

Application Delivery Fundamentals 2.0 B:

Prometheus and Grafana



High performance. Delivered.



Goals

- Observability Concepts
- Prometheus fundamentals
- PromQL
- Instrumentation and Exporters
- Alerting & Dashboarding
- Alerting advanced
- Alert Manager HA
- Prometheus Federation
- Grafana

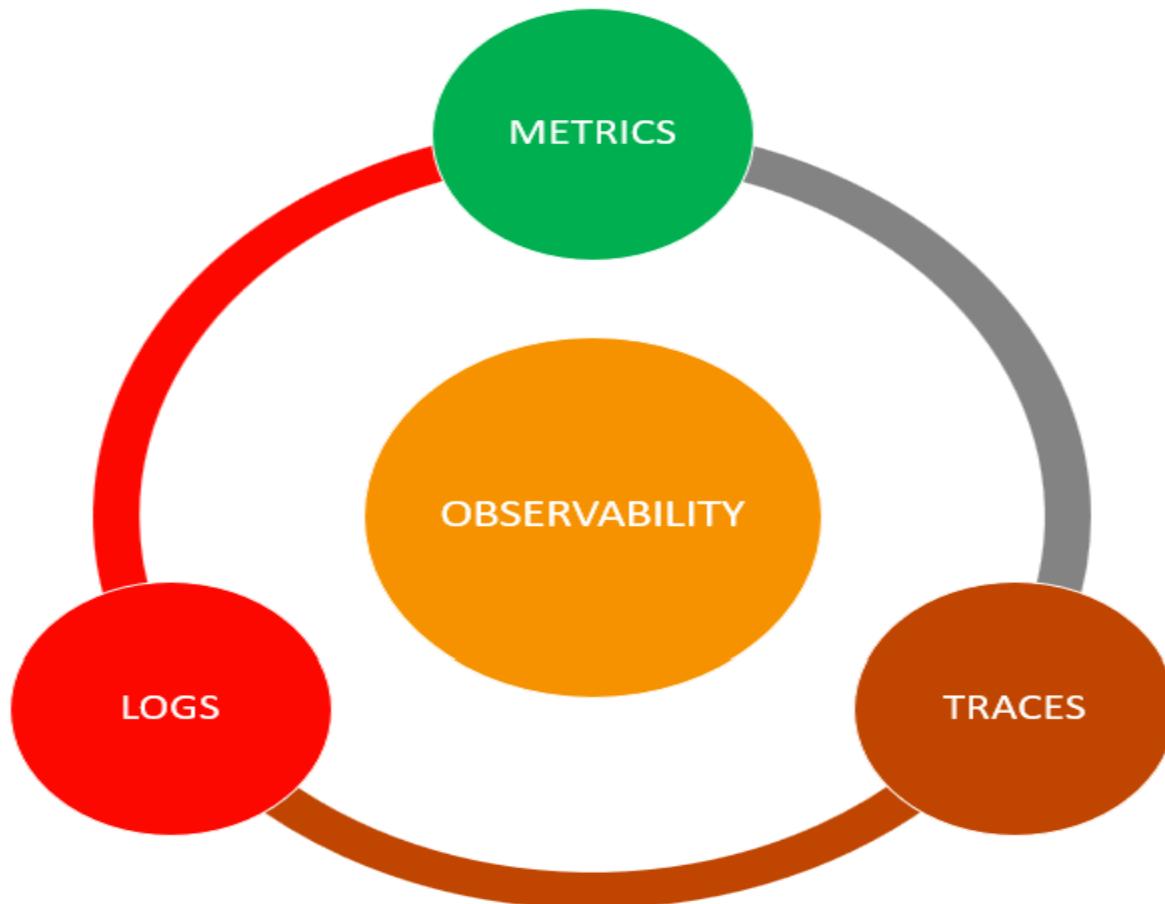


Observability

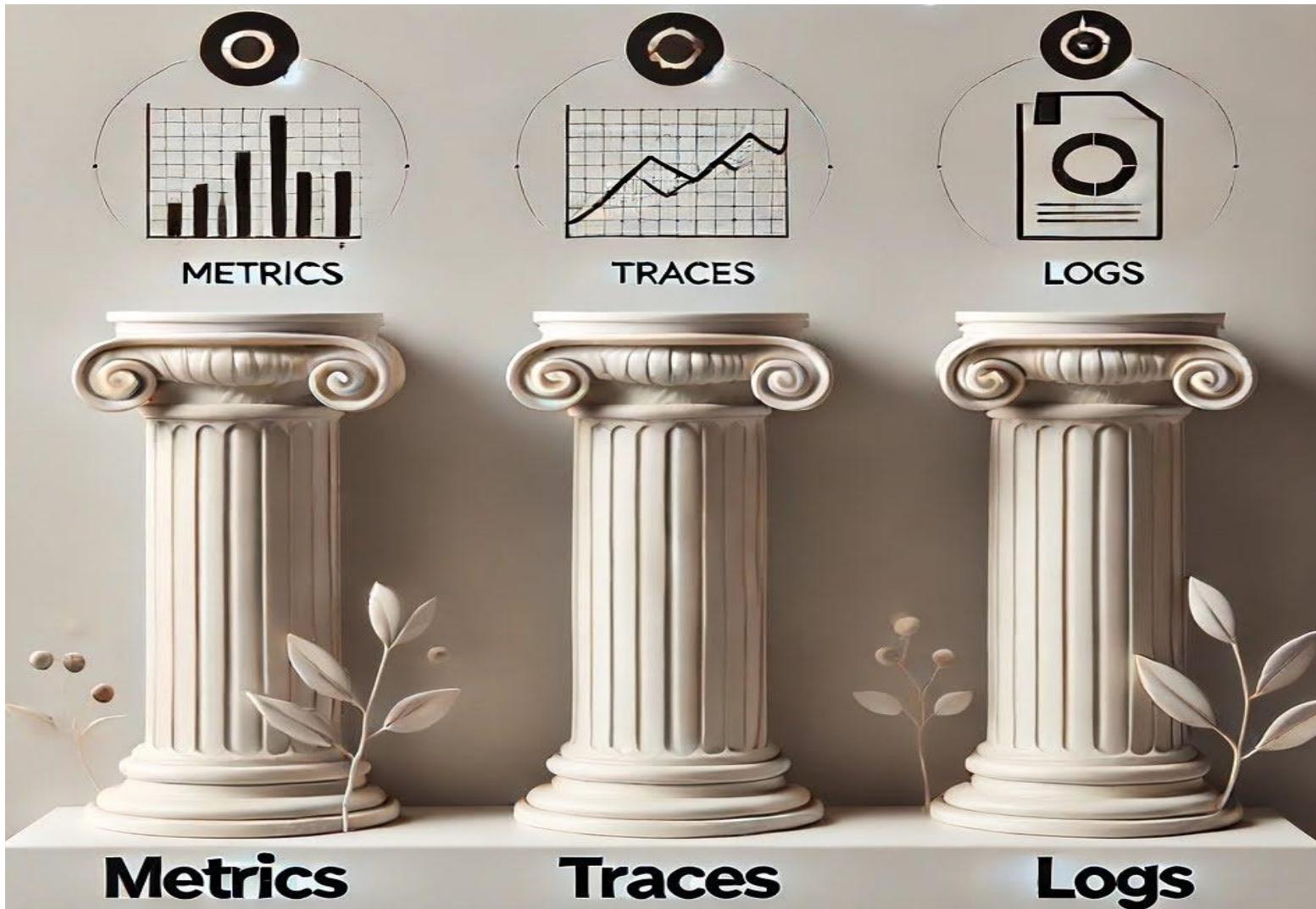
- Observability is the measure of how well the internal states of a system can be inferred from knowledge of its external outputs.
- Often confused with monitoring, observability goes a step further by identifying issues in a system and why those issues occur.
- It's from control theory and has become super crucial in software engineering, especially with the rise of complex, distributed systems and microservices.
- Observability collects data from metrics, logs, and traces, the three pillars, to give a complete view of a system's health and performance.



Observability

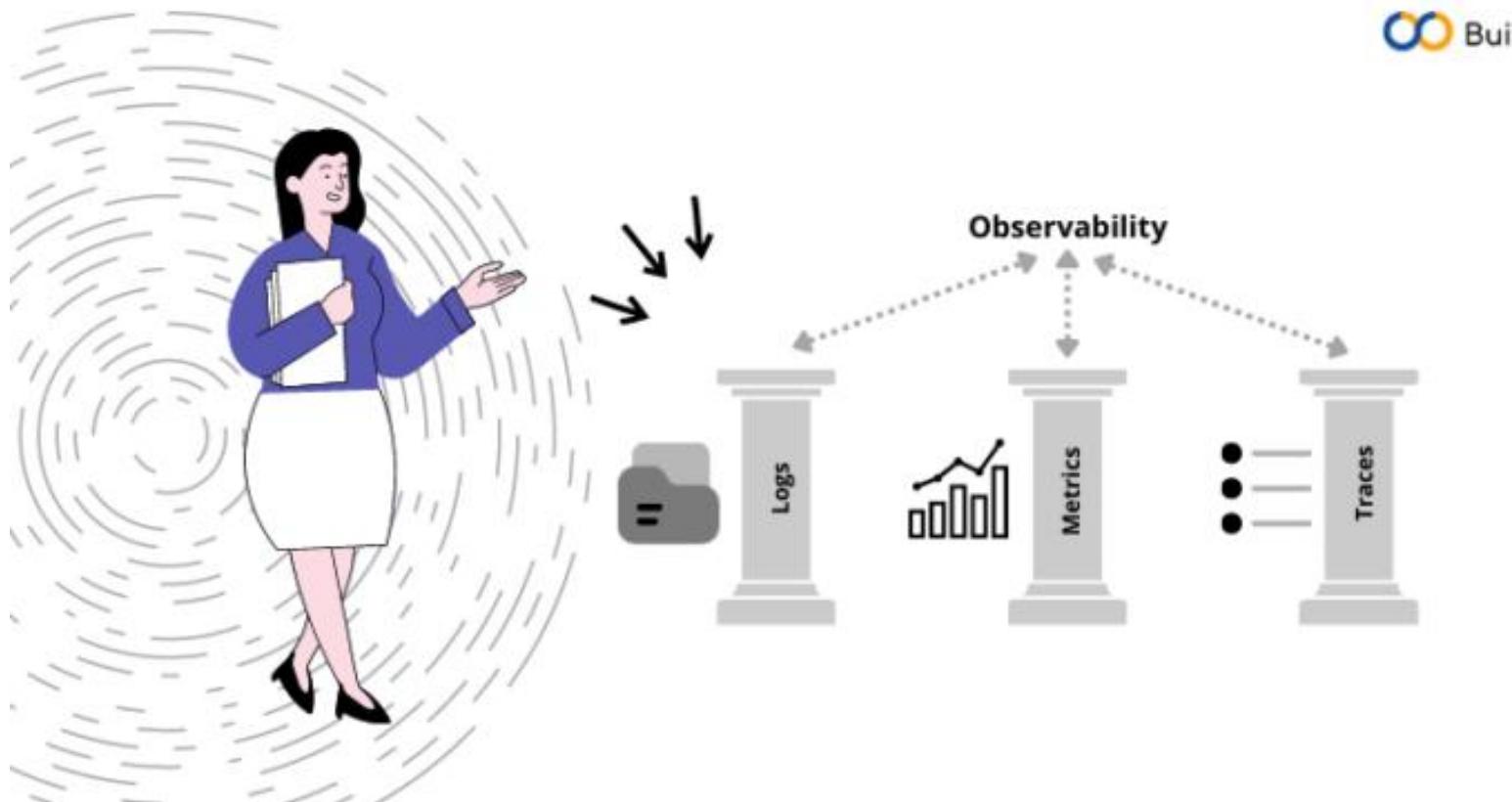


Observability





Observability



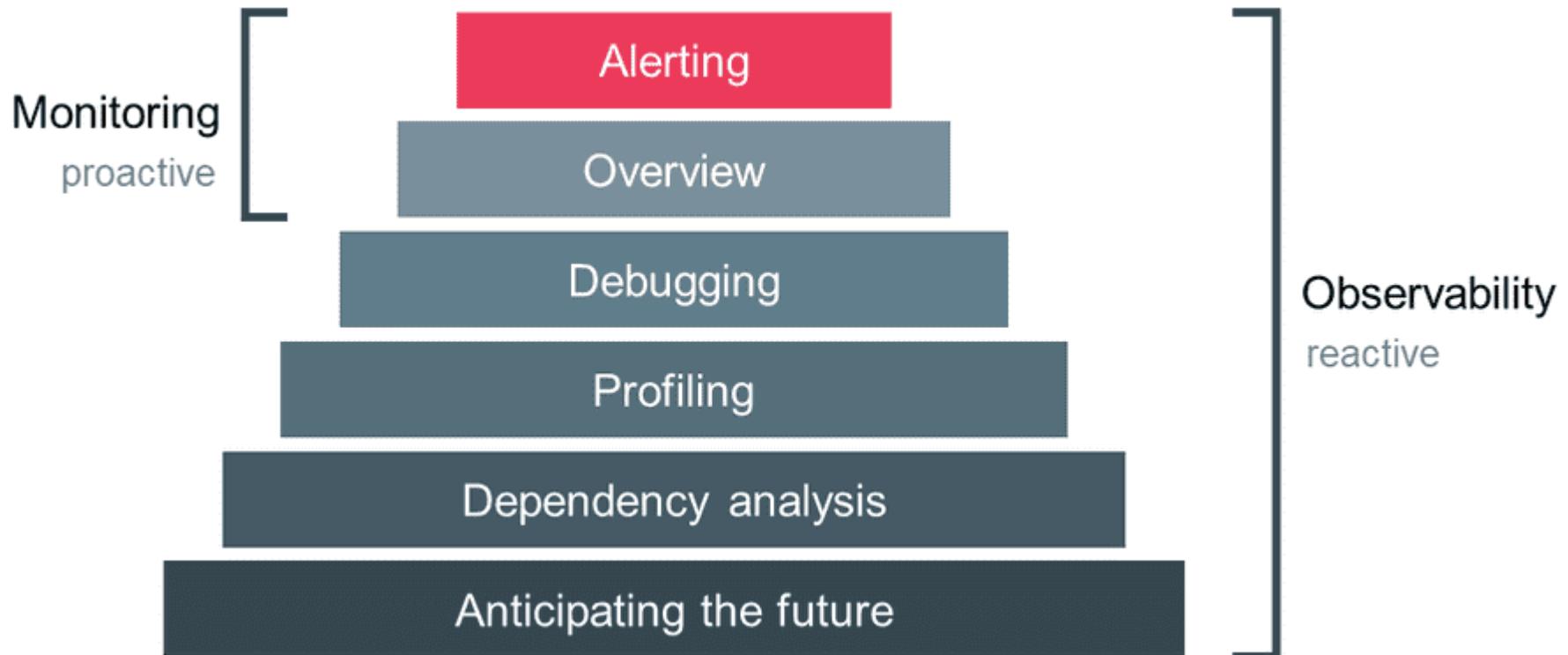


Need for Comprehensive Observability

- Less downtime: Faster detection and fixing means reduced downtime and more customer service.
- Better decisions: With visibility into system performance, management can make better decisions on resource allocation and strategic improvements.
- Develop and run faster: Developers and ops teams can catch bottlenecks or failures before they become big problems.
- Better customer experience: Smoother system operation means less customer friction and better service overall.



Observability vs Monitoring





Observability vs Monitoring

Observability

VS

Monitoring

- Tells you why a system is at fault
- Acts as a knowledge base in defining what to monitor
- Focuses on giving context to the data
- Gives a more complete assessment of the overall environment
- Observability is a traversable map
- It gives you complete Information
- Observability creates the potential to monitor different events

- Notifies you that a system is at fault
- Focuses on monitoring the systems & discovering faults
- Focuses on collecting data
- Focuses on monitoring KPIs
- Monitoring is a single plane
- It gives you limited Information
- Monitoring is the process of using observability

Profiling



APPLICATION PROFILING

- APPLICATION
 - PERFORMANCE
 - PERFORMANCE



OBSERVABILITY

- SYSTEM HEALTH
 - MONITORING
 - REALM HEALTH
 - DISTRIBUTED SYSTEMS



Profiling



Aspect	Application Profiling	Observability
Focus	Code-level performance and resource usage (CPU, memory, etc.)	System-wide behavior and health (logs, metrics, traces)
Purpose	Identify performance bottlenecks in specific functions or code	Provide real-time monitoring and insights into system health
Granularity	Highly granular, focused on individual threads or functions	Broader scope, focusing on entire applications or services
Timing	Typically done offline or in development	Continuous, real-time monitoring in production environments
Data Type	Low-level data such as CPU cycles, memory usage, and I/O	High-level data like logs, metrics, and distributed traces
Use Case	Debugging, performance optimization, resource tuning	Troubleshooting, detecting issues, and ensuring reliability
Tool Examples	Profilers (e.g., CPU profilers, memory profilers)	Monitoring tools (e.g., Prometheus, Grafana, Datadog)
Scope	Specific code segments or modules	Entire application stack or distributed systems
Insight	Provides detailed insights into code behavior	Provides insights into how the system behaves as a whole
Outcome	Identifies inefficiencies at the code level	Monitors and maintains system performance and availability



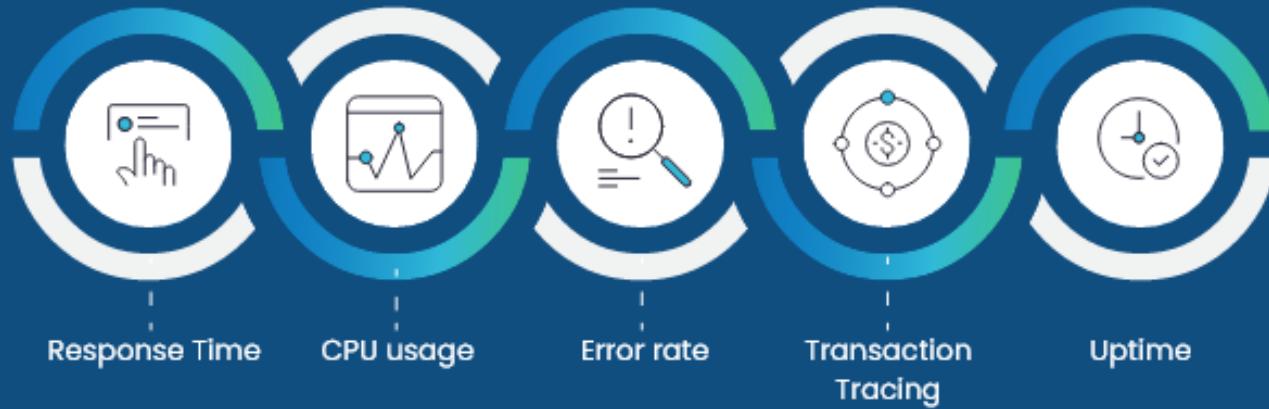
Metrics

- "Metrics" refers to quantifiable measurements used to track and assess the performance or progress of a specific process, activity, or objective.
- Metrics are essential in various fields, including business, technology, healthcare, and education, to provide objective data for decision-making, analysis, and improvement.
- There are many types of metrics, depending on the context.

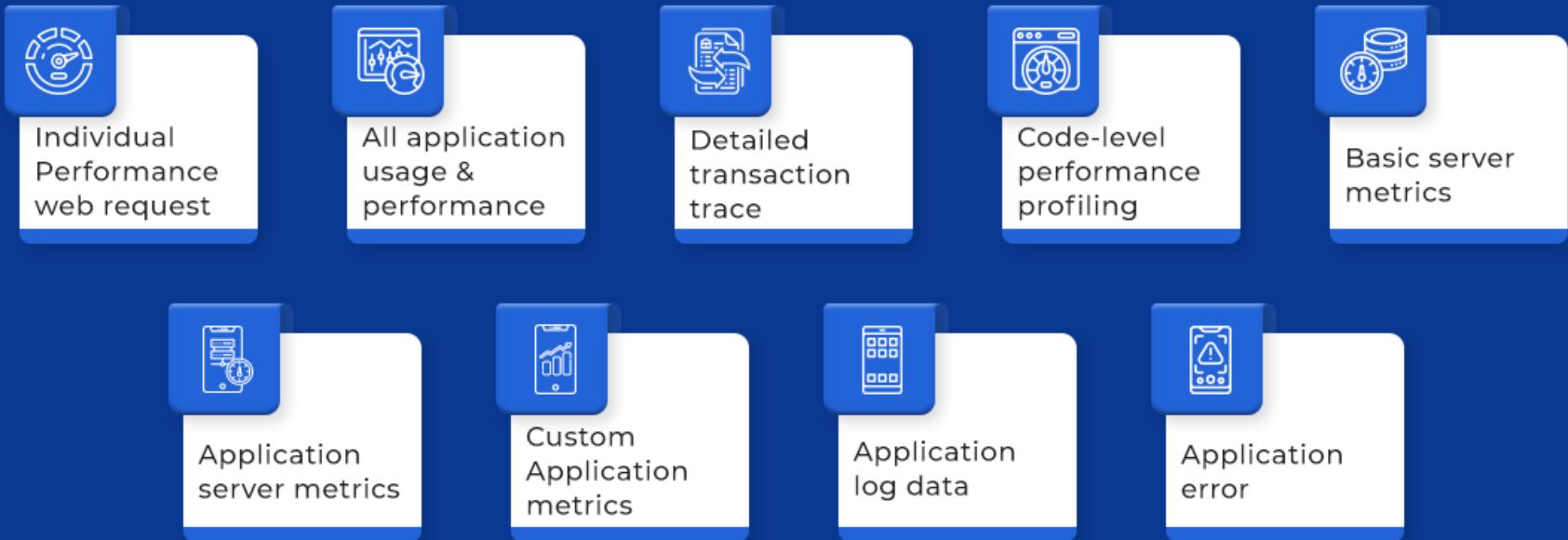
Metrics



What metrics does APM track?



Application Performance Management Solution Includes





Types of Metrics

- **Business Metrics:**
 - **Revenue Growth:** Measures the increase in revenue over a certain period.
 - **Profit Margin:** Percentage of profit generated from total revenue.
 - **Customer Acquisition Cost (CAC):** The cost of acquiring a new customer.
 - **Net Promoter Score (NPS):** Measures customer satisfaction and likelihood to recommend a service/product.
 - **Churn Rate:** The percentage of customers who stop using a service/product.



Types of Metrics

- **Financial Metrics:**
 - **Return on Investment (ROI):** Measures profitability relative to the investment cost.
 - **Gross Profit Margin:** A company's total sales revenue minus its cost of goods sold (COGS).
 - **Debt-to-Equity Ratio:** A measure of a company's financial leverage.



Types of Metrics

- **Project Management Metrics:**
 - **On-time Delivery:** The percentage of projects delivered by the scheduled due date.
 - **Budget Variance:** The difference between the budgeted cost and the actual cost.
 - **Scope Changes:** Number or percentage of changes made to the project scope.



Types of Metrics

- **Technology/Software Development Metrics:**
 - **Velocity:** Measures the amount of work completed during a sprint (agile metric).
 - **Mean Time to Repair (MTTR):** Average time to fix a system after a failure.
 - **Defect Density:** The number of defects found in each amount of code.



Types of Metrics

- **Healthcare Metrics:**
 - **Patient Satisfaction Score:** Measures patient satisfaction with healthcare services.
 - **Average Length of Stay (ALOS):** The average number of days a patient spends in the hospital.
 - **Readmission Rates:** The percentage of patients who return to the hospital after discharge.



Types of Metrics

- **Marketing Metrics:**
 - **Conversion Rate:** The percentage of visitors who take a desired action (e.g., purchasing).
 - **Click-through Rate (CTR):** The percentage of people who clicked on a link after seeing an ad or email.
 - **Cost Per Lead (CPL):** The average cost of acquiring a lead.



Types of Metrics

- **Human Resources Metrics:**
 - **Employee Turnover Rate:** The percentage of employees who leave a company over a given period.
 - **Absenteeism Rate:** Measures employee absences as a percentage of the workforce.
 - **Employee Engagement Score:** A measure of employees' emotional investment in their work.



Importance of Metrics

- Performance Measurement:
 - Metrics allow organizations to gauge how well they are meeting objectives or standards.
 - Decision-Making: With data from metrics, businesses can make informed decisions regarding strategy, resource allocation, and priorities.
 - Benchmarking: Metrics help compare performance over time or against competitors.
 - Continuous Improvement: Metrics enable tracking of progress and identification of areas needing improvement.



Understand logs and events

- Logs and events are essential components in monitoring and analyzing systems, networks, and applications.
- They provide critical insights into the performance, security, and operational status of infrastructure, helping in troubleshooting, auditing, and improving overall system health.
- Understanding the distinction between logs and events and how they are used can help us better interpret system behaviors.

Understand logs and events





Understand logs and events

Aspect	Logs	Events
Definition	A record of system activities captured over time	A specific action or occurrence within the system
Content	Contains detailed information such as timestamps, messages, errors, etc.	Describes a significant change or state in the system, like an alert or notification
Granularity	Typically continuous and detailed, capturing all system activities	Typically discrete, representing a single instance of change
Purpose	Used for tracking, diagnosing, and understanding system behavior over time	Used to trigger alerts, notifications, or actions based on system state
Use Case	Analyzing system health, debugging, and root cause analysis	Monitoring system state, triggering automation, or handling real-time responses
Storage	Stored as log files or in log management systems	Stored in event logs, metrics, or monitoring systems
Example	"2024-10-11 10:45:32: Error connecting to database."	"User login successful" or "Server down at 10:45 AM"
Relation to Monitoring	Provides context for events, giving detailed system history	Indicates when something of importance occurs
Tool Examples	Logstash, Fluentd, Splunk	Prometheus, Nagios, AWS CloudWatch Events
Time Aspect	Often continuous and spans over a long period	Discrete and represents a point-in-time occurrence



Understand logs and events

- Logs:
 - A log is a detailed, time-stamped record of activity generated by systems, applications, devices, or users.
 - Logs capture various types of information and are used for debugging, auditing, monitoring, and analyzing the state of a system.



