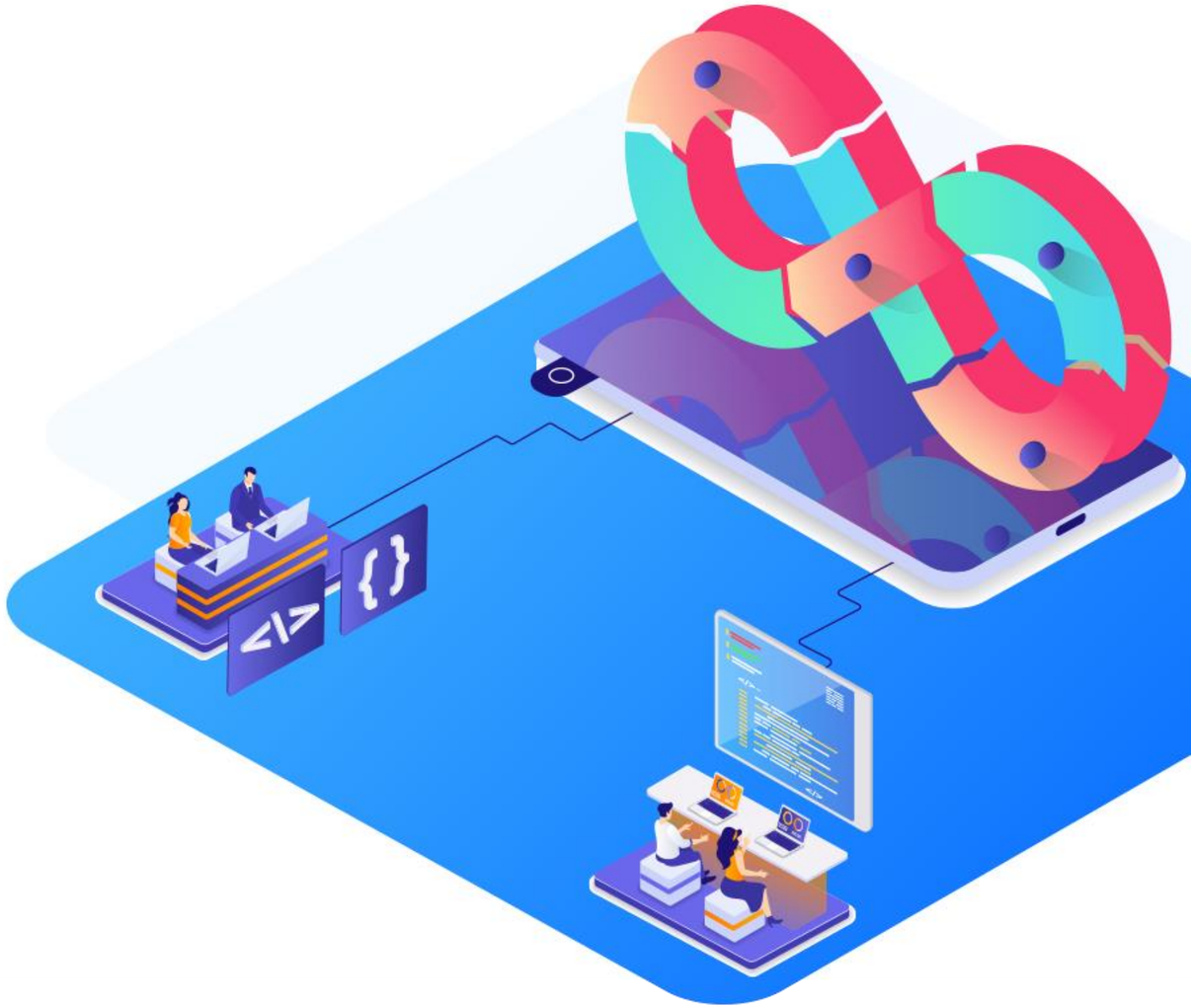
Monitoring and Logging in DevOps



Advance Concepts of Prometheus in Monitoring

Learning Objectives

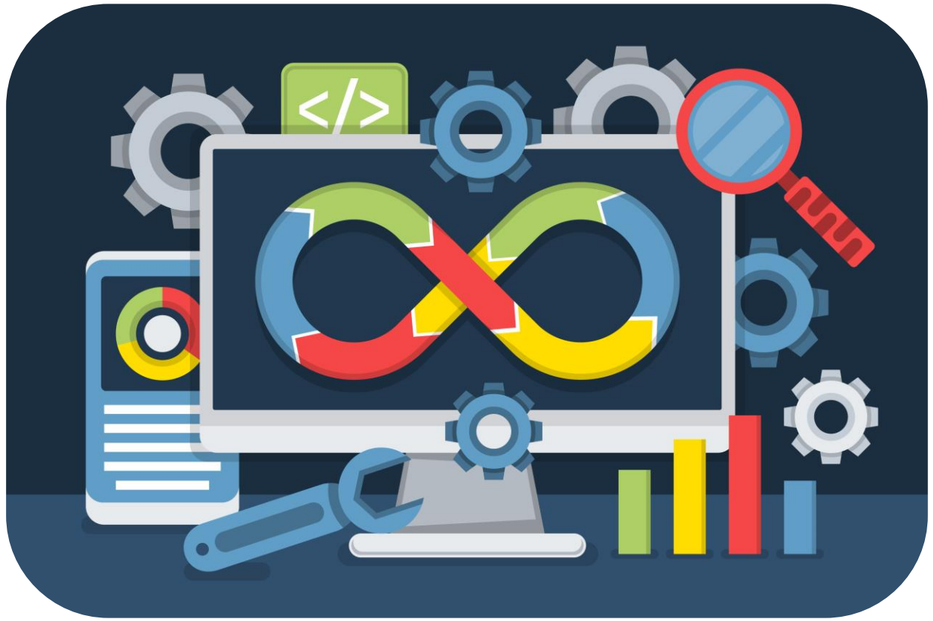
By the end of this lesson, you will be able to:

- Identify key instrumentation techniques and their role in monitoring to ensure effective performance tracking
- Create and demonstrate recording rules in Prometheus to efficiently evaluate and store metrics
- Create and implement alert rules in Prometheus to monitor and respond to different alert states effectively
- Utilize Alertmanager to manage alerts by focusing on routing, grouping, and notification setup for efficient incident response
- Deploy Pushgateway and automate metric pushing to integrate applications seamlessly within the Prometheus ecosystem

Instrumentation in Monitoring

What Is Instrumentation?

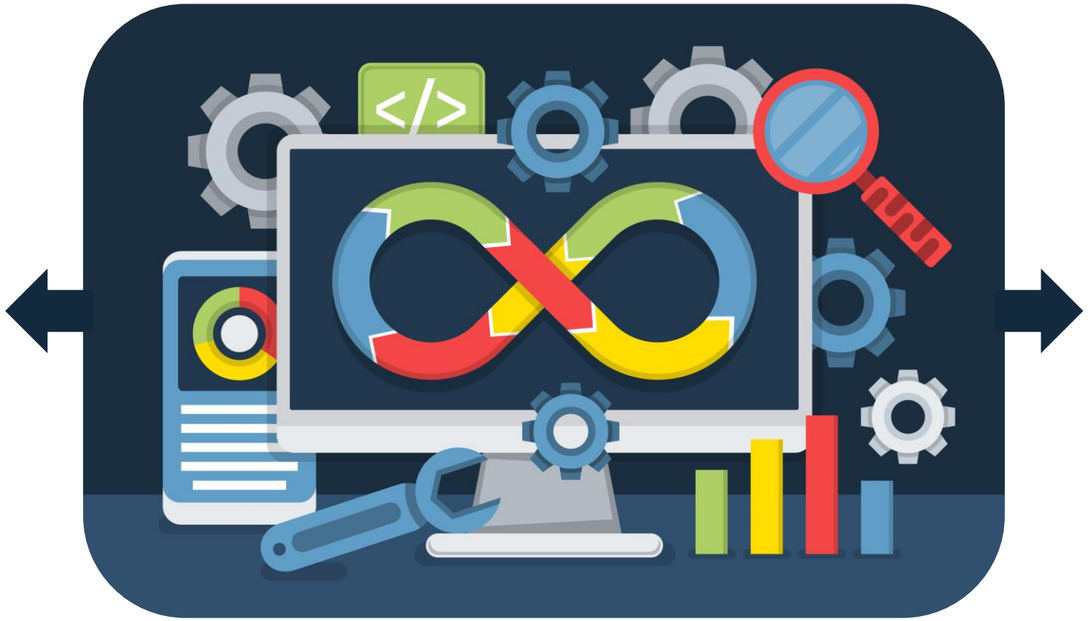
It is the process of adding code or libraries within an application to expose metrics and other system performance data for monitoring.



