1 PCA

Generated on: 2026-01-13 15:04:41 **Version:** 1.0

1.1 Table of Contents

1.2 Overview

The **Prometheus Certified Associate (PCA)** exam demonstrates an engineer's foundational knowledge of observability and skills using Prometheus, the open source systems monitoring and alerting toolkit.

1.3 Exam Overview

Detail	Information
Exam Format	Multiple Choice
Number of Questions	60
Duration	90 minutes
Passing Score	75%
Certification Validity	3 years
Cost	$250 USD
Retake Policy	1 free retake

1.4 Exam Domains & Weights

Domain	Weight
Observability Concepts	18%
Prometheus Fundamentals	20%
PromQL	28%
Instrumentation and Exporters	16%
Alerting & Dashboarding	18%

1.5 Prerequisites

- Basic understanding of Linux command line
- Familiarity with containerized environments (Docker/Kubernetes)
- General understanding of monitoring concepts
- Basic knowledge of YAML configuration

1.6 Study Resources

1.6.1 Official Resources

- PCA Exam Curriculum
- Prometheus Documentation
- Prometheus GitHub Repository

1.6.2 Recommended Courses

- Monitoring Systems and Services with Prometheus (LFS241)
- PromLabs Training

1.6.3 Practice Resources

- Prometheus Playground
- PromQL Basics
- Grafana Play

1.7 Exam Tips

1. **Understand the Three Pillars of Observability**: Logs, Metrics, and Traces
2. **Master PromQL**: This is 28% of the exam - practice writing queries
3. **Know SLA/SLO/SLI**: Understand the differences and relationships
4. **Practice with Real Prometheus**: Set up a local instance and experiment
5. **Understand Push vs Pull**: Know when to use each approach
6. **Learn Alertmanager**: Configuration, routing, and notification channels

1.8 Key Topics to Master

1.8.1 Observability Concepts

- Metrics, Logs, and Traces
- Push vs Pull model
- Service Discovery
- SLAs, SLOs, and SLIs

1.8.2 Prometheus Architecture

- Prometheus Server components
- Time Series Database (TSDB)
- Scrape configuration
- Data model and labels

1.8.3 PromQL

- Selectors and matchers
- Aggregation operators
- Rate and increase functions
- Histogram queries

1.8.4 Exporters

- Node Exporter
- Blackbox Exporter
- Custom exporters
- Client libraries

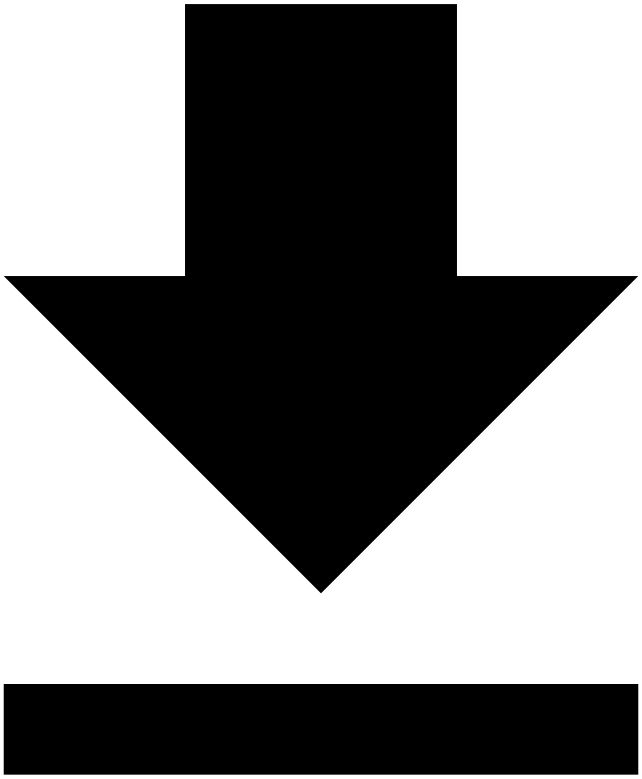
1.8.5 Alerting

- Alert rules
- Alertmanager configuration
- Routing and grouping
- Notification channels

1.9 Navigation

- Next: Observability Concepts →

1.10 Observability Concepts



Download PDF Version

1.11 Overview

This domain covers the foundational concepts of observability, including the three pillars (metrics, logs, traces), service discovery, and understanding SLAs, SLOs, and SLIs.

1.12 The Three Pillars of Observability

1.12.1 1. Metrics

Metrics are numerical values that measure some aspect of a system over intervals of time.

Characteristics: - Aggregatable and compressible - Low storage overhead - Good for alerting and trending - Examples: CPU usage, request count, error rate

Prometheus Metric Types:

```
# Counter - only increases (resets on restart)
http_requests_total{method="GET", status="200"} 1234

# Gauge - can go up or down
temperature_celsius{location="server_room"} 23.5

# Histogram - samples observations into buckets
http_request_duration_seconds_bucket{le="0.1"} 24054
http_request_duration_seconds_bucket{le="0.5"} 33444
http_request_duration_seconds_sum 53423
http_request_duration_seconds_count 144320

# Summary - similar to histogram with quantiles
go_gc_duration_seconds{quantile="0.5"} 0.000107458
go_gc_duration_seconds{quantile="0.9"} 0.000262326
```

1.12.2 2. Logs

Logs are immutable records that describe discrete events that have happened over time.

Characteristics: - High cardinality data - Detailed context for debugging - Higher storage requirements - Examples: Application errors, access logs, audit trails

Log Levels: - **DEBUG**: Detailed information for debugging - **INFO**: General operational information - **WARN**: Warning conditions - **ERROR**: Error conditions - **FATAL/CRITICAL**: Severe errors causing shutdown

1.12.3 3. Traces

Traces are records of the full paths or sequences of events that occur as requests flow through a system.

Key Concepts: - **Trace**: Complete journey of a request through the system - **Span**: A single operation within a trace - **Context Propagation**: Passing trace context between services

```
Trace ID: abc123
├── Span 1: API Gateway (10ms)
│   ├── Span 2: Auth Service (5ms)
│   └── Span 3: Backend Service (50ms)
│       ├── Span 4: Database Query (30ms)
│       └── Span 5: Cache Lookup (2ms)
```

1.13 Push vs Pull Model

1.13.1 Pull Model (Prometheus Default)

Prometheus actively scrapes metrics from targets at regular intervals.

Advantages: - Prometheus controls scrape timing - Easy to detect if a target is down - No need for targets to know about Prometheus - Simpler target configuration

Configuration Example:

```yaml
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
    scrape_interval: 15s
```

1.13.2 Push Model (Pushgateway)

Applications push metrics to an intermediary (Pushgateway).

Use Cases: - Short-lived batch jobs - Jobs behind firewalls - Legacy systems that can't expose endpoints

When to Use Push:

```
# Push metrics to Pushgateway
echo "job_completion_time $(date +%s)" | curl --data-binary @-
        http://pushgateway:9091/metrics/job/batch_job
```

Important: Pushgateway should NOT be used as a general metrics aggregator.

1.14 Service Discovery

Service discovery automatically finds and monitors targets without manual configuration.

1.14.1 Static Configuration

```yaml
scrape_configs:
  - job_name: 'static_targets'
    static_configs:
      - targets: ['server1:9090', 'server2:9090']
```

1.14.2 Kubernetes Service Discovery

```yaml
scrape_configs:
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod
```

```
    relabel_configs:
      - source_labels:
        [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: true
```

1.14.3 File-based Service Discovery

```
scrape_configs:
  - job_name: 'file_sd'
    file_sd_configs:
      - files:
          - '/etc/prometheus/targets/*.json'
        refresh_interval: 5m
```

1.14.4 DNS Service Discovery

```
scrape_configs:
  - job_name: 'dns_sd'
    dns_sd_configs:
      - names:
          - 'myservice.example.com'
        type: 'A'
        port: 9090
```

1.15 SLAs, SLOs, and SLIs

1.15.1 Service Level Agreement (SLA)

A formal agreement between a service provider and customer defining expected service levels.

Example: > “The service will be available 99.9% of the time, measured monthly. If availability falls below this threshold, customers will receive a 10% credit.”

1.15.2 Service Level Objective (SLO)

Internal targets that teams aim to achieve to meet SLAs.

Example:

```
SLO: 99.95% availability (stricter than SLA)
SLO: 95th percentile latency < 200ms
SLO: Error rate < 0.1%
```

1.15.3 Service Level Indicator (SLI)

The actual metrics used to measure service performance.