Understand logs and events

- **Characteristics of Logs:**

1. **Time-stamped:** Logs are usually recorded with the exact time and date when the action or event occurred.
2. **Detailed Information:** Logs typically contain detailed information such as user actions, system processes, requests, errors, and system messages.
3. **Continuous Generation:** Logs are continuously generated by systems, often creating large amounts of data over time.
4. **Text-based:** Most logs are stored as plain text or in structured formats like JSON or XML.
5. **Retained for Auditing:** Logs can be stored for long periods to serve as an audit trail of system activities.



Understand logs and events

- **Types of Logs:**
 - **System Logs:** Captures operating system-related activities such as boot sequences, services starting or stopping, and system errors.
 - **Application Logs:** Records events within an application, such as user requests, database queries, and application errors.
 - **Security Logs:** Includes information about user logins, permissions, access control, and failed login attempts.
 - **Network Logs:** Logs data about network traffic, firewall rules, and packet exchanges.



Understand logs and events

- **Examples of Logs:**
 - Web server access logs capturing incoming HTTP requests.
 - System logs capturing user logins, reboots, or file access.
 - Error logs generated by applications when an error or crash occurs.



Understand logs and events

- **Events:**

- An event is a specific occurrence or action within a system or application, often defined by triggers or conditions.
- Events represent key moments that may require attention, such as a security alert, system crash, or significant change in state.



Understand logs and events

- **Characteristics of Events:**

1. **Discrete Occurrences:** An event is usually tied to a specific action or change, such as a system startup, security breach, or completed transaction.
2. **Higher-Level Information:** Events tend to summarize important activities, unlike logs which may capture more granular data.
3. **Trigger-Based:** Events are often triggered by specific conditions or thresholds, such as a system reaching a performance limit, or a user accessing restricted data.
4. **Alerting and Monitoring:** Events are often tied to monitoring systems that generate alerts, notifications, or reports for administrators.
5. **Structured Data:** Events are often captured in a more structured way, helping systems and humans process them efficiently.



Understand logs and events

- **Examples of Events:**

- A user successfully logs in to a system (authentication event).
- A network intrusion detection system identifies a potential attack (security event).
- A scheduled backup process successfully completes (system event).
- An application generates a "transaction completed" event when a purchase is made.



Understand logs and events

- **Differences Between Logs and Events:**
 - **Granularity:** Logs are more granular and detailed, capturing every action or change in a system, while events highlight more significant occurrences.
 - **Purpose:** Logs are used for detailed analysis, diagnostics, and audit trails, whereas events focus on triggering alerts, monitoring system health, and identifying critical issues.
 - **Data Structure:** Logs are often unstructured or semi-structured (e.g., text files), while events tend to be structured and can be more easily parsed by monitoring tools.
 - **Storage:** Logs are often stored for a long time to track past actions, while events are typically used in real-time for immediate responses or alerts.



Understand logs and events

- **Use Cases of Logs and Events:**
 - **Troubleshooting:** When a system issue occurs, logs help pinpoint the exact sequence of actions that led to the issue.
 - For example, an error log in a web server could reveal why a particular page failed to load.
 - **Security Monitoring:** Security logs and events help detect suspicious behavior such as failed login attempts, unauthorized access, or malware activity.
 - Event monitoring tools can raise an alert if certain thresholds are breached (e.g., multiple failed login attempts).



Understand logs and events

- Performance Analysis:
 - Logs can capture metrics such as CPU usage, memory consumption, and network traffic, helping administrators detect performance bottlenecks or inefficient processes.
 - Compliance Auditing: Logs serve as a record of system activity, which is often required for regulatory compliance.
 - Logs can show who accessed data, when, and what actions were taken.
 - Real-Time Monitoring and Alerting: Event management systems (like SIEM or APM tools) continuously monitor systems for events that may require immediate action, such as security breaches or system outages.



Tracing and Spans

Trace And Span





Tracing and Spans

Aspect	Tracing	Spans
Definition	A method used to track the entire journey of a request as it moves through different services in a distributed system.	A single unit of work within a trace, representing a specific operation or step in the request's journey.
Scope	Captures the flow of a request across multiple systems or services, providing a high-level overview.	Represents a specific operation, such as a function call or database query, within the trace.
Granularity	High-level, focused on the end-to-end lifecycle of the request.	Fine-grained, focused on individual components or steps within the trace.
Purpose	To give visibility into how requests traverse through services, helping to identify bottlenecks, failures, or latency issues.	To break down the trace into specific operations, allowing detailed performance analysis for each step.
Hierarchy	A trace consists of multiple spans that represent the steps involved in processing a request.	A span is a segment of a trace, and spans can be nested or linked to show relationships between operations.

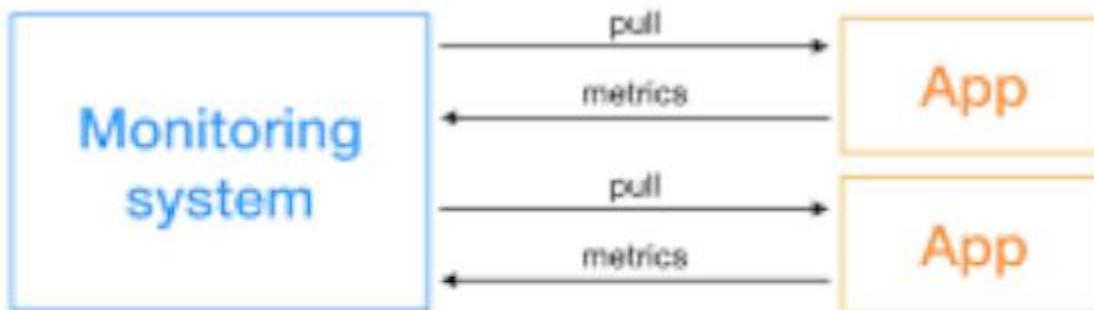


Tracing and Spans

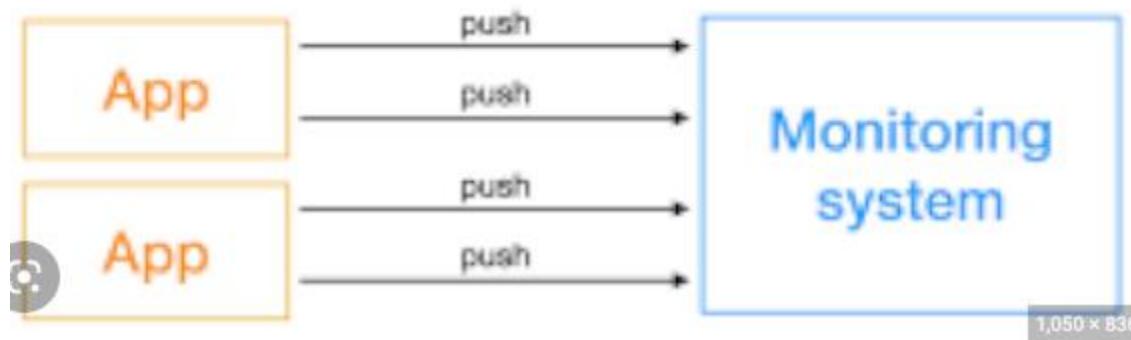
Data Captured	Collects metadata such as trace ID, timestamps, and relationships between spans.	Captures operation-specific details like start/end time, duration, and metadata (tags, logs).
Visualization	Typically visualized as a flow of connected spans, showing the end-to-end path of a request.	Visualized as individual nodes or blocks within the trace, often nested to show dependencies.
Use Case	Helps diagnose latency issues, track request flow, and identify problematic services in distributed architectures.	Helps to pinpoint delays or errors in specific operations within a service.
Example	A trace could show a user's request passing through a web server, an application server, and a database.	A span might represent the database query or the web server processing the request.
Tool Examples	Jaeger, OpenTelemetry, Zipkin	Spans are part of trace data in the same tools (Jaeger, Zipkin).

Push vs Pull

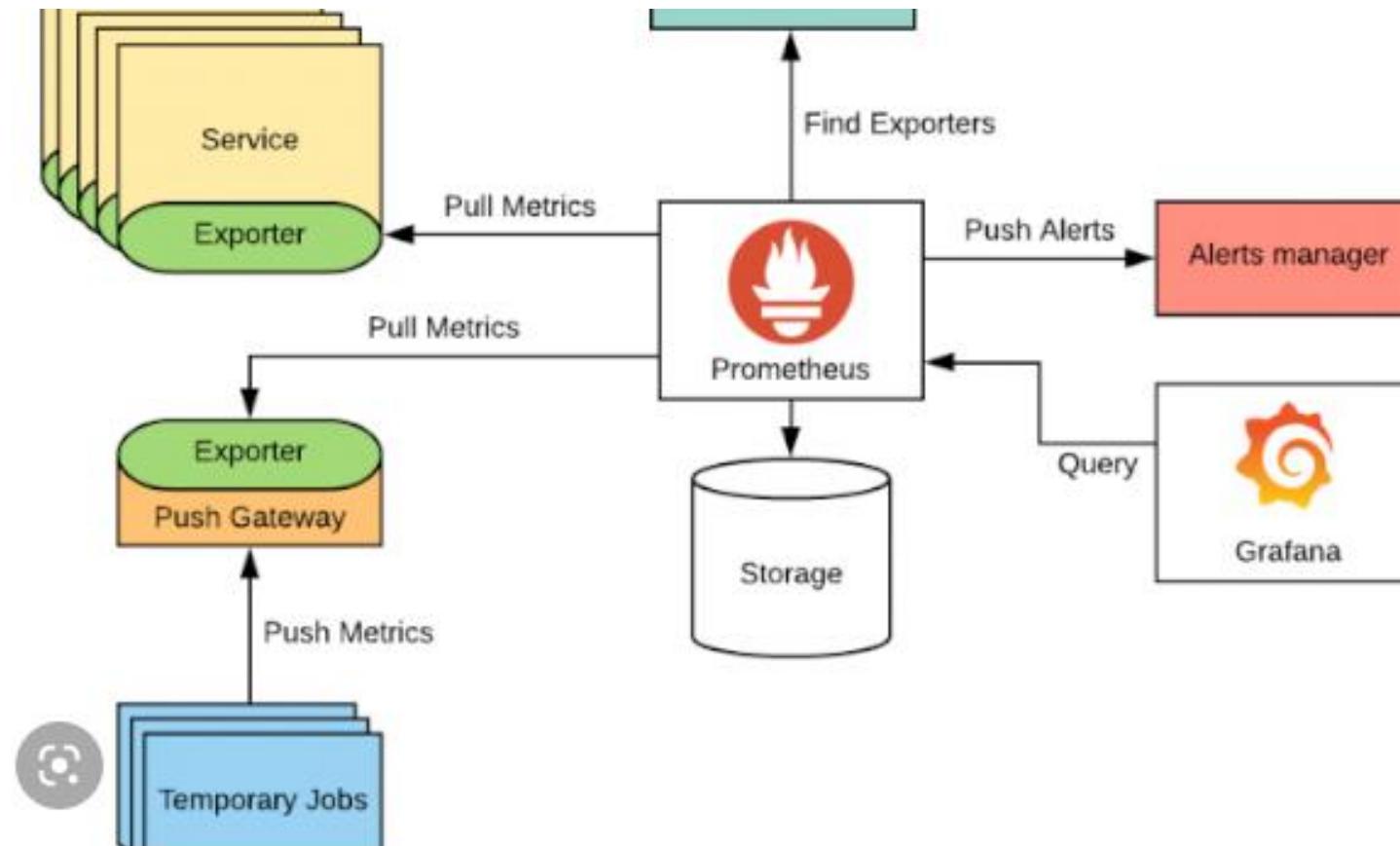
Pull-based system



Push-based monitoring system



Push vs Pull





Push vs Pull

- Pull Model (Prometheus's Native Model)
- In the pull model, Prometheus regularly scrapes metrics from targets (services, servers, or applications) at specified intervals.
- It pulls metrics data by sending HTTP requests to a predefined endpoint (usually /metrics).
- How it works:
 - Prometheus server queries the /metrics endpoint of each target.
 - The target exposes metrics in a specific format that Prometheus understands.
 - Prometheus pulls data at intervals specified in its configuration.



Push vs Pull

- Advantages:
 - Prometheus can control the frequency of data collection.
 - It can handle a dynamic list of targets and retry if one fails to respond.
 - Easier to monitor targets behind load balancers since Prometheus pulls from the available endpoint.
 - The target can remain stateless as it doesn't need to track which systems want its metrics.



Push vs Pull

- Disadvantages:
 - May not work well for short-lived jobs, as these may disappear before Prometheus scrapes their metrics.
 - Not ideal for environments where network access from Prometheus to the target is restricted.



Push vs Pull

- Push Model (Not Native to Prometheus but Achievable)
 - The push model involves the target sending (pushing) its metrics to a central location.
 - Prometheus does not natively support a push mechanism, but the Pushgateway component fills this gap.
 - The Pushgateway acts as an intermediary that receives metrics from applications (especially short-lived jobs) and exposes them to Prometheus.



Push vs Pull

- How it works:
 - Targets push their metrics to a Push gateway.
 - The Push gateway exposes a /metrics endpoint for Prometheus to scrape the pushed data.
 - The Push gateway stores the metrics until Prometheus pulls them.



Push vs Pull

- Advantages:
 - Suitable for short-lived jobs (e.g., batch jobs or CI jobs) that might finish execution before Prometheus can scrape them.
 - Can work in environments where the network setup restricts Prometheus from pulling metrics directly from targets (e.g., firewalls, private networks).
 - Useful when the target cannot keep an HTTP server running to expose metrics continuously.



Push vs Pull

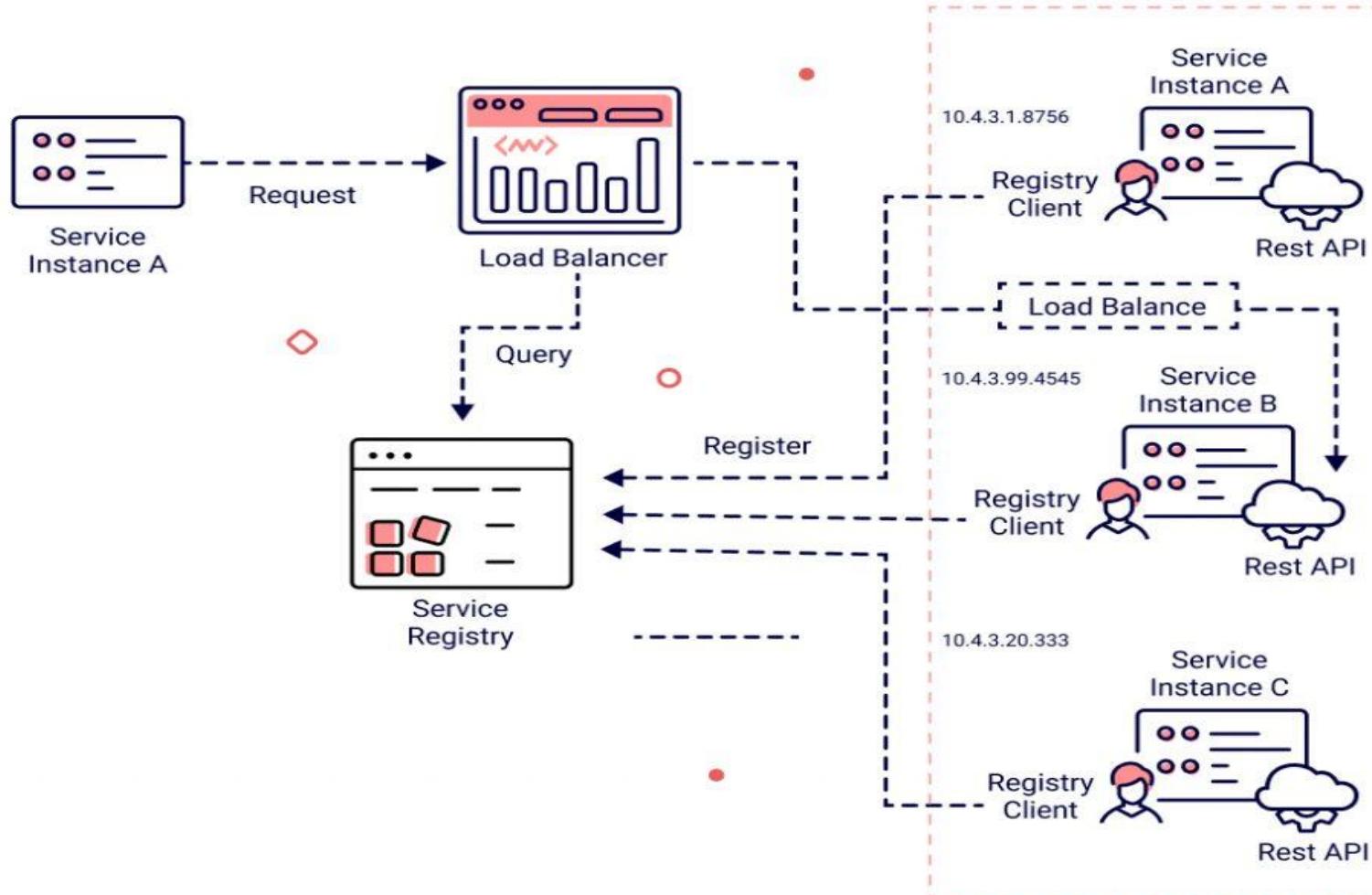
- Disadvantages:
 - Introduces an extra component (Push gateway), which adds complexity.
 - The state of the Push gateway must be managed, as it retains data until explicitly deleted.
 - Push gateway is not designed for continuous metrics (e.g., long-lived services); it's primarily intended for short-lived jobs.



Service Discovery

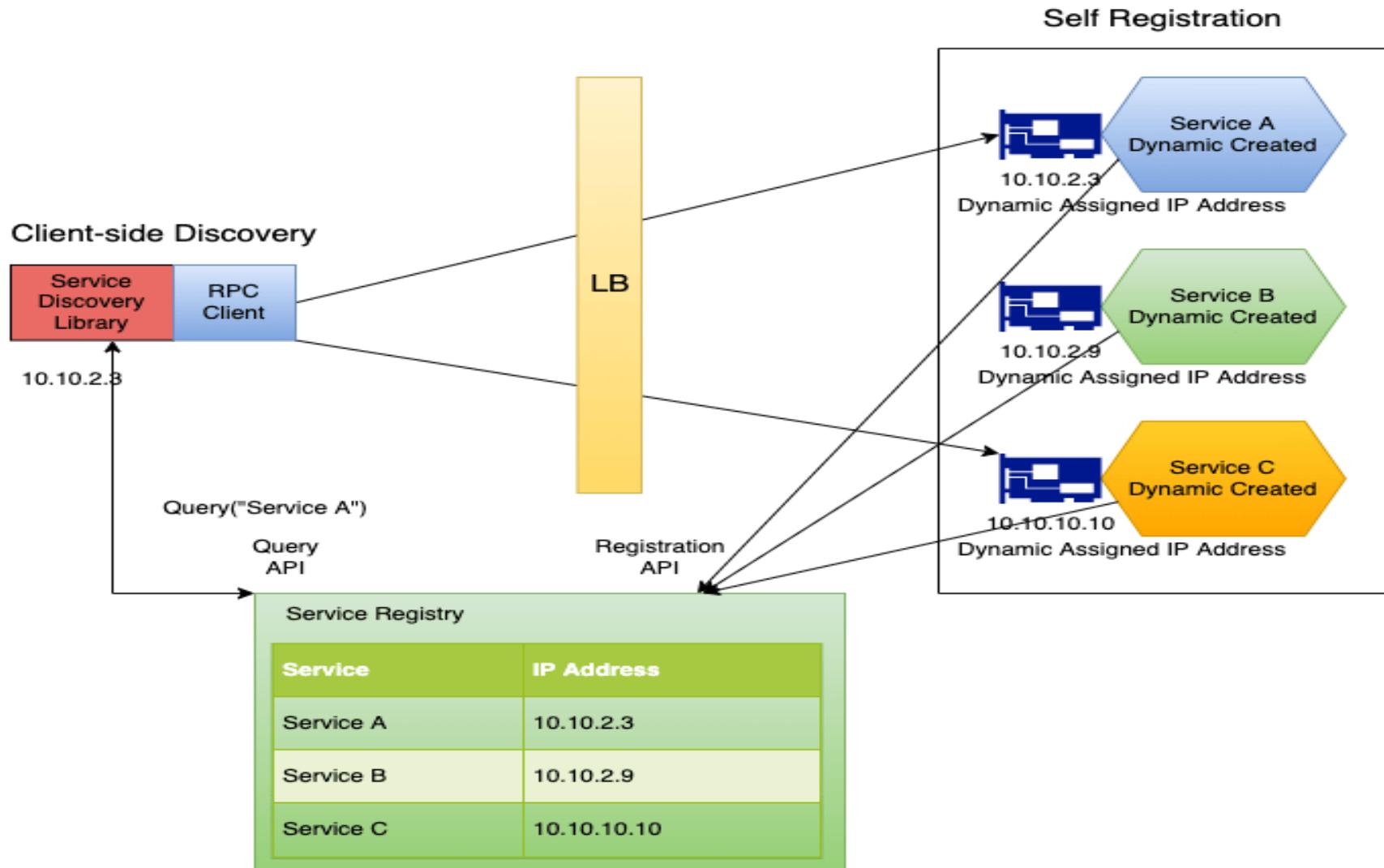
- Service discovery is a key concept in microservices and distributed systems.
- It refers to the process of automatically detecting and connecting microservices in a network.
- In these environments, services communicate with each other, and their locations (IP addresses, ports) may change dynamically due to scaling, failures, or deployments.
- Service discovery helps maintain seamless communication between services despite these changes.

Service Discovery





Service Discovery





Service Discovery

- 1. Client-side discovery:
 - In client-side discovery, the client (the service that wants to make a request) is responsible for determining the location of service instances.
 - A service registry contains the addresses of all available services.
 - The client queries the registry to find a valid service instance and then sends a request directly to the selected instance.



Service Discovery

- **Components:**
 - **Service Registry:** A centralized database that tracks all instances of services.
 - **Client:** The service that queries the service registry and selects a service instance.
- **Examples:**
 - Netflix Eureka (client-side service discovery)
 - Consul (can be used for client-side)



Service Discovery

- **Server-side discovery:**

- In server-side discovery, the client sends a request to a load balancer, which in turn queries the service registry and forwards the request to an appropriate service instance.
- The client is unaware of where the actual service instance is located; it simply sends the request to the load balancer.



Service Discovery

- **Components:**
 - **Service Registry:** Stores the locations of services.
 - **Load Balancer/Service Proxy:** Acts as an intermediary between the client and service instances.
- **Examples:**
 - AWS Elastic Load Balancing (ELB)
 - Kubernetes (through the use of DNS or environment variables)



Service Discovery

- **Common Technologies for Service Discovery:**
 - **Eureka:** A service discovery tool developed by Netflix for client-side discovery.
 - **Consul:** A service mesh and service discovery tool from HashiCorp that supports both client-side and server-side discovery.
 - **Kubernetes:** Uses internal DNS or environment variables for service discovery within a cluster.
 - **Zookeeper:** Often used with Apache frameworks for service discovery.



Service Discovery

- **How it Works:**
 - **Service Registration:** When a new service instance starts, it registers itself with the service registry, providing details like IP address, port, and metadata.
 - **Service Deregistration:** When a service instance stops, it deregisters itself from the registry to avoid sending traffic to unavailable instances.
 - **Health Checks:** The registry may periodically perform health checks to ensure that the registered services are available and healthy.



Service Discovery

- **Benefits of Service Discovery:**
 - **Scalability:** Helps services find and communicate with each other dynamically, supporting high levels of scaling.
 - **Resilience:** Allows systems to adapt to changes in the environment, such as instances going down or new ones being added.
 - **Flexibility:** Can support different deployment environments (cloud, containers, on-premises).



SLA vs SLO vs SLI

SLA



SERVICE LEVEL AGREEMENT

the agreement you make with your clients or users

SLOs



SERVICE LEVEL OBJECTIVES

the objectives your team must hit to meet that agreement

SLIs



SERVICE LEVEL INDICATORS

the real numbers on your performance



Basics of SLOs, SLAs, and SLIs

- **SLA (Service Level Agreement):**
 - An SLA is a formal agreement between a service provider and a customer.
 - It defines the specific measurable service performance standards that the provider is obligated to meet.
 - SLAs are typically legally binding and include consequences (such as penalties or service credits) if the service provider fails to meet the agreed-upon standards.



Basics of SLOs, SLAs, and SLIs

- **Key Points:**

- **Purpose:** Sets expectations for service performance and outlines responsibilities between the provider and customer.
- **Enforceability:** Often legally binding with penalties for non-compliance.
- **Example:** "The service provider guarantees 99.9% uptime over a month. If this is not met, the customer will receive a 5% refund of the monthly service fee."



Basics of SLOs, SLAs, and SLIs

- **SLO (Service Level Objective):**
 - An SLO is a specific, measurable goal that defines the acceptable performance level of a service.
 - SLOs are part of the SLA but are less formal than the SLA itself.
 - They act as internal targets for the service provider, helping to ensure that the SLAs are met.
 - The SLO is used to define performance objectives like availability, latency, or response times.



Basics of SLOs, SLAs, and SLIs

- **Key Points:**

- **Purpose:** Sets measurable objectives that help the service provider track and maintain service levels.
- **Measurable:** Clearly defined metrics such as uptime percentage, error rates, or latency thresholds.
- **Example:** "The system should have an uptime of 99.9% each month."



Basics of SLOs, SLAs, and SLIs

- **SLI (Service Level Indicator):**
 - An SLI is the actual measurement or metric that indicates how well a service is performing against the SLOs.
 - SLIs are quantitative measurements, typically expressed as percentages or time units, that track specific characteristics of a service (such as availability, latency, throughput, or error rate).
 - SLIs provide the data that allows a provider to understand whether they are meeting their SLOs and SLAs.



Basics of SLOs, SLAs, and SLIs

- **Key Points:**

- **Purpose:** Provides real-time metrics or measurements of service performance.
- **Monitoring:** Often collected through monitoring tools and compared against SLOs.
- **Example:** "The service achieved 99.95% uptime last month" (where uptime is the SLI).