This data, which includes metrics, logs, and traces, can then be collected and analyzed by monitoring tools such as Prometheus.

Key Objectives of Instrumentation

To understand the internal state, performance, and behavior of an application



To provide the necessary data for monitoring tools to deliver insights

Benefits of Instrumentation

Here are some key business benefits of deploying instrumentation:

Improved data quality

Provides precise and real-time data collection, minimizing human error

Cost efficiency

Lowers labor costs and prevents expensive failures with early issue detection

Faster decision-making

Enables quick issue detection and response, reducing downtime

Scalability

Easily adapts to business growth while maintaining operational efficiency

Instrumentation Process

It includes the following steps:

1. Add client libraries to the code
2. Define and create specific metrics
3. Expose metrics through HTTP endpoint

Example: This process can be performed on critical web applications to monitor parameters such as the number of requests per second, response times, and error rates.

Role of Instrumentation in Monitoring

Instrumentation plays a key role in monitoring and observability.

Exposes key data: It exposes metrics, logs and traces in a format consumable by monitoring tools.

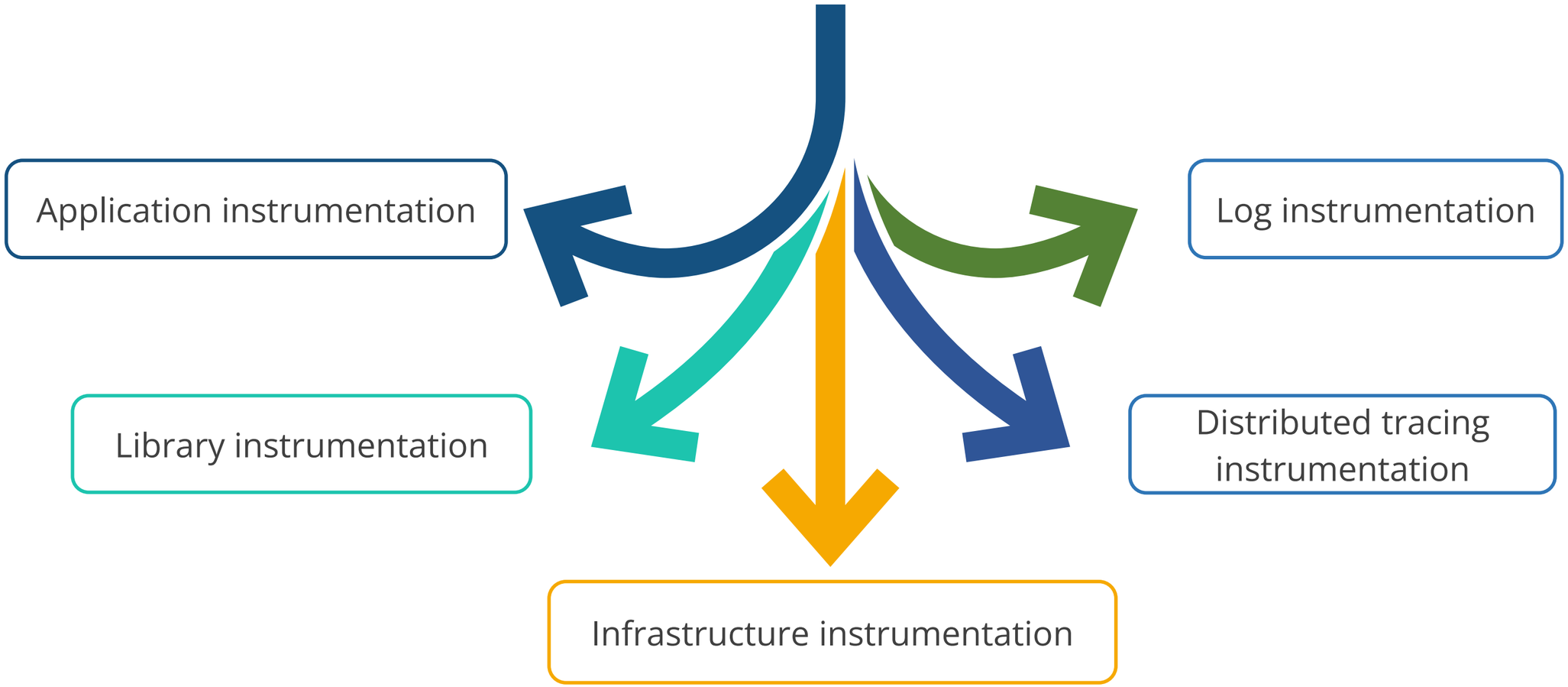
Enables compatibility: It provides essential data for tracking the internal state and performance of components and enables compatibility of the application with the monitoring tool.

Supports critical functions: It enables alerting, anomaly detection, root cause analysis, and performance optimization.

Facilitates capacity planning: It monitors resource utilization and helps in effective planning.

Types of Instrumentation

The following types of instrumentation are generally used in monitoring and observability:



Types of Instrumentation

Application instrumentation

- Adds code or libraries to applications to collect and expose metrics, logs, and traces
- Offers detailed insights into application performance and behavior

Example

A development team integrates Prometheus Client Libraries into their Python-based web service to collect metrics such as request rates and response times. These metrics are exposed through an endpoint, enabling Prometheus to scrape the data regularly and monitor the performance.

Types of Instrumentation

Library instrumentation

- Utilizes third-party libraries or frameworks that are pre-instrumented to expose metrics and traces
- Simplifies monitoring and observability by leveraging built-in instrumentation

Example

Frameworks like Flask (Python) or Spring Boot (Java) come with built-in capabilities to collect metrics related to HTTP requests and database connections.

Types of Instrumentation

Infrastructure instrumentation

- Instruments infrastructure components such as operating systems, containers, and cloud services to expose system-level metrics
- Monitors resource utilization, network traffic, and system performance

Example

A team uses Node Exporter on their Linux servers to collect metrics like CPU usage and memory usage. These metrics are then scraped by Prometheus, helping the team monitor server health and performance in real time and address any emerging issues.

Types of Instrumentation

Distributed tracing instrumentation	• Captures and propagates information about requests as they travel through multiple services in distributed systems to understand the flow of requests • Enables end-to-end visibility into request flows and helps to identify performance issues

Example

In a microservices-based e-commerce app, OpenTelemetry captures trace data across services like authentication and payment. Jaeger visualizes the request flow, helping the team quickly identify and fix performance issues.

Types of Instrumentation

Log instrumentation

- Includes adding structural and contextual data in application logs to enhance the analysis and correlation with metrics and traces
- Aids in better log analysis and integrates logs with metrics and traces

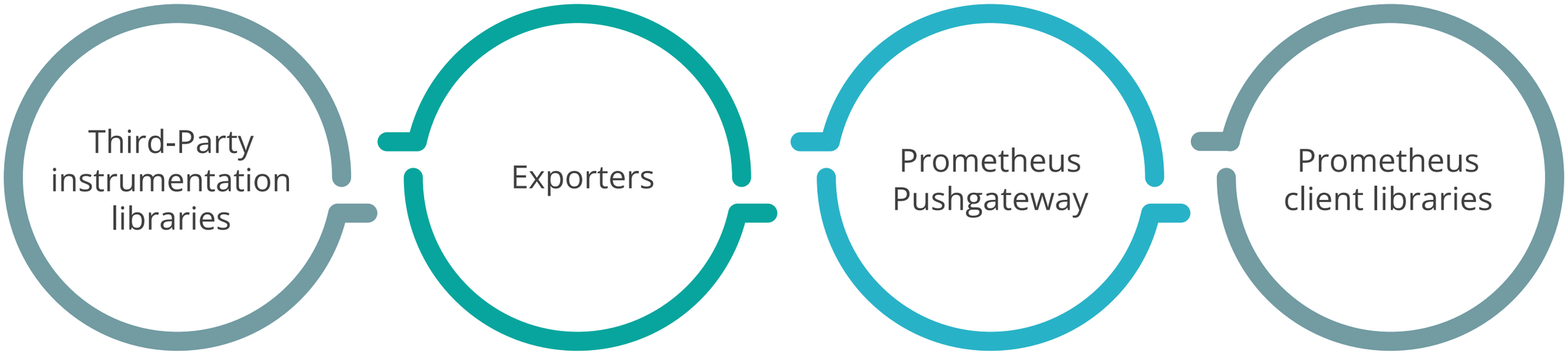
Example

A team working on a microservices-based application uses structured logging in JSON format, enriching logs with user and request IDs, and correlation IDs to link logs across services. This enables easy request tracing, log metric correlation, and quick issue resolution.

Instrumentation Techniques

Prometheus offers various instrumentation techniques for different languages and frameworks.