Common SLIs:

```
# Availability SLI
sum(rate(http_requests_total{status!~"5.."}[5m])) /
sum(rate(http_requests_total[5m]))

# Latency SLI (95th percentile)
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))

# Error Rate SLI
sum(rate(http_requests_total{status=~"5.."}[5m])) /
sum(rate(http_requests_total[5m]))
```

1.15.4 Error Budgets

The acceptable amount of unreliability based on SLO.

```
SLO: 99.9% availability
Error Budget: 0.1% (43.2 minutes/month)

If error budget is exhausted:
- Freeze feature releases
- Focus on reliability improvements
```

1.16 Key Concepts Summary

Concept	Description	Example
Metrics	Numerical measurements over time	CPU usage, request count
Logs	Discrete event records	Error messages, access logs
Traces	Request flow through system	Distributed transaction path
Pull	Prometheus scrapes targets	Default Prometheus model
Push	Targets push to gateway	Batch jobs, short-lived processes
SLA	Customer agreement	99.9% uptime guarantee
SLO	Internal target	99.95% availability target
SLI	Actual measurement	Current availability percentage

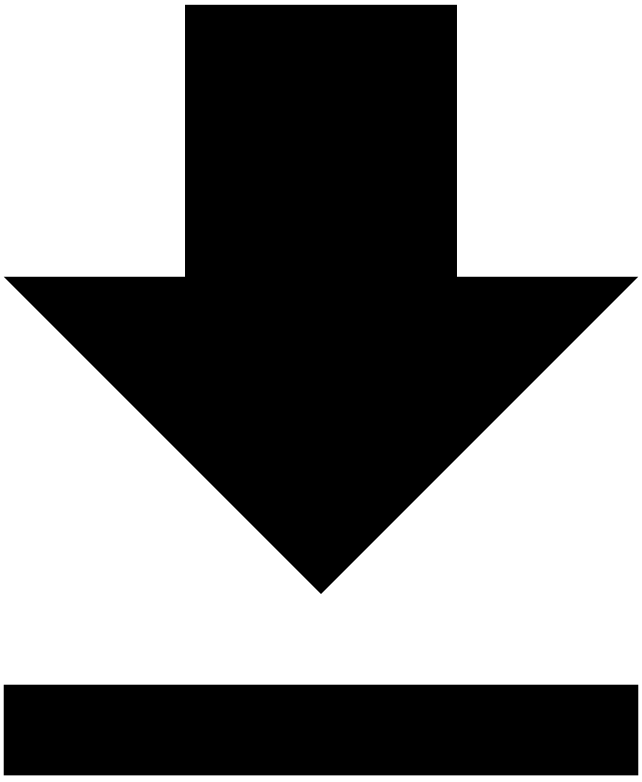
1.17 Practice Questions

1. What are the three pillars of observability?
2. When should you use the Pushgateway instead of the pull model?
3. What is the difference between an SLA and an SLO?
4. Name three types of service discovery supported by Prometheus.
5. What is a span in the context of distributed tracing?

1.18 Navigation

- ← Back to Overview
- Next: Prometheus Fundamentals →

1.19 Prometheus Fundamentals



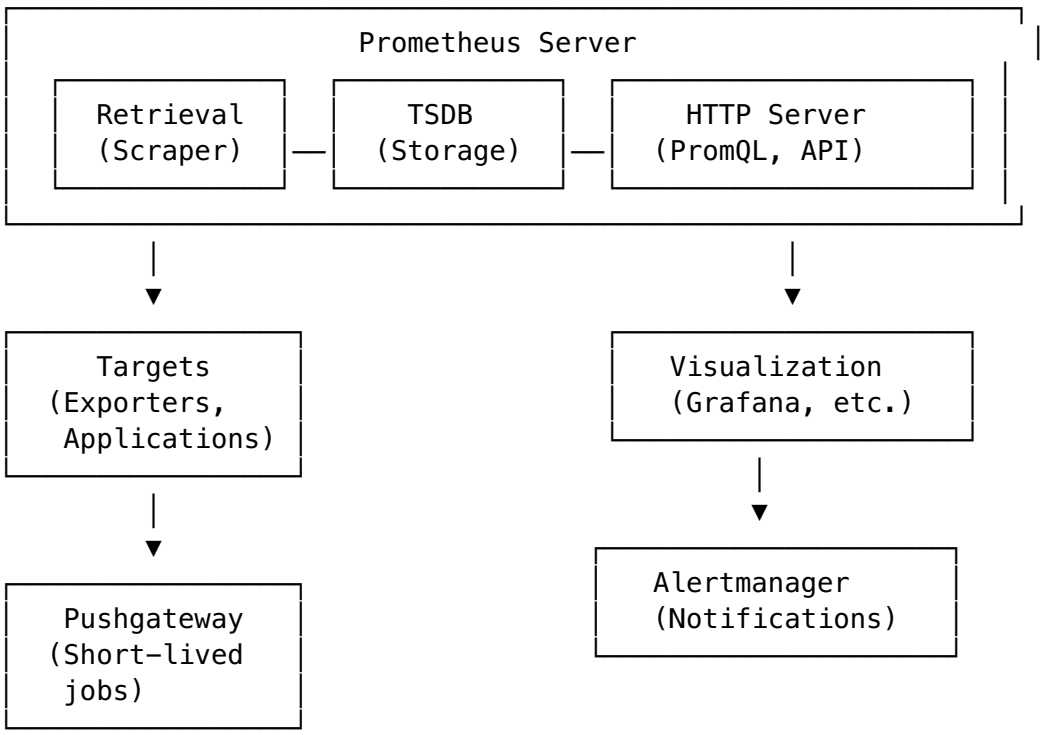
Download PDF Version

1.20 Overview

This domain covers Prometheus architecture, configuration, scraping, data model, labels, and understanding Prometheus limitations.

1.21 Prometheus Architecture

1.21.1 Core Components



1.21.2 Component Descriptions

Component	Purpose
Prometheus Server	Core component that scrapes and stores metrics
TSDB	Time Series Database for efficient storage
Retrieval	Scrapes metrics from configured targets
HTTP Server	Serves PromQL queries and API requests
Alertmanager	Handles alerts, routing, and notifications
Pushgateway	Accepts pushed metrics from short-lived jobs
Exporters	

Component	Purpose
	Expose metrics from third-party systems

1.22 Configuration

1.22.1 Basic Configuration Structure

```yaml
# prometheus.yml
global:
  scrape_interval: 15s      # How often to scrape targets
  evaluation_interval: 15s  # How often to evaluate rules
  scrape_timeout: 10s       # Timeout for scrape requests

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - alertmanager:9093

# Rule files
rule_files:
  - "rules/*.yml"
  - "alerts/*.yml"

# Scrape configurations
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

1.22.2 Scrape Configuration Options

```yaml
scrape_configs:
  - job_name: 'example'
    # Override global settings
    scrape_interval: 30s
    scrape_timeout: 10s

    # Metrics path (default: /metrics)
    metrics_path: /metrics

    # Scheme (http or https)
    scheme: https

    # Basic authentication
    basic_auth:
      username: admin
```

```
      password: secret

    # TLS configuration
    tls_config:
      ca_file: /path/to/ca.crt
      cert_file: /path/to/client.crt
      key_file: /path/to/client.key

    # Static targets
    static_configs:
      - targets: ['host1:9090', 'host2:9090']
        labels:
          env: production
          team: backend
```

1.22.3 Relabeling Configuration

Relabeling allows modifying labels before scraping or storing.

```
scrape_configs:
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod

    relabel_configs:
      # Keep only pods with annotation
      - source_labels:
        [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: true

      # Replace metrics path
      - source_labels:
        [__meta_kubernetes_pod_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)

      # Add namespace label
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: namespace

      # Drop specific labels
      - action: labeldrop
        regex: __meta_kubernetes_pod_label_(.+)
```

1.23 Data Model

1.23.1 Time Series Structure

Every time series is uniquely identified by: - **Metric name**: Describes what is being measured - **Labels**: Key-value pairs for dimensions

```
<metric_name>{<label_name>=<label_value>, ...}

# Examples
http_requests_total{method="GET", status="200", path="/api"}
node_cpu_seconds_total{cpu="0", mode="idle"}
```

1.23.2 Metric Naming Conventions

```
# Format: <namespace>_<name>_<unit>_<suffix>

# Good examples
http_requests_total              # Counter
http_request_duration_seconds    # Histogram
node_memory_bytes_total          # Gauge
process_cpu_seconds_total        # Counter

# Suffixes
_total    - Counter
_count    - Number of observations (histogram/summary)
_sum      - Sum of observations (histogram/summary)
_bucket   - Histogram bucket
_info     - Info metric (gauge with value 1)
```

1.23.3 Label Best Practices

```
# Good labels - low cardinality
http_requests_total{method="GET", status="200"}
http_requests_total{method="POST", status="201"}

# Bad labels - high cardinality (avoid!)
http_requests_total{user_id="12345", request_id="abc-123"}
```

Cardinality Warning: Each unique label combination creates a new time series. High cardinality can cause performance issues.

1.24 Metric Types

1.24.1 Counter

A cumulative metric that only increases (or resets to zero on restart).