Prometheus fundamentals



Prometheus

- Prometheus is an open-source technology designed to provide monitoring and alerting functionality for cloud-native environments, including Kubernetes.
- It can collect and store metrics as time-series data, recording information with a timestamp.
- It can also collect and record labels, which are optional key-value pairs.



Prometheus Features

- Multidimensional data model – Using time-series data, which is identified by metric name and key-value pairs.
- PromQL – A flexible querying language that can leverage the multi-dimensional data model.
- No reliance on distributed storage – All single server nodes remain autonomous.
- Pull model – Prometheus can collect time-series data by actively “pulling” data over HTTP.
- Pushing time-series data – Available through the use of an intermediary gateway.
- Monitoring target discovery – Available through static configuration or service discovery.
- Visualization – Prometheus offers multiple types of graphs and dashboards.



Micrometer Prometheus

- Micrometer provides a simple facade over the instrumentation clients for the most popular monitoring systems.
- It allows us to instrument JVM-based application code without vendor lock-in.
- Micrometer is an open-source project and provides a metric facade that exposes metric data in a vendor-neutral format that a monitoring system can understand.



Monitoring Systems

These monitoring systems are supported:

- AppOptics
- Azure Monitor
- Netflix Atlas
- CloudWatch
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx/Telegraf
- JMX
- KairosDB
- New Relic
- Prometheus
- SignalFx
- Google Stackdriver
- StatsD
- Wavefront



How Does Prometheus Monitoring Work?

- To get metrics, Prometheus requires an exposed HTTP endpoint.
- Once an endpoint is available, Prometheus can start scraping numerical data, capture it as a time series, and store it in a local database suited to time-series data.
- Prometheus can also be integrated with remote storage repositories.



How Does Prometheus Monitoring Work?

- Users can leverage queries to create temporary time series from the source.
- These series are defined by metric names and labels.
- Queries are written in PromQL, a unique language that allows users to choose and aggregate time-series data in real time.
- PromQL can also help us establish alert conditions, resulting in notifications to external systems like email, PagerDuty, or Slack.



How Does Prometheus Monitoring Work?

- Prometheus can display collected data in tabular or graph form, shown in its web-based user interface.
- We can also use APIs to integrate with third-party visualization solutions like Grafana.



What Can You Monitor with Prometheus?

- Prometheus is a versatile monitoring tool, which you can use to monitor a variety of infrastructure and application metrics.
- Use Cases
 - Service Metrics
 - Host Metrics
 - Website Uptime/Up Status
 - Cronjobs



Service Metrics

- Prometheus is typically used to collect numeric metrics from services that run 24/7.
- It allows metric data to be accessed via HTTP endpoints.
- This can be done manually or with various client libraries.
- Prometheus exposes data using a simple format, with a new line for each metric, separated with line feed characters.
- The file is published on an HTTP server that Prometheus can query and scrape metrics from based on the specified path, port, and hostname.
- Prometheus can also be used for distributed services, which are run on multiple hosts.
- Each instance publishes its own metrics and has a name that Prometheus can distinguish.



Host Metrics

- We can monitor the operating system to identify when a server's hard disk is full or if a server operates constantly at 100% CPU.
- We can install a special exporter on the host to collect the operating system information and publish it to an HTTP-reachable location.



Website Uptime/Up Status

- Prometheus doesn't usually monitor website status, but we can use a black box exporter to enable this.
- We specify the target URL to query an endpoint.
- It performs an uptime check to receive information such as the website's response time.
- We define the hosts to be queried in the `prometheus.yml` configuration file, using `relabel_configs` to ensure Prometheus uses the black box exporter.



Cronjobs

- To check if a cronjob is running at the specified intervals, we can use the Push Gateway to display metrics to Prometheus through an HTTP endpoint.
- We can push the timestamp of the last successful job (i.e. a backup job) to the Gateway and compare it with the current time in Prometheus.
- If the time exceeds the specified threshold, the monitor times out and triggers an alert.



Prometheus Metric Types

- The client libraries of Prometheus offer three core types of metrics.
- However, the Prometheus server does not currently save these metrics as different data types.
- Instead, it flattens all information into an untyped time series.



Prometheus Metric Types

- Counter
 - This is a cumulative metric. It represents a single monotonically-increasing counter, and its value can either increase or be reset to zero on restart.
- Gauge
 - This metric represents one numerical value, which can arbitrarily go down and up.
 - A gauge is often used to measure values like current memory usage or temperatures.
- Histogram
 - A histogram samples observations, such as request durations or response sizes.
 - It then counts the observations in a configurable bucket.
 - A histogram can also provide a total sum of all the observed values.



Prometheus Architecture

- Prometheus follows a **pull-based model** where it scrapes metrics from monitored targets at specified intervals.
- It uses a multi-dimensional data model for storing time-series data and provides a flexible query language called **PromQL** to retrieve and analyze this data.
- Prometheus is a standalone service that doesn't require external storage.



Prometheus Architecture

- **Key Components of Prometheus Architecture:**

1. Prometheus Server:

1. The Prometheus server is at the core of the system architecture. It is responsible for:
 1. **Data Collection:** Scraping metrics from target services at defined intervals.
 2. **Data Storage:** Storing scraped time-series data in a local time-series database (TSDB).
 3. **Querying:** Processing queries written in PromQL to retrieve and analyze the collected data.



Prometheus Architecture

- **Targets (Instrumented Services):**
 - Targets are the systems and applications being monitored by Prometheus.
 - These targets expose metrics via an HTTP endpoint (typically /metrics), which Prometheus scrapes.
 - Prometheus interacts with these services by pulling metrics from them at regular intervals.
 - Targets can be instrumented applications, exporters, or third-party systems like databases or cloud services.



Prometheus Architecture

- **Exporters:**
- Exporters are specialized services that expose metrics from third-party systems or non-instrumented services in a Prometheus-compatible format.
- Common exporters:
 - **Node Exporter:** Exposes system-level metrics (CPU, memory, disk, etc.).
 - **Blackbox Exporter:** Probes endpoints (HTTP, DNS, ICMP) to check for uptime and reachability.
 - **Database Exporters (e.g., MySQL, PostgreSQL):** Export database-specific metrics.



Prometheus Architecture

- **Service Discovery:**
- Prometheus supports dynamic service discovery, which allows it to automatically find and scrape targets in dynamic environments (e.g., cloud, Kubernetes).
- It supports a variety of service discovery mechanisms:
 - Kubernetes
 - Consul
 - EC2 instances
 - DNS-based discovery
- Service discovery helps in automatically adapting to infrastructure changes, such as when services scale up or down.



Prometheus Architecture

- **Time-Series Database (TSDB):**
 - Prometheus stores all metrics as time-series data, identified by a unique metric name and label set.
 - The time-series data is stored locally on disk and compressed to optimize space.
 - Data is kept for a configurable retention period, after which it is deleted.



Prometheus Architecture

- **Alert manager:**
- **Alert manager** handles alerts generated by Prometheus when a rule condition is met (e.g., high CPU usage).
- Prometheus evaluates alerting rules against the data and pushes alerts to the Alertmanager.
- **Alert manager's Role:**
 - **Alert Deduplication:** Prevents repeated alerts for the same issue.
 - **Grouping:** Groups related alerts to reduce noise.
 - **Routing:** Sends alerts to the appropriate receiver (email, Slack, PagerDuty, etc.).
 - **Silencing:** Temporarily suppresses alerts during planned maintenance.



Prometheus Architecture

- **Prometheus Client Libraries:**

- Prometheus provides client libraries for different programming languages (Go, Python, Java, Ruby, etc.) to enable custom metrics instrumentation in applications.
- Developers can use these libraries to instrument their code and expose custom application-specific metrics.



Prometheus Architecture

- **PromQL (Prometheus Query Language):**
 - Prometheus comes with its own powerful query language, PromQL, which allows users to select and aggregate time-series data.
 - PromQL supports complex operations like rate calculations, aggregations, and joins across metrics.

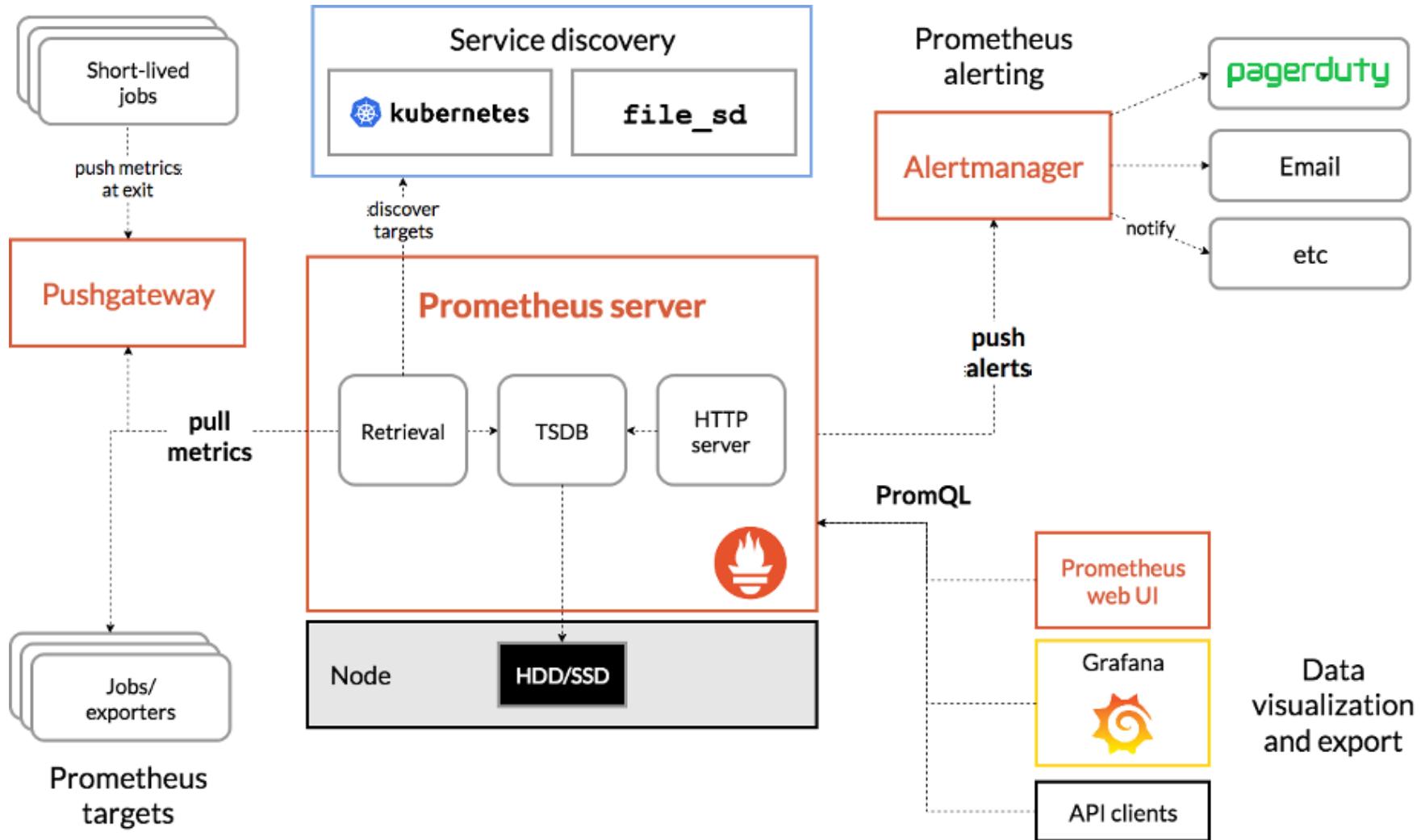


Prometheus Architecture

1. Pushgateway (Optional):

1. Prometheus uses a pull model, but sometimes systems may not be suited to be scraped (e.g., batch jobs or short-lived processes).
2. The **Pushgateway** allows these systems to push metrics to Prometheus. It acts as an intermediary for these metrics, storing them until Prometheus pulls them.

Prometheus Component Architecture





Prometheus Server

- The Prometheus server handles the scraping and storing of metrics.
- The server manages scheduling of monitoring jobs – querying data sources (called “instances”) at a predefined polling frequency.
- Monitoring jobs are configured via one or more “scrape config” directives, managed via a YAML configuration file.
- It can be live-reloaded using a SIGHUP or the Management API.



Prometheus Server

- Prometheus relies heavily on various service discovery (SD) mechanisms to identify targets to scrape.
- These service discovery integrations range from generic interfaces like a file-based service discovery that custom SD implementations can leverage by managing a JSON or YAML file containing a list of targets.



Prometheus Server

- Prometheus also provides number of platform-specific SD implementations, including: Kubernetes, AWS EC2, Azure, GCE, Docker Swarm, OpenStack, and more.
- These generally integrate with the corresponding APIs to query the platform for targets running applications and exporters that can be scraped by Prometheus.



Prometheus Data Flow

- Scraping:
 - Prometheus scrapes metrics from the targets (applications, services, exporters) at a configured interval via HTTP. Each target exposes an endpoint that Prometheus polls for metrics.
- Storing:
 - Scrapped metrics are stored in Prometheus's internal time-series database.
 - Each metric is stored as a time-series, identified by a combination of a metric name and key-value labels.
- Example metric format:

```
http_requests_total{method="GET", handler="/api"} 12893
```



Prometheus Data Flow

- Querying:
 - Users or external systems (like dashboards) can query the stored time-series data using PromQL.
 - Queries can be used to retrieve data for monitoring, debugging, or performance analysis.
- Alerting:
 - Prometheus evaluates alerting rules against the stored data at regular intervals.
 - When an alert condition is met (e.g., CPU usage > 90% for 5 minutes), an alert is triggered and sent to the Alert manager.
 - Alert manager processes and routes the alerts to defined destinations like Slack, PagerDuty, or email.

Key Features and Benefits of Prometheus Architecture



- Pull-based Model:
 - Prometheus's pull model allows it to periodically scrape metrics from targets, making it more efficient and flexible in discovering new services or handling dynamic infrastructure.
- Time-Series Data Storage:
 - Prometheus uses a highly optimized time-series database, enabling efficient storage and querying of millions of metrics.
- Multi-dimensional Data Model:
 - Prometheus uses labels to add dimensions to metrics, making it easier to filter and group metrics based on these labels.

Key Features and Benefits of Prometheus Architecture



- PromQL:
 - The powerful query language enables deep insights into system behavior through metrics aggregation, filtering, and real-time analysis.
- Service Discovery:
 - Dynamic environments like Kubernetes benefit from Prometheus's ability to auto-discover new targets, making it a natural choice for cloud-native monitoring.
- Self-Contained:
 - Prometheus operates independently, without external dependencies, and can be scaled by federating multiple Prometheus servers.



Integration Systems

- Grafana:
 - Prometheus is often integrated with Grafana for visualizing metrics using dashboards.
- Thanos/Cortex:
 - For long-term storage and scalability, Prometheus can be extended with projects like Thanos or Cortex to store metrics across multiple Prometheus instances and enable long-term storage and queries.



Push gateway send metrics - Bashscript

- echo "example_metric 123" | curl --data-binary @- http://localhost:9091/metrics/job/my_job

```
MINGW64:/c/Users/Dell
DeLL@DESKTOP-B08BAAN MINGW64 ~
$ echo "example_metric 123" | curl --data-binary @- http://localhost:9091/metrics/job/my_job
% Total    % Received % Xferd  Average Speed   Time      Time      Current
          Dload  Upload   Total   Spent    Left  Speed
100      19      0      0  100      19       0    781 --:--:-- --:--:-- --:--:--  950

DeLL@DESKTOP-B08BAAN MINGW64 ~
$
```

localhost:9091/#

Pushgateway Metrics Status Help

job="my_job"

Delete Group

example_metric UNTYPED last pushed: 2024-10-21T13:15:38Z

push_failure_time_seconds Last Unix time when changing this group in the Pushgateway failed. GAUGE last pushed: 2024-10-21T13:15:38Z

push_time_seconds Last Unix time when changing this group in the Pushgateway succeeded. GAUGE last pushed: 2024-10-21T13:15:38Z



Push gateway send metrics - Bashscript

Administrator: Command Prompt

```
F:\Local disk\prometheus\PrometheusGrafana_Latest\prometheushelm\cronjobs>schtasks /create /sc minute /mo 1 /tn "MyCronJob" /tr "F:\Local disk\prometheus\PrometheusGrafana_Latest\prometheushelm\cronjobs\job.bat" /st 00:00  
SUCCESS: The scheduled task "MyCronJob" has successfully been created.
```

```
F:\Local disk\prometheus\PrometheusGrafana_Latest\prometheushelm\cronjobs>
```

Push gateway send metrics – Spring boot scheduler



Pushgateway Metrics Status Help

<input type="checkbox"/> job="my_job"	Delete Group
<input type="checkbox"/> job="spring_boot_job"	Delete Group
<input type="checkbox"/> job="spring_boot_job216"	Delete Group
<input type="checkbox"/> job="spring_boot_job543"	Delete Group
<input type="checkbox"/> job="spring_boot_job72"	Delete Group
<input type="checkbox"/> job="spring_boot_job97"	Delete Group
<input type="checkbox"/> job="spring_boot_job998"	Delete Group

Push gateway send metrics – Spring boot scheduler



```
// This method simulates a job that runs every 15 seconds
@scheduled(cron = "0 * * * *")
public void pushMetrics() {
    // Record the start time
    long startTime = System.currentTimeMillis();

    // Simulate a job (you can replace this with actual logic)
    performJob();

    // Record the end time and calculate the job duration
    long duration = (System.currentTimeMillis() - startTime) / 1000;
    jobDurationGauge.set(duration); // Set the gauge with job duration

    // Labels for the job (you can add additional labels as needed)
    Map<String, String> groupingKey = new HashMap<>();
    groupingKey.put("job", "spring_boot_job"+new Random().nextInt(1, 1000));

    try {
        // Push the metrics to the Prometheus Pushgateway
        pushGateway.pushAdd(registry, job: "spring_boot_job"+new Random().nextInt(1, 1000), groupingKey);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



1. What is PromQL?

- **PromQL** (Prometheus Query Language) is a **functional, time-series–oriented DSL** (domain-specific language).
- Goal: query, aggregate, and transform time-series stored in the **Prometheus TSDB**.
- Inspired by SQL, but designed for **metrics, not rows**.
- Always returns **time series** (vectors), not tables.



2. Data model theory

- Prometheus organizes all data as **time series**:
- **2.1 Time series**
- Each series = **metric name + set of labels**.
 - Example:
 - `http_requests_total{method="GET", handler="/api", status="200"}`
- Labels = key–value pairs → give **dimensionality**.



2. Data model theory

- **2.2 Sample**
- Each series is a sequence of samples:
(timestamp_ms, float64 value)
- **2.3 Metric types**
- **Counter** → cumulative, monotonic ↑, reset on restart.
(e.g., http_requests_total)
- **Gauge** → instantaneous value that can ↑/↓.
(e.g., node_memory_usage_bytes)
- **Histogram** → series of _bucket counters with le labels +
_sum and _count.
- **Summary** → quantiles precomputed client-side (less used;
hard to aggregate).



Expression types in PromQL

- Every PromQL expression evaluates to one of these **four data types**:
- **Instant vector**
 - Set of time series with a single sample (latest value).
 - Example:
 - `up{job="api"}`
- **Range vector**
 - Set of time series with multiple samples over a time window.
 - Example:
 - `http_requests_total[5m]`



Expression types in PromQL

- Every PromQL expression evaluates to one of these **four data types**:
- **Instant vector**
 - Set of time series with a single sample (latest value).
 - Example:
 - `up{job="api"}`
- **Range vector**
 - Set of time series with multiple samples over a time window.
 - Example:
 - `http_requests_total[5m]`



Expression types in PromQL

- **Scalar**
 - Single floating-point value (e.g., 2, 0.95).
- **String**
 - Rarely used, just literal strings.

4. Selectors (theory)

- **4.1 Instant vector selector**
- `metric_name{label1="x", label2=~"y.*"}`
- Filters which series to include.
- Returns **latest sample** within lookback-delta (default 5m).
- **4.2 Range vector selector**
- `metric_name{label="foo"}[10m]`
- Returns a *window of samples* for each matching series.



4. Selectors (theory)

- **4.3 Offset**
- `rate(http_requests_total[5m] offset 1h)`
- Shifts the evaluation window into the past (good for comparisons).
- **4.4 @ modifier**
- `up @ 1695800000`
- Evaluate at a specific timestamp (useful for debugging historical events).



4. Selectors (theory)

- **4.3 Offset**
- `rate(http_requests_total[5m] offset 1h)`
- Shifts the evaluation window into the past (good for comparisons).
- **4.4 @ modifier**
- `up @ 1695800000`
- Evaluate at a specific timestamp (useful for debugging historical events).



5. Operators (semantics)

- **Arithmetic:** + - * / % ^
- **Comparisons:** == != > < >= <=
 - With bool modifier, return 1/0 instead of filtering.
- **Logical/set:** and, or, unless
- **Important:** PromQL operators are defined **element-wise across series**, requiring label matching rules.



6. Vector matching theory

- When you apply binary operators between two vectors:
- Series are matched by their labels.
- You can control matching with:
 - `on (label1, label2)` → only match on specific labels.
 - `ignoring (labelX)` → ignore those labels in matching.
 - `group_left(...)` or `group_right(...)` → allow 1:N matches and propagate labels.
- This is how PromQL achieves **joins** without being a relational DB.

7. Functions (categories)

- PromQL functions transform vectors:
- **Counter functions**
 - `rate()`, `irate()`, `increase()` → apply to *range vectors of counters*.
- **Gauge functions**
 - `avg_over_time`, `max_over_time`, `min_over_time`,
`quantile_over_time`.
- **Statistical**
 - `stddev_over_time`, `stdvar_over_time`, `predict_linear`,
`holt_winters`.
- **Label manipulation**
 - `label_replace`, `label_join`.
- **Math**
 - `deriv`, `iDelta`.



8. Aggregation theory

- Aggregators reduce across labels:
- sum, avg, count, min, max, stddev, quantile.
- Two modes:
- **by(labels)** → group only on those labels (everything else dropped).
- **without(labels)** → drop those labels, keep rest.



9. Subqueries theory

- Let you run a query **over time**, then re-run another function.
- Syntax:
- `expr[range:step]`
- Example:
- `avg_over_time(rate(http_requests_total[5m])[1h:5m])`
- Think: **window over a derived metric**.



10. Histograms in PromQL

- Histograms in Prometheus → multiple _bucket series:
- Each bucket: cumulative counter of observations $\leq le$.
- To get quantiles:
- `histogram_quantile(0.95, sum by
(le)(rate(request_duration_seconds_bucket[5m])))`
- **Key theory:** must aggregate by `le` before applying `histogram_quantile`, otherwise meaningless.



11. Recording vs alerting rules (why they exist)

- **Recording rules:** precompute heavy queries (store them back as new metrics).
- **Alerting rules:** evaluate PromQL → send alerts when condition holds.
- This decouples **query evaluation cost** from **dashboard latency**.



11. Recording vs alerting rules (why they exist)

- **Recording rules:** precompute heavy queries (store them back as new metrics).
- **Alerting rules:** evaluate PromQL → send alerts when condition holds.
- This decouples **query evaluation cost** from **dashboard latency**.



12. Staleness, gaps & absent theory

- Prometheus marks missing series as **stale**.
- Default lookback-delta = 5m means instant vectors search back 5m for a sample.
- `absent(metric{...})` returns 1 if **no series exist**.
- Very important in alerting (to catch missing exporters).



13. Why PromQL is different from SQL

- **SQL**: set-based, joins, relational algebra.
- **PromQL**: time-series calculus.
 - Values are always floats.
 - "Rows" = label dimensions.
 - "Time" is first-class: every query is **time-aware**.
- Joins restricted to **label matching**, not arbitrary keys.



14. Theoretical strengths

- **Composability:** functions & operators can be nested indefinitely.
- **Dimensionality:** labels allow flexible aggregations.
- **Declarative:** you state *what you want*, Prometheus computes.



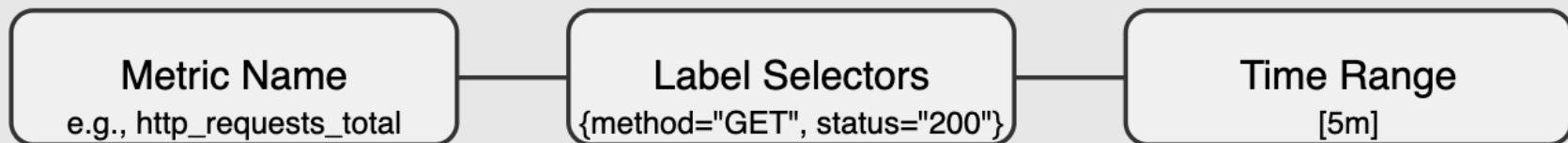
15. Limitations (theory)

- No arbitrary joins (restricted to label-based matches).
- No strings/text search (numeric only).
- High cardinality can explode memory.
- Designed for metrics, not logs/events/traces.



Core Concepts in PromQL

Anatomy of a Basic PromQL Query



Complete Query Example