The following key methods help achieve instrumentation:



Instrumentation Techniques: Third-Party Instrumentation Libraries

- They offer advanced features and deeper integration with different frameworks.
- They support different programming languages with instrumentation libraries that add additional functionality.
- **Example:** A team integrates Micrometer with their Spring Boot application to monitor performance metrics like request rates and memory usage. These metrics are exported to Prometheus for real-time monitoring and analysis.

Instrumentation Techniques: Exporters

- They collect and expose metrics from different systems and services in a Prometheus-compatible format.
- They act as a bridge between non-instrumented systems and Prometheus, enabling the monitoring of components like databases and network devices.
- **Example:** A team uses MySQL Exporter with Prometheus to monitor query response times, connection counts, and buffer pool usage, enabling real-time database health monitoring and performance optimization.

Instrumentation Techniques: Prometheus Pushgateway

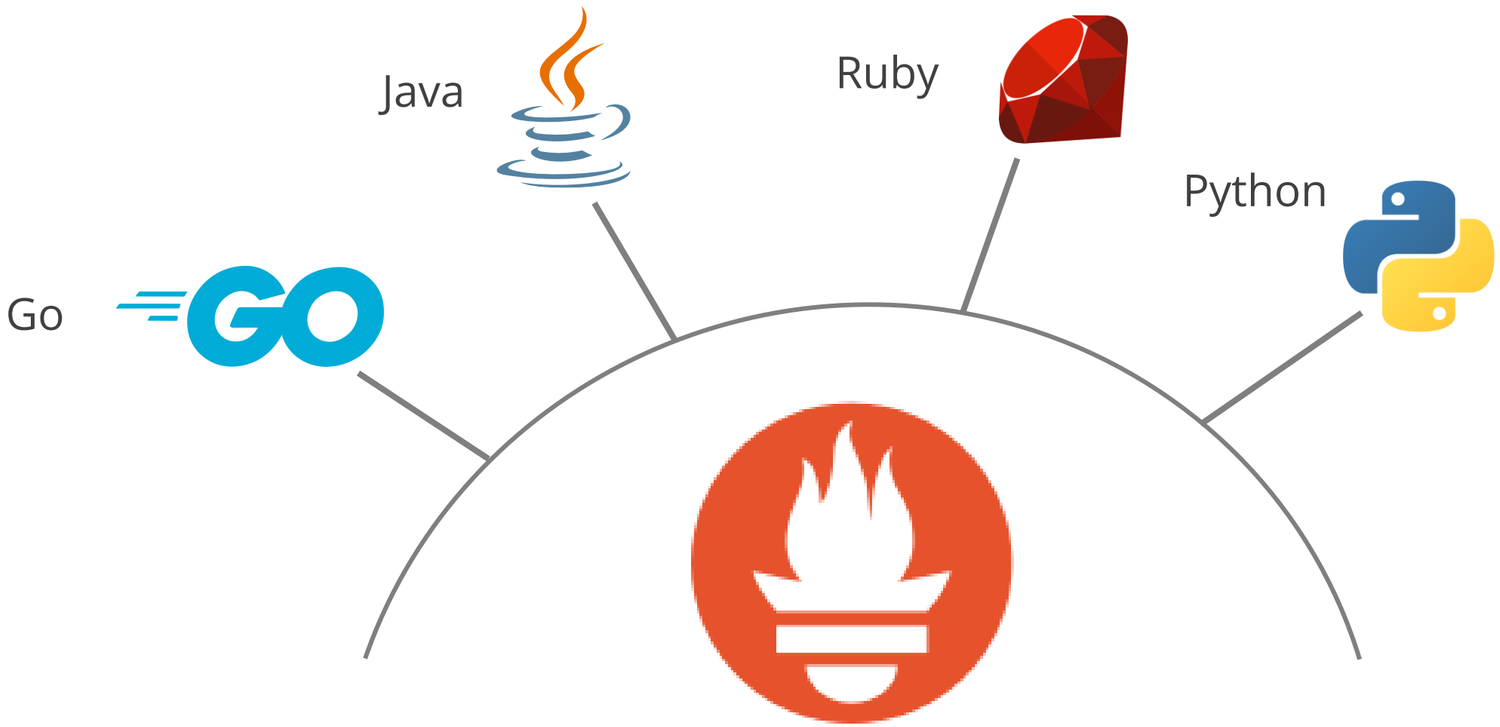
- It enables short-lived or batch jobs to push their metrics to a temporary storage location, which Prometheus can then scrape.
- It collects metrics from jobs that cannot be scraped directly because they are short-lived.
- **Example:** A team uses Prometheus Pushgateway to push metrics from a short-lived batch job, such as records processed and execution time, allowing Prometheus to monitor job performance.

Instrumentation Techniques: Prometheus Client Libraries

- They provide official client libraries for various programming languages to expose metrics.
- They simplify instrumentation by providing APIs that help create and expose metrics in Prometheus format.
- **Example:** A team uses the Prometheus Python client to instrument their Flask app, exposing metrics like HTTP requests and response times via an HTTP endpoint.

Client Libraries

They simplify the process of integrating applications and exposing metrics in a Prometheus-compatible format using different programming languages, such as:



They provide assistance and APIs for generating, updating, and exporting metrics that the Prometheus server scrapes.

Client Libraries: Go

It is a part of the official Prometheus Go repository that enables Go applications to expose metrics that Prometheus can easily collect.

- Provides simple-to-use interface for handling metrics
- Enables the starting of an HTTP server for Prometheus to scrape metrics

Example

A team developing a Go-based microservice uses the Prometheus Go client library to track request counts, memory usage, and request latencies, exposing these metrics via an HTTP endpoint.

Client Libraries: Java

The Java client library, also known as Prometheus Java Simpleclient, enables Java applications to export metrics for monitoring.

- Supports the creation and management of all four Prometheus metric types
- Integrates with Sprint Boot frameworks and provides metric naming conventions

Example

A Java-based microservice uses the Prometheus Java client library to track request counts, memory usage, and latency, exposing these metrics via an HTTP endpoint for monitoring.

Client Libraries: Python

The Prometheus Python client enables applications written in Python to export metrics that Prometheus can scrape.

- Supports all types of metrics with an easy interface for creating and modifying them
- Provides tools to expose metrics via an HTTP server for Prometheus scraping

Example

A Flask-based microservice uses the Prometheus Python client library to track request counts, memory usage, and latency, exposing these metrics for monitoring.

Client Libraries: Ruby

The Prometheus Ruby client helps Ruby applications to instrument and expose metrics for Prometheus to monitor.

- Offers a Ruby-friendly way to create and manage different types of metrics
- Exposes metrics through an HTTP endpoint, making them accessible for Prometheus

Example

A Ruby on Rails application uses the Prometheus Ruby client library to monitor request counts, memory usage, and latency, exposing these metrics for monitoring.

Client Libraries: Key Benefits

Here is a quick comparison of client libraries, highlighting key benefits for each tool:

Tools	Key Benefits
Go	Best for high performance and concurrency, outperforming Java, Ruby, and Python in scalable systems like microservices
Java	Most robust for enterprise applications, offering better scalability and security than Python, Ruby, or Go
Ruby	Ideal for rapid web development, enabling faster cycles than Java, Go, and Python
Python	Most versatile, with broad use in automation, data science, and more, offering greater flexibility than Java, Ruby, or Go

How to Implement Instrumentation?

The instrumentation process uses official Prometheus client libraries or third-party instrumentation libraries.

The following are the steps to implement instrumentation in application:

01 Install Prometheus client

02 Import required libraries

03 Define metrics

04 Instrument the application

05 Expose metrics for scraping

Assisted Practice

Adding Instrumentation to a Java Application **Duration: 10 Min.**

Problem statement:

You have been assigned a task to add instrumentation to a Java application for enabling metric collection and visualizing the metrics using Prometheus.

Outcome:

By the end of this demo, you will be able to instrument a Java application to expose Prometheus metrics, configure a Prometheus server to scrape these metrics, and visualize the data in the Prometheus UI.

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

Assisted Practice: Guidelines

Steps to be followed:

1. Clone the GitHub repository for the Java application
2. Build the Java application using Maven
3. Run the Java application to expose Prometheus metrics
4. Visualize the metrics using the Prometheus UI

ASSISTED PRACTICE

Quick Check

A company has implemented Prometheus to monitor its infrastructure, including network devices and databases. However, some of these systems do not have built-in support for Prometheus. What is the best approach to collect and expose metrics from these non-instrumented systems in a format that Prometheus can understand?