```
# Raw counter value
http_requests_total

# Rate of increase per second
rate(http_requests_total[5m])

# Total increase over time window
increase(http_requests_total[1h])
```

1.24.2 Gauge

A metric that can go up or down.

```
# Current value
node_memory_MemAvailable_bytes

# Changes over time
delta(node_memory_MemAvailable_bytes[1h])

# Derivative (rate of change)
deriv(node_memory_MemAvailable_bytes[1h])
```

1.24.3 Histogram

Samples observations and counts them in configurable buckets.

```
# Bucket counts
http_request_duration_seconds_bucket{le="0.1"}
http_request_duration_seconds_bucket{le="0.5"}
http_request_duration_seconds_bucket{le="+Inf"}

# Calculate percentiles
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))

# Average duration
rate(http_request_duration_seconds_sum[5m]) /
rate(http_request_duration_seconds_count[5m])
```

1.24.4 Summary

Similar to histogram but calculates quantiles on the client side.

```
# Pre-calculated quantiles
go_gc_duration_seconds{quantile="0.5"}
go_gc_duration_seconds{quantile="0.9"}
go_gc_duration_seconds{quantile="0.99"}
```

Histogram vs Summary: | Feature | Histogram | Summary | |———|———–|———–| | Quantile calculation | Server-side (PromQL) | Client-side | | Aggregatable | Yes | No | | Bucket configuration | Required | Not needed | | Accuracy | Depends on buckets | Configurable |

1.25 Exposition Format

1.25.1 Text Format

```
# HELP http_requests_total Total number of HTTP requests
# TYPE http_requests_total counter
http_requests_total{method="GET",status="200"} 1234 1609459200000
http_requests_total{method="POST",status="201"} 567

# HELP node_cpu_seconds_total CPU time spent in each mode
# TYPE node_cpu_seconds_total counter
node_cpu_seconds_total{cpu="0",mode="idle"} 123456.78
node_cpu_seconds_total{cpu="0",mode="user"} 45678.90
```

1.25.2 Format Components

```
# HELP <metric_name> <description>
# TYPE <metric_name> <type>
<metric_name>{<labels>} <value> [<timestamp>]
```

1.26 Prometheus Limitations

1.26.1 Storage Limitations

- **Local storage only**: No built-in clustering
- **Retention**: Limited by disk space
- **No long-term storage**: Use remote write for long-term

```yaml
# Storage configuration
storage:
  tsdb:
    path: /prometheus/data
    retention.time: 15d
    retention.size: 50GB
```

1.26.2 Scalability Limitations

- **Single server**: Prometheus is designed for single-server operation
- **High availability**: Requires running multiple instances
- **Federation**: For hierarchical scaling

```yaml
# Federation configuration
scrape_configs:
```

```yaml
  - job_name: 'federate'
    honor_labels: true
    metrics_path: '/federate'
    params:
      'match[]':
        - '{job="prometheus"}'
        - '{__name__=~"job:.*"}'
    static_configs:
      - targets:
          - 'prometheus-1:9090'
          - 'prometheus-2:9090'
```

1.26.3 Other Limitations

Limitation	Workaround
No event logging	Use logging systems (Loki, ELK)
Pull-only model	Pushgateway for batch jobs
No built-in auth	Use reverse proxy
Single-node	Federation, Thanos, Cortex

1.27 Remote Write/Read

1.27.1 Remote Write Configuration

```yaml
remote_write:
  - url: "http://remote-storage:9201/write"
    queue_config:
      max_samples_per_send: 1000
      batch_send_deadline: 5s
    write_relabel_configs:
      - source_labels: [__name__]
        regex: 'expensive_metric_.*'
        action: drop
```

1.27.2 Remote Read Configuration

```yaml
remote_read:
  - url: "http://remote-storage:9201/read"
    read_recent: true
```

1.28 Practice Questions

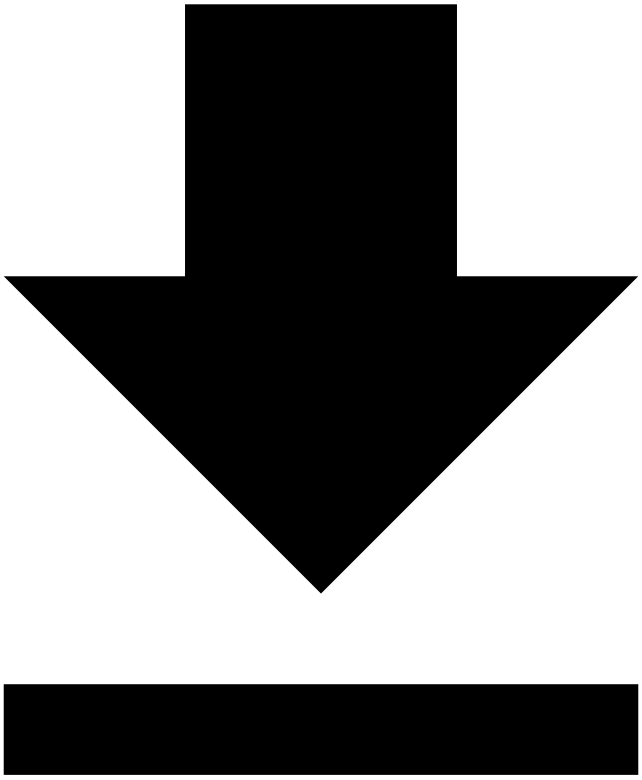
1. What are the main components of Prometheus architecture?
2. What is the difference between `scrape_interval` and `evaluation_interval`?
3. How do you configure basic authentication for a scrape target?
4. What is the purpose of relabeling in Prometheus?

5. What are the four metric types in Prometheus?
6. Why should you avoid high-cardinality labels?
7. What is the difference between histogram and summary metrics?
8. How can you scale Prometheus for high availability?

1.29 Navigation

- ← Back to Observability Concepts
- Next: PromQL →

1.30 PromQL



Download PDF Version

1.31 Overview

PromQL (Prometheus Query Language) is the most heavily weighted domain in the PCA exam. This section covers selecting data, rates, aggregations, binary operators, and histogram queries.

1.32 Data Types

1.32.1 Instant Vector

A set of time series with a single sample value at a given timestamp.

```
# Returns current value for all matching series
http_requests_total
http_requests_total{method="GET"}
```

1.32.2 Range Vector

A set of time series with a range of samples over time.

```
# Returns samples from the last 5 minutes
http_requests_total[5m]
http_requests_total{method="GET"}[1h]
```

1.32.3 Scalar

A simple numeric floating-point value.

```
# Scalar values
42
3.14
```

1.32.4 String

A simple string value (rarely used).

```
"hello world"
```

1.33 Selectors and Matchers

1.33.1 Label Matchers

```
# Exact match
http_requests_total{method="GET"}

# Not equal
http_requests_total{method!="GET"}
```

```
# Regex match
http_requests_total{method=~"GET|POST"}

# Regex not match
http_requests_total{method!~"DELETE|PUT"}

# Multiple matchers (AND logic)
http_requests_total{method="GET", status="200"}
```

1.33.2 Metric Name Matching

```
# Match metric name with regex
{__name__=~"http_.*"}

# All metrics with specific label
{job="prometheus"}
```

1.34 Time Ranges and Offsets

1.34.1 Range Vectors

```
# Last 5 minutes
http_requests_total[5m]

# Last 1 hour
http_requests_total[1h]

# Last 1 day
http_requests_total[1d]
```

1.34.2 Time Units

Unit	Description
`ms`	Milliseconds
`s`	Seconds
`m`	Minutes
`h`	Hours
`d`	Days
`w`	Weeks
`y`	Years

1.34.3 Offset Modifier

```
# Value from 1 hour ago
http_requests_total offset 1h

# Rate from 1 day ago
```

```
rate(http_requests_total[5m] offset 1d)

# Compare current to yesterday
http_requests_total - http_requests_total offset 1d
```

1.34.4 @ Modifier

```
# Value at specific timestamp
http_requests_total @ 1609459200

# Value at start of query range
http_requests_total @ start()

# Value at end of query range
http_requests_total @ end()
```

1.35 Rates and Derivatives

1.35.1 rate()

Calculates per-second average rate of increase for counters.

```
# Requests per second over 5 minutes
rate(http_requests_total[5m])

# CPU usage rate
rate(node_cpu_seconds_total{mode="user"}[5m])
```

1.35.2 irate()

Instant rate - uses last two data points (more volatile).