```
http_requests_total{method="GET", status="200"}[5m]
```

This query selects the 'http_requests_total' metric for GET requests with status 200 over the last 5 minutes



Selecting Data

- The very basic beginnings of any query language is being able to select data from your metrics collection.
- Metric name: Querying for all time series data collected for that metric.
- Label: Using one or more assigned labels filters metric output.
- Timestamp: Fixes a query single moment in time of your choosing.
- Range of time: Setting a query to select over a certain period.
- Time-shifted data: Setting a query to select over a period while adding an offset from the time of query execution.



Selecting Data

- **1. Metric Names**
- Every query starts with a **metric name**.
Examples:
 - up
 - http_requests_total
 - node_memory_Active_bytes
 - This retrieves **all time series** for that metric.

Selecting Data

- **2. Label Selectors**
- Labels refine which series you want.
Syntax:
- `{label="value"}` Exact match
- `{label!="value"}` Exclude match
- `{label=~"regex"}` Regex match
- `{label!~"regex"}` Regex exclude



Selecting Data

- Examples:
- # All HTTP requests with status 500
- http_requests_total{status="500"}
- # All GET requests except to /health
- http_requests_total{method="GET", handler!="health"}
- # Match multiple statuses using regex
- http_requests_total{status=~"4..|5.."}



Selecting Data

- Examples:
- # All HTTP requests with status 500
- http_requests_total{status="500"}
- # All GET requests except to /health
- http_requests_total{method="GET", handler!="health"}
- # Match multiple statuses using regex
- http_requests_total{status=~"4..|5.."}



Selecting Data

- **3. Instant Vector Selectors**
- Select the **latest value** of the matching series.
- This is the **default** query type if you just specify metric + labels.
- Example:
- `up{job="api"}`
- → “Are API targets up right now?”



Selecting Data

- **4. Range Vector Selectors**
- Select **all values within a time window**.
- Syntax:
- `metric_name{...}[range]`
- Common ranges: [5m], [1h], [24h].
- Example:
- `http_requests_total{job="api"}[5m]`
- → All samples of that counter over the last 5 minutes.



Selecting Data

- **5. Offsets**
- Look back in time without changing the window size.
- Useful for comparisons (e.g., today vs yesterday).
- Example:
- `rate(http_requests_total[5m] offset 1h)`
- → Request rate 1 hour ago.
- .



Selecting Data

- ◆ **6. @ Modifier**
- Query at a specific timestamp (absolute time).
- Example:
- `up @ 1695900000`
- → Status of targets at that Unix time.



Selecting Data

- ◆ **7. Putting it together**
- **Example A – Error requests right now**
- `http_requests_total{status=~"5..")}`
- **Example B – Requests per second last 5m**
- `rate(http_requests_total{job="frontend"}[5m])`
- **Example C – Compare current vs 24h ago**
- `rate(http_requests_total[5m]) / rate(http_requests_total[5m] offset 24h)`
- `.`



Basic PromQL Queries

1. Select All Data for a Metric

To select the latest value of a specific metric:

```
promql
```

 Copy code

```
metric_name
```

For example:

```
promql
```

 Copy code

```
http_requests_total
```

This query retrieves the current value for the `http_requests_total` metric.



Basic PromQL Queries

2. Select Data with Specific Labels

You can filter results based on specific label values:

```
promql
```

Copy code

```
metric_name{label1="value1", label2="value2"}
```

Example:

```
promql
```

Copy code

```
http_requests_total{method="GET", handler="/api/users"}
```

This selects the total number of HTTP GET requests for the `/api/users` endpoint.



Basic PromQL Queries

3. Select a Range of Data (Time Range)

To query data over a specified time range:

```
promql
```

 Copy code

```
metric_name[<duration>]
```

Example:

```
promql
```

 Copy code

```
http_requests_total[5m]
```

This returns the `http_requests_total` metric over the last 5 minutes.



Basic PromQL Queries

Aggregation Queries

4. Sum of All Values

To sum the values of a metric across all instances or labels:

```
promql
```

Copy code

```
sum(metric_name)
```

Example:

```
promql
```

Copy code

```
sum(http_requests_total)
```

This returns the total number of HTTP requests across all instances.



Basic PromQL Queries

5. Average of Values

To calculate the average value across instances:

```
promql
```

Copy code

```
avg(metric_name)
```

Example:

```
promql
```

Copy code

```
avg(cpu_usage_seconds_total)
```

This returns the average CPU usage across all instances.



Basic PromQL Queries

6. Max or Min of Values

To find the maximum or minimum value across instances:

- Max:

```
promql  
  
max(metric_name)
```

[Copy code](#)

- Min:

```
promql  
  
min(metric_name)
```

[Copy code](#)

Example:

```
promql  
  
max(memory_usage_bytes)
```

[Copy code](#)

This returns the maximum memory usage across instances.



Basic PromQL Queries

7. Group by Label

You can group the results by a label using the `by` clause:

```
promql
```

Copy code

```
sum(metric_name) by (label)
```

Example:

```
promql
```

Copy code

```
sum(http_requests_total) by (handler)
```

This returns the total number of HTTP requests grouped by the `handler` label.



Basic PromQL Queries

Rate Queries

9. Rate of Change

To calculate the per-second rate of change for a counter over a time range, use the `rate` function:

```
promql
```

Copy code

```
rate(metric_name[<duration>])
```

Example:

```
promql
```

Copy code

```
rate(http_requests_total[5m])
```

This calculates the per-second rate of HTTP requests over the last 5 minutes.



Basic PromQL Queries

10. Increase in Counter Over Time

To calculate how much a counter has increased over a time range:

```
promql
```

 Copy code

```
increase(metric_name[<duration>])
```

Example:

```
promql
```

 Copy code

```
increase(http_requests_total[1h])
```



Basic PromQL Queries

11. Changes Over Time (for non-counters)

To compute the difference over time for non-counter metrics (like gauges):

```
promql
```

Copy code

```
delta(metric_name[<duration>])
```

Example:

```
promql
```

Copy code

```
delta(cpu_usage_seconds_total[1h])
```

This calculates the change in CPU usage over the last hour.



Basic PromQL Queries

13. Top N

To retrieve the top N results based on a metric value:

```
promql
```

Copy code

```
topk(N, metric_name)
```

Example:

```
promql
```

Copy code

```
topk(5, sum(rate(http_requests_total[5m])) by (instance))
```



Basic PromQL Queries

14. Bottom N

To retrieve the bottom N results based on a metric value:

```
promql
```

 Copy code

```
bottomk(N, metric_name)
```

Example:

```
promql
```

 Copy code

```
bottomk(5, sum(rate(http_requests_total[5m])) by (instance))
```

This returns the 5 instances with the lowest request rate.



Basic PromQL Queries

15. Adding, Subtracting, Multiplying Metrics

You can apply mathematical operations to metrics.

- **Addition:**

```
promql
```

 [Copy code](#)

```
metric_name1 + metric_name2
```

- **Subtraction:**

```
promql
```

 [Copy code](#)

```
metric_name1 - metric_name2
```

- **Multiplication:**

```
promql
```

 [Copy code](#)

```
metric_name1 * metric_name2
```



Basic PromQL Queries

Example:

```
promql
```

 Copy code

```
cpu_usage_seconds_total * 100 / instance_cpu_quota
```



Basic PromQL Queries

Example:

```
promql
```

 Copy code

```
cpu_usage_seconds_total * 100 / instance_cpu_quota
```



Basic PromQL Queries

Subquery

Return the 5-minute `rate` of the `http_requests_total` metric for the past 30 minutes, with a resolution of 1 minute.

```
rate(http_requests_total[5m])[30m:1m]
```

This is an example of a nested subquery. The subquery for the `deriv` function uses the default resolution. Note that using subqueries unnecessarily is unwise.

```
max_over_time(deriv(rate(distance_covered_total[5s])[30s:5s])[10m:])
```



Prometheus Query – JVM Memory Usage

- `jvm_memory_used_bytes{area="heap"}`
- `jvm_memory_used_bytes{area="nonheap"}`
- `jvm_memory_max_bytes{area="heap"}`
- `(jvm_memory_used_bytes{area="heap"} / jvm_memory_max_bytes{area="heap"}) * 100`



Prometheus Query – JVM Garbage Collection

- `sum(jvm_gc_pause_seconds_sum)`
- `sum(jvm_gc_pause_seconds_count)`
- `sum(jvm_gc_pause_seconds_sum) / sum(jvm_gc_pause_seconds_count)`
- `sum(jvm_gc_pause_seconds_sum{gc="G1 Old Generation"})`



Prometheus Query – JVM Threads

- jvm_threads_live_threads
- jvm_threads_peak_threads
- jvm_threads_daemon_threads



Prometheus Query – HTTP Request Metrics

- `sum(http_server_requests_seconds_count)`
- `sum(http_server_requests_seconds_sum) / sum(http_server_requests_seconds_count)`
- `sum(http_server_requests_seconds_count) by (status)`

Prometheus Query – Rate of Change (GC, Memory, etc.)



- `rate(jvm_gc_pause_seconds_count[5m])`
- `rate(jvm_memory_used_bytes{area="heap"}[5m])`
- `rate(http_server_requests_seconds_count[1m])`



Types of Metrics

- **Types of Metrics**
- **Counter:**
 - Monotonic increasing until reset.
 - Examples: http_requests_total, bytes_sent_total.
- **Gauge:**
 - Current snapshot value (can go up/down).
 - Examples: memory_usage_bytes, temperature_celsius.
- **Histogram:**
 - Multiple series ending with _bucket, plus _sum and _count.
 - Example: request_duration_seconds_bucket{le="0.1"}.
- **Summary:**
 - Pre-computed quantiles at the client side (harder to aggregate; less used in distributed systems)



Common PromQL Query Types

PromQL also supports standard operators for combining and comparing time-series.

1. Arithmetic Operators:

These include basic mathematical operations such as `+`, `-`, `*`, `/`.

Example:

```
promql
(copy code)
(rate(http_requests_total[5m]) / sum(rate(http_requests_total[5m]))) * 100
```

This calculates the percentage of total HTTP requests handled by each instance.

2. Comparison Operators:

Used to filter time-series based on a comparison, such as `>`, `<`, `==`, `!=`, etc.

Example:

```
promql
(copy code)
http_requests_total > 1000
```

This returns all time-series where the number of HTTP requests is greater than 1,000.



Common PromQL Query Types

3. Boolean Modifiers:

These operators can also be used with a boolean modifier (e.g., `http_requests_total > 1000 == true`).

4. Logical/Set Operators:

These operators combine time-series based on their label sets:

- `and` : Returns time-series present in both operands.
- `or` : Returns time-series present in either operand.
- `unless` : Excludes time-series from the left operand that are also present in the right operand.

Example (and):

```
promql
cpu_usage and memory_usage
Copy code
```

This returns time-series where both CPU usage and memory usage metrics are present.



Limitations in Prometheus Data Model

Limitation	Description
High Cardinality	Too many unique combinations of label sets (label cardinality) can explode memory usage and slow down queries.
No Native Histograms/Percentiles	Only approximations using buckets (via <code>histogram_quantile()</code>); can't compute exact percentiles.
No Native Support for Complex Events	It's metric-based, not event-based. Lacks native support for logs/traces (though integrations like Loki and Tempo help).
No Join Across Metrics	You cannot perform SQL-style joins between different metrics (only label-based vector matching).
Limited Data Types	Only <code>float64</code> values are stored; no native support for strings, booleans, or complex structures.
No Multi-Tenancy Built-In	Prometheus itself isn't multi-tenant; workarounds require using Cortex, Thanos, or Mimir.
Local Storage Only	By default, Prometheus uses local TSDB which isn't horizontally scalable (Thanos/Cortex help here too).
Global View is Manual	Federation must be set up manually to get a global view across Prometheus instances.



Limitations in Prometheus Labels

Limitation	Description
Cardinality Explosion	Using dynamic values like <code>pod_id</code> , <code>user_id</code> , or <code>transaction_id</code> as labels leads to explosive series counts.
Not Indexed like a DB	Labels are stored in a custom index format; querying non-existent labels can be expensive.
No Reserved Label Names Enforcement	Misusing labels like <code>job</code> , <code>instance</code> , or <code>_name_</code> can break queries silently.
No Schema Enforcement	Labels are free-form; no schema validation, which can lead to inconsistent metrics if not controlled.
Label Value Size	Long label values increase memory usage and can cause performance degradation.
Label Set Uniqueness	Each unique label set defines a new series , even if the metric name is the same. Misuse can cause duplication.



Prometheus Exposition Format

- The Prometheus Exposition Format is a text-based format used by Prometheus to scrape metrics from instrumented applications (exporters, apps, etc.). It's the default format for exposing metrics over HTTP.

- ◆ **Structure of Exposition Format**

Each scraped page (e.g., at `/metrics`) contains a mix of:

1. **HELP** line (optional)
2. **TYPE** line (optional)
3. **Metric lines** – actual values with labels

- ◆ **Basic Example:**

```
text
```

```
# HELP http_requests_total Total number of HTTP requests
# TYPE http_requests_total counter
http_requests_total{method="GET",code="200"} 1027
http_requests_total{method="POST",code="500"} 3
```



Prometheus Exposition Format

Component	Purpose
# HELP	Describes what the metric does
# TYPE	Declares the metric type (<code>counter</code> , <code>gauge</code> , <code>histogram</code> , <code>summary</code> , or <code>gauge_histogram</code>)
Metric line	Key-value pairs in <code>{}</code> are labels ; the last value is the metric's current value



Exposition

- # HELP go_gc_duration_seconds A summary of the wall-time pause (stop-the-world) duration in garbage collection cycles.
- # TYPE go_gc_duration_seconds summary
- go_gc_duration_seconds{quantile="0"} 1.5378e-05
- go_gc_duration_seconds{quantile="0.25"} 2.6879e-05
- go_gc_duration_seconds{quantile="0.5"} 3.2515e-05
- go_gc_duration_seconds{quantile="0.75"} 3.9957e-05
- go_gc_duration_seconds{quantile="1"} 0.00037568
- go_gc_duration_seconds_sum 0.041221056
- go_gc_duration_seconds_count 674

Understanding Prometheus Metrics and Buckets

- Prometheus is a powerful monitoring tool that collects and stores time-series data as metrics.
- It supports four types of metrics:
- Counter: A cumulative metric that only increases or resets to zero.
- Gauge: A metric that can increase or decrease.
- Histogram: A metric that samples observations and counts them in configurable buckets.
- Summary: Like histogram but calculates configurable quantiles over a sliding time window.

Understanding Prometheus Metrics and Buckets

- Buckets are primarily associated with histogram metrics in Prometheus.
- They represent ranges of observed values and are essential for aggregating data efficiently.
- By using buckets, Prometheus can provide insights into the distribution of metric values without storing every individual data point.



What is a Bucket in Prometheus?

- A bucket in Prometheus is a predefined range of values used to categorize and count observations in histogram metrics.
- Each bucket represents a cumulative count of all observations less than or equal to its upper bound.
- This structure allows Prometheus to efficiently store and query large volumes of data while providing valuable insights into the distribution of metric values.



Anatomy of a Prometheus Bucket

- Prometheus buckets have a specific structure:
- Metric name: The base name of the histogram metric.
- `_bucket` suffix: Appended to the metric name to indicate a bucket.
- `le` label: Stands for "less than or equal to" and defines the upper bound of the bucket.

```
http_request_duration_seconds_bucket{le="0.1"} 2036
```



How Prometheus Histograms Use Buckets

- Prometheus histograms organize observed values into predefined buckets.
- When a new observation is made, Prometheus increments the counters for all buckets with upper bounds greater than or equal to the observed value.
- This process creates a cumulative frequency distribution of the data.

```
http_request_duration_seconds_bucket{le="0.1"} 2036
http_request_duration_seconds_bucket{le="0.5"} 2127
http_request_duration_seconds_bucket{le="1"} 2198
http_request_duration_seconds_bucket{le="5"} 2205
http_request_duration_seconds_bucket{le="+Inf"} 2205
```

Practical Applications of Prometheus Buckets



- Measuring request latencies: Buckets help identify performance bottlenecks by showing the distribution of response times.
- Monitoring system resource utilization: Buckets can track CPU, memory, or disk usage patterns over time.
- Calculating Apdex scores: Buckets facilitate the computation of application performance index scores.
- Capacity planning: By analyzing bucket distributions, you can make informed decisions about resource allocation and scaling.



Prometheus Query

- Counters
 - Counter metrics are used for measurements that only increase.
 - Therefore, they are always cumulative—their value can only go up.
 - The only exception is when the counter is restarted, in which case its value is reset to zero.
 - The actual value of a counter is not typically very useful on its own.
 - A counter value is often used to compute the delta between two timestamps or the rate of change over time.



Prometheus Query

- Counters
- The PromQL query below calculates the average requests per second over the last five minutes
 - `rate(prometheus_http_requests_total[5m])`
- To calculate the absolute change over a time period, we would use a delta function which in PromQL is called `increase()`:
 - `increase(prometheus_http_requests_total[5m])`



Prometheus Query

- Counters
- This would return the total number of requests made in the last five minutes, and it would be the same as multiplying the per second rate by the number of seconds in the interval (five minutes in our case):
- `rate(prometheus_http_requests_total[5m])*5*60`
- `count_over_time(process_cpu_usage{instance="customer-app:7074"}[10m])`



Prometheus Query

Count by (job) (up)

Screenshot of the Prometheus Query interface showing the results of the query `count by (job) (up)`.

The results table shows the following data:

Job	Count
(job="prometheus-pushgateway")	1
(job="pushgateway")	1
(job="kubernetes-service-endpoints")	4
(job="kubernetes-apiservers")	1
(job="customer-app")	1
(job="kubernetes-nodes-cadvisor")	1
(job="prometheus")	1
(job="kubernetes-nodes")	1

Query configuration: `count by (job) (up)`. Evaluation time: [].

Metrics used: `up`.

Load time: 57ms Resolution: 14s Result series: 8

Remove Panel



Prometheus Query

- Gauges
 - Gauge metrics are used for measurements that can arbitrarily increase or decrease.
 - the actual value with no additional processing is meaningful and they are often used.
 - For example, metrics to measure temperature, CPU, and memory usage, or the size of a queue are gauges.



Prometheus Query

- `avg_over_time(application_ready_time_seconds{instance="customer-app:7074"}[5m])`



Prometheus Query

- **Histograms**
 - Histogram metrics are useful to represent a distribution of measurements.
 - They are often used to measure request duration or response size.
 - Histograms divide the entire range of measurements into a set of intervals—named buckets—and count how many measurements fall into each bucket.



Prometheus Query (Histogram)

- Histogram is a more complex metric type when compared to the previous two.
- Histogram can be used for any calculated value which is counted based on bucket values.
- Bucket boundaries can be configured by the developer.
- A common example would be the time it takes to reply to a request, called latency.

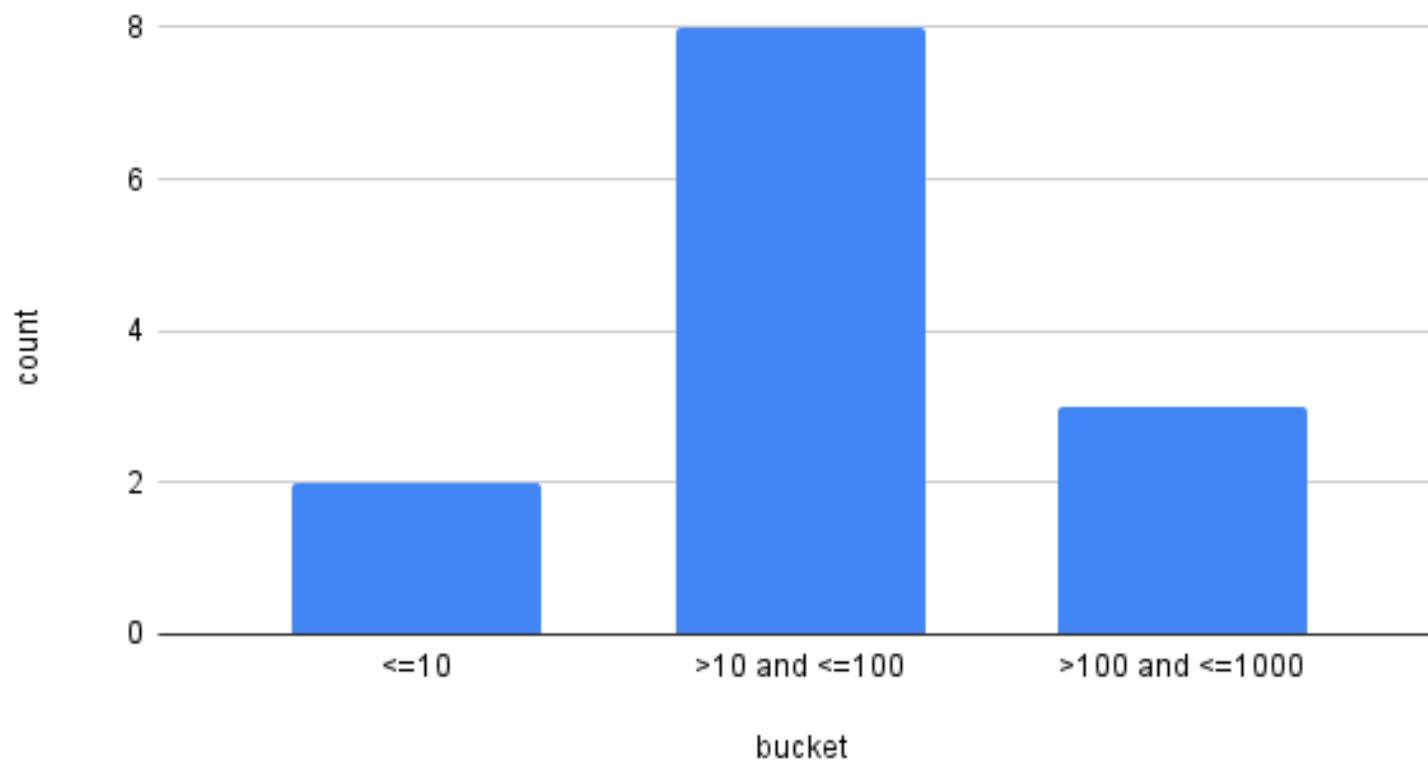


Prometheus Query (Histogram)

- The buckets are cumulative - each bucket contains values less than or equal to the bucket's upper threshold.
- The time series itself is cumulative - the buckets in the histogram are always increasing so that the most recent instance of the histogram shows the total values for each of the buckets since the metric was first recorded.

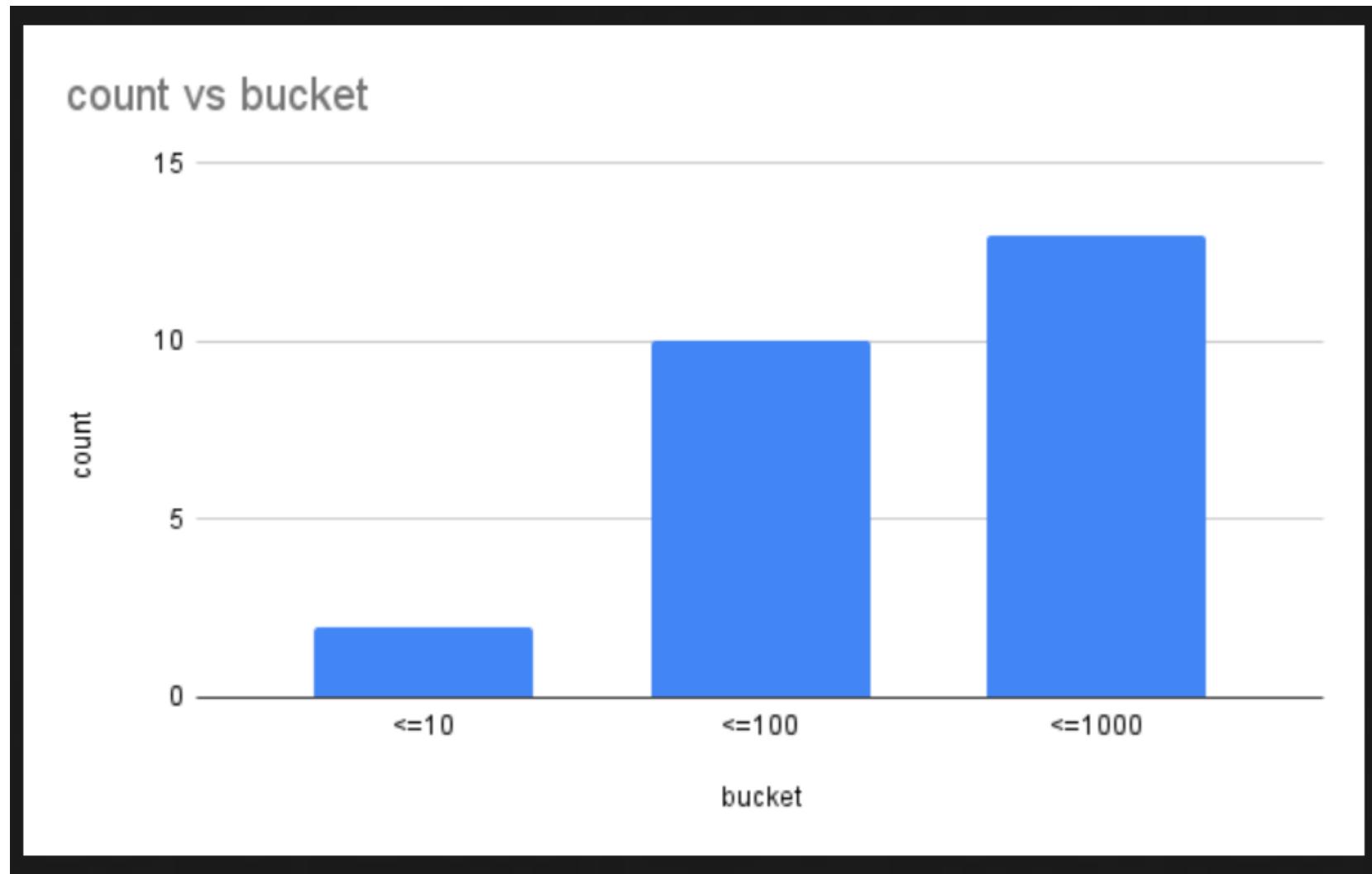
Prometheus Query (Counter)

count vs bucket





Prometheus Query (Histogram-Cumulative)





Prometheus Query (Histogram-Cumulative)

Enable query history

[Try experimental React UI](#)

histogram_metric_bucket

Load time: 51ms
Resolution: 14s
Total time series: 6

[Execute](#)

- insert metric at cursor ▾

[Graph](#)

[Console](#)



Moment



Element

Value

histogram_metric_bucket{instance="application:8080",job="application",le="1"}

44

histogram_metric_bucket{instance="application:8080",job="application",le="2"}

89

histogram_metric_bucket{instance="application:8080",job="application",le="3"}

132

histogram_metric_bucket{instance="application:8080",job="application",le="4"}

177

histogram_metric_bucket{instance="application:8080",job="application",le="5"}

210

histogram_metric_bucket{instance="application:8080",job="application",le="+Inf"}

210



Prometheus Query (Histogram)

- Example: Let's assume we want to observe the time taken to process API requests.
- Instead of storing the request time for each request, histograms allow us to store them in buckets.
- We define buckets for time taken, for example lower or equal 0.3, le 0.5, le 0.7, le 1, and le 1.2.
- So, these are our buckets and once the time taken for a request is calculated it is added to the count of all the buckets whose bucket boundaries are higher than the measured value.



Prometheus Query (Histogram)

Let's say Request 1 for endpoint "/ping" takes 0.25 s. The count values for the buckets will be.

/ping

Bucket	Count
0 - 0.3	1
0 - 0.5	1
0 - 0.7	1
0 - 1	1
0 - 1.2	1
0 - +Inf	1

Note: +Inf bucket is added by default.



Prometheus Query (Histogram)

(Since the histogram is a cumulative frequency 1 is added to all the buckets that are greater than the value)

Request 2 for endpoint "/ping" takes 0.4s The count values for the buckets will be this.

/ping

Bucket	Count
0 - 0.3	1
0 - 0.5	2
0 - 0.7	2
0 - 1	2
0 - 1.2	2
0 - +Inf	2



Prometheus Query (Histogram)