A. Use Prometheus client libraries to modify the systems

B. Employ an exporter to convert and expose metrics in a Prometheus-compatible format

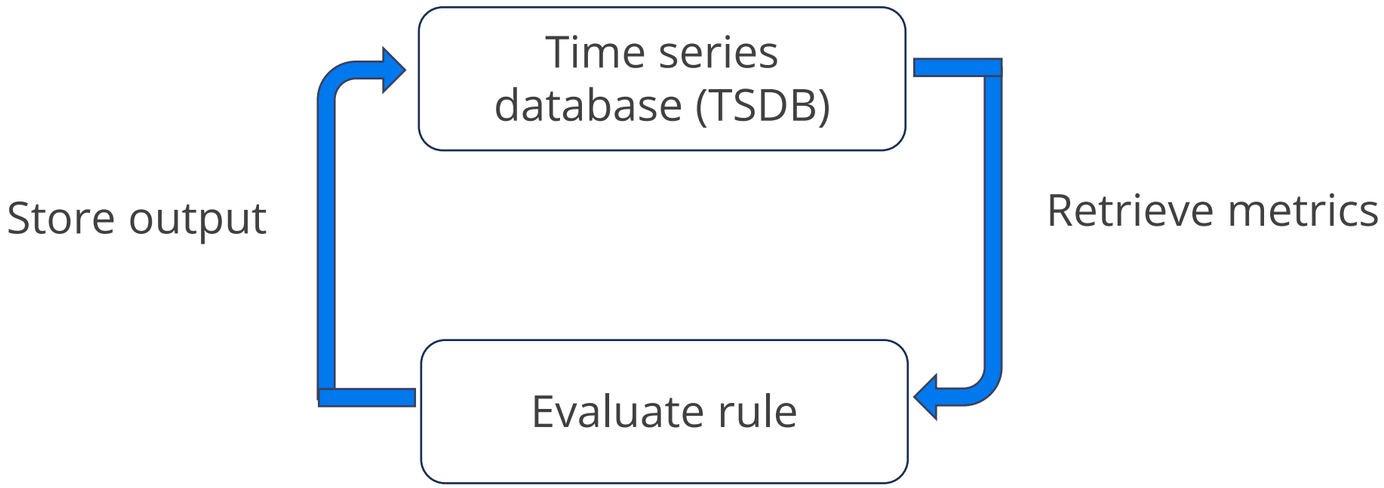
C. Develop custom scripts to manually gather metrics

D. Use built-in tracing tools to monitor these systems

Recording Rules

What Are Recording Rules?

It is a mechanism that enables the pre-computation of frequently used queries and stores the result as a new time series in Prometheus.



This approach significantly speeds up query performance, particularly for dashboards that repeatedly run the same queries upon refresh.

Key Elements of Recording Rules

In Prometheus, recording rules are organized into groups, each with specific rules and expressions, to efficiently manage and optimize PromQL queries. Here is the description of each element:

Groups

- The recording and alerting rules are organized within rule groups, allowing multiple rules to be evaluated together.

Rules

- The recording rule generates a new time series, identified by a unique name within the Prometheus instance.

PromQL Expressions

- This expression defines how a new time series is created, using existing metrics and applying math functions or filters.

The Role of Recording Rules in Prometheus

Recording rules help resolve performance issues as the number of metrics increases and queries become more complex. It offers the following advantages:

Reduces query load

Precomputes results and minimizes load on the Prometheus server

Improves query speed

Executes queries significantly faster, leading to quicker retrieval of data

Enhances reliability

Minimizes the risk of query timeouts and failures during critical periods

Writing Recording Rules

The following steps describe the creation of recording rules:

Define rule component

- Specify a unique rule name for identification
- Write the PromQL expression for the new metric calculation

Create YAML File

- Create a dedicated YAML file to store rules
- Format the rules according to Prometheus YAML syntax

Configure Prometheus

- Add the rule file to the *<rule_files>* section in the configuration file
- Reload Prometheus to apply the new recording rules

Recording Rules: Define Rule Components

The following is the syntax of the YAML file used to record rules:

```
groups:
  - name: group_name
    rules:
      - record: new_metric_name
        expr: expression_to_calculate_new_metric
      - record: another_new_metric
        expr: another_expression
```

Here,

- **groups**: Contains one or more group of rules
- **name:** Each group has a unique name
- **rules:** List of individual or multiple rules
- **record:** Name of the new metric to be created
- **expr:** PromQL expression to calculate the new metric

Recording Rules: Create YAML File

The following example demonstrates how to create recording rules in Prometheus for monitoring HTTP requests and their success rates:

```
groups:
  - name: my_recording_rules
    rules:
      - record: requests_per_minute_total
        expr: sum(rate(http_requests_total[1m]))
      - record: successful_requests_percentage
        expr: sum(rate(http_requests_success_total[1m])) /
              sum(rate(http_requests_total[1m])) * 100
```

- **http_requests_per_minute:** Calculates total HTTP requests per minute
- **http_success_percentage:** Calculates percentage of successful HTTP requests

Recording Rules: Configure Prometheus

It is essential to include the file path in the *rule_files* section of the Prometheus configuration file to use the new recording rules.

Syntax

```
rule_files:
  - /path/to/recording_rules.yml
```

Steps:

- Restart Prometheus after configuring the *rule_files* to apply the changes
- Access new metrics for PromQL queries and visualization tools

Recording Rules: Evaluation

Recording rules are periodically evaluated as instant queries, which assess data at a specific point in time and return results for that moment. The syntax is:

Syntax

```
[ interval: <duration> | default =
  global.evaluation_interval ]
```

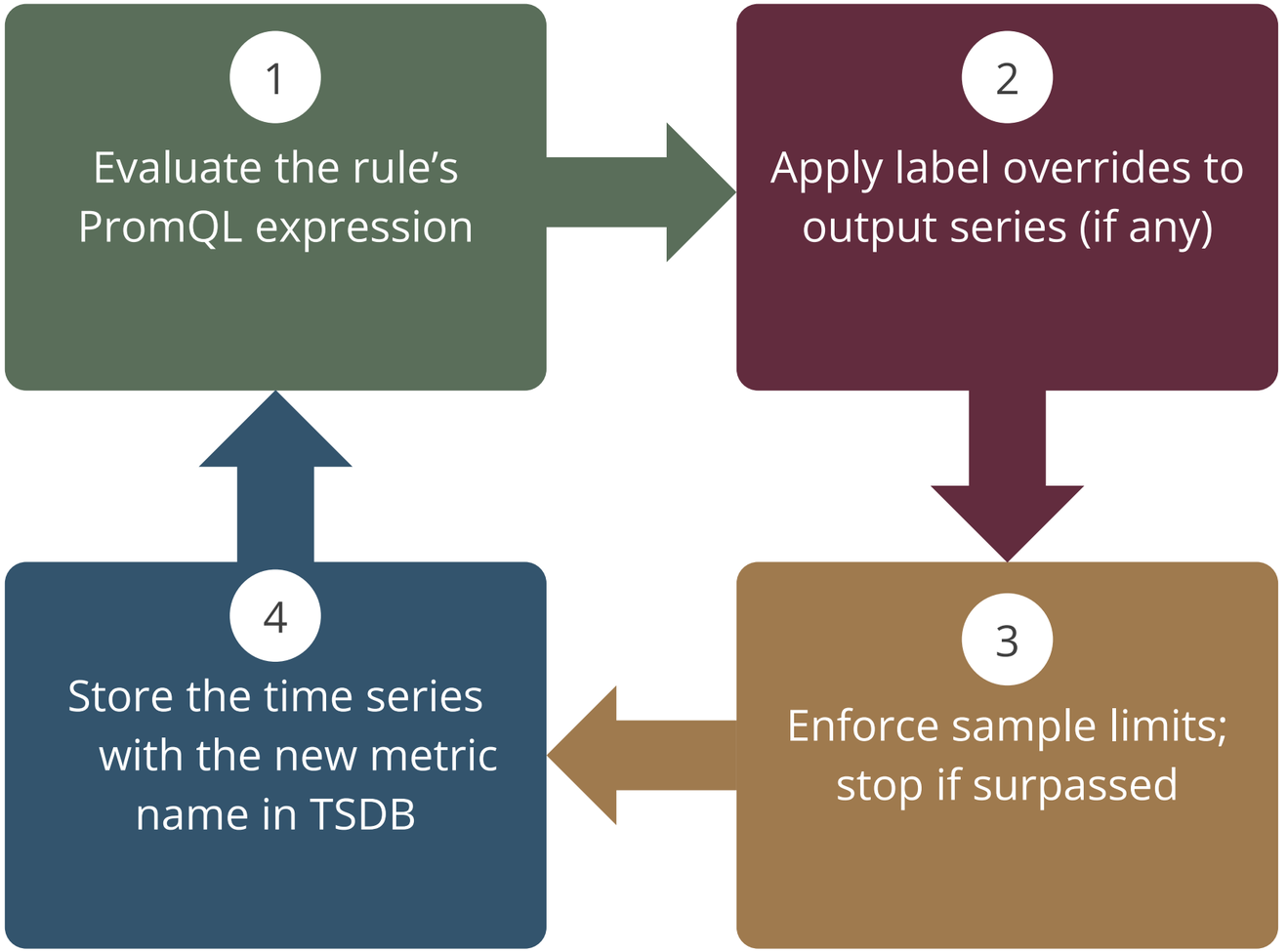
- **global.evaluation_interval:** Determines the evaluation interval if it is not specified
- **Interval:** Configures the evaluation interval for a specific group
- **<duration>:** Defines time interval; by default, it is 1 minute.