```
# Instant rate (more responsive to spikes)
irate(http_requests_total[5m])
```

1.35.3 increase()

Total increase over the time range.

```
# Total requests in last hour
increase(http_requests_total[1h])

# Total errors in last day
increase(http_errors_total[1d])
```

1.35.4 delta()

Difference between first and last value (for gauges).

```
# Memory change over 1 hour
delta(node_memory_MemAvailable_bytes[1h])
```

1.35.5 deriv()

Per-second derivative using linear regression (for gauges).

```
# Rate of memory change
deriv(node_memory_MemAvailable_bytes[1h])
```

1.35.6 Rate vs irate vs increase

Function	Use Case	Behavior
`rate()`	General rate calculation	Average over range
`irate()`	Volatile metrics, graphs	Last two points only
`increase()`	Total count over period	Total increase

1.36 Aggregation Operators

1.36.1 Basic Aggregations

```
# Sum all values
sum(http_requests_total)

# Average
avg(http_requests_total)

# Minimum
min(http_requests_total)

# Maximum
max(http_requests_total)

# Count of series
count(http_requests_total)

# Standard deviation
stddev(http_requests_total)

# Standard variance
stdvar(http_requests_total)
```

1.36.2 Aggregation with Dimensions

```
# Sum by specific label
sum by (method) (rate(http_requests_total[5m]))

# Sum excluding specific label
```

```
sum without (instance) (rate(http_requests_total[5m]))

# Multiple dimensions
sum by (method, status) (rate(http_requests_total[5m]))
```

1.36.3 topk and bottomk

```
# Top 5 by request rate
topk(5, rate(http_requests_total[5m]))

# Bottom 3 by memory usage
bottomk(3, node_memory_MemAvailable_bytes)
```

1.36.4 count_values

```
# Count occurrences of each value
count_values("version", build_info)
```

1.36.5 quantile

```
# 90th percentile of values
quantile(0.9, http_request_duration_seconds)
```

1.37 Aggregation Over Time

```
# Average over time range
avg_over_time(node_cpu_seconds_total[1h])

# Sum over time range
sum_over_time(http_requests_total[1h])

# Min/Max over time range
min_over_time(temperature_celsius[1d])
max_over_time(temperature_celsius[1d])

# Count samples in range
count_over_time(up[1h])

# Quantile over time
quantile_over_time(0.95, http_request_duration_seconds[1h])

# Standard deviation over time
stddev_over_time(response_time_seconds[1h])

# Last value in range
last_over_time(up[5m])

# Check if present in range
present_over_time(up[5m])
```

1.38 Binary Operators

1.38.1 Arithmetic Operators

```
# Addition
node_memory_MemTotal_bytes + node_memory_SwapTotal_bytes

# Subtraction
node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes

# Multiplication
rate(http_requests_total[5m]) * 100

# Division
node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes

# Modulo
http_requests_total % 100

# Power
2 ^ 10
```

1.38.2 Comparison Operators

```
# Greater than
http_requests_total > 1000

# Less than
node_memory_MemAvailable_bytes < 1073741824

# Equal
up == 1

# Not equal
up != 0

# Greater than or equal
rate(http_requests_total[5m]) >= 10

# Less than or equal
error_rate <= 0.01
```

1.38.3 Logical Operators

```
# AND - returns left side where both sides have matching labels
http_requests_total and on(instance) up

# OR - returns all series from both sides
http_requests_total or http_errors_total
```

```
# UNLESS - returns left side where right side doesn't match
http_requests_total unless on(instance) up == 0
```

1.38.4 Vector Matching

```
# One-to-one matching
http_requests_total / on(instance, job) http_requests_duration_sum

# Many-to-one matching
http_requests_total / ignoring(status) group_left
http_requests_duration_sum

# One-to-many matching
http_requests_duration_sum * ignoring(status) group_right
http_requests_total
```

1.39 Histogram Functions

1.39.1 histogram_quantile()

Calculate quantiles from histogram buckets.

```
# 95th percentile latency
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))

# 50th percentile (median) by endpoint
histogram_quantile(0.50, sum by (le, endpoint)
(rate(http_request_duration_seconds_bucket[5m])))

# Multiple quantiles
histogram_quantile(0.99,
rate(http_request_duration_seconds_bucket[5m]))
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))
histogram_quantile(0.50,
rate(http_request_duration_seconds_bucket[5m]))
```

1.39.2 Average from Histogram

```
# Average request duration
rate(http_request_duration_seconds_sum[5m]) /
rate(http_request_duration_seconds_count[5m])
```

1.39.3 Histogram Bucket Analysis

```
# Requests under 100ms
rate(http_request_duration_seconds_bucket{le="0.1"}[5m])

# Percentage of requests under 500ms
```

```
rate(http_request_duration_seconds_bucket{le="0.5"}[5m]) /
rate(http_request_duration_seconds_count[5m])
```

1.40 Useful Functions

1.40.1 Label Functions

```
# Add/modify labels
label_replace(up, "host", "$1", "instance", "(.*):.*")

# Join labels
label_join(up, "full_name", "-", "job", "instance")
```

1.40.2 Math Functions

```
# Absolute value
abs(delta(temperature[1h]))

# Ceiling
ceil(http_requests_total / 100)

# Floor
floor(http_requests_total / 100)

# Round
round(http_requests_total / 100, 0.1)

# Clamp values
clamp(cpu_usage, 0, 100)
clamp_min(value, 0)
clamp_max(value, 100)
```

1.40.3 Time Functions

```
# Current timestamp
time()

# Day of month (1-31)
day_of_month()

# Day of week (0-6, Sunday=0)
day_of_week()

# Hour (0-23)
hour()

# Minute (0-59)
minute()

# Month (1-12)
```

```
month()

# Year
year()
```

1.40.4 Sorting

```
# Sort ascending
sort(http_requests_total)

# Sort descending
sort_desc(http_requests_total)
```

1.40.5 Other Functions

```
# Check if vector is empty
absent(nonexistent_metric)

# Check if specific series is absent
absent(up{job="missing"})

# Changes in value
changes(process_start_time_seconds[1h])

# Resets (counter resets)
resets(http_requests_total[1h])
```

1.41 Common Query Patterns

1.41.1 Error Rate

```
# Error percentage
sum(rate(http_requests_total{status=~"5.."}[5m])) /
sum(rate(http_requests_total[5m])) * 100
```

1.41.2 Availability

```
# Service availability
avg(up{job="myservice"}) * 100
```

1.41.3 Saturation

```
# CPU saturation
100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}
[5m])) * 100)
```

1.41.4 Request Rate

```
# Requests per second by endpoint
sum by (endpoint) (rate(http_requests_total[5m]))
```

1.41.5 Latency Percentiles

```
# P50, P95, P99 latencies
histogram_quantile(0.50, sum by (le)
(rate(http_request_duration_seconds_bucket[5m])))
histogram_quantile(0.95, sum by (le)
(rate(http_request_duration_seconds_bucket[5m])))
histogram_quantile(0.99, sum by (le)
(rate(http_request_duration_seconds_bucket[5m])))
```

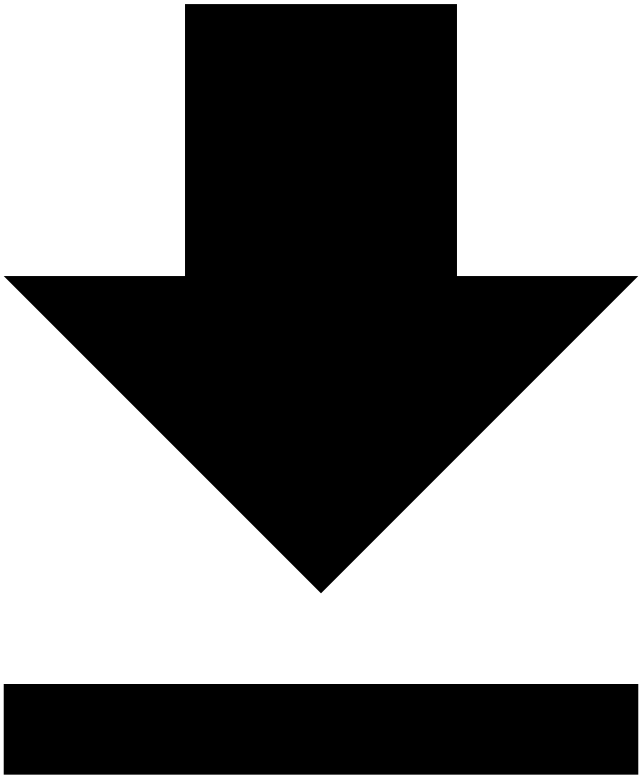
1.42 Practice Questions

1. What is the difference between an instant vector and a range vector?
2. How do you calculate the per-second rate of a counter?
3. What is the difference between `rate()` and `irate()`?
4. How do you aggregate metrics by a specific label?
5. Write a query to get the 95th percentile latency from a histogram.
6. How do you compare current values to values from 1 hour ago?
7. What does the `absent()` function do?
8. How do you calculate error rate as a percentage?

1.43 Navigation

- ← Back to Prometheus Fundamentals
- Next: Instrumentation and Exporters →

1.44 Instrumentation and Exporters



Download PDF Version

1.45 Overview

This domain covers client libraries, instrumentation best practices, exporters, and metric naming conventions.

1.46 Client Libraries

Prometheus provides official client libraries for instrumenting your applications.

1.46.1 Official Libraries

Language	Library
Go	`prometheus/client_golang`
Java	`prometheus/client_java`

Language	Library
Python	`prometheus/client_python`
Ruby	`prometheus/client_ruby`
Rust	`prometheus/client_rust`

1.46.2 Go Client Example