```
# /metrics response

# HELP http_request_duration_seconds_bucket The latency of the HTTP requests
# TYPE http_request_duration_seconds_bucket histogram
http_request_duration_seconds_bucket_bucket{le="0.0005"} 1
http_request_duration_seconds_bucket_bucket{le="0.001"} 1
http_request_duration_seconds_bucket_bucket{le="0.005"} 2
http_request_duration_seconds_bucket_bucket{le="0.01"} 2
http_request_duration_seconds_bucket_bucket{le="0.015"} 2
http_request_duration_seconds_bucket_bucket{le="0.02"} 3
http_request_duration_seconds_bucket_bucket{le="0.025"} 3
http_request_duration_seconds_bucket_bucket{le="+Inf"} 3
```

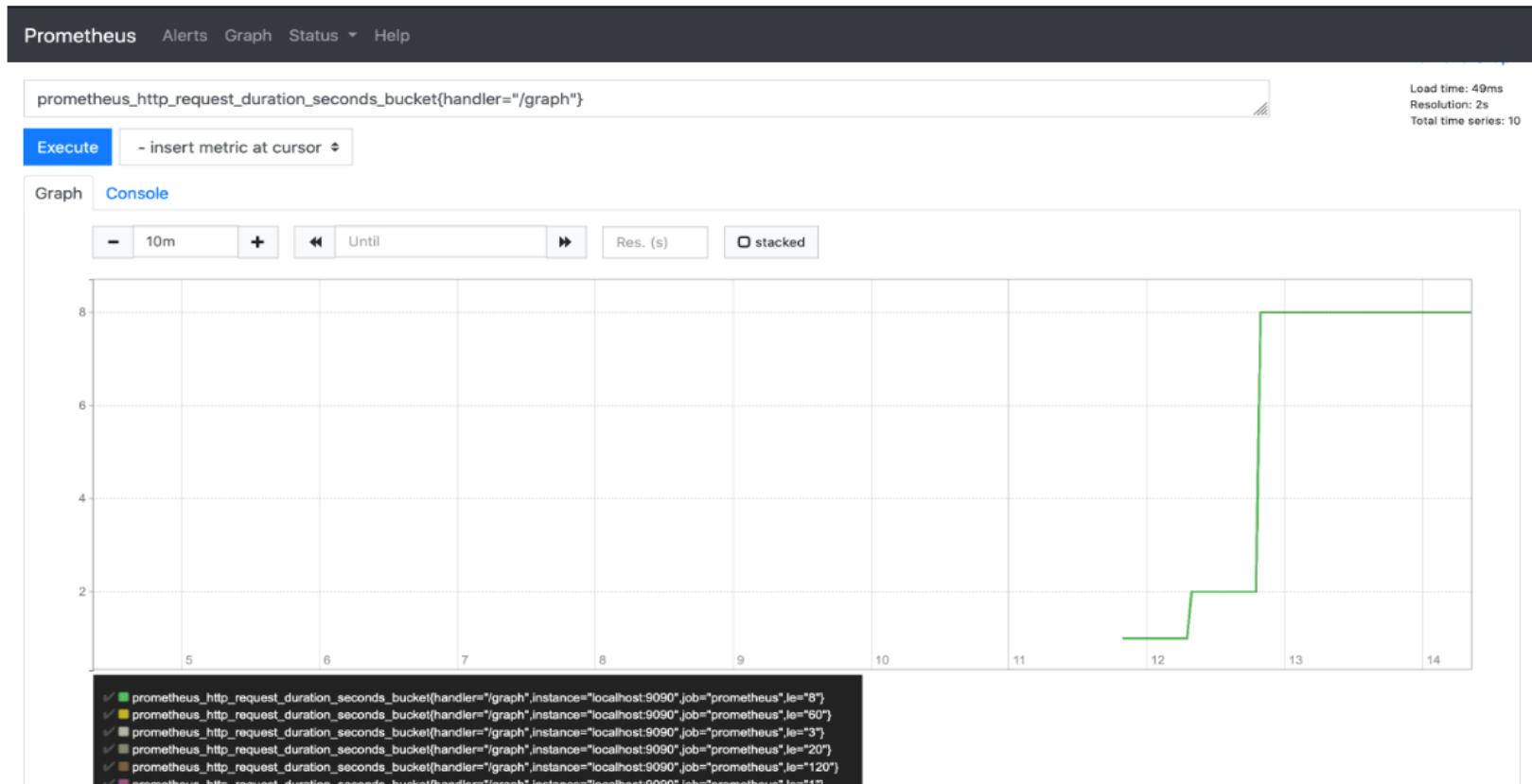


Prometheus Query (Histogram)

Since 0.4 is below 0.5, all buckets up to that boundary increase their counts.

Let's explore a histogram metric from the Prometheus UI and apply a few functions.

```
prometheus_http_request_duration_seconds_bucket{handler="/graph"}
```



Prometheus Query (Histogram) Visualization methods



- Percentiles
- It's a general way for histogram visualization, and in most cases that's enough.
- We can build the graph using the histogram_quantile function.
- PromQL query for getting 95 percentile
- `histogram_quantile(0.95, sum by(handler, code, le) (rate(http_request_duration_seconds_bucket{handler="/payload"}[1m])))`

Prometheus Query (Histogram) Visualization methods



Fig 1.
95th percentile

The graph shows that 95% of all requests took 20.1ms. Similarly, we can build any percentile graph.



Prometheus Query

- **Histograms**

- A histogram metric includes a few items:
- A counter with the total number of measurements.
- The metric name uses the `_count` suffix.
- A counter with the sum of the values of all measurements.
- The metric name uses the `_sum` suffix.
- The histogram buckets are exposed as counters using the metric name with a `_bucket` suffix and a le label indicating the bucket upper inclusive bound.
- Buckets in Prometheus are inclusive, that is a bucket with an upper bound of N (i.e., le label) includes all data points with a value less than or equal to N.



Prometheus Query

- **Histograms**

- `http_server_requests_seconds_count{instance="customer-app:7074"}`
- `prometheus_http_request_duration_seconds_bucket{instance="localhost:9090"}`



Advanced Query Examples

1. Alerting Threshold Example:

```
promql Copy code
```

```
node_cpu_usage_seconds_total{job="node"} > 0.9
```

This query triggers an alert if CPU usage on any node exceeds 90%.

2. Percentage of Error Rate:

```
promql Copy code
```

```
(rate(http_requests_total{status="500"}[5m]) / rate(http_requests_total[5m])) * 100
```

This calculates the percentage of HTTP requests that returned a 500 error code.

3. Memory Usage Over Time:

```
promql Copy code
```

```
avg_over_time(node_memory_usage_bytes[1h])
```

This calculates the average memory usage over the last hour.



Advanced Query Examples

1. Selecting Data in PromQL

Selecting data in PromQL is the foundation of querying time-series data. You can use **instant vectors** and **range vectors** to retrieve data.

Instant Vector:

- An instant vector returns the latest value of a time-series at a given timestamp.

Example:

```
promql
```

 Copy code

```
http_requests_total
```

This returns the current value of `http_requests_total` for all time-series (with different labels).



Advanced Query Examples

Range Vector:

- A range vector retrieves time-series data over a specified time range.

Example:

```
promql
```

Copy code

```
http_requests_total[5m]
```

This returns all time-series for the last 5 minutes.

Filtering by Labels:

You can filter data by using label selectors.

Example:

```
promql
```

Copy code

```
http_requests_total{method="GET", status="200"}
```

This selects only the series where `method` is "GET" and `status` is "200".



Advanced Query Examples

2. Rates and Derivatives in PromQL

Rates and derivatives are used to calculate the change in a time-series over time, especially useful for metrics that increment continuously (like counters).

rate():

- `rate()` calculates the per-second average rate of increase for a counter over a specified time window. This is useful for computing the rate at which a metric is increasing over time.

Example:

```
promql
```

 Copy code

```
rate(http_requests_total[5m])
```

This calculates the per-second rate of HTTP requests over the last 5 minutes.



Advanced Query Examples

irate():

- `irate()` calculates the instantaneous rate of change based on the last two data points within a time range. It's more sensitive to recent changes.

Example:

```
promql
```

Copy code

```
irate(http_requests_total[5m])
```

increase():

- `increase()` calculates the total increase in a counter over a specified time range.

Example:

```
promql
```

Copy code

```
increase(http_requests_total[1h])
```

This calculates how many HTTP requests occurred in the last hour.





Advanced Query Examples

```
delta():
```

- `delta()` calculates the difference between the start and end values of a time-series in a given time range, used for gauges (non-counters).

Example:

```
promql
```

 Copy code

```
delta(node_memory_usage_bytes[10m])
```

This calculates the difference in memory usage over the last 10 minutes.



Advanced Query Examples

3. Aggregating Over Time

PromQL allows you to aggregate time-series data over a time window.

`avg_over_time()`:

- This function calculates the average value of the time-series in a given range.

Example:

promql

Copy code

```
avg_over_time(http_requests_total[1h])
```

This calculates the average number of requests over the past hour.

`sum_over_time()`:

- This function calculates the sum of all values in a time range.

Example:

promql

Copy code

```
sum_over_time(node_cpu_seconds_total[30m])
```

This calculates the total CPU time over the last 30 minutes.



Advanced Query Examples

```
min_over_time() and max_over_time():
```

- These functions return the minimum or maximum value over a time range.

Example (min):

```
promql
```

Copy code

```
min_over_time(node_memory_usage_bytes[1h])
```

This returns the minimum memory usage in the last hour.

```
count_over_time():
```

- Counts the number of data points in the specified range.

Example:

```
promql
```

Copy code

```
count_over_time(http_requests_total[10m])
```



Advanced Query Examples

4. Aggregating Over Dimensions

Aggregation over dimensions allows you to reduce the dimensionality of the data by applying functions (like sum, avg) across different labels.

sum() :

- Aggregates across time-series by summing the values across a common label set.

Example:

```
promql
```

Copy code

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

This sums the rate of HTTP requests grouped by the `method` label.

avg() :

- Calculates the average across time-series.

Example:

```
promql
```

Copy code

```
avg(rate(cpu_usage_seconds_total[1m])) by (instance)
```

This calculates the average CPU usage rate per instance.



Advanced Query Examples

`max()` and `min()`:

- These return the maximum and minimum values across time-series.

Example (max):

promql

Copy code

```
max(rate(node_cpu_seconds_total[5m])) by (instance)
```

`count()`:

- Counts the number of time-series that match a given query.

Example:

promql

Copy code

```
count(up) by (job)
```

This counts the number of instances that are up, grouped by `job`.



Advanced Query Examples

5. Binary Operators

PromQL supports binary operators to combine time-series.

Arithmetic Operators:

- Operators like `+`, `-`, `*`, `/` can be used to perform arithmetic on time-series.

Example:

```
promql Copy code  
rate(http_requests_total[5m]) / sum(rate(http_requests_total[5m])) * 100
```

This calculates the percentage of total HTTP requests handled by each instance.

Comparison Operators:

- Use `>`, `<`, `>=`, `<=`, `==`, `!=` to filter time-series based on comparison.

Example:

```
promql Copy code  
http_requests_total > 1000
```

This selects time-series where the total number of HTTP requests exceeds 1000.



Advanced Query Examples

Logical/Set Operators:

- These operators (e.g., `and`, `or`, `unless`) can combine time-series sets based on their labels.

Example (and):

```
promql
```

 Copy code

```
cpu_usage and memory_usage
```

Example (or):

```
promql
```

 Copy code

```
cpu_usage or memory_usage
```



Advanced Query Examples

Logical/Set Operators:

- These operators (e.g., `and`, `or`, `unless`) can combine time-series sets based on their labels.

Example (and):

```
promql
```

 Copy code

```
cpu_usage and memory_usage
```

Example (or):

```
promql
```

 Copy code

```
cpu_usage or memory_usage
```



Advanced Query Examples

6. Histograms

Prometheus uses **histograms** to measure distributions. A histogram measures the frequency distribution of events over a range of values (e.g., request durations).

Histogram Metrics:

Prometheus stores histograms in multiple time-series for each bucket, along with a `_bucket` suffix.

Example metric:

- `request_duration_seconds_bucket{le="0.1"}` : The number of requests with a duration of 0.1 seconds or less.

`histogram_quantile()` :

- This function is used to calculate quantiles from histogram buckets. It's commonly used to calculate percentiles like the 95th percentile of request durations.

Example:

```
promql
```

Copy code

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

This calculates the 95th percentile of request durations over the last 5 minutes.



Advanced Query Examples

7. Timestamp Metrics

You can extract the timestamp of the most recent data point from a time-series using the `timestamp()` function.

`timestamp()`:

- Returns the timestamp (in Unix time) of the current value of a time-series.

Example:

```
promql
```

Copy code

```
timestamp(http_requests_total)
```

This returns the Unix timestamp of the last `http_requests_total` value.



Prometheus Query

localhost:9090/graph?g0.expr=hikaricp_connections_max%20or%20jvm_memory_max_bytes&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h

Insert title here Empire New Tab How to use Asserti... Browser Automatio... desktop-55agi0i.ms... Freelancer-dev-810... Courses node.js - How can I... New Tab Airtel 4G Hotspot nt8F83 »

Prometheus Alerts Graph Status Help

Use local time Enable query history Enable autocomplete Enable highlighting Enable linter

hikaricp_connections_max or jvm_memory_max_bytes Execute

Table Graph Evaluation time < >

Load time: 22ms Resolution: 14s Result series: 9

hikaricp_connections_max{instance="host.docker.internal:7074", job="customer-api", pool="HikariPool-1"}	10
jvm_memory_max_bytes{area="heap", id="G1 Eden Space", instance="host.docker.internal:7074", job="customer-api"}	-1
jvm_memory_max_bytes{area="heap", id="G1 Old Gen", instance="host.docker.internal:7074", job="customer-api"}	2061500416
jvm_memory_max_bytes{area="heap", id="G1 Survivor Space", instance="host.docker.internal:7074", job="customer-api"}	-1
jvm_memory_max_bytes{area="nonheap", id="CodeHeap 'non-nmethods'", instance="host.docker.internal:7074", job="customer-api"}	5836800
jvm_memory_max_bytes{area="nonheap", id="CodeHeap 'non-profiled nmethods'", instance="host.docker.internal:7074", job="customer-api"}	122912768
jvm_memory_max_bytes{area="nonheap", id="CodeHeap 'profiled nmethods'", instance="host.docker.internal:7074", job="customer-api"}	122908672
jvm_memory_max_bytes{area="nonheap", id="Compressed Class Space", instance="host.docker.internal:7074", job="customer-api"}	1073741824
jvm_memory_max_bytes{area="nonheap", id="Metaspace", instance="host.docker.internal:7074", job="customer-api"}	-1

Remove Panel

Add Panel

JSON API Grafana



Data sources

+ Add new data source

View and manage your connected data source connections

Q Search by name or type

Sort by A-Z



marcusolsson-json-datasource

JSON API | https://jsonplaceholder.typicode.com/users

Build a dashboard

Explore



prometheus

Prometheus | http://host.docker.internal:9090

default

Build a dashboard

Explore

JSON API Grafana



Searched: Search or jump to... ctrl+k

Home > Dashboards > New dashboard > Edit panel

Table view Fill Actual Last 6 hours

	A id	A username	A email	A city	A phone
1	Bret		Sincere@april.biz	Gwenborough	1-770-736-8031 x56442
2	Antonette		Shanna@melissa.tv	Wisokyburgh	010-692-6593 x09125
3	Samantha		Nathan@yesenia.net	McKenziehaven	1-463-123-4447
4	Karianne		Julianne.OConner@kory.org	South Elvis	493-170-9623 x156

Query 1 Transform data 0 Alert 0

Data source marcusolsson-json-datasource ? Query options MD = auto = 1138 Interval = 20s Query inspector

Users_Query (marcusolsson-json-datasource) Copy Open Delete More

Fields	Path	Params	Headers	Body	Experimental	Cache Time ?	5m				
Field ?	\$..id					JSONPath ?	Type ?	Auto ?	Alias ?		+ -
Field ?	\$..username					JSONPath ?	Type ?	Auto ?	Alias ?		+ -
Field ?	\$..email					JSONPath ?	Type ?	Auto ?	Alias ?		+ -



Prometheus Exporters

- Prometheus exporters are components or services that translate metrics from various sources into a format that Prometheus can scrape and understand.
- Since Prometheus natively supports scraping metrics from its instrumentation clients, exporters are used to collect metrics from systems that do not natively support Prometheus, such as databases, hardware devices, or external applications.



Prometheus Exporters

- Applications are instrumented using client libraries.
- It enables an HTTP endpoint where internal metrics are exposed and collected by Prometheus servers.
- Other monitoring tools relies on agents or embedded instrumentation (e.g. APM client libraries) to collect data and "push" metrics to the monitoring backend.



Types of Prometheus Exporters

- Node Exporter:
 - Collects hardware and OS-level metrics like CPU usage, memory, disk I/O, and network stats from Linux or Unix-based systems.
 - Commonly used to monitor servers.
- Blackbox Exporter:
 - Allows probing of endpoints over various network protocols like HTTP, HTTPS, DNS, TCP, and ICMP. Great for uptime monitoring and checking service availability.



Types of Prometheus Exporters

- Database Exporters:
 - MySQL Exporter:
 - Exposes MySQL server metrics, such as query statistics, table locks, and replication status.
 - PostgreSQL Exporter:
 - Similar for PostgreSQL databases, providing connection, query, and performance metrics.
 - MongoDB Exporter, Redis Exporter, Elasticsearch Exporter, etc., work similarly for other databases.
- JMX Exporter:
 - Collects metrics from Java applications running with Java Management Extensions (JMX), typically for applications like Apache Kafka, Tomcat, and Cassandra.



Types of Prometheus Exporters

- HAProxy Exporter:
 - Used to gather metrics from HAProxy instances, like requests, sessions, and backend stats.
- Nginx Exporter:
 - Exposes metrics from the Nginx web server, such as request rates, response times, and error counts.
- cAdvisor Exporter:
 - Collects metrics from running containers and exposes resource usage statistics like CPU, memory, and network utilization.
- SNMP Exporter:
 - Collects metrics from devices via the Simple Network Management Protocol (SNMP), often used for monitoring network devices like switches, routers, and printers.



How Prometheus Exporters Work

- Data Collection:
 - Exporters collect metrics from an underlying source (like a database or hardware).
- Metrics Exposure:
 - Exporters transform the collected data into the Prometheus metrics format and expose it at an HTTP endpoint.
- Prometheus Scraping:
 - Prometheus scrapes these endpoints at defined intervals and stores the metrics for querying and alerting.



Example: The Prometheus Node Exporter

- It is one of the most popular exporters in the Prometheus ecosystem.
- It is used to expose hardware and operating system metrics, particularly from Linux/Unix systems.
- These metrics are essential for monitoring the health and performance of a machine or virtual server.



Example: The Prometheus Node Exporter

- **Features of Node Exporter:**
- **Wide Range of Metrics:** Collects various hardware and OS-related metrics, such as CPU usage, memory usage, disk I/O, network stats, filesystem information, and more.
- **Modular Design:** Supports many collectors, allowing for enabling or disabling specific metric groups based on your monitoring needs.
- **Minimal Resource Usage:** The exporter is designed to be lightweight, so it has minimal performance impact on the host system.



Example: The Prometheus Node Exporter

- **Common Metrics**

- Exposed by Node Exporter:
 - CPU:node_cpu_seconds_total:
 - Total CPU time spent in each mode (user, system, idle, etc.).
 - Memory:
 - node_memory_MemAvailable_bytes: Available memory.
 - node_memory_MemFree_bytes: Free memory.
 - node_memory_SwapFree_bytes: Free swap memory.



Example: The Prometheus Node Exporter

- **Disk I/O:**
 - `node_disk_io_time_seconds_total`: Time spent doing I/O operations.
 - `node_disk_read_bytes_total`: Bytes read from disk.
 - `node_disk_written_bytes_total`: Bytes written to disk.
- **Filesystem:**
 - `node_filesystem_size_bytes`: Total size of a filesystem.
 - `node_filesystem_free_bytes`: Free space available.
 - `node_filesystem_files`: Total number of inodes.



Example: The Prometheus Node Exporter

- **Network:**
 - `node_network_receive_bytes_total`: Total bytes received on a network interface.
 - `node_network_transmit_bytes_total`: Total bytes transmitted on a network interface.
- **System Load:**
 - `node_load1`: Load average over the last 1 minute.
 - `node_load5`: Load average over the last 5 minutes.



Example: The Prometheus Node Exporter

- **scrape_configs:**
- - **job_name: 'node_exporter'**
- **static_configs:**
- - **targets: ['<node-exporter-host>:9100']**



Node Exporter

- docker run -d -p 9100:9100 --net="boanetwork" prom/node-exporter

Metric	Meaning
<code>rate(node_cpu_seconds_total{mode="system"}[1m])</code>	The average amount of CPU time spent in system mode, per second, over the last minute (in seconds)
<code>node_filesystem_avail_bytes</code>	The filesystem space available to non-root users (in bytes)
<code>rate(node_network_receive_bytes_total[1m])</code>	The average network traffic received, per second, over the last minute (in bytes)



Node Exporter

Prometheus Time Series Collector X Monitoring Linux host metrics w X Grafana X | +

localhost:9090/graph?g0.expr=rate(node_network_receive_bytes_total[1m])&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h

Insert title here Empire New Tab How to use Assertions... Browser Automation... desktop-55agi0i.ms... Freelancer-dev-810... Courses node.js - How can I... New Tab Airtel 4G Hotspot nt8F83

Prometheus Alerts Graph Status Help

Use local time Enable query history Enable autocomplete Enable highlighting Enable linter

rate(node_network_receive_bytes_total[1m])

Table Graph Evaluation time < > Load time: 25ms Resolution: 14s Result series: 2

Labels	Value
{device="eth0", instance="node-exporter:9100", job="node"}	36.866666666666666
{device="lo", instance="node-exporter:9100", job="node"}	0

Remove Panel

Add Panel

29°C
Partly sunny





Windows Exporter

- https://github.com/prometheus-community/windows_exporter/releases

A screenshot of a GitHub releases page for the "windows_exporter" repository. The page has a dark theme. At the top, there's a section labeled "Assets" with a dropdown arrow and a number "8" in a circle. Below this, a list of files is displayed:

- sha256sums.txt
- windows_exporter-0.22.0-386.exe
- windows_exporter-0.22.0-386.msi
- windows_exporter-0.22.0-amd64.exe
- windows_exporter-0.22.0-amd64.msi
- windows_exporter-0.22.0-arm64.exe
- Source code (zip)
- Source code (tar.gz)



Windows Exporter

- `http://localhost:9182/metrics`



Custom Exporter

```
G:\Local disk\prometheus\customexporter>python metricexporter.py -f "/test" -e "txt"
2023-06-29 22:20:09,991 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:20:19,992 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:20:29,993 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:20:39,994 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:20:49,995 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:20:59,996 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:21:09,997 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:21:19,998 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:21:29,999 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:21:40,000 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:21:50,001 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:00,002 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:10,003 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:20,004 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:30,005 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:40,006 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:22:50,007 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:00,008 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:10,009 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:20,010 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:30,010 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:40,011 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:23:50,012 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:24:00,013 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:24:10,014 - root - INFO - Finds 2 file(s) in folder /test with extension txt
2023-06-29 22:24:20,015 - root - INFO - Finds 2 file(s) in folder /test with extension txt
```



Custom Exporter