NOTE

If the evaluation of a rule group exceeds its set interval, subsequent evaluations are delayed until the current one is completed.

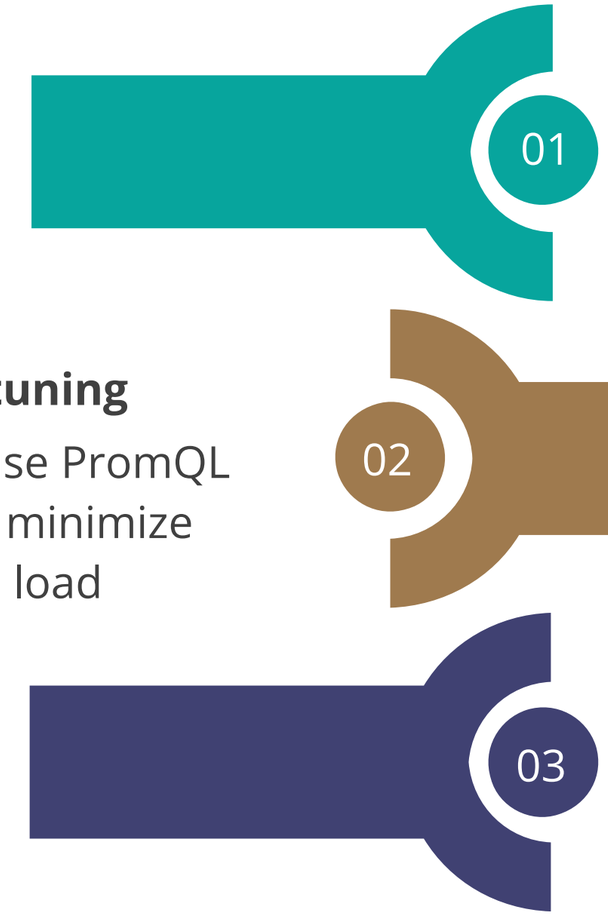
Recording Rules: Evaluation Cycle

Prometheus processes recording rules through the following steps:



Best Practices for Writing Recording Rules

The following best practices help create efficient and reliable recording rules, ensuring a robust monitoring system:



Simplicity and precision

Capture specific metrics and calculations to keep rules simple and precise

Performance tuning

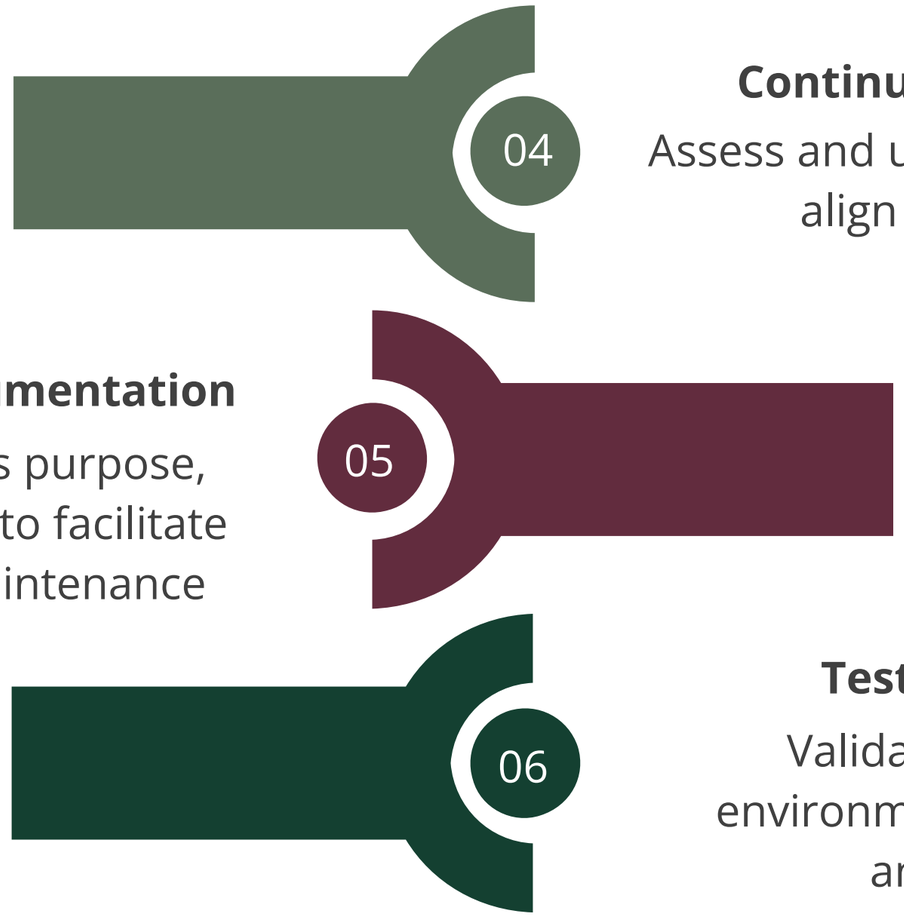
Optimize rules and use PromQL functions wisely to minimize computational load

Strategic labeling

Apply labels to organize time series data and improve query precision

Best Practices for Writing Recording Rules

The following best practices help create efficient and reliable recording rules, ensuring a robust monitoring system:



Continuous improvement

Assess and update rules periodically to align with current needs

Comprehensive documentation

Document each rule's purpose, inputs, and outcomes to facilitate collaboration and maintenance

Testing and validation

Validate rules in a staging environment to ensure accuracy and performance

Quick Check

John (M) is facing slow query performance on his Prometheus dashboard due to complex queries. He wants to optimize this using recording rules. Which approach should he take?

A. Precompute and store results of frequent queries

B. Use metrics directly without modifications

C. Increase the frequency of real-time queries

D. Limit the number of metrics collected

Alerting in Prometheus

What Is Alerting ?

It is the process of creating rules to trigger notifications or alerts when specific conditions in the collected metrics are met.

Function

Triggers notifications based on predefined metric conditions

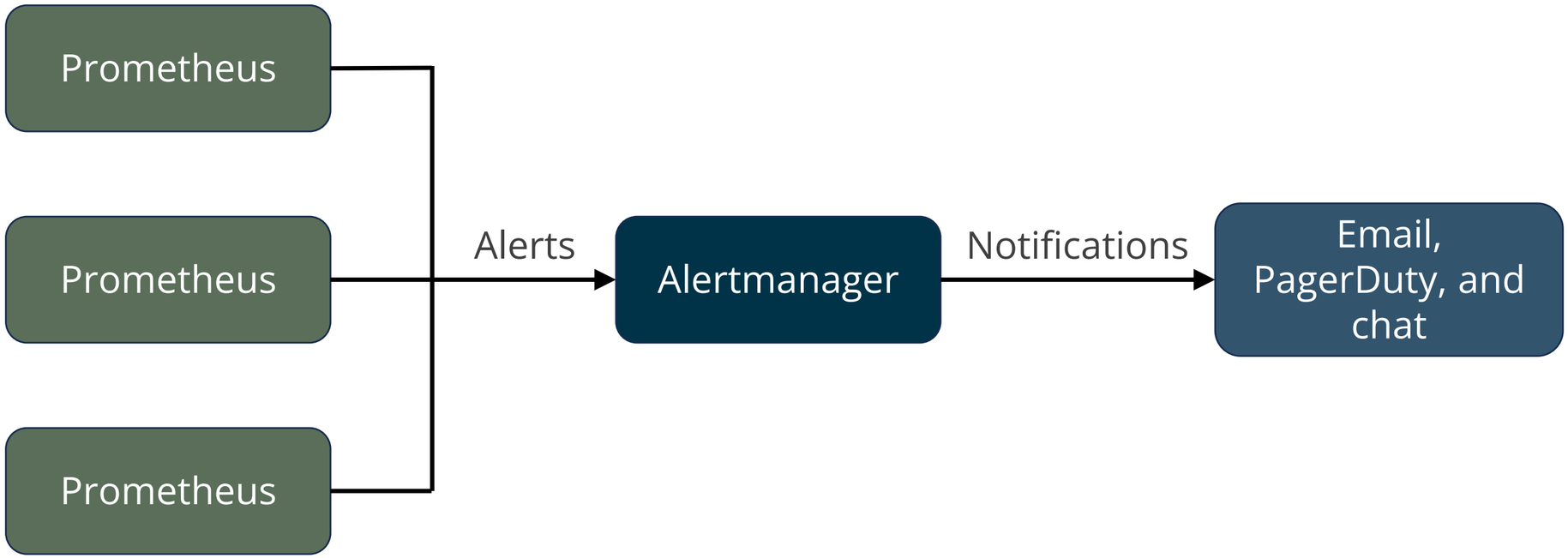
Purpose

Identifies and addresses system and application anomalies

Prometheus Alerting Process

It is the process of identifying and alerting users about potential issues in monitored systems.