```
package main

import (
    "net/http"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
    // Counter
    httpRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "http_requests_total",
            Help: "Total number of HTTP requests",
        },
        []string{"method", "status"},
    )

    // Gauge
    activeConnections = prometheus.NewGauge(
        prometheus.GaugeOpts{
            Name: "active_connections",
            Help: "Number of active connections",
        },
    )

    // Histogram
    requestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name:    "http_request_duration_seconds",
            Help:    "HTTP request duration in seconds",
            Buckets: prometheus.DefBuckets,
        },
        []string{"method", "path"},
    )
)

func init() {
    prometheus.MustRegister(httpRequestsTotal)
    prometheus.MustRegister(activeConnections)
    prometheus.MustRegister(requestDuration)
}

func main() {
```

```
    http.Handle("/metrics", promhttp.Handler())
    http.ListenAndServe(":8080", nil)
}
```

1.46.3 Python Client Example

```
from prometheus_client import Counter, Gauge, Histogram,
        start_http_server

# Counter
http_requests_total = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'status']
)

# Gauge
active_connections = Gauge(
    'active_connections',
    'Number of active connections'
)

# Histogram
request_duration = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'path'],
    buckets=[0.01, 0.05, 0.1, 0.5, 1.0, 5.0]
)

# Usage
http_requests_total.labels(method='GET', status='200').inc()
active_connections.set(42)

with request_duration.labels(method='GET', path='/api').time():
    # Your code here
    pass

# Start metrics server
start_http_server(8000)
```

1.46.4 Java Client Example

```
import io.prometheus.client.Counter;
import io.prometheus.client.Gauge;
import io.prometheus.client.Histogram;
import io.prometheus.client.exporter.HTTPServer;

public class Application {
    static final Counter requests = Counter.build()
        .name("http_requests_total")
        .help("Total HTTP requests")
```

```
            .labelNames("method", "status")
            .register();

    static final Gauge connections = Gauge.build()
            .name("active_connections")
            .help("Active connections")
            .register();

    static final Histogram duration = Histogram.build()
            .name("http_request_duration_seconds")
            .help("Request duration")
            .labelNames("method", "path")
            .buckets(0.01, 0.05, 0.1, 0.5, 1.0, 5.0)
            .register();

    public static void main(String[] args) throws Exception {
        HTTPServer server = new HTTPServer(8080);

        // Usage
        requests.labels("GET", "200").inc();
        connections.set(42);

        Histogram.Timer timer = duration.labels("GET", "/
        api").startTimer();
        try {
            // Your code here
        } finally {
            timer.observeDuration();
        }
    }
}
```

1.47 Instrumentation Best Practices

1.47.1 What to Instrument

The Four Golden Signals (Google SRE):

1. **Latency**: Time to service a request
2. **Traffic**: Demand on your system (requests/second)
3. **Errors**: Rate of failed requests
4. **Saturation**: How "full" your service is

RED Method (for services):

- **Rate**: Requests per second
- **Errors**: Failed requests per second
- **Duration**: Time per request

USE Method (for resources):

- **Utilization**: Percentage of resource used

- **Saturation**: Amount of work queued
- **Errors**: Error events

1.47.2 Counter Best Practices

```
// Good: Use _total suffix
http_requests_total

// Good: Include relevant labels
http_requests_total{method="GET", status="200"}

// Bad: Don't use counters for values that can decrease
// Use gauge instead for things like queue size
```

1.47.3 Gauge Best Practices

```
// Good: Current state metrics
active_connections
queue_size
temperature_celsius

// Good: Use for values that go up and down
memory_usage_bytes
```

1.47.4 Histogram Best Practices

```
// Good: Choose appropriate buckets for your use case
prometheus.HistogramOpts{
    Name:    "http_request_duration_seconds",
    Buckets: []float64{0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5,
        10},
}

// Good: Use seconds as the base unit
http_request_duration_seconds

// Bad: Don't use milliseconds
http_request_duration_milliseconds  // Avoid this
```

1.47.5 Label Best Practices

```
// Good: Low cardinality labels
http_requests_total{method="GET", status="200"}

// Bad: High cardinality labels (avoid!)
http_requests_total{user_id="12345", request_id="abc-123"}

// Good: Bounded set of values
http_requests_total{status_class="2xx"}
```

```
// Bad: Unbounded values
http_requests_total{path="/users/12345/orders/67890"}
```

1.48 Metric Naming Conventions

1.48.1 Format

```
<namespace>_<name>_<unit>_<suffix>
```

1.48.2 Examples

```
# Application namespace
myapp_http_requests_total
myapp_database_connections_active

# Unit in name
http_request_duration_seconds
node_memory_bytes_total
process_cpu_seconds_total

# Suffixes
_total      # Counter
_count      # Histogram/Summary count
_sum        # Histogram/Summary sum
_bucket     # Histogram bucket
_info       # Info metric (value always 1)
_created    # Creation timestamp
```

1.48.3 Naming Rules

1. Use snake_case
2. Include unit in name (seconds, bytes, etc.)
3. Use base units (seconds not milliseconds, bytes not kilobytes)
4. Use `_total` suffix for counters
5. Don't include label names in metric name

1.49 Exporters

Exporters expose metrics from third-party systems in Prometheus format.

1.49.1 Node Exporter

Exposes hardware and OS metrics from Linux/Unix systems.

```
# Install and run
./node_exporter --web.listen-address=":9100"

# Key metrics
```

```
node_cpu_seconds_total
node_memory_MemTotal_bytes
node_memory_MemAvailable_bytes
node_filesystem_size_bytes
node_network_receive_bytes_total
node_load1, node_load5, node_load15
```

Prometheus Configuration:

```yaml
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

1.49.2 Blackbox Exporter

Probes endpoints over HTTP, HTTPS, DNS, TCP, ICMP.

```yaml
# blackbox.yml
modules:
  http_2xx:
    prober: http
    timeout: 5s
    http:
      valid_http_versions: ["HTTP/1.1", "HTTP/2.0"]
      valid_status_codes: [200, 201, 202]
      method: GET

  tcp_connect:
    prober: tcp
    timeout: 5s

  dns_lookup:
    prober: dns
    timeout: 5s
    dns:
      query_name: "example.com"
```

Prometheus Configuration:

```yaml
scrape_configs:
  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
          - https://example.com
          - https://api.example.com
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
```

```
        target_label: instance
      - target_label: __address__
        replacement: blackbox-exporter:9115
```

1.49.3 Other Common Exporters

Exporter	Purpose	Default Port
node_exporter	Linux/Unix system metrics	9100
blackbox_exporter	Endpoint probing	9115
mysqld_exporter	MySQL metrics	9104
postgres_exporter	PostgreSQL metrics	9187
redis_exporter	Redis metrics	9121
nginx_exporter	NGINX metrics	9113
kafka_exporter	Kafka metrics	9308
mongodb_exporter	MongoDB metrics	9216
elasticsearch_exporter	Elasticsearch metrics	9114
cloudwatch_exporter	AWS CloudWatch metrics	9106

1.49.4 Writing Custom Exporters

```python
from prometheus_client import Gauge, start_http_server
import time
import random

# Define metrics
custom_metric = Gauge(
    'custom_application_metric',
    'A custom metric from our application',
    ['environment', 'service']
)

def collect_metrics():
    """Collect metrics from your data source"""
    while True:
        # Simulate collecting data
        value = random.random() * 100
        custom_metric.labels(
            environment='production',
            service='api'
        ).set(value)
        time.sleep(15)

if __name__ == '__main__':
    start_http_server(8000)
    collect_metrics()
```

1.50 Pushgateway

For short-lived jobs that can't be scraped.