```
localhost:9877/metrics

# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 209.0
python_gc_objects_collected_total{generation="1"} 356.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 42.0
python_gc_collections_total{generation="1"} 3.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="11",patchlevel="4",version="3.11.4"} 1.0
# HELP cust_txt_files_in_test_total number of *txt files in /test
# TYPE cust_txt_files_in_test_total gauge
cust_txt_files_in_test_total 2.0
```

Grafana



A screenshot of a web browser showing the Grafana homepage. The browser tab bar includes: Prometheus Time Series Co, Swagger UI, GitHub - prometheus-compr, Windows Server Monitoring, localhost:9182/metrics, Grafana, Insert title here, Empire, New Tab, How to use Asserti..., Browser Automatio..., desktop-55agi0i.ms..., Freelancer-dev-810..., Courses, node.js - How can I..., New Tab, Airtel 4G Hotspot, nt8F83. The address bar shows localhost:3000/?orgId=1. The main content area features a search bar, a navigation menu with Home, and a welcome message: "Welcome to Grafana". To the right, there's a "Need help?" section with links to Documentation, Tutorials, Community, and Public Slack. A dropdown menu is open in the top right corner, showing options: New dashboard (selected), Import dashboard, and Create alert rule. A blue arrow points from the top right towards the "Import dashboard" option. Below the main content are three panels: "Basic" (with sections for TUTORIAL, DATA SOURCE AND DASHBOARDS, and Grafana fundamentals), "DATA SOURCES" (with a "Add your first data source" button and a "Learn how in the docs" link), and "DASHBOARDS" (with a "Create your first dashboard" button and a "Learn how in the docs" link). At the bottom, there are sections for "Dashboards" (localhost:3000/dashboard/import) and "Latest from the blog", along with a "Show all" link. The status bar at the bottom shows weather (31°C, Mostly cloudy), system icons, and the date/time (28/06/2023, 21:40).



localhost:3000/dashboard/import

Search or jump to... ctrl+k

Home > Dashboards > Import dashboard

Import dashboard

Import dashboard from file or Grafana.com

Upload dashboard JSON file

Drag and drop here or click to browse
Accepted file types: .json, .txt

Import via grafana.com

 Load

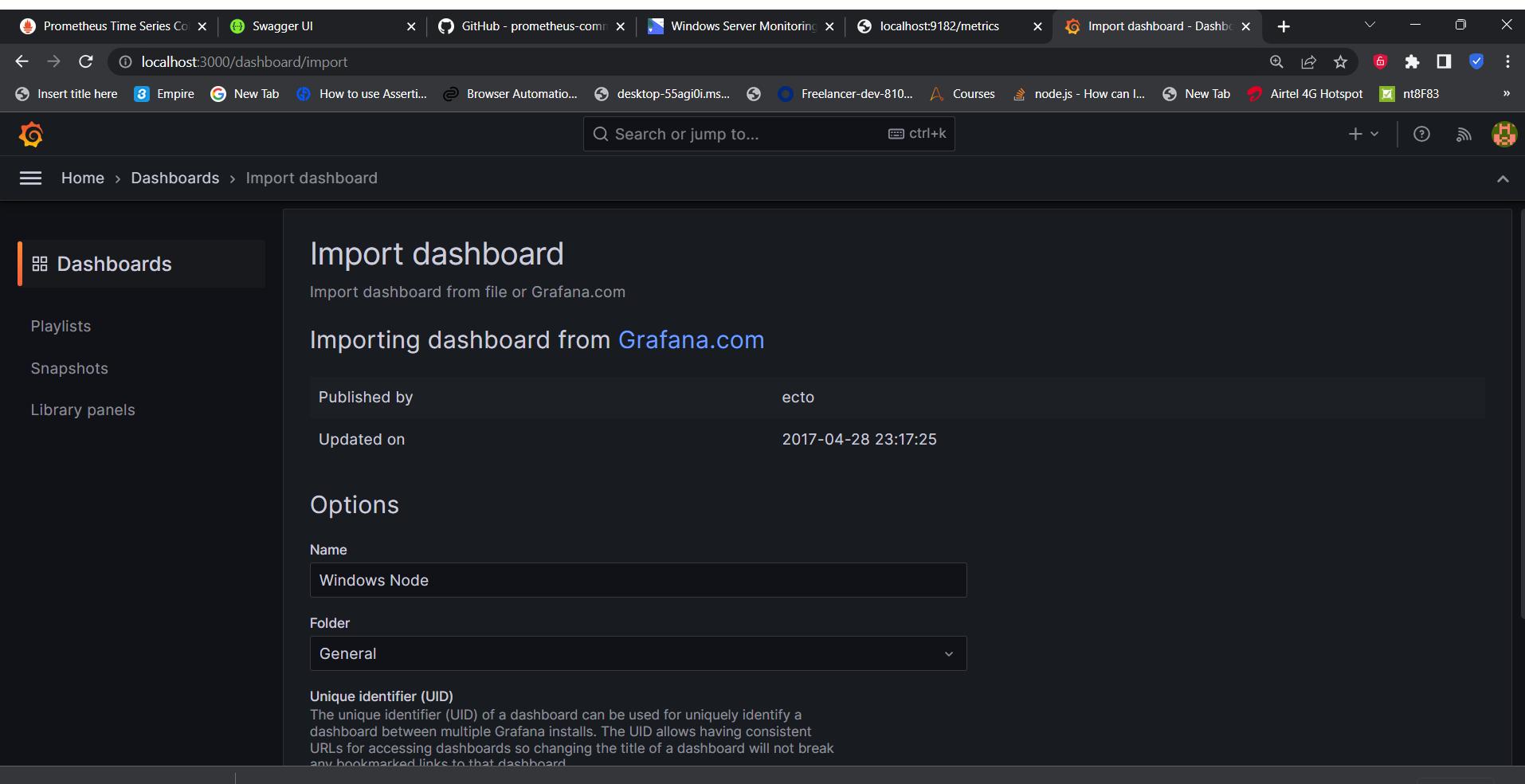
Import via panel json

31°C Mostly cloudy Show all

21:41 28/06/2023



Grafana





Example: The Prometheus Node Exporter

- The Prometheus node_exporter is officially distributed as a binary archive.
- Like other exporters, the node_exporter must be configured to listen on a dedicated port (9100 by default).
- For example, if a host is running a MySQL database, the official Prometheus MySQL exporter would expose metrics on port 9104 by default.
- Node_exporter would expose machine metrics on port 9100.
- Caution is advised to avoid prolonged runtimes (as measured by the built-in scrape_duration_seconds metric).



Statsd-Exporter

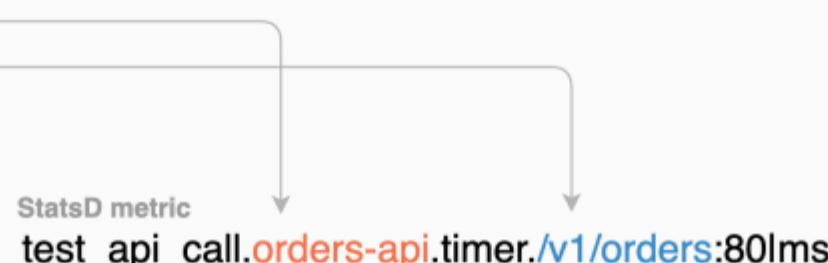
- The statsd-exporter is used to convert Statsd metrics into Prometheus metrics.
- This is because the StatsD text-format represents metrics differently to Prometheus' exposition format which is represented like this:
- <time series name>{<label name>=<label value>, ...}

Statsd-Exporter

Prometheus statsd-exporter mapping (YAML)

```
- match: "test_api_call.*.timer.*"
  name: "test_api_call"
  labels:
```

```
    api_name: "$1"
    api_endpoint: "$2"
```



Prometheus metrics (summary)

```
test_api_call{api_name="orders-api",api_endpoint="/v1/orders",quantile="0.5"} 0.08
test_api_call{api_name="orders-api",api_endpoint="/v1/orders",quantile="0.9"} 0.08
test_api_call{api_name="orders-api",api_endpoint="/v1/orders",quantile="0.99"} 0.08
test_api_call_sum{api_name="orders-api",api_endpoint="/v1/orders"} 0.7999999999999999
test_api_call_count{api_name="orders-api",api_endpoint="/v1/orders"} 10
```

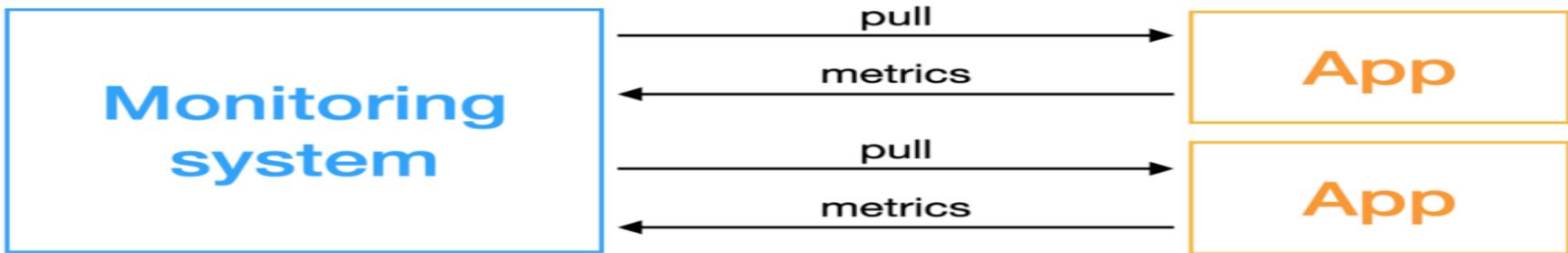
↑
time series name

↑
labels

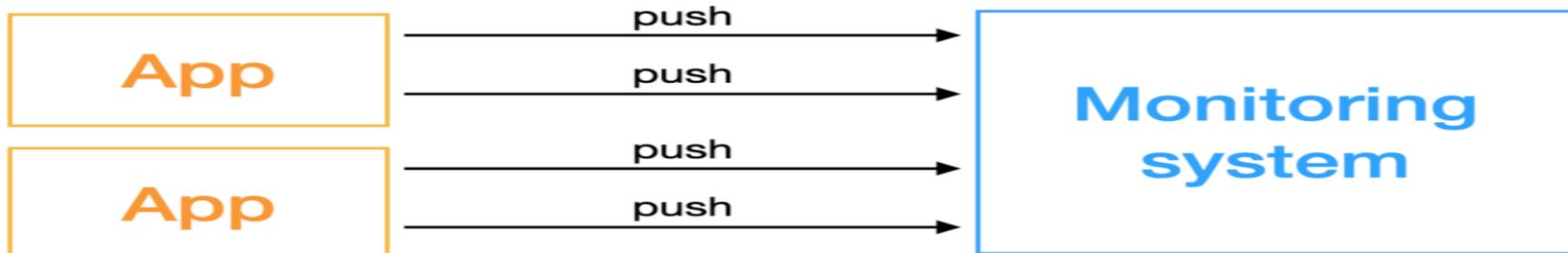


Types of Monitoring Systems

Pull-based system



Push-based monitoring system





Types of Monitoring Systems

- The Pull-based monitoring system actively obtains indicators, and the objects.
- Monitoring system should have the capability to be accessed remotely.
- Push-based monitoring systems do not actively obtain data, but the monitored objects actively push indicators.
- There are differences between the two methods in many aspects.
- For the construction and selection of monitoring systems, we must understand the advantages and disadvantages of these two methods in advance and choose the appropriate scheme for implementation.



Types of Monitoring Systems

Feature	Details	Pull Monitoring	Push Monitoring
Principle and Deployment	Configuration	Native centralized configuration	End configuration, support centralization through configuration center
	Monitoring object discovery	Depending on service discovery mechanisms, such as Zookeeper, Etcd, and Consul registries	Reported by applications and agents without service discovery module
	Deployment method	<ol style="list-style-type: none"> Application exposes port and access service discovery, with native support for Pull protocol Hosts, MySQL, NGINX, and other middleware rely on adapters (also known as exporters) to capture metrics and then provide Pull ports 	<ol style="list-style-type: none"> Unified proxy of Agent, capture data from host, MySQL, and other middleware and push to the monitoring system: Agent can also Applications proactively push to the monitoring system
Scalability	Scalability	Depending on Pull-end for extension; Pull Agent needs to be decoupled from storage (native Prometheus does not support); Push Agent is divided according to shards	Simple, agent can be scaled horizontally



Types of Monitoring Systems

Capabilities Comparison	Monitoring object survivability	Easy	Unable to find the reason behind why the object is not alive
	Data Completeness Calculation	1. Easy when deploying at the pull-end and storage coupling-end	Difficult
		2. Difficult in the distributed deployment of Pull Agent	
	Short lifecycle (Job, Serverless)/Real-time data acquisition	Difficult to apply	Applicable
	Metric acquisition flexibility	On-demand acquisition	Passive acceptance, requires some filters for additional support
	Application coupling	Applications are decoupled from the monitoring system. Applications do not need to care about the peer address of Push and handling Push errors	Higher coupling than Pull
Resource consumption	Resource consumption	1. Low resource consumption in the application exposing port mode	1. Low resource consumption in application push mode
		2. High resource consumption in Exporter mode	2. Low resource consumption in Agent mode (multiple systems can be collected at the same time)
Security assurance	The workload is heavy. It is necessary to ensure the security of the exposed port of the application and the security of the exporter port, which are vulnerable to DDoS attacks.	Low, agents and servers generally perform data transmission with encryption and authentication	



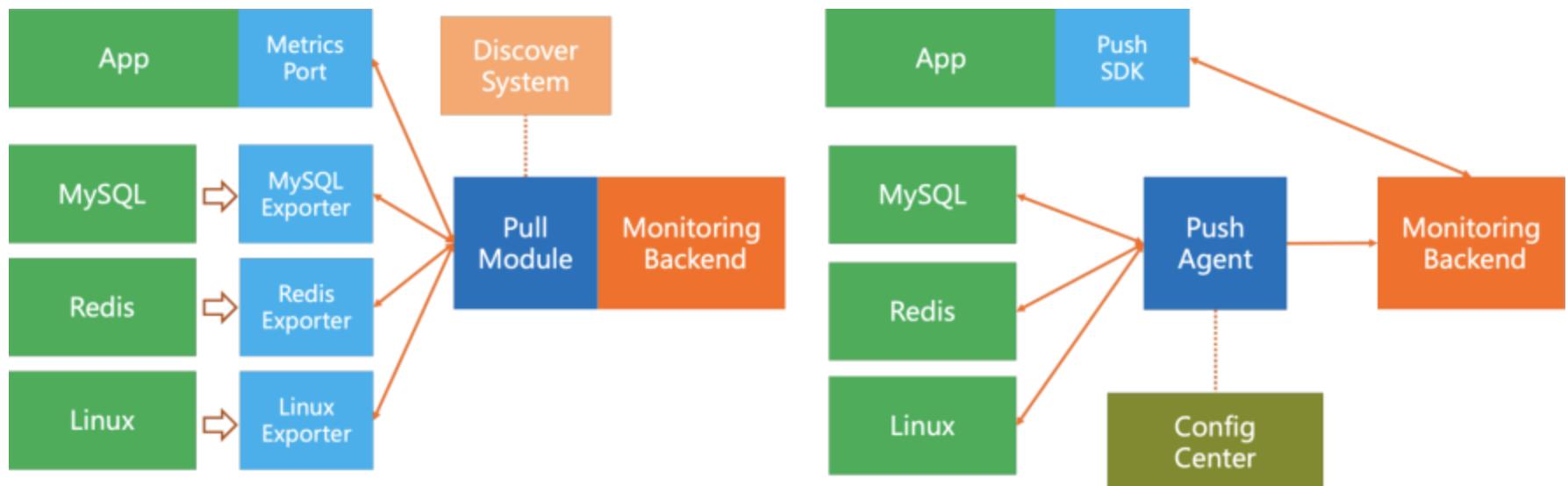
Types of Monitoring Systems

Machine and Labor Cost Core O&M consumption	1. Pull Agent stability and scale-out	1. Push Agent stability
	2. Server stability and scale-out	2. Server stability and scale-out
	3. Service discovery system stability	3. Configuration center stability and scale-out (optional)
	4. Exporter stability and scale-out	4. Network connectivity assurance (forward connectivity, relatively simple)
	5. Network connectivity assurance (reverse connectivity, cross-cluster, and network ACL)	



Types of Monitoring Systems

Pull Model Architecture





Types of Monitoring Systems

- Pull Architecture
- Service discovery system, including host service discovery (generally depends on the company's CMDB system), application service discovery (such as Consul), and PaaS service discovery (such as Kubernetes).
- The pull module needs to have the capability to connect to these service discovery systems.
- In addition to the service discovery part, the Pull core module generally uses the common protocol to pull data from the remote end.
- It usually supports configuration pull interval, timeout interval, metric filtering, rename, and simple process capabilities.
- The application-end SDK supports listening to a fixed port to provide the capability to be pulled.
- Since various middleware and other systems are incompatible with the Pull protocol, it is necessary to develop an Agent corresponding to Exporter to support pulling the metrics of these systems and provide standard Pull interfaces.



Types of Monitoring Systems

- Push Model Architecture
 - The Push model is simple and can be described as follows.
 - Push Agent supports pulling the metric data of various monitored objects and pushing them to the server.
 - It can be deployed in a manner coupled with the monitoring system or deployed separately.
 - ConfigCenter (optional) can provide centralized dynamic configuration capabilities, such as monitoring targets, collection intervals, metric filtering, metric processing, and remote targets.
 - The application-end SDK supports sending data to the monitoring backend or the local agent (usually, the local agent also implements a set of backend interfaces).



Prometheus Push gateway

- Prometheus is a primarily pull-based monitoring system.
- An additional component called the "Push gateway" is available for pushing metrics from external applications and services.
- The Push gateway is useful for collecting metrics from systems that are not compatible with the otherwise pull-based infrastructure.
- For instance, short-lived batch jobs that are ephemeral in nature may start and end before Prometheus can discover and scrape metrics from them.
- The Prometheus Pushgateway can be used to push the metrics of such processes to prevent losing essential data before they get a chance to get scraped.



Prometheus Client Libraries

- Applications only generate metrics after we add instrumentation to their code.
- We can do this via one of the Prometheus client libraries.
- Prometheus official client libraries are compatible with Java/JVM, Go, Python, and Ruby.
- Third-party client libraries are also available for Node.js, Haskell, C#/.Net, Common Lisp, Dart, Erlang, Rust, and more.
- Choose a Prometheus client library that matches the language of our application.

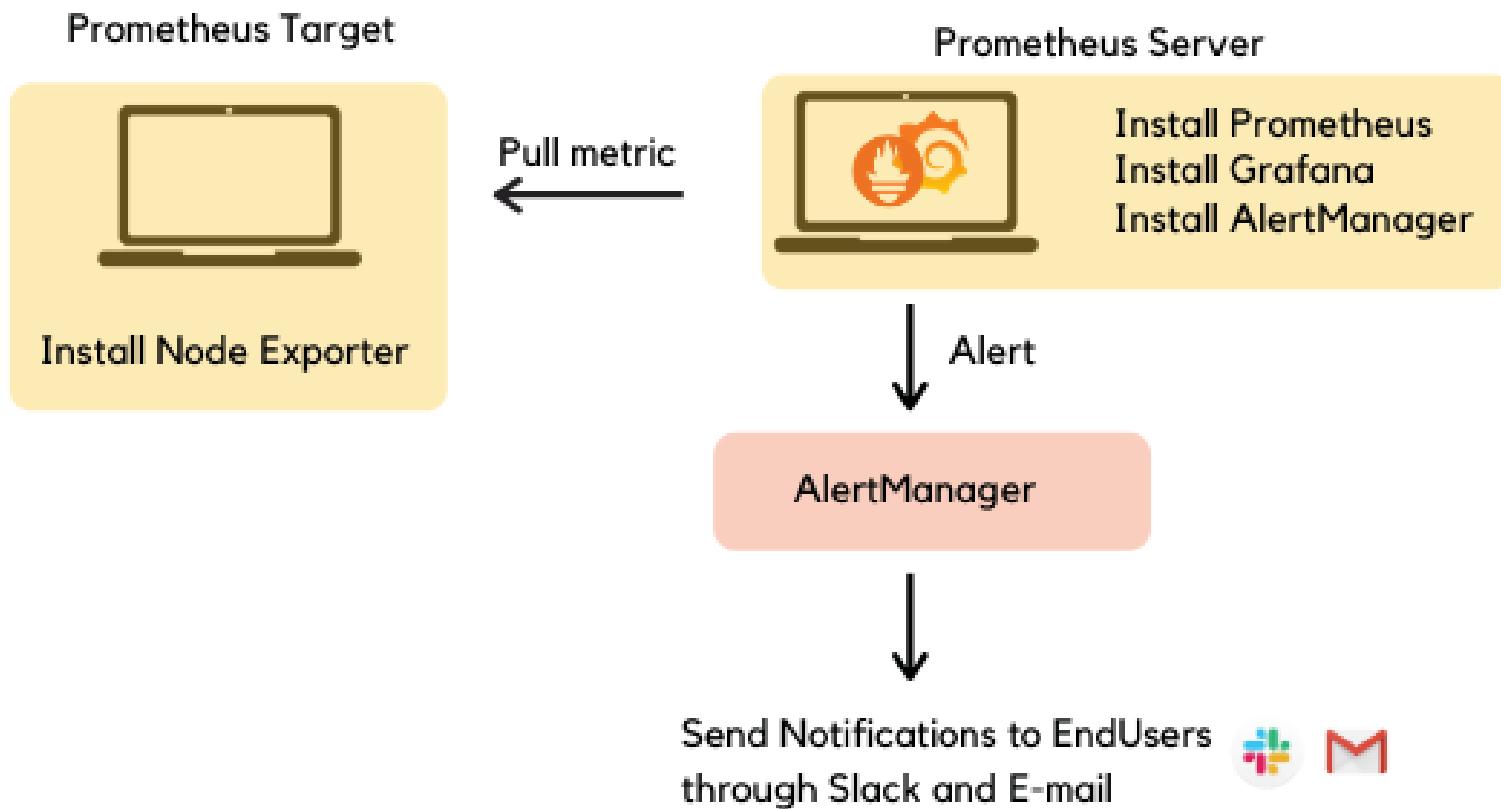


Alert Manager Architecture

- Prometheus Alert manager is a critical component of the Prometheus ecosystem responsible for handling alerts generated by Prometheus and managing how these alerts are delivered.
- The architecture of Prometheus Alert manager focuses on receiving, processing, and routing alerts based on defined rules.



Prometheus Alert manager





Alert Manager Architecture

- **1. Alert Generation in Prometheus**
- The process starts in **Prometheus**, where rules are defined to trigger alerts based on conditions from the data being scraped. These rules are defined in **alerting rules**, which continuously evaluate time series data for conditions like high CPU usage or disk space exhaustion.
- **Alerting Rules:** These are defined in a Prometheus configuration file. Each rule specifies:
 - **Condition:** The condition that needs to be met (e.g., CPU usage > 90% for 5 minutes).
 - **Labels:** Metadata about the alert (e.g., severity, instance).
 - **Annotations:** Additional information for the alert (e.g., description, runbook URL).
- Once the condition is met, the alerting rule triggers an alert, and Prometheus pushes these alerts to the **Alert manager**.



Alert Manager Architecture

- **2. Alert manager's Role**
- The Prometheus **Alert manager** is a standalone service that manages these alerts. It handles:
 - **Deduplication:** If Prometheus sends multiple alerts for the same condition, Alert manager deduplicates them to avoid multiple notifications for the same issue.
 - **Grouping:** Similar alerts can be grouped together based on labels (e.g., alerts from the same instance, same service, or same severity). This helps reduce noise and sends a single notification for multiple related alerts.
 - **Silencing:** Temporary suppression of alerts based on defined conditions (e.g., maintenance periods). Alerts matching the silence condition will not trigger notifications.

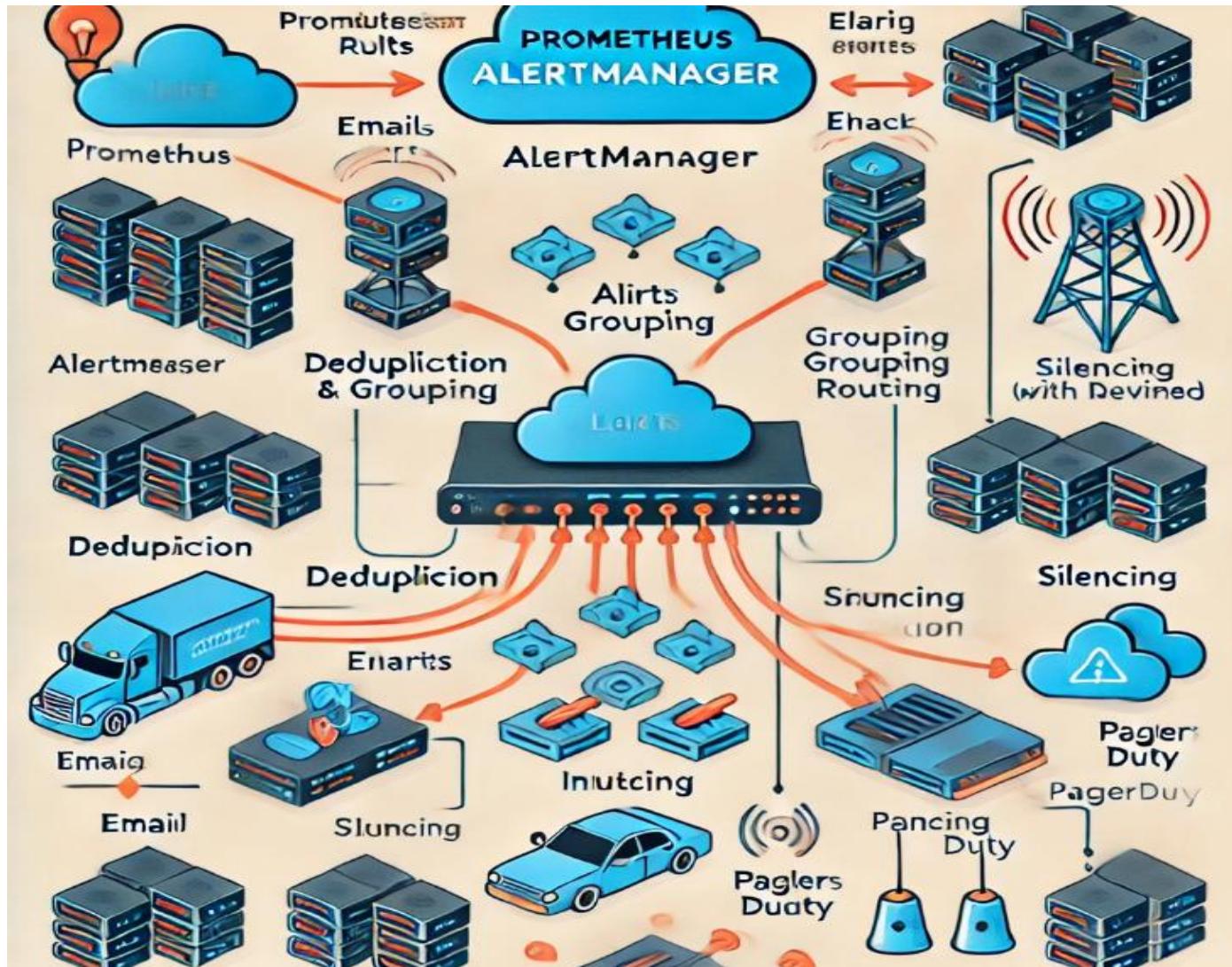


Alert Manager Architecture

- **Inhibition:** Prevents certain alerts from being sent if another alert is already firing (e.g., an "Instance Down" alert might inhibit related "Service Unavailable" alerts, as the root cause is the instance being down).
- **Routing:** Alert manager routes alerts to the right recipients based on rules defined in its configuration. It supports multiple notification integrations like:
 - Email
 - Slack
 - PagerDuty
 - Webhooks
 - Custom integrations



Alert Manager Architecture





Alert Manager Architecture

- 3. Alert manager Components
- a) Alert Receivers
 - These are systems or individuals who receive the alerts. Alerts can be sent to multiple receivers, such as:
 - Email
 - PagerDuty
 - Slack
 - Webhook (for custom systems)
 - OpsGenie
 - The routing of alerts to specific receivers is based on routing rules defined in the Alert manager configuration file (`alertmanager.yml`).



Alert Manager Architecture

- b) Notification Pipeline
 - Once an alert is routed, Alert manager sends notifications to the configured receivers.
 - These notifications can be customized with templates, and their frequency can be controlled to prevent alert flooding (via grouping and rate-limiting).
- c) Grouping and Deduplication
 - Grouping:
 - Alerts with the same label set (like service=backend or severity=critical) can be grouped together, reducing noise and allowing for more concise notifications.
 - Deduplication:
 - Alertmanager will suppress sending multiple identical alerts for the same issue, ensuring that the same alert doesn't overwhelm recipients.



Alert Manager Architecture

- d) Silencing and Inhibition

- Silencing:

- You can create silences to mute alerts during planned downtimes, maintenance, or other expected disruptions.
 - Silences can be configured to expire automatically after a certain time.

- Inhibition:

- This feature allows for defining relationships between alerts, where one alert can suppress another based on certain conditions.
 - For example, if a "Database Down" alert is firing, it can inhibit related service-level alerts since the root cause is the database being down.



Alert Manager Architecture

- 4. Configuration Files
- Alert manager is configured using a yaml configuration file (`alertmanager.yml`), which defines all of the key components, including:
 - Global settings: Timeouts, retry policies, etc.
 - Route settings: Rules to route alerts to the correct receivers.
 - Receivers: Definitions of where alerts should be sent (email, Slack, PagerDuty, etc.).
 - Inhibit rules: Definitions of which alerts should inhibit others.
 - Templates: Customize the format of the alert messages.



Alert Manager Architecture

- 5. Integration with Prometheus
- Alert API:
 - Prometheus pushes alerts to Alertmanager via the API (`/api/v1/alerts`).
 - Prometheus is responsible for sending alerts to Alert manager based on its alerting rules.
- Metrics Scraping:
 - Alertmanager exposes metrics (e.g., number of alerts, notifications sent) that can be scraped by Prometheus itself for monitoring the state of Alertmanager.



Alert Manager Architecture

- **6. Scaling and High Availability**
- **High Availability (HA)**: Alertmanager supports clustering for high availability.
- Multiple instances of Alertmanager can be deployed to form a cluster, and they will share state, allowing them to process alerts in a fault-tolerant way.
- Alerts will only be sent once by the cluster to avoid duplicates.
- Clustering is configured with peers that communicate with each other to synchronize alert states (active, silenced, etc.).
- **Replication**: When running multiple instances of Alert manager, alerts are replicated across instances in the cluster to ensure they are processed consistently.



Alert Manager Architecture

- **7. Alert manager Web UI**
- Alert manager provides a basic web interface that can be used to:
 - View active alerts.
 - Manage silences.
 - View inhibited alerts.
 - See the status of Alertmanager itself (e.g., active alerts, number of silences, number of notifications sent).



Alert Manager Architecture

- **Example Alertmanager Workflow**

1. **Prometheus** detects that a node's CPU usage has been above 90% for more than 5 minutes (based on the alerting rule).
2. **Prometheus** generates an alert and sends it to the **Alertmanager**.
3. **Alertmanager** receives the alert and:
 1. **Deduplicates**: If the same alert is repeatedly sent by Prometheus, Alertmanager deduplicates it.
 2. **Groups**: If there are other CPU-related alerts from different nodes, they might be grouped together.
 3. **Routes**: Based on routing rules, the alert is sent to the appropriate receiver (e.g., a Slack channel).
 4. **Notifies**: The alert is sent with the relevant metadata (severity, description, etc.) to the receiver.



Alert Manager Installation

2d3h					
prometheus-alertmanager-headless		ClusterIP	None	<none>	9093/TCP
2d3h					
prometheus-kube-state-metrics		ClusterIP	10.108.61.45	<none>	8080/TCP
2d3h					
prometheus-prometheus-node-exporter		ClusterIP	10.109.231.19	<none>	9100/TCP
2d3h					
prometheus-prometheus-pushgateway		ClusterIP	10.108.6.83	<none>	9091/TCP
2d3h					
prometheus-server		ClusterIP	10.101.15.213	<none>	80/TCP
2d3h					
C:\Windows\System32>kubectl port-forward -n prometheus svc/prometheus-alertmanager 9093:9093					
Forwarding from 127.0.0.1:9093 -> 9093					
Forwarding from [::1]:9093 -> 9093					
Handling connection for 9093					
Handling connection for 9093					
Handling connection for 9093					
Handling connection for 9093					
Handling connection for 9093					



Alert Manager Installation

Screenshot of a web browser showing the Alertmanager interface at localhost:9093/#/alerts.

The browser toolbar includes: Insert title here, Empire, New Tab, How to use Assertio..., Browser Automatio..., node.js - How can I..., Freelancer-dev-810..., Courses, New Tab, Airtel 4G Hotspot, Tryit Editor v3.6, All Bookmarks.

The Alertmanager navigation bar includes: Alertmanager, Alerts, Silences, Status, Settings, Help, and a New Silence button.

The main content area has tabs for Filter and Group. It shows a search input field, a "Custom matcher, e.g. env="production"" placeholder, and buttons for "+" and "Silence". Above the search input, there are filters for Receiver: All, Silenced, and Inhibited.

A message at the bottom states: "+ Expand all groups" and "No alert groups found".



Alerting Rules Overview

- Alerting rules are used in Prometheus to evaluate and trigger alerts based on certain conditions in your system metrics.
- These rules are created using PromQL (Prometheus Query Language), and they define when an alert should be fired if a specific condition is met.



Alerting Rules Overview

- **1. What are Alerting Rules?**
- Alerting rules allow Prometheus to evaluate conditions on metrics data and trigger alerts when certain conditions are met.
- Alerts are typically sent to an alert manager (like Prometheus Alertmanager) for further handling, which could include routing, silencing, grouping, or sending notifications to different systems like email, Slack, or PagerDuty.



Alerting Rules Overview

- **2. How Alerting Rules Work**
- Alerting rules are defined in YAML files, usually in Prometheus configuration files.
- They are evaluated periodically by Prometheus, based on their evaluation interval.
- If a defined condition is met, Prometheus "fires" the alert and sends it to Alertmanager.
- Alertmanager manages the alerts (e.g., de-duplicates, sends notifications, or handles escalations).



Alerting Rules Overview

3. Alerting Rule Syntax

Alerting rules are defined in a YAML format under `groups`. Here is an example of an alerting rule:

yaml

Copy code