Here is the illustration of the Prometheus alerting process:



PromQL defines alerting rules, which it regularly evaluates against stored data. When these conditions are met, it generates alerts and sends them to Alertmanager for processing and routing.

Defining Alert Rules

Alert rules are defined in the following format using PromQL and are stored in separate configuration files:

Alert rule format:

```
groups:
  - name: group_name
    rules:
      - alert: alert_name
        expr: promql_expression
        for: duration
        labels:
          label_key: label_value
        annotations:
          summary: alert_summary
          description:
alert_description
```

groups: Groups alert rules into sets with a specific name

alert: Specifies the unique name of the alert rule

expr: Specifies the condition for triggering the alert

for: (optional) Specifies the duration of the alert

labels: (optional) Adds extra labels to the alert

annotations: (optional) Provides descriptive text and additional context for the alert

Configuring Alert Rules

Once the alert rule is defined, specify the path to the alert rules file in the rule_files section of the Prometheus configuration. The path specification is as follows:

Specifying path:

```
rule_files:
  - "alert_rules.yml"
```

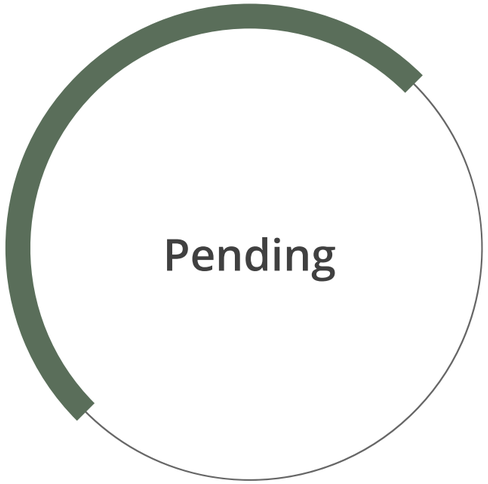
rule_files: Specifies the configuration files that contain alert rules

alert_rules.yml: Indicates that the file alert_rules.yml is used to define these alert rules

Prometheus loads and evaluates alert rules every minute, sending alerts to Alertmanager for processing and sending notifications when conditions are met.

Understanding Alert States

There are three main alert states, including:



The alert has breached the threshold, but the recording interval has not passed yet.

The alert has breached the threshold and the recording interval, and notifications are being sent.

The alert is no longer active because the threshold is no longer breached.

Managing Alerts with Alertmanager

The Alertmanager manages the alerts from client applications such as the Prometheus server.

The responsibilities of the Alertmanager include:

Alert receiving

Acts as the centralized entry point for alerts from multiple servers

Alert deduplication

Filters out duplicate alerts using labels and annotations

Alert grouping

Organizes related alerts by grouping them based on shared labels

Alert routing

Directs alerts to the appropriate channels using configurable rules

Managing Alerts with Alertmanager

The responsibilities of the Alertmanager include:

Notification delivery

Sends alerts through integrated channels (email, Slack, PagerDuty)

Alert silencing

Suppresses notifications temporarily or permanently for specific alerts or groups

Alert lifecycle tracking

Monitors alert states, including firing, resolution, and re-firing, for detailed analysis

Alert inhibition

Suppresses alerts when related alerts are already firing, focusing attention on root causes

Alert Routing

It refers to the configuration of alerts in a monitoring system to ensure that different types of alerts are directed to the appropriate channels or teams.

Example

```
route:
  receiver: 'team-X-pager'
routes:
  - match:
      severity: critical
    receiver: 'team-X-pager'
  - match:
      severity: warning
    receiver: 'team-X-email'
```

In this example, 'team-X-pager' receives **critical** alerts whereas 'team-X-email' receives **warning** alerts

Alert Grouping

It is the process of organizing related alerts based on shared attributes or labels.

Example

```
route:
  group_by: ['alertname', 'cluster', 'service']
```

In this example, alerts are grouped based on their *alertname*, *cluster*, and *service* labels.

Alert Inhibition

It inhibits the notification for specific alerts when some other alerts are already active.

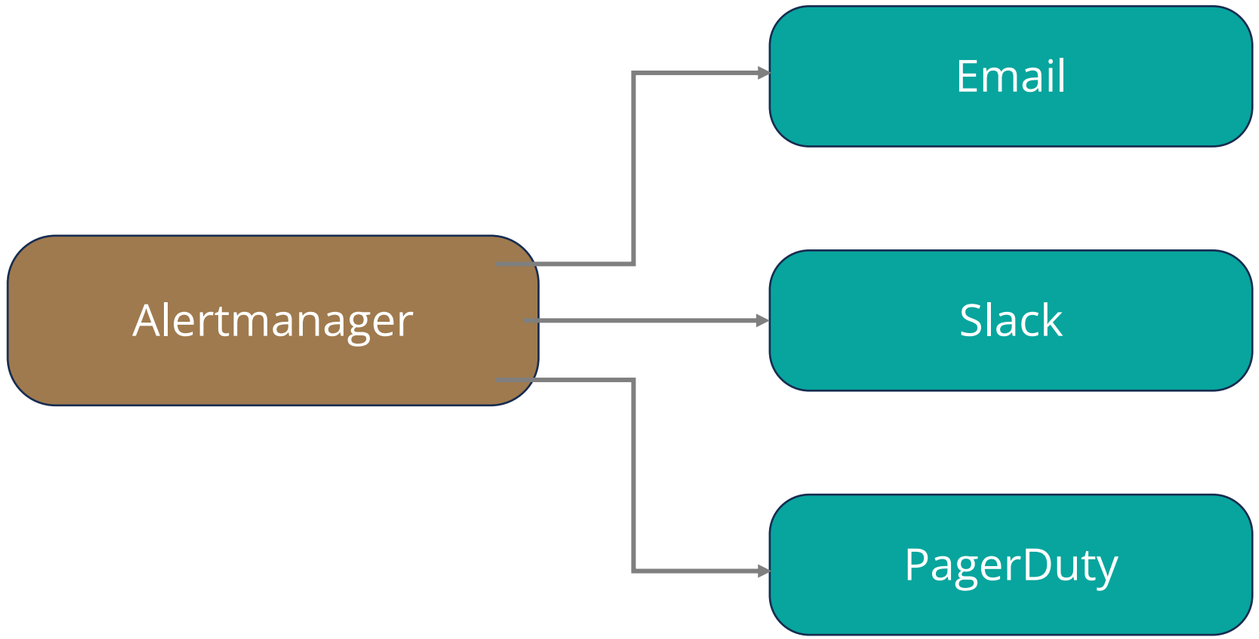
Example

```
inhibit_rules:
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['alertname', 'cluster', 'service']
```

In this example, an active critical alert will inhibit any warning alerts with the same name, cluster, and service labels to prioritize the critical alert and reduce similar problem alerts.

Alertmanager Notification Channels

Alertmanager in Prometheus enables users to configure the following notification channels:



It also supports various other notification channels, such as OpsGenie, Webhooks, and more.

Email Notifications

The following example demonstrates the configuration of an SMTP server in the Alertmanager's configuration file to send notifications through email:

Example:

```
receivers:
  - name: 'team-email'
    email_configs:
      - to: 'team@example.com'
        from: 'alertmanager@example.com'
        smarthost: 'smtp.example.com:587'
        auth_username: 'alertmanager@example.com'
        auth_identity: 'alertmanager@example.com'
        auth_password: 'somepassword'
```

Slack Notifications

These notifications are configured in Alertmanager by adding a Slack webhook URL to the configuration file as shown in the below example:

Example:

```
receivers:
  - name: 'team-slack'
    slack_configs:
      - send_resolved: true
        channel: '#alerts'
        api_url:
'https://hooks.slack.com/services/YOUR/WEBHOOK/URL'
```

PagerDuty Notifications

The following example explains how to set up a PagerDuty integration key in the Alertmanager's configuration file for sending notifications to PagerDuty:

Example:

```
receivers:
  - name: 'team-pagerduty'
    pagerduty_configs:
      - service_key: 'YOUR_PAGERDUTY_INTEGRATION_KEY'
        send_resolved: true
```

Routing Rules

These are rules that are used to assign alerts to appropriate channels. The below example shows sending critical alerts to PagerDuty and warnings to Slack:

Example:

```
route:
  receiver: 'team-email'
  routes:
    - match:
        severity: critical
      receiver: 'team-pagerduty'
    - match:
        severity: warning
      receiver: 'team-slack'
```

In this example, alerts with *severity: critical* are sent to team-pagerduty and alerts with *severity: warning* are sent to team-slack.