1.50.1 When to Use

- Batch jobs
- Cron jobs
- Short-lived processes
- Jobs behind firewalls

1.50.2 Pushing Metrics

```
# Push a single metric
echo "job_last_success_timestamp $(date +%s)" | \
  curl --data-binary @- http://pushgateway:9091/metrics/job/
        batch_job

# Push multiple metrics
cat <<EOF | curl --data-binary @- http://pushgateway:9091/metrics/
        job/batch_job/instance/host1
# TYPE job_duration_seconds gauge
job_duration_seconds 42.5
# TYPE job_records_processed counter
job_records_processed 1234
EOF

# Delete metrics for a job
curl -X DELETE http://pushgateway:9091/metrics/job/batch_job
```

1.50.3 Python Example

```
from prometheus_client import CollectorRegistry, Gauge,
        push_to_gateway

registry = CollectorRegistry()

duration = Gauge(
    'job_duration_seconds',
    'Duration of batch job',
    registry=registry
)
duration.set(42.5)

records = Gauge(
    'job_records_processed',
    'Records processed by batch job',
    registry=registry
)
records.set(1234)
```

```
push_to_gateway(
    'pushgateway:9091',
    job='batch_job',
    registry=registry
)
```

1.50.4 Pushgateway Caveats

- Metrics persist until deleted
- No automatic cleanup
- Single point of failure
- Not for general metrics collection

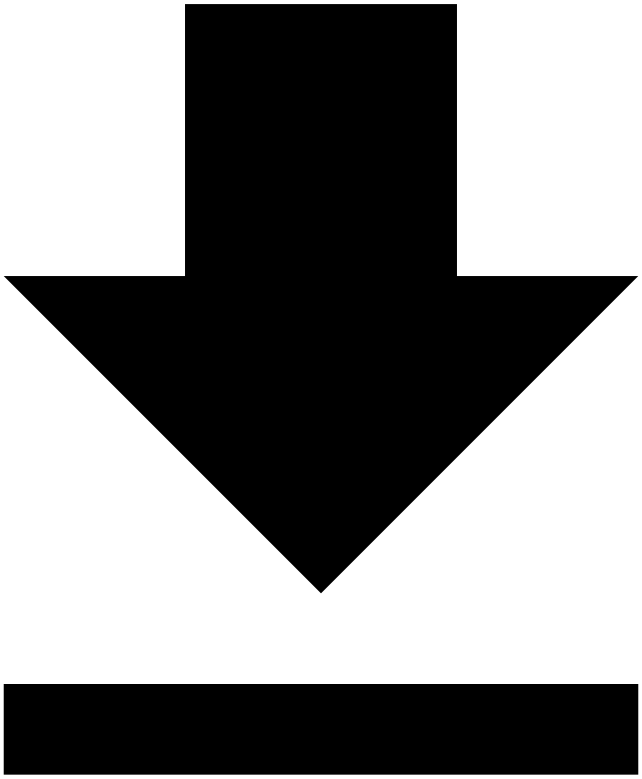
1.51 Practice Questions

1. What are the four golden signals of monitoring?
2. Name three official Prometheus client libraries.
3. What is the difference between the RED and USE methods?
4. When should you use the Pushgateway?
5. What metrics does the Node Exporter provide?
6. How do you configure the Blackbox Exporter to probe HTTP endpoints?
7. What are the naming conventions for Prometheus metrics?
8. Why should you avoid high-cardinality labels?

1.52 Navigation

- ← Back to PromQL
- Next: Alerting & Dashboarding →

1.53 Alerting & Dashboarding



Download PDF Version

1.54 Overview

This domain covers configuring alerting rules, understanding Alertmanager, and dashboarding basics with Grafana.

1.55 Alerting Rules

1.55.1 Rule Configuration

Alert rules are defined in YAML files and loaded by Prometheus.

```yaml
# alerts.yml
groups:
  - name: example-alerts
    rules:
```

```yaml
    - alert: HighErrorRate
      expr: rate(http_requests_total{status=~"5.."}[5m]) /
       rate(http_requests_total[5m]) > 0.05
      for: 5m
      labels:
        severity: critical
        team: backend
      annotations:
        summary: "High error rate detected"
        description: "Error rate is {{ $value |
       humanizePercentage }} for {{ $labels.instance }}"

    - alert: InstanceDown
      expr: up == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "Instance {{ $labels.instance }} is down"
        description: "{{ $labels.instance }} of job
       {{ $labels.job }} has been down for more than 1 minute."
```

1.55.2 Rule Components

Component	Description
`alert`	Name of the alert
`expr`	PromQL expression that triggers the alert
`for`	Duration the condition must be true before firing
`labels`	Additional labels to attach to the alert
`annotations`	Informational labels (summary, description, runbook)

1.55.3 Alert States

1. **Inactive**: Condition is not met
2. **Pending**: Condition is met but `for` duration hasn't elapsed
3. **Firing**: Condition met for the `for` duration

```
Inactive → Pending → Firing
                ↓
           Inactive (if condition becomes false)
```

1.55.4 Recording Rules

Pre-compute frequently used or expensive expressions.

```yaml
groups:
  - name: recording-rules
    rules:
      - record: job:http_requests_total:rate5m
        expr: sum by (job) (rate(http_requests_total[5m]))
```

```
      - record: instance:node_cpu_utilization:ratio
        expr: 1 - avg by (instance)
        (rate(node_cpu_seconds_total{mode="idle"}[5m]))

      - record: job:http_request_duration_seconds:p95
        expr: histogram_quantile(0.95, sum by (job, le)
        (rate(http_request_duration_seconds_bucket[5m])))
```

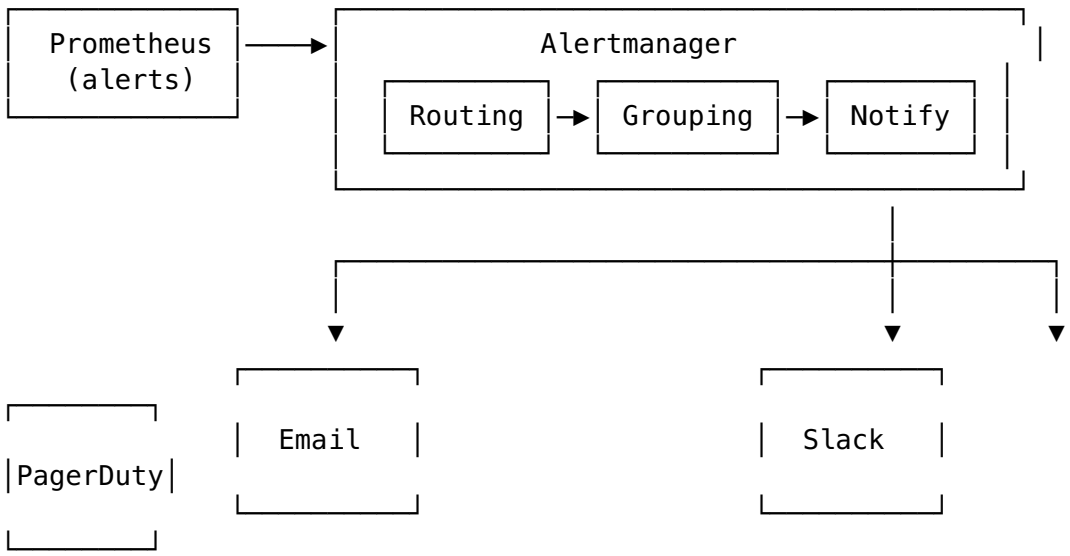
1.55.5 Prometheus Configuration

```
# prometheus.yml
rule_files:
  - "rules/*.yml"
  - "alerts/*.yml"

alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - alertmanager:9093
```

1.56 Alertmanager

1.56.1 Architecture



1.56.2 Configuration Structure

```
# alertmanager.yml
global:
  resolve_timeout: 5m
  smtp_smarthost: 'smtp.example.com:587'
  smtp_from: 'alertmanager@example.com'
```

```yaml
route:
  receiver: 'default-receiver'
  group_by: ['alertname', 'cluster']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - match:
        severity: critical
      receiver: 'pagerduty-critical'
    - match:
        severity: warning
      receiver: 'slack-warnings'

receivers:
  - name: 'default-receiver'
    email_configs:
      - to: 'team@example.com'

  - name: 'pagerduty-critical'
    pagerduty_configs:
      - service_key: '<pagerduty-service-key>'

  - name: 'slack-warnings'
    slack_configs:
      - api_url: 'https://hooks.slack.com/services/xxx/yyy/zzz'
        channel: '#alerts'

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

1.56.3 Routing

Routes determine which receiver handles an alert.