```
groups:
- name: example-alerts
  rules:
  - alert: HighCPUUsage
    expr: 100 * (1 - avg by(instance)(irate(node_cpu_seconds_total{mode="idle"}[5m]))) > 90
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "Instance {{ $labels.instance }} CPU usage is high"
      description: "CPU usage on instance {{ $labels.instance }} is above 90% for the last 5 minutes"
```



Alerting Rules Overview

4. Components of an Alerting Rule

- `alert:`

This is the name of the alert, e.g., `HighCPUUsage`.

- `expr:`

The PromQL expression used to evaluate the alert condition. For example:

less

Copy code

```
100 * (1 - avg by(instance)(irate(node_cpu_seconds_total{mode="idle"}[5m]))) > 90
```

This expression triggers the alert if CPU usage exceeds 90%.

- `for:`

Duration for which the condition must be met before the alert is fired. For example, `5m` means the condition must hold true for 5 minutes before the alert is triggered. This is useful for avoiding false positives.



Alerting Rules Overview

- **labels:**

Labels provide additional metadata for the alert, such as severity level:

```
yaml
```

[Copy code](#)

```
severity: warning
```

These labels help Alertmanager classify and route alerts.

- **annotations:**

Annotations are additional, non-indexed pieces of information that can provide more context about the alert:

```
yaml
```

[Copy code](#)

```
summary: "Instance {{ $labels.instance }} CPU usage is high"
```

```
description: "CPU usage on instance {{ $labels.instance }} is above 90% for the last 5
```

These can be used to provide descriptions or other contextual information for notifications.



Alerting Rules Overview

5. Evaluation Interval

The frequency of evaluating an alerting rule is determined by the `evaluation_interval` setting in the Prometheus configuration. For example:

yaml

```
global:  
  scrape_interval: 15s  
  evaluation_interval: 15s
```

 Copy code

This configuration means that Prometheus will evaluate the alerting rules every 15 seconds.



Alerting Rules Overview

6. Best Practices for Writing Alerting Rules

- **Avoid Noise:** Write alerting rules that are meaningful and actionable. Too many alerts can overwhelm the team and reduce their effectiveness.
- **Use Sufficient `for:` Duration:** To avoid false positives, especially for short-lived issues, set a `for:` value to ensure the problem is consistent before firing the alert.
- **Group Alerts with Labels:** Labels help in grouping and categorizing alerts so they can be managed more effectively.
- **Use Annotations to Provide Context:** Adding summaries and descriptions can help quickly understand why an alert was triggered.



Alerting Rules Overview

6. Best Practices for Writing Alerting Rules

- **Avoid Noise:** Write alerting rules that are meaningful and actionable. Too many alerts can overwhelm the team and reduce their effectiveness.
- **Use Sufficient `for:` Duration:** To avoid false positives, especially for short-lived issues, set a `for:` value to ensure the problem is consistent before firing the alert.
- **Group Alerts with Labels:** Labels help in grouping and categorizing alerts so they can be managed more effectively.
- **Use Annotations to Provide Context:** Adding summaries and descriptions can help quickly understand why an alert was triggered.



Alerting Rules Overview

- High CPU Usage

yaml

 Copy code

```
- alert: HighCPUUsage
  expr: 100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: "High CPU usage on {{ $labels.instance }}"
    description: "The CPU usage is above 90% for more than 10 minutes."
```



Alerting Rules Overview

- Low Available Memory

yaml

 Copy code

```
- alert: LowAvailableMemory
  expr: node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes < 0.1
  for: 5m
  labels:
    severity: warning
  annotations:
    summary: "Low available memory on {{ $labels.instance }}"
    description: "Available memory is less than 10% on instance {{ $labels.instance }}
```



Alerting Rules Overview

- High Disk Usage

yaml

 Copy code

```
- alert: HighDiskUsage
  expr: (node_filesystem_size_bytes{mountpoint="/"} - node_filesystem_avail_bytes{mountpoint="/"}) > 85
  for: 15m
  labels:
    severity: warning
  annotations:
    summary: "High disk usage on {{ $labels.instance }}"
    description: "The disk usage has been above 85% for the past 15 minutes."
```



Alerting Rules Overview

8. Deploying Alerting Rules

You can create an alerting rules file and mount it as a configMap into Prometheus using Kubernetes.

Here is an example:

bash

 Copy code

```
kubectl create configmap prometheus-alert-rules --from-file=alert-rules.yaml
```

Then, mount this configMap into your Prometheus configuration.

```
kubectl create configmap prometheus-alert-rules --from-file=alert-rules.yaml
```



Alerting Rule in values.yml

```
serverFiles:
## Alerts configuration
## Ref: https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/
alerting_rules.yml:
groups:
- name: Instances
  rules:
  - alert: InstanceDown
    expr: up == 0
    for: 5m
    labels:
      severity: page
    annotations:
      description: '{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes.'
      summary: 'Instance {{ $labels.instance }} down'
## DEPRECATED DEFAULT VALUE, unless explicitly naming your files, please use alerting_rules.yml
alerts: {}

## Records configuration
## Ref: https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/
recording_rules.yml: {}
## DEPRECATED DEFAULT VALUE, unless explicitly naming your files, please use recording_rules.yml
rules: {}

prometheus.yml:
rule_files:
- /etc/config/recording_rules.yml
- /etc/config/alerting_rules.yml
```



Changes in Up History

```
_ changes(up[6h])
```

⋮

Execute

Table Graph Explain

< 2025-04-09 07:10:30 >

Load time: 20ms Result series: 13

```
{app_kubernetes_io_component="metrics", app_kubernetes_io_instance="prometheus", app_kubernetes_io_managed_by="Helm", app_kubernetes_io_name="kube-state-metrics", app_kubernetes_io_part_of="kube-state-metrics", app_kubernetes_io_version="2.15.0", helm_sh_chart="kube-state-metrics-5.31.0", instance="10.1.0.244:8080", job="kubernetes-service-endpoints", namespace="prometheus", node="docker-desktop", service="prometheus-kube-state-metrics"} 0

{app_kubernetes_io_component="metrics", app_kubernetes_io_instance="prometheus", app_kubernetes_io_managed_by="Helm", app_kubernetes_io_name="prometheus-node-exporter", app_kubernetes_io_part_of="prometheus-node-exporter", app_kubernetes_io_version="1.9.0", helm_sh_chart="prometheus-node-exporter-4.45.0", instance="192.168.65.3:9100", job="kubernetes-service-endpoints", namespace="prometheus", node="docker-desktop", service="prometheus-prometheus-node-exporter"} 0

{beta_kubernetes_io_arch="amd64", beta_kubernetes_io_os="linux", instance="docker-desktop", job="kubernetes-nodes", kubernetes_io_arch="amd64", kubernetes_io_hostname="docker-desktop", kubernetes_io_os="linux"} 0

{beta_kubernetes_io_arch="amd64", beta_kubernetes_io_os="linux", instance="docker-desktop", job="kubernetes-nodes-cadvisor", kubernetes_io_arch="amd64", kubernetes_io_hostname="docker-desktop", kubernetes_io_os="linux"} 0

{instance="10.1.0.246:9153", job="kubernetes-service-endpoints", k8s_app="kube-dns", kubernetes_io_cluster_service="true", kubernetes_io_name="CoreDNS", namespace="kube-system", node="docker-desktop", service="kube-dns"} 0

{instance="10.1.0.248:9153", job="kubernetes-service-endpoints", k8s_app="kube-dns", kubernetes_io_cluster_service="true", kubernetes_io_name="CoreDNS", namespace="kube-system", node="docker-desktop", service="kube-dns"} 0

{instance="192.168.0.108:9182", job="windowsexporter"} 0
```

8

FNC

15:07



Alerting Rule in values.yml

Prometheus Alerts Graph Status ▾ Help ⚙️ ⚡ ⏱ ⏴

Inactive (0) Pending (1) Firing (0) Show annotations

/etc/config/alerting_rules.yml > Instances pending (1)

▼ InstanceDown (1 active)

```
name: InstanceDown
expr: up == 0
for: 1m
labels:
  severity: page
annotations:
  description: {{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes.
  summary: Instance {{ $labels.instance }} down
```

Labels	State	Active Since	Value
alertname=InstanceDown instance=host.docker.internal:7074 job=customer-app severity=page	PENDING	2024-10-22T18:44:14.323620173Z	0



Slack Receiver

training



Messages Add canvas +

This channel is for everything #training. Hold meetings, share docs and make decisions together with your team. [Edit description](#)

Add colleagues

Today ▾

 **Parameswari Bala** 19:29
joined #training.

 **Parameswari Bala** 19:41
added an integration to this channel: [incoming-webhook](#)

 **incoming-webhook** APP 20:04
Hello, World!

Hello, World!



 Hello, team!

Let's use this space to...

Status update:

x



Slack Receiver

vebconsulting ▾

Channels

all-vebconsulting

social

training

+ Add channels

Direct messages

Parameswari Bala

Parameswari Bala you

+ Add colleagues

training

Messages Add canvas +

Today ▾

Grafana APP 06:42 [FIRING:1] cpu_usage_alert cpu_usage_behaviour (host.docker.internal:7074 customer-app)
Firing
Value: C=1, CPU_USAGE_QUERY=0.006517239860848122
Labels:
- alertname = cpu_usage_alert
- grafana_folder = cpu_usage_behaviour
- instance = host.docker.internal:7074
Show more
Grafana v11.2.2+security-01 | Today at 06:42

[FIRING:1] DatasourceNoData promqlevaluator (de1m96lsbrojkc Evaluator CPU_Usage_Alert)
Firing

Hello, team! Let's use this space to... Status update: ×



Slack Receiver

vebconsulting ▾

Channels

all-vebconsulting

social

training

+ Add channels

Direct messages

Parameswari Bala

Parameswari Bala you

+ Add colleagues

Messages Add canvas +

Add colleagues

Today

Parameswari Bala 19:29 joined #training.

Parameswari Bala 19:41 added an integration to this channel: incoming-webhook

incoming-webhook APP 20:04 Hello, World!

Hello, World!

Sending Test Messages!

Hello, team! Let's use this space to... Status update: x

A screenshot of a Slack interface showing the '# training' channel. The sidebar on the left lists channels like '# all-vebconsulting', '# social', and '# training'. The '# training' channel is selected and highlighted with a yellow bar. The channel page shows messages from Parameswari Bala and an 'incoming-webhook' app. The messages from Parameswari Bala indicate she joined the channel and added an integration. The messages from the 'incoming-webhook' app show 'Hello, World!' and 'Sending Test Messages!'. At the bottom, there are input fields for 'Hello, team!', 'Let's use this space to...', and 'Status update:'.



Webhook Receiver

Administrator: Command Prompt - kubectl port-forward svc/nodejs-service 4000:80					
prometheus-operated		ClusterIP	None	<none>	9
090/TCP	165m				
prometheus-operator		ClusterIP	None	<none>	8
080/TCP	165m				
prometheus-operator-grafana		ClusterIP	10.97.67.35	<none>	8
0/TCP	167m				
prometheus-operator-kube-p-alertmanager		ClusterIP	10.102.8.24	<none>	9
093/TCP,8080/TCP	167m				
prometheus-operator-kube-p-operator		ClusterIP	10.107.146.240	<none>	4
43/TCP	167m				
prometheus-operator-kube-p-prometheus		ClusterIP	10.111.74.17	<none>	9
090/TCP,8080/TCP	167m				
prometheus-operator-kube-state-metrics		ClusterIP	10.109.96.44	<none>	8
080/TCP	167m				
prometheus-operator-prometheus-node-exporter		ClusterIP	10.98.86.227	<none>	9
100/TCP	167m				


```
E:\prometheusjava\ webhookdemo>kubectl port-forward svc/nodejs-service 4000:80
Forwarding from 127.0.0.1:4000 -> 4000
Forwarding from [::1]:4000 -> 4000
Handling connection for 4000
```



Webhook Receiver

```
Administrator: Command Prompt
prometheus-prometheus-operator-kube-p-prometheus-0      0/2      Pending      0
  0s

F:\Local disk\prometheus\PrometheusGrafana_Latest\prometheushelm>kubectl logs nodejs-deployment-bc5fc9c7-2ptxb
Server is running on port 4000
Received Webhook: {
  receiver: 'web-hook',
  status: 'firing',
  alerts: [
    {
      status: 'firing',
      labels: [Object],
      annotations: [Object],
      startsAt: '2024-10-23T17:30:14.323Z',
      endsAt: '0001-01-01T00:00:00Z',
      generatorURL: 'http://prometheus-server-6459547474-kmpdt:9090/graph?g0.expr=up%3D%3D+0&g0.tab=1',
      fingerprint: '1a08cdb1dfbab6fe'
    }
  ],
  groupLabels: { alertname: 'InstanceDown' },
  commonLabels: {
```



Search



ENG
IN



23:28
23-10-2024



Pager Duty

PagerDuty

Incidents Services People Automation AIOps Analytics Integrations Status

Search



Service Directory

A service in PagerDuty represents a component, microservice or piece of infrastructure a team operates, manages, and monitors. Usually it's something you'd go on call for. [Learn more about the service directory.](#)

Services

Maintenance Windows

+ New Service

Create a Service

Name

Assign

Reduce Noise

4 Integrations

Integrations

Alert feeds can come into PagerDuty from a number of sources. We apply our AI to these alerts and can trigger incidents and notify the right people at the right time.

Select the integration(s) you use to send alerts to this service

Search for an integration(s)



Most popular integrations



Events API V2



Prometheus



Amazon
CloudWatch



Splunk



Nagios



Pager Duty

PagerDuty

Incidents Services People Automation AIOps Analytics Integrations Status

Search



Service Directory

A service in PagerDuty represents a component, microservice or piece of infrastructure a team operates, manages, and monitors. Usually it's something you'd go on call for. [Learn more about the service directory.](#)

+ New Service

Services

Maintenance Windows

prometheus service	TEAM <i>No team is assigned to the prometheus service - ep escalation policy.</i>	ON CALL NOW PARAMESWARI E	LAST INCIDENT <i>No incidents so far.</i>	OPEN INCIDENTS 0 triggered 0 acknowledged	 2/9 Standards met	More ▾
------------------------------------	--	--	--	---	-----------------------	------------------------

[Activity](#)[Integrations](#)[Workflows](#)[Settings](#)[Service Dependencies](#)

Integrations (1)

Follow the steps below to integrate and test each integration, or [add an integration](#).

 **Prometheus** No Test Alert Received  

Prometheus Integration Guide

Prometheus + PagerDuty Benefits

- Send richly formatted event data from Prometheus to PagerDuty, allowing you to engage the right people, accelerate resolution and improve learning.
- Create high and low urgency incidents based on the severity of the event from the Prometheus event payload.

How it Works

- Prometheus sends events to PagerDuty via a Prometheus Alertmanager configuration file. Events from Prometheus will trigger a new incident on the corresponding PagerDuty service, or group as alerts into an existing incident.
- You can resolve PagerDuty incidents from the Prometheus server as

Integration Name 

Integration Key 

Integration URL 

Pager Duty



localhost:9093/#/alerts

Insert title here Empire New Tab How to use Assertio... Browser Automatio... node.js - How can I... Freelancer-dev-810... Courses All Bookmarks

Filter Group Receiver: All Silenced Inhibited Muted

Custom matcher, e.g. env="production"

+ Expand all groups

- pager-duty-notifications alertname="HighProcessCPUUsage" + 1 alert

2025-03-06T12:08:14.323Z + Info Source Silence Link

instance="host.docker.internal:7076" + job="customer-app" + severity="warning" +

- slack-notifications alertname="InstanceDown" + 2 alerts

2025-03-06T06:30:14.323Z + Info Source Silence Link

app_kubernetes_io_component="metrics" + app_kubernetes_io_instance="prometheus" + app_kubernetes_io_managed_by="Helm" +

app_kubernetes_io_name="prometheus-node-exporter" + app_kubernetes_io_part_of="prometheus-node-exporter" +

Pager Duty



! Awaiting response	PARAMESWARI E	prometheus service-ep	No team is assigned to the prometheus service-ep escalation policy.	<i>No channel for this service.</i> Add one.
--	---------------	-----------------------	---	---

[Activity](#)[Integrations](#)[Workflows](#)[Settings](#)[Service Dependencies](#) 1+

Open Incidents (2)

[! Acknowledge](#)[✓ Resolve](#)[⌚ Snooze ▾](#)[All statuses ▾](#)[Go to incident #](#)

25 per page ▾ 1 - 2 of 2

< >

[Merge Incidents](#)

<input type="checkbox"/>	Status	Priority ▾	Urgency	Alerts	Title	Assigned To	Created ▾
<input type="checkbox"/>	Triggered		High	1	[FIRING:1] HighProcessCPUUsage (host.docker.internal:7076 customer-app warning) #4 + SHOW DETAILS (1 triggered alert)	PARAMESWARI E	Today at 12:17 PM
<input type="checkbox"/>	Triggered	P1	High	0	Prometheus incident #3 + SHOW DETAILS	PARAMESWARI E	Today at 11:59 AM

Pager Duty



outlook.office.com.mcas.ms/mail/inbox/id/AAQkAGFiZWQ1MDZhLTkzZjEtNDNmNi05NDI2LTY5M2Q3ZGQ0MmlzMQAQAD...

Insert title here Empire New Tab How to use Assertio... Browser Automatio... node.js - How can I... Freelancer-dev-810... Courses All Bookmarks

Outlook Search

Home View Help

New mail

Favorites

- Inbox 6285
- Sent Items
- Drafts 5
- Deleted Items 88

Folders

- Inbox 6285
- Drafts 5
- Sent Items
- Deleted Items 88
- Junk Email 3
- Notes
- Archive
- Conversation History

Focused Other

Other: New messages (15)
Google Cloud; Chargebee; Splunk AppDynamics; Team ...

Today

PagerDuty [PagerDuty ALERT] You have 4 TRIGG... 5:47 PM
trainingmar2025.pageerduty.com Hello PARAMES...

PagerDuty [PagerDuty ALERT] You have 3 TRIGG... 5:29 PM
trainingmar2025.pageerduty.com Hello PARAMES...

PagerDuty [PagerDuty ALERT] You have 2 TRIGG... 5:11 PM
trainingmar2025.pageerduty.com Hello PARAMES...

PagerDuty Welcome to PagerDuty. Get started h... 5:10 PM
View in Browser | Forward to a Friend Hello and ...

PagerDuty [PagerDuty ALERT] You have 1 TRIGGER... 5:09 PM
trainingmar2025.pageerduty.com Hello PARAMES...

[PagerDuty ALERT] You have 4 TRIGGERED Incidents (4f500)

PagerDuty <no-reply@pagerduty.com>
To: parameswari bala
Thu 3/6/2025 5:47 PM

Some content in this message has been blocked because the sender isn't in your Safe senders list.

Trust sender Show blocked content

PagerDuty trainingmar2025.pageerduty.com

Hello PARAMESWARI E, you have 4 incidents assigned to you:

INCIDENT #4

[FIRING:1] HighProcessCPUUsage (host.docker.internal:7076 customer-app warning)

[View Incident](#)

DETAILS

firing: Labels: - alertname = HighProcessCPUUsage - instance = host.docker.internal:7076 - job = customer-app - severity = warning Annotations: - description = Job Request host.docker.internal:7076 is using high cpu for the last 1 minute. - summary = Job Request host.docker.internal:7076 is using high cpu Source: http://prometheus-server-6c586d99cd-nhmf:9090/graph?g0.expr=rate%28process_cpu_usage%5B5m%5D%29+%2A+100%+3E+0.0001&g0.tab=1 num_firing: 1 num_resolved: 0 ...

← → ⌂ ⌂ eswaribala70.app.opsgenie.com/teams/dashboard/ea67e8b9-2e1d-4ae0-beb6-5eccce9ac0e8/main

Insert title here Empire New Tab How to use Assertio... Browser Automatio... node.js - How can I... Freelancer-dev

Atlassian uses cookies to improve your browsing experience, perform analytics and research, and conduct advertising. Accept all cookies to indicate that you agree to our use of cookies on your device. [Atlassian cookies and tracking notice](#)

Opsgenie Alerts Incidents Who is on-call Teams Services Analytics Settings

HpTeamAPR2025

On-call

Integrations

Services

Members

Activity stream

i Your trial will expire in 13 days.

Teams / HpTeamAPR2025

On-call

<https://eswaribala70.app.opsgenie.com/quick-start-guide>

Routing rules

Route alerts to HpTeamAPR2025_escalation

Add integration

Integrate with powerful applications
to sync alert data and to streamline your IT operation workflows.



ALL PRODUCTS

API



Prometheus

OpgPrometheus

[Turn on integration](#)

Close

...

Integration settings

[Learn how to integrate with Prometheus](#)

Status OFF

[Edit](#)

Assignee team HpTeamAPR2025

API key 14ed2176-f589-496c-944c-0717db5402bc

Notifications ON

Set up your integration

We've got the basic settings for the integration covered. Customise them to make sure you get the most out of the integration.

[OK](#)

cessing alerts

inst events in the integrated applications. You can set rules to create an alert, acknowledge an alert, add a note, or close an alert. This integration comes with a default set of rules. [Read more about incoming integrations](#).

← → ⌛ 🏠 eswaribala70.app.opsgenie.com/alert/detail/753f16e1-43ae-40bf-a43a-4cb5c5adfb6-1744818129495/details ⭐ 🌐 📁

Insert title here 3 Empire G New Tab F How to use Assertio... 🎯 Browser Automatio... node.js - How can I... Freelancer-dev-810... A Courses

Opsgenie Alerts Incidents Who is on-call Teams Services Analytics Settings 33% 🌐

Your trial will expire in 13 days.

Alerts

P3 Apr 16, 2025 9:12 PM (GMT+05:30) No Owner [Prometheus]: [FIRING:2] HighDiskUsage (/dev/sdc ext4 /run/desktop/mnt warning)

x1 + Add tag #1

Details Activity log Responder states

Source	http://prometheus-alertmanager-0:9093/#/alerts?receiver=opsgenie-receiver
Integration	OpgPrometheus (Prometheus)
Responders	HpTeamAPR2025
Owner Team	HpTeamAPR2025
Alias	4bdccc20bf53ee852655783029b75bb19a998feed70a6c54d7b5a3258c323ae5
Last Updated At	Apr 16, 2025 9:28 PM (GMT+05:30)
Description	Alerts Firing: Labels: - alertname = HighDiskUsage - app_kubernetes_io_component = metrics - app_kubernetes_io_instance = prometheus

MS Teams



Chat ... 🔍 ⌂ ⌂

Unread Channels Chats Meeting chats

Copilot Discover Mentions Followed threads

Favourites

Chats

Teams and channels

Meet General

RPS Consulting Pvt.Ltd General

training

See all your teams

grafanachannel Posts Files Notes +

Unknown user 15:15

[FIRING:1] TestAlert Test Folder Grafana

Firing

Value: B=22, C=1 Labels:

- alertname = TestAlert
- grafana_folder = Test Folder
- instance = Grafana

Annotations:

- summary = Notification test

Source: ?orgId=1

Silence: http://localhost:3000/alerting/silence/new?alertmanager=grafana&matcher=alertname%3DTestAlert&matcher=grafana_folder%3DTest+Folder&matcher=instance%3DGrafana&orgId=1

Dashboard: http://localhost:3000/d/dashboard_uid?from=1759308334202&orgId=1&to=1759311934205

Panel: http://localhost:3000/d/dashboard_uid?from=1759308334202&orgId=1&to=1759311934205&viewPanel=1

Post in channel

MS Teams



Home > Alerting > Contact points Search...

Choose how to notify your contact points when an alert instance fires

Create contact point

Name *

Integration

Microsoft Teams Test

URL

<https://rpsconsulting.webhook.office.com/webhookb2/e5678fe3-dde6-4a2d>

Optional Microsoft Teams settings

Notification settings

+ Add contact point integration

Save contact point Cancel



Inhibition

- Alert inhibition is a feature that suppresses specific alerts when related higher-priority alerts are already firing.
- For example, if you've set up your Prometheus server to warn at 70% memory usage and alert at 90%, you will get two alerts if your application rapidly spikes to 95%.
- The 70% warning will be suppressed with inhibition since it is redundant when the critical 90% alert is already firing.



Inhibition

- `inhibit_rules:`
- - `source_matchers:`
- - `alername = HighMemoryUsage`
- - `target_matchers:`
- - `alername = MemoryUsageWarning`
- `equal: ['instance', 'job']`



Inhibition

- The `source_matchers` specifies the alerts that will suppress others (inhibiting alerts), while `target_matchers` specifies the alerts that will be suppressed.
- The `equal` property then defines the labels that must match between the source and target alerts for inhibition to apply.
- This essentially means that if an alert with `alertname=HighMemoryUsage` is firing for a specific instance and job, it will inhibit any `MemoryUsageWarning` alert for the same instance and job.



Inhibition

- The `source_matchers` specifies the alerts that will suppress others (inhibiting alerts), while `target_matchers` specifies the alerts that will be suppressed.
- The `equal` property then defines the labels that must match between the source and target alerts for inhibition to apply.
- This essentially means that if an alert with `alertname=HighMemoryUsage` is firing for a specific instance and job, it will inhibit any `MemoryUsageWarning` alert for the same instance and job.

Inhibition

alerts.yml