Assisted Practice

Configuring Alertmanager for Email Notifications **Duration: 10 Min.**

Problem statement:

You have been assigned a task to set up Prometheus to initiate alerts and validate the alerting and notification system with practical tests for enhanced monitoring and timely responses in system management.

Outcome:

By the end of this demo, you will be able to configure Prometheus and Alertmanager for sending email notifications when alerts are triggered, testing the alerting system, and monitoring alert status through the Prometheus and Alertmanager UI.

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

Assisted Practice: Guidelines

Steps to be followed:

1. Configure Prometheus to work with Alertmanager
2. Install and configure Alertmanager
3. Start Node Exporter and Prometheus
4. Review Alertmanager settings via the Prometheus UI

Quick Check

The operations team wants to optimize their alerting setup in Prometheus to avoid overwhelming the team with redundant alerts. They plan to use Alertmanager for this purpose. Suggest how they should configure Alertmanager to manage and prioritize alerts effectively.

A. By sending all alerts via a single channel

B. By ignoring alerts for non-critical issues

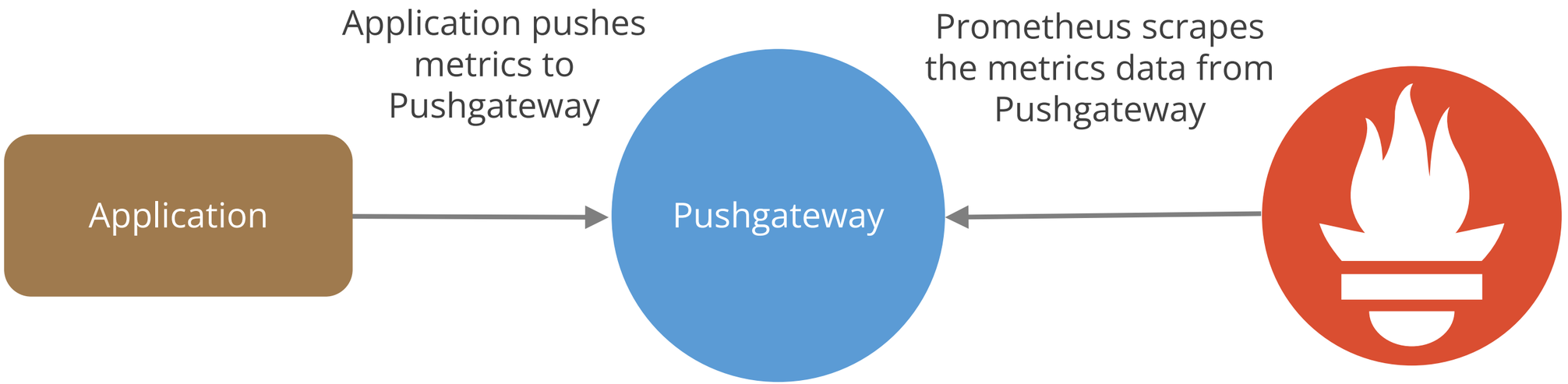
C. By directly sending all alerts without modification

D. By using alert routing based on severity

Pushgateway

Introduction to Pushgateway

It is a feature of Prometheus designed for ephemeral and batch jobs to expose their metrics to Prometheus despite their short lifecycle. The below shows the working of Pushgateway:

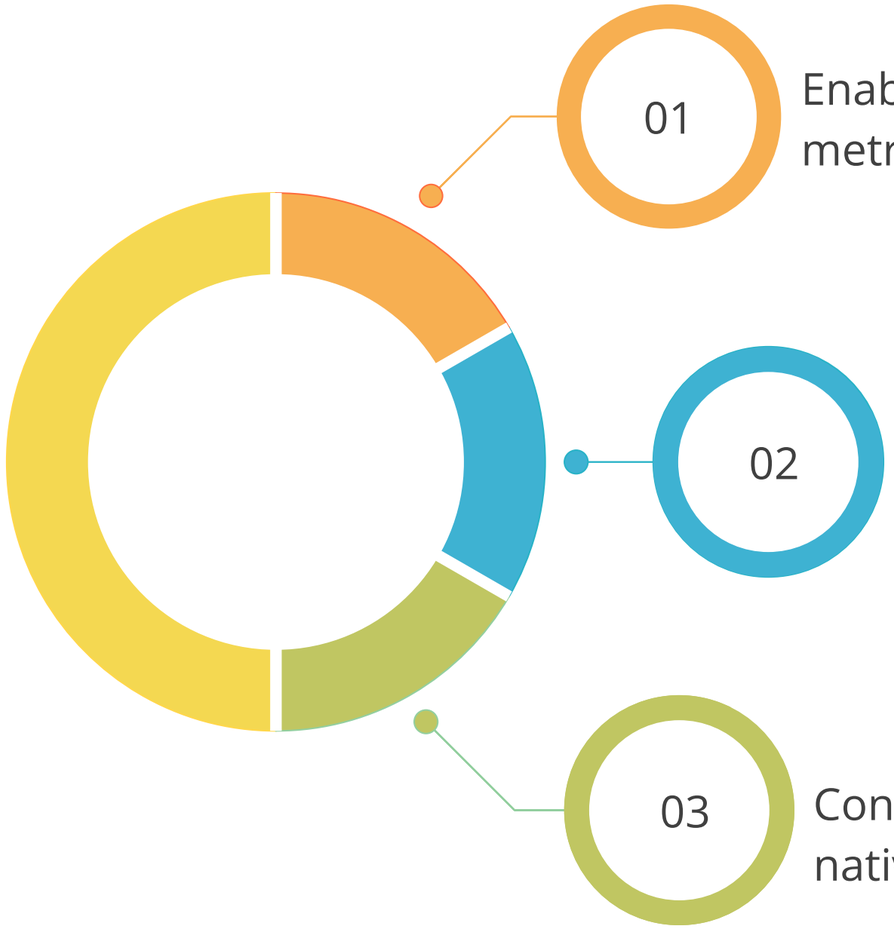


It acts as a cache for short-lived jobs, enabling metrics collection even after they end. It supports client libraries for Go, Java, Scala, Python, and Ruby.

Role of Pushgateway in Prometheus Ecosystem

Pushgateway extends Prometheus monitoring to scenarios where direct scraping is not practical.

It serves the following roles:



01 Enables temporary jobs to push metrics for Prometheus

02 Works with serverless and offline systems

03 Connects systems that do not natively expose Prometheus metrics

Role of Pushgateway in Prometheus Ecosystem

Pushgateway serves the following roles:

04

Separates metrics from the application lifecycle for flexible monitoring

05

Simplifies monitoring in resource-limited or secure settings

06

Complements pull model with a push-based approach

Setting up Pushgateway

Pushgateway exposes its metrics and those pushed by other jobs at the "/metrics" HTTP path.

The Pushgateway configuration is as follows:

Example:

```
scrape_configs:
  - job_name: 'pushgateway'
    honor_labels: true
    static_configs:
      - targets: ['<pushgateway-address>:9091']
```

NOTE

The IP address or hostname of the device executing Pushgateway should be substituted for *<pushgateway-address>*.

Pushing Metrics to Pushgateway

After setting up, jobs or applications can push their metrics to Pushgateway using its HTTP API.

The format for pushing metrics is as follows:

Syntax

```
http://<pushgateway-address>:9091/metrics/job/<job-name>/<metric-labels>
```

This structure is used to push metrics to Pushgateway, which then makes them available for Prometheus to scrape.

Pushing Metrics to Pushgateway: Example

The example below defines a gauge metric to track batch job progress and demonstrates how to use a curl command to push this metric to Pushgateway:

Gauge metrics

```
# TYPE batch_progress gauge
batch_progress{task="data_processing", run_id="123"} 0.75
```

Use curl to send POST requests to Pushgateway

```
curl --request POST \
  --data "# TYPE batch_progress gauge
batch_progress{task=\"data_processing\", run_id=\"123\"} 0.75" \

http://pushgateway.example.org:9091/metrics/job/batch_job/instance/server1/reg
ion/us-west
```

Automating Metric Pushing

It is more effective and reliable than manual methods like using curl, especially for long-running or recurring jobs and applications. Some of the approaches for automating the process are:

Client
libraries utilization

Build or deployment
tools integration

Utilizing Client Libraries

Prometheus provides official client libraries in various programming languages for metric pushing.