```yaml
route:
  receiver: 'default'
  routes:
    # Critical alerts go to PagerDuty
    - match:
        severity: critical
      receiver: 'pagerduty'
      continue: false

    # Database alerts go to DBA team
    - match_re:
        alertname: ^(MySQL|Postgres).*
```

```
      receiver: 'dba-team'

    # Multiple matchers (AND logic)
    - match:
        team: backend
        severity: warning
      receiver: 'backend-slack'
```

1.56.4 Grouping

Groups related alerts together to reduce notification noise.

```
route:
  group_by: ['alertname', 'cluster', 'service']
  group_wait: 30s      # Wait before sending first notification
  group_interval: 5m   # Wait before sending updates to group
  repeat_interval: 4h  # Wait before re-sending same alert
```

1.56.5 Silences

Temporarily mute alerts.

```
# Create silence via API
curl -X POST http://alertmanager:9093/api/v2/silences \
  -H "Content-Type: application/json" \
  -d '{
    "matchers": [
      {"name": "alertname", "value": "HighMemoryUsage", "isRegex":
        false},
      {"name": "instance", "value": "server1", "isRegex": false}
    ],
    "startsAt": "2024-01-01T00:00:00Z",
    "endsAt": "2024-01-01T06:00:00Z",
    "createdBy": "admin",
    "comment": "Maintenance window"
  }'
```

1.56.6 Inhibition

Suppress alerts when related alerts are firing.

```
inhibit_rules:
  # If critical alert fires, suppress warning for same alertname
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['alertname', 'instance']

  # If cluster is down, suppress all other cluster alerts
  - source_match:
```

```
      alertname: 'ClusterDown'
    target_match_re:
      alertname: '.+'
    equal: ['cluster']
```

1.56.7 Receivers

1.56.7.1 Email

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

1.56.7.2 Slack

```
receivers:
  - name: 'slack-notifications'
    slack_configs:
      - api_url: 'https://hooks.slack.com/services/xxx/yyy/zzz'
        channel: '#alerts'
        username: 'Alertmanager'
        icon_emoji: ':warning:'
        title: '{{ .Status | toUpper }}:
        {{ .CommonAnnotations.summary }}'
        text: '{{ range .Alerts }}{{ .Annotations.description }}
        {{ end }}'
```

1.56.7.3 PagerDuty

```
receivers:
  - name: 'pagerduty'
    pagerduty_configs:
      - service_key: '<integration-key>'
        severity: '{{ if eq .Status "firing" }}critical{{ else }}
        info{{ end }}'
        description: '{{ .CommonAnnotations.summary }}'
```

1.56.7.4 Webhook

```
receivers:
  - name: 'webhook'
    webhook_configs:
      - url: 'http://webhook-handler:8080/alerts'
        send_resolved: true
```

1.57 Dashboarding with Grafana

1.57.1 Data Source Configuration

```yaml
# Grafana datasource provisioning
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
    isDefault: true
    editable: false
```

1.57.2 Panel Types

Panel Type	Use Case
Time Series	Metrics over time
Stat	Single value display
Gauge	Value with thresholds
Bar Gauge	Horizontal/vertical bars
Table	Tabular data
Heatmap	Distribution over time
Logs	Log data display

1.57.3 Common Dashboard Patterns

1.57.3.1 Request Rate Panel

```
sum(rate(http_requests_total[5m])) by (service)
```

1.57.3.2 Error Rate Panel

```
sum(rate(http_requests_total{status=~"5.."}[5m])) by (service)
/
sum(rate(http_requests_total[5m])) by (service) * 100
```

1.57.3.3 Latency Percentiles Panel

```
histogram_quantile(0.99,
sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
histogram_quantile(0.95,
sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
histogram_quantile(0.50,
sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
```

1.57.3.4 Resource Utilization Panel

```
# CPU Usage
100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}
[5m])) * 100)

# Memory Usage
(1 - node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)
* 100

# Disk Usage
(1 - node_filesystem_avail_bytes / node_filesystem_size_bytes) *
100
```

1.57.4 Variables (Template Variables)

```
# Query variable for instances
Name: instance
Type: Query
Query: label_values(up, instance)

# Custom variable for time ranges
Name: interval
Type: Custom
Values: 1m,5m,15m,1h

# Using variables in queries
rate(http_requests_total{instance="$instance"}[$interval])
```

1.57.5 Dashboard Best Practices

1. **Use consistent naming**: Follow a naming convention
2. **Add descriptions**: Document what each panel shows
3. **Set appropriate time ranges**: Match to your SLOs
4. **Use variables**: Make dashboards reusable
5. **Group related panels**: Use rows to organize
6. **Set thresholds**: Visual indicators for good/bad states
7. **Include links**: Link to runbooks and related dashboards

1.58 Alerting Best Practices

1.58.1 When to Alert

Alert on symptoms, not causes:

```
# Good: Alert on user-facing impact
- alert: HighErrorRate
  expr: rate(http_errors_total[5m]) /
        rate(http_requests_total[5m]) > 0.01
```

```yaml
# Avoid: Alerting on every possible cause
- alert: HighCPU
  expr: cpu_usage > 80  # May not indicate a problem
```

1.58.2 Alert Fatigue Prevention

1. **Set appropriate thresholds**: Not too sensitive
2. **Use `for` duration**: Avoid flapping alerts
3. **Group related alerts**: Reduce notification volume
4. **Use inhibition**: Suppress redundant alerts
5. **Regular review**: Remove or tune noisy alerts

1.58.3 Alert Annotations

```yaml
annotations:
  summary: "Brief description of the alert"
  description: "Detailed information with {{ $labels.instance }}
        and {{ $value }}"
  runbook_url: "https://wiki.example.com/runbooks/high-error-rate"
  dashboard_url: "https://grafana.example.com/d/abc123"
```

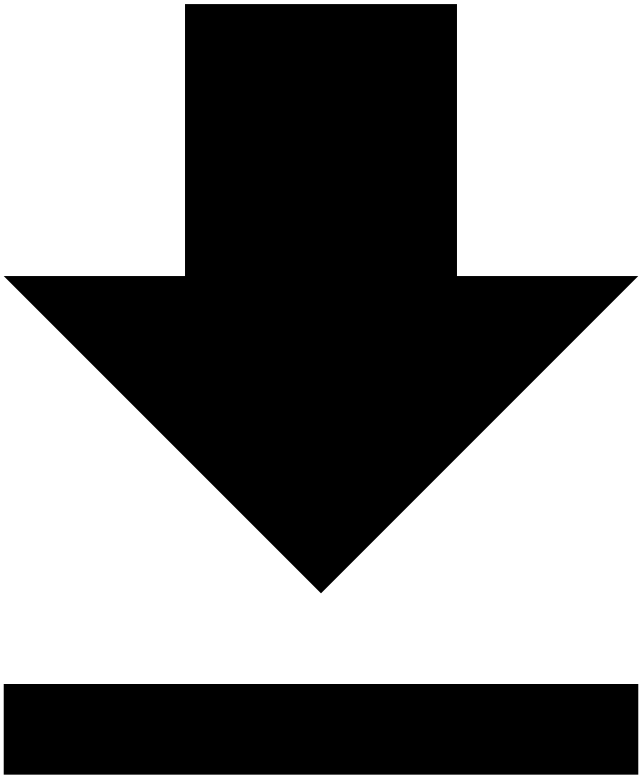
1.59 Practice Questions

1. What are the three states of an alert in Prometheus?
2. What is the purpose of the `for` clause in an alert rule?
3. How does Alertmanager group alerts?
4. What is the difference between silences and inhibition?
5. Name three notification channels supported by Alertmanager.
6. What is a recording rule and when should you use one?
7. How do you configure routing in Alertmanager?
8. What are template variables in Grafana used for?

1.60 Navigation

- ← Back to Instrumentation and Exporters
- Next: Sample Practice Questions →

1.61 Sample Practice Questions



Download PDF Version

1.62 Practice Resources

- Prometheus Documentation
- PromLabs Training
- Prometheus Playground

1.63 Domain 1: Observability Concepts (18%)

1.63.1 Question 1

What are the three pillars of observability?

Show Answer

Answer: Metrics, Logs, and Traces

- **Metrics**: Numerical values that measure aspects of a system over time
- **Logs**: Immutable records of discrete events
- **Traces**: Records of request paths through distributed systems

1.63.2 Question 2

What is the difference between an SLA, SLO, and SLI?

Show Answer

- **SLA (Service Level Agreement)**: A formal agreement with customers defining expected service levels
- **SLO (Service Level Objective)**: Internal targets that teams aim to achieve
- **SLI (Service Level Indicator)**: The actual metrics used to measure service performance

Example: SLA promises 99.9% uptime, SLO targets 99.95%, SLI measures actual availability.

1.63.3 Question 3

When should you use the Push model (Pushgateway) instead of the Pull model?

Show Answer

Use Pushgateway for: - Short-lived batch jobs - Cron jobs that complete before scraping - Jobs behind firewalls that can't be scraped - Legacy systems that can't expose endpoints

Important: Pushgateway should NOT be used as a general metrics aggregator.

1.63.4 Question 4

What is a span in the context of distributed tracing?

Show Answer

A **span** represents a single operation within a trace. It provides: - Start and end timestamps - Operation name - Tags/labels - Logs/events - Parent span reference

Multiple spans together form a complete trace showing the request flow through a system.

1.64 Domain 2: Prometheus Fundamentals (20%)

1.64.1 Question 5

What are the four metric types in Prometheus?

Show Answer

1. **Counter**: Cumulative metric that only increases (resets on restart)
2. **Gauge**: Metric that can go up or down
3. **Histogram**: Samples observations into configurable buckets
4. **Summary**: Similar to histogram but calculates quantiles client-side

1.64.2 Question 6

What is the purpose of relabeling in Prometheus?

Show Answer

Relabeling allows you to: - Modify labels before scraping (`relabel_configs`) - Filter which targets to scrape - Modify labels before storing (`metric_relabel_configs`) - Drop unwanted metrics - Rename labels - Extract values from labels using regex

1.64.3 Question 7

Why should you avoid high-cardinality labels?

Show Answer

High-cardinality labels (like user IDs or request IDs) create problems because: - Each unique label combination creates a new time series - Increases memory usage significantly - Slows down queries - Can cause Prometheus to run out of memory

Best practice: Use labels with bounded, low-cardinality values.

1.64.4 Question 8

What is the difference between `scrape_interval` and `evaluation_interval`?

Show Answer

- **scrape_interval**: How often Prometheus scrapes targets for metrics (default: 1m)
- **evaluation_interval**: How often Prometheus evaluates recording and alerting rules (default: 1m)

These can be set globally and overridden per scrape job.

1.65 Domain 3: PromQL (28%)

1.65.1 Question 9

What is the difference between `rate()` and `irate()`?

Show Answer

- **rate()**: Calculates per-second average rate over the entire range
  - More stable, better for alerting
  - Uses all data points in the range
- **irate()**: Calculates instant rate using only the last two data points
  - More responsive to changes
  - Better for volatile metrics in graphs
  - Can miss spikes between scrapes

1.65.2 Question 10

Write a PromQL query to calculate the 95th percentile latency from a histogram.

Show Answer

```
histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))
```

Or with aggregation by service:

```
histogram_quantile(0.95, sum by (service, le)
(rate(http_request_duration_seconds_bucket[5m])))
```

1.65.3 Question 11

How do you calculate error rate as a percentage?

Show Answer

```
sum(rate(http_requests_total{status=~"5.."}[5m]))
/
sum(rate(http_requests_total[5m]))
* 100
```

This divides error requests by total requests and multiplies by 100 for percentage.

1.65.4 Question 12

What does the `absent()` function do?

Show Answer

`absent()` returns 1 if the vector has no elements, otherwise returns nothing.

Use cases: - Alert when a metric is missing - Detect when a service stops reporting

```
# Alert if no data from job
absent(up{job="myservice"})
```

1.65.5 Question 13

How do you compare current values to values from 1 hour ago?

Show Answer

Use the `offset` modifier:

```
# Difference from 1 hour ago
http_requests_total - http_requests_total offset 1h

# Percentage change
(http_requests_total - http_requests_total offset 1h) /
http_requests_total offset 1h * 100
```

1.65.6 Question 14

What is the difference between `sum by` and `sum without`?

Show Answer

- **sum by (label)**: Aggregates and keeps only the specified labels
- **sum without (label)**: Aggregates and removes the specified labels, keeping all others

```
# Keep only 'method' label
sum by (method) (rate(http_requests_total[5m]))

# Remove 'instance' label, keep everything else
sum without (instance) (rate(http_requests_total[5m]))
```

1.66 Domain 4: Instrumentation and Exporters (16%)

1.66.1 Question 15

What are the Four Golden Signals of monitoring?

Show Answer

From Google SRE: 1. **Latency**: Time to service a request 2. **Traffic**: Demand on your system (requests/second) 3. **Errors**: Rate of failed requests 4. **Saturation**: How “full” your service is

1.66.2 Question 16

What metrics does the Node Exporter provide?

Show Answer

Node Exporter provides hardware and OS metrics: - CPU usage (`node_cpu_seconds_total`) - Memory (`node_memory_*`) - Disk (`node_filesystem_*`, `node_disk_*`) - Network (`node_network_*`) - Load average (`node_load1`, `node_load5`, `node_load15`) - System info

1.66.3 Question 17

What is the correct naming convention for Prometheus metrics?

Show Answer

Format: `<namespace>_<name>_<unit>_<suffix>`

Rules: - Use snake_case - Include unit in name (seconds, bytes) - Use base units (seconds not milliseconds) - Use `_total` suffix for counters - Use `_info` suffix for info metrics

Examples: - `http_requests_total` - `http_request_duration_seconds` - `node_memory_bytes_total`

1.66.4 Question 18

When should you use the Blackbox Exporter?

Show Answer

Use Blackbox Exporter for: - HTTP/HTTPS endpoint probing - TCP port checks - DNS lookups - ICMP ping checks - SSL certificate expiry monitoring

It's useful for monitoring external services or endpoints where you can't install an exporter.

1.67 Domain 5: Alerting & Dashboarding (18%)

1.67.1 Question 19

What are the three states of an alert in Prometheus?

Show Answer

1. **Inactive**: The alert condition is not met
2. **Pending**: Condition is met but `for` duration hasn't elapsed
3. **Firing**: Condition has been true for the `for` duration

1.67.2 Question 20

What is the purpose of the `for` clause in an alert rule?

Show Answer

The `for` clause specifies how long the condition must be true before the alert fires.

Benefits: - Prevents flapping alerts - Reduces false positives from brief spikes - Ensures the issue is persistent

```yaml
- alert: HighErrorRate
  expr: error_rate > 0.05
  for: 5m  # Must be true for 5 minutes
```

1.67.3 Question 21

What is the difference between silences and inhibition in Alertmanager?

Show Answer

Silences: - Manually created to mute specific alerts - Time-bounded (start and end time) - Used for maintenance windows - Created via UI or API

Inhibition: - Automatic suppression based on rules - Suppresses alerts when related alerts are firing - Configured in alertmanager.yml - Example: Suppress warnings when critical is firing

1.67.4 Question 22

What is a recording rule and when should you use one?

Show Answer

Recording rules pre-compute frequently used or expensive PromQL expressions.

Use when: - Query is computationally expensive - Query is used in multiple dashboards/alerts - You need to aggregate across federation - Query performance is critical

```yaml
- record: job:http_requests:rate5m
  expr: sum by (job) (rate(http_requests_total[5m]))
```

1.67.5 Question 23

How does Alertmanager group alerts?

Show Answer

Alertmanager groups alerts based on: - `group_by` labels in the route configuration - Alerts with matching group labels are batched together

Configuration:

```yaml
route:
  group_by: ['alertname', 'cluster']
  group_wait: 30s      # Wait before first notification
  group_interval: 5m   # Wait between group updates
  repeat_interval: 4h  # Wait before re-sending
```

1.67.6 Question 24

What notification channels does Alertmanager support?

Show Answer

Built-in receivers: - Email (SMTP) - Slack - PagerDuty - OpsGenie - VictorOps - Webhook (for custom integrations) - Pushover - WeChat - Telegram

Custom integrations can be built using the webhook receiver.

1.68 Bonus Questions

1.68.1 Question 25

What is meta-monitoring?

Show Answer

Meta-monitoring is monitoring the monitoring system itself (Prometheus monitoring Prometheus).

Important metrics to monitor: - `prometheus_tsdb_head_series` - Number of time series - `prometheus_engine_query_duration_seconds` - Query performance - `prometheus_target_scrape_pool_sync_total` - Scrape health - `up{job="prometheus"}` - Prometheus availability

1.68.2 Question 26

How can you scale Prometheus for high availability?

Show Answer

Options for scaling: 1. **Multiple instances**: Run identical Prometheus servers 2. **Federation**: Hierarchical Prometheus setup 3. **Remote storage**: Thanos, Cortex, Mimir for long-term storage 4. **Sharding**: Split targets across multiple Prometheus instances

Note: Prometheus itself doesn't support clustering natively.

1.69 Navigation

- ← Back to Alerting & Dashboarding
- Back to Overview