The example below shows how to use the Python client library to push custom metrics from a job to Pushgateway:

Example

```
from prometheus_client import push_to_gateway

job_name = "batch_job"
instance = "server1"
metrics = {"batch_progress": 0.75}
labels = {"task": "data_processing", "run_id": "123"}

push_to_gateway("pushgateway.example.org:9091", job=job_name,
instance=instance, metrics=metrics, labels=labels)
```

Integrating Build or Deployment Tools

Prometheus allows the integration of metrics collection into build or deployment pipelines for batch jobs or scripts.

The example below shows a Bash script that runs a batch job and then uses a curl command to push the progress metrics:

Example

```
#!/bin/bash

# Run the batch job
run_batch_job.sh

# Push metrics to Pushgateway
BATCH_PROGRESS=$(get_batch_progress)
curl --request POST \
  --data "# TYPE batch_progress gauge
batch_progress{task=\"data_processing\",
run_id=\"${BATCH_JOB_ID}\"} ${BATCH_PROGRESS}" \

http://pushgateway.example.org:9091/metrics/job/batch_job/instan
ce/server1/region/us-west
```

Integrating Applications

Applications that do not natively support Prometheus can be integrated with Pushgateway to expose their metrics. Here are some methods to integrate these applications:

Instrument the application	Add Prometheus metrics to the application code and push them to Pushgateway using client libraries
Integrate with existing monitoring solutions	Use a custom adapter to push metrics from other monitoring solutions to Pushgateway
Leverage application logs	Extract metrics from log files and push them to Pushgateway with a parsing script
Use external monitoring agents	Deploy agents to collect and push metrics to Pushgateway when direct integration is not feasible

Integrating Applications: Example

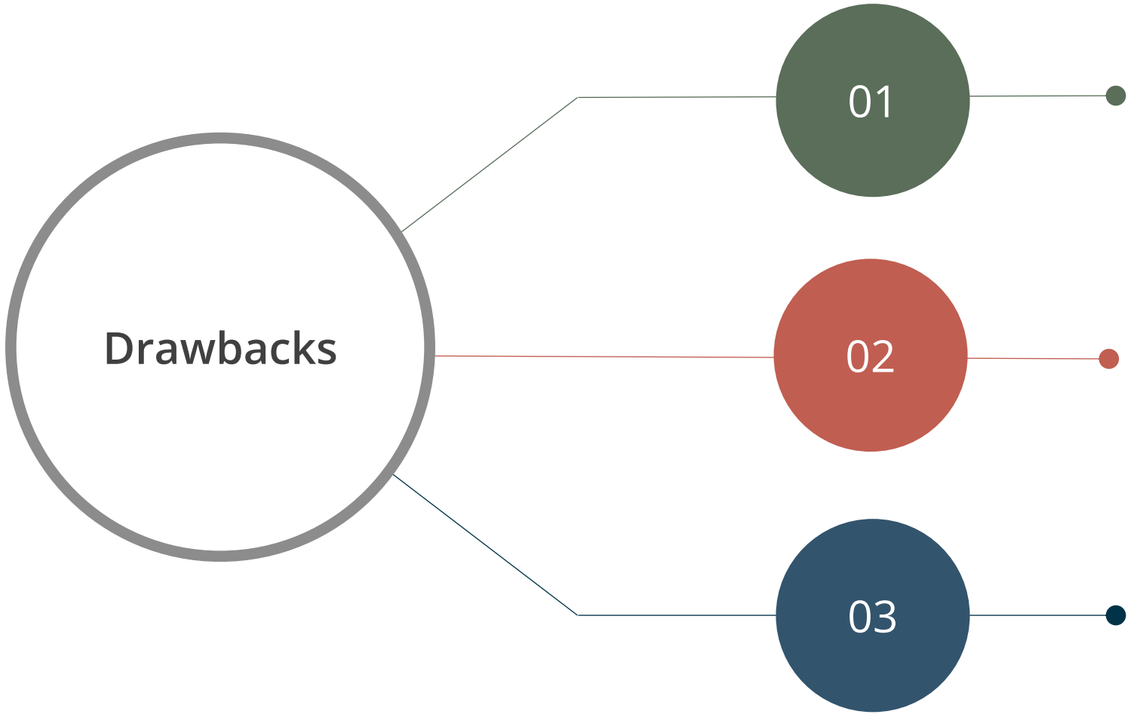
The below code instruments an application by setting up a metric, exposing it via a local HTTP server, and continuously pushing updates to Pushgateway for Prometheus monitoring:

Example:

```
from prometheus_client import start_http_server, Gauge, push_to_gateway
import time

# Create a Gauge metric
app_metric = Gauge('app_metric', 'Application-specific metric')

# Start a local metrics endpoint
start_http_server(8000)

while True:
    # Update the metric value
    app_metric.set(get_metric_value())

    # Push metrics to Pushgateway
    push_to_gateway("pushgateway.example.org:9091", job="my_app",
instance="app_instance", metrics={"app_metric": app_metric.collect()})

    # Push metrics every minute
    time.sleep(60)
```

Using Pushgateway: Challenges

Pushgateway has limitations that should be considered before utilizing it, making it an imperfect solution for pushing metrics to Prometheus. Some of the disadvantages are:



Single point of failure
Pushgateway can become a bottleneck when monitoring multiple instances.

Loss of automatic health monitoring
Lacks support for Prometheus up metric health checks

Persistent metrics
Metrics remain stored indefinitely unless manually deleted, causing potential clutter.

Assisted Practice

Automating Metric Pushing with a Cron Job **Duration: 10 Min.**

Problem statement:

You have been assigned a task to automate metrics pushing from short-lived jobs to Pushgateway using a cron job for enabling metrics collection and visualization through Prometheus.

Outcome:

By the end of this demo, you will be able to set up Pushgateway for automating metric pushing via cron jobs and configure Prometheus to scrape and visualize these metrics in real time.

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

Assisted Practice: Guidelines

Steps to be followed:

1. Download and initialize Pushgateway
2. Configure Prometheus for Pushgateway integration
3. Set up a bash script to push metrics to Pushgateway
4. Verify Pushgateway and Prometheus functionality

ASSISTED PRACTICE

Quick Check

A company needs to monitor ephemeral batch jobs that do not exist long enough for direct scraping by Prometheus. They decide to use Pushgateway to handle these metrics. Which feature of Pushgateway is crucial for integrating these short-lived jobs into their Prometheus monitoring setup?

A. Pushgateway only stores metrics for currently running jobs.

B. Pushgateway uses a push-based model to collect metrics from jobs.

C. Pushgateway automatically deletes metrics after a fixed period.

D. Pushgateway scrapes metrics directly from the running jobs.

Key Takeaways

- Instrumentation adds code to applications to collect metrics for performance monitoring.
- Recording rules precompute and store metrics to optimize query performance in Prometheus.
- Recording rules are defined using PromQL expressions and evaluated at set intervals to enhance monitoring efficiency.
- Alerting rules trigger notifications based on specific conditions and are evaluated regularly by Prometheus.
- Alertmanager handles deduplication, routing, and notification of alerts, integrating with various channels like Slack and PagerDuty.
- Pushgateway enables ephemeral and batch jobs to push metrics for Prometheus to scrape, acting as a metrics cache.

Monitoring Apache Server Metrics Using Prometheus

Duration: 25 Min.

Project agenda: To configure Prometheus for monitoring a MySQL database, collect key performance metrics using MySQL Exporter, and set up alerting rules using Alertmanager to notify of any issues for proactive database management and performance optimization

Description: You are a DevOps engineer responsible for maintaining the performance and stability of MySQL databases within your organization. Due to performance bottlenecks during peak traffic periods, you have been tasked with implementing a monitoring solution. By using Prometheus and MySQL Exporter, you will gather essential MySQL metrics, analyze performance, and set up alerting rules through Alertmanager to notify the team in real time of any potential performance degradation or failures.

LESSON-END PROJECT

Monitoring Apache Server Metrics Using Prometheus

Duration: 25 Min.

Perform the following:

1. Set up MySQL using Docker
2. Set up MySQL Server Exporter with Docker
3. Configure Alertmanager using Docker
4. Configure and start Prometheus in a Docker container
5. Explore MySQL Exporter metrics and alerting config on Prometheus
6. Simulate MySQL failure

Expected deliverables: A setup for MySQL monitoring with Prometheus, including MySQL Exporter configuration, PromQL queries for performance analysis, and alerting rules for key MySQL performance indicators

LESSON-END PROJECT

Thank You