```
groups:
- name: Example group
  rules:
    - alert: HighMemoryUsage # The rule that suppresses should come before the
      expr: 1
      for: 5m
      labels:
        inhibit: "true" ✓
      annotations:
        summary: "This is an inhibiting rule"
    - alert: MemoryUsageWarning
      expr: 1
      for: 5m
      labels:
        inhibited: "true" ✓
      annotations:
        summary: "This is an inhibited rule"
```

Inhibition



Filter by rule state ▾ Filter by rule name or labels

Instances /etc/config/alerting_rules.yml

Rule	Status	Count
InstanceDown		FIRING (2)
HighCPUUsage		PENDING (1)
LowAvailableMemory		FIRING (1)
HighMemoryUsage		FIRING (1)

Inhibition



Alertmanager Alerts Silences Status Settings Help

New Silence

Filter Group Receiver: All Silenced Inhibited Muted

Custom matcher, e.g. `env="production"`

[+](#) [Silence](#)

[+](#) Expand all groups

— pager-duty-notifications alertname="LowAvailableMemory" [+](#) 1 alert

2025-04-10T13:10:14.323Z [+ Info](#) [Source](#) [Silence](#) [Link](#)

[inhibit="true"](#) [+](#) [instance="prometheus-prometheus-node-exporter:9100"](#) [+](#) [job="node_exporter"](#) [+](#) [severity="critical"](#) [+](#)



Silencing alerts

- Alert silencing allows you to temporarily mute specific alerts based on predefined criteria.
- It is particularly useful for avoiding unnecessary notifications during planned maintenance, known issues, or when testing.
- A silence in Alertmanager is a set of matchers (conditions) that suppress alerts for a specified period.
- Alerts that match the silence criteria will not trigger notifications to the configured receivers during the silencing period.



Silencing alerts

New Silence

Start

2025-04-10T13:14:36.765Z

Duration

2h

End

2025-04-10T15:14:36.765Z

**Matchers** Alerts affected by this silence +Custom matcher, e.g. `env="production"`**Creator****Comment**



Grafana JSON API with JWT Token

Home > Connections > Data sources > marcusolsson-json-datasource-1

Search or jump to...

Connection

URL * <http://host.docker.internal:7072/metrics/v1.0>

Authentication

Authentication methods

Choose an authentication method to access the data source

No Authentication

TLS settings

Additional security measures that can be applied on top of authentication

- Add self-signed certificate ⓘ
- TLS Client Authentication ⓘ
- Skip TLS certificate validation ⓘ

HTTP headers

Pass along additional context and metadata about the request/response

Header Authorization Value configured Reset ⌫

Beamer ↗

Grafana JSON API with JWT Token



Home > Dashboards > New dashboard > Edit panel

Search or jump to... ctrl+k

Back to dashboard Discard panel Save dashboard

Table view Last 6 hours Refresh

Time series

Search options

All Overrides

Panel options

Title Panel Title

Description

Transparent background

Panel links

Repeat options

Queries 1 Transformations 0 Alert 0

(marcusoisonn-json-datasource-1)

Cache Time 5m

Fields Path Params Headers Body Experimental

Field	Type	Alias	+	-
\$[*].catalogId	JSONPath	Auto		
\$[*].catalogName	JSONPath	Auto		

303

The screenshot shows the Grafana interface for editing a dashboard. A table panel is displayed, showing two rows of data from a JSON API response. The columns are labeled 'catalogId' and 'catalogName'. The first row has an ID of 1 and a name of 'Electronics'. The second row has an ID of 2 and a name of 'Furniture'. Below the table, there are sections for 'Queries', 'Transformations', and 'Alerts'. The 'Queries' section is active, showing one query named '(marcusoisonn-json-datasource-1)'. The 'Fields' tab is selected, displaying two fields: '\$[*].catalogId' and '\$[*].catalogName'. Both fields are set to 'JSONPath' type and 'Auto' alias. The 'Cache Time' is set to 5m. On the right side, there are various panel configuration options like 'Title', 'Description', and 'Transparent background'. The bottom right corner shows the page number 303.



Configuring Prometheus to monitor itself

- Prometheus collects metrics from targets by scraping metrics HTTP endpoints.
- Since Prometheus exposes data to itself, it can also scrape and monitor its own health.



Configuring Prometheus to monitor itself

```
global:
  scrape_interval:      15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```



Custom Metrics

- We can also define some custom metrics.
- The demo contains a Scheduler class which periodically runs the included scheduling Task method.
- To be able to send custom metrics we need to import Meter Registry from the Micrometer library and inject it into our class.



Custom Metrics

- It is possible to instantiate these types of meters from MeterRegistry:
 - Counter: reports merely a count over a specified property of an application
 - Gauge: shows the current value of a meter
 - Timers: measures latencies or frequency of events
 - DistributionSummary: provides distribution of events and a simple summary
 - **Refer implementation of a counter and a gauge for demonstration purposes**

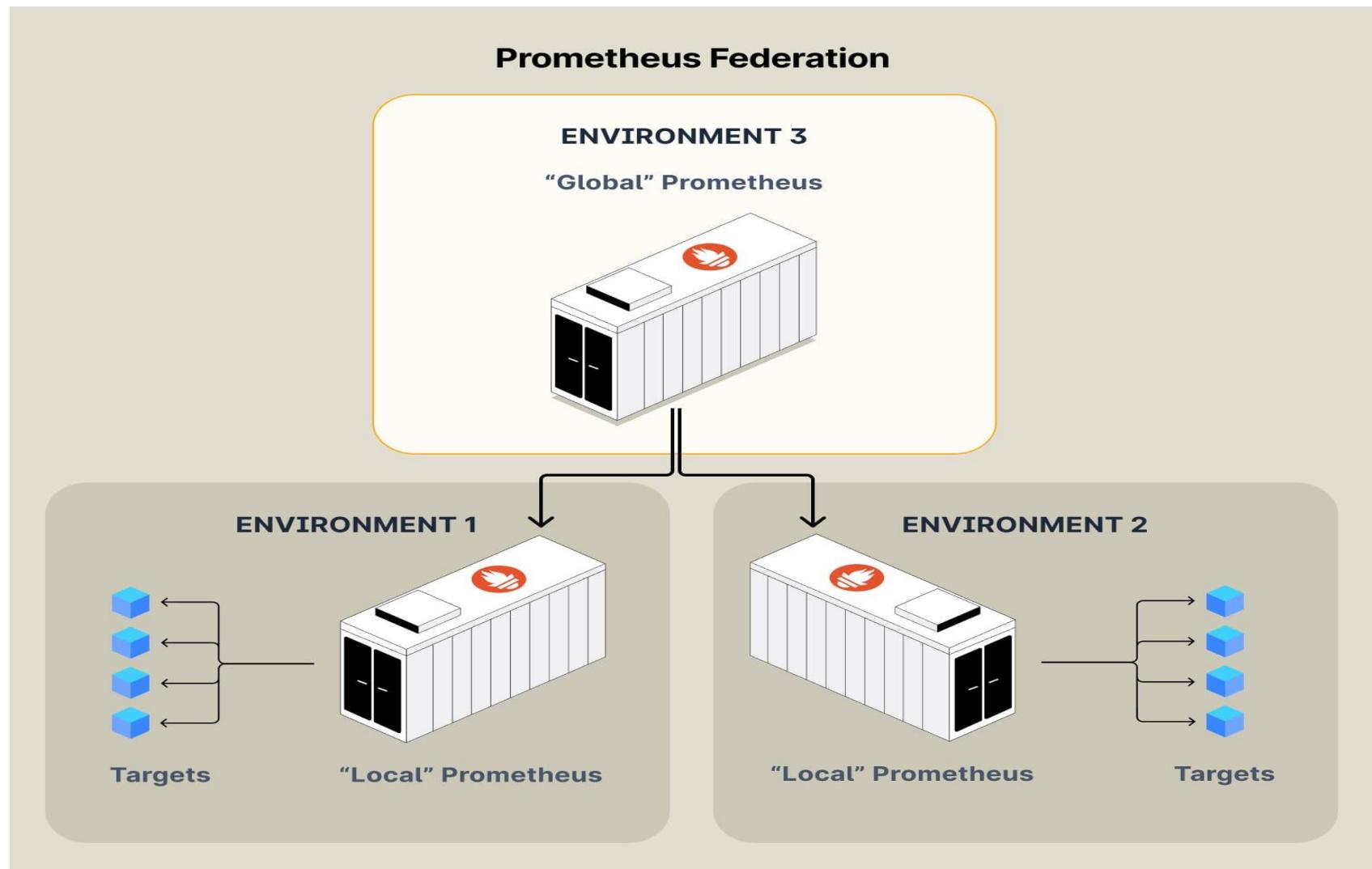


Prometheus Federation

- Federation is needed to handle complex and large infrastructures efficiently, where a single Prometheus instance would be insufficient due to load or geographical distribution.
- A federation strategy in Prometheus is typically required in environments that are either large-scale, geographically distributed, or have complex infrastructure setups.



Prometheus Federation





Prometheus Federation

- This includes scenarios like:
 - Multi-Datacenter Operations: Organizations operating across multiple data centers, needing aggregated views of their distributed systems.
 - Large Enterprises: Large enterprises with extensive infrastructure, where a single Prometheus instance would be insufficient due to the sheer volume of data.
 - High Availability Requirements: Systems requiring high availability, where monitoring must be robust against individual instances' failures.



Prometheus Federation

- This includes scenarios like:
 - Complex Service Architectures: Environments with complex service architectures where specific cross-service metrics must be aggregated for a comprehensive overview.
 - Multi-Cluster K8s Architectures: Each Kubernetes cluster has its own Prometheus instance (aka “leaf” node), which scrapes metrics from that specific cluster. Then, a higher-level, or “global,” Prometheus instance is used to federate data from these “leaf” nodes.



Prometheus Federation

- helm install secondary-prometheus prometheus-community/prometheus --namespace prometheus-1 --create-namespace
- helm install secondary-prometheus prometheus-community/prometheus --namespace prometheus-2 --create-namespace



How to configure federation in Prometheus

```
global:  
  scrape_interval: 15s  
  
scrape_configs:  
  - job_name: 'global-view'  
    honor_labels: true  
    metrics_path: '/federate'  
    params:  
      'match[]':  
        - '{job="app1-aggregate"}'  
        - '{job="app2-aggregate"}'  
    static_configs:  
      - targets:  
          - 'prom.domain1.com:9090'  
          - 'prom.domain2.com:9090'
```



How to configure federation in Prometheus

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:7074/actuator/prometheus	DOWN	instance="host.docker.internal:7074" job="customer-app" ▾	1m 0s ago	12.698ms	Get "http://host.docker.internal:7074/actuator/prometheus": dial tcp 192.168.65.254:7074: connect: connection refused

federate (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:9080/metrics match[]={_name_=~".+"}	UP	instance="host.docker.internal:9080" job="federate" ▾	1.838s ago	87.199ms	



Aggregate Queries in Prometheus

yaml

 Copy code

```
rule_files:
  - "recording_rules.yml"

recording_rules.yml:
  groups:
    - name: job:request_rate:sum
      rules:
        - record: job:request_rate:sum
          expr: sum(rate(http_requests_total[5m])) by (job)
```

[Open source](#)[Solutions](#)[Learn](#)[Company](#)[Downloads](#)[Contact us](#)[Sign in](#)

Using grafana cli

If you are using grafana-cli, execute the following command to install the plugin

[Shell](#)

```
grafana-cli plugins install yesoreyeram-boomtheme-panel
```

or for specific versions

[Shell](#)

```
grafana-cli --pluginUrl https://github.com/yesoreyeram/yesoreyeram-boomtheme-panel/releases/download/v0.2.1/yeso
```

Using helm chart

If you use helm chart to provision grafana, use the following config to install the plugin

[YAML](#)

```
plugins:  
  - yesoreyeram-boomtheme-panel
```

or for any specific versions

Grafana



```
Administrator: Command Prompt - docker exec -it prometheus_grafana_1 /bin/bash
1784f8db487a:/usr/share/grafana/public/sass$ cd ..
1784f8db487a:/usr/share/grafana/public$ cd ..
1784f8db487a:/usr/share/grafana$ ls
bin    conf    public
1784f8db487a:/usr/share/grafana$ cd bin
1784f8db487a:/usr/share/grafana/bin$ ls
grafana      grafana-cli      grafana-cli.md5      grafana-server      grafana-server.md5  grafana.md5
1784f8db487a:/usr/share/grafana/bin$ grafana-cli plugins install yesoreyeram-boomtheme-panel
Deprecation warning: The standalone 'grafana-cli' program is deprecated and will be removed in the future. Please update all uses of 'grafana-cli' to 'grafana cli'
✓Downloaded and extracted yesoreyeram-boomtheme-panel v0.2.1 zip successfully to /var/lib/grafana/plugins/yesoreyeram-boomtheme-panel

Please restart Grafana after installing or removing plugins. Refer to Grafana documentation for instructions if necessary.

1784f8db487a:/usr/share/grafana/bin$
```



Telegraf

- Telegraf is an agent for collecting and sending metrics and events from a wide array of sources to databases like InfluxDB, Prometheus, or via OpenTelemetry.



Telegraf

Key Features:

- **Plugins:** 300+ input, output, and processor plugins.
- **Input Sources:** System stats, logs, databases, services (Docker, Kubernetes).
- **Output Options:** InfluxDB, Prometheus, Kafka, OpenTelemetry.
- **Lightweight & Efficient:** Suitable for edge and server environments.



Telegraf

Use Case:

- Collect system-level metrics (CPU, memory, disk, network).
- Send custom metrics to monitoring backends.
- Acts as a **bridge** between services and monitoring tools.

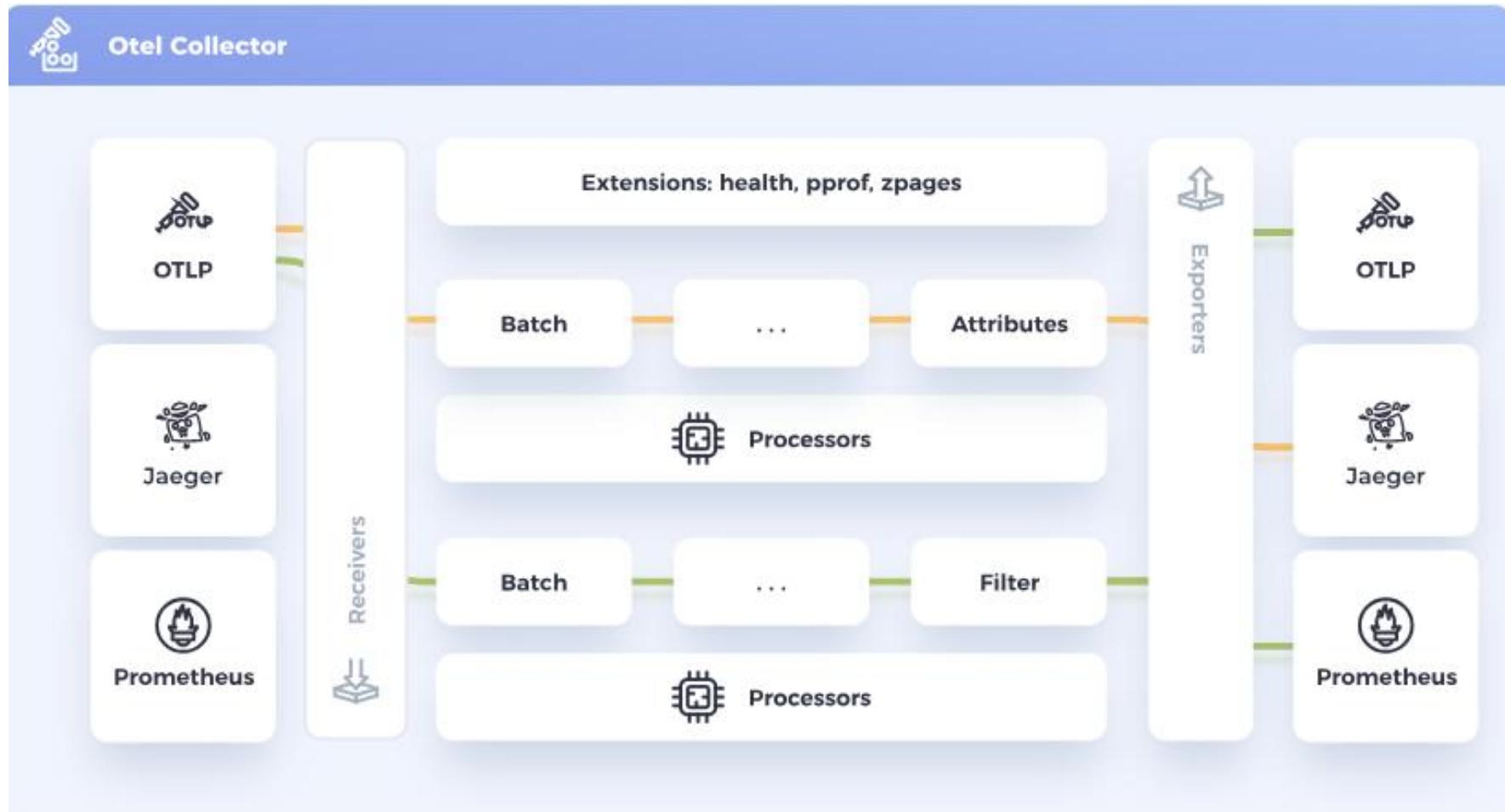


Open Telemetry Collector

- The Open Telemetry Collector is a vendor-agnostic service designed to receive, process, and export telemetry data like traces, metrics, and logs from applications and infrastructure.
- It's a key component in observability pipelines.

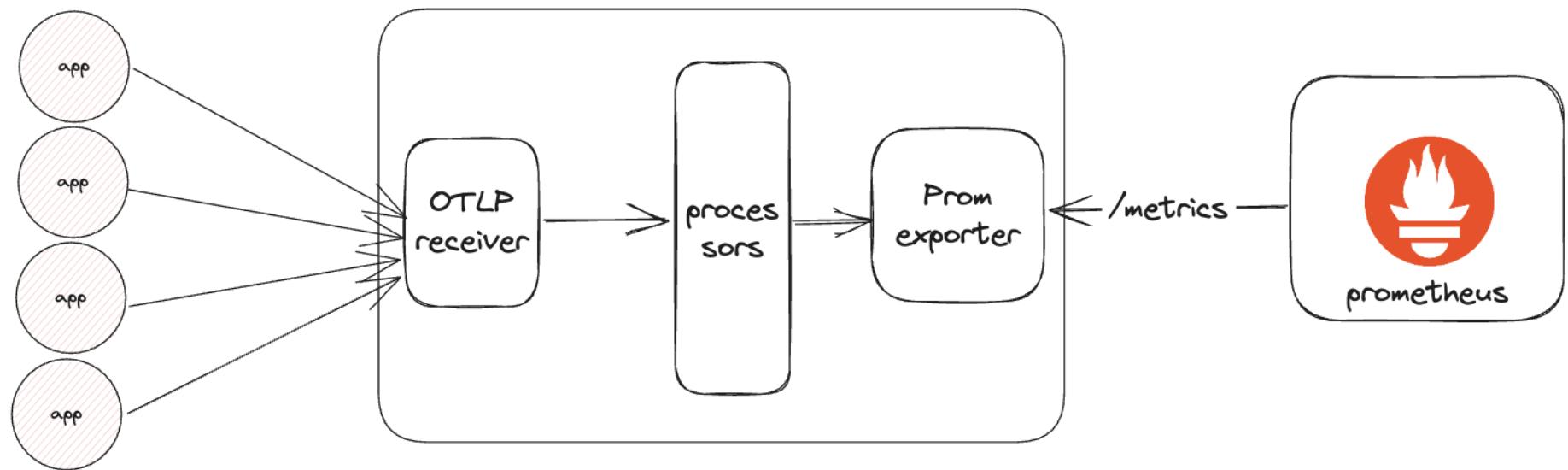


Open Telemetry Collector





Open Telemetry Collector





Open Telemetry Collector

Objectives

- *Usability*: Reasonable default configuration, supports popular protocols, runs and collects out of the box.
- *Performance*: Highly stable and performant under varying loads and configurations.
- *Observability*: An exemplar of an observable service.
- *Extensibility*: Customizable without touching the core code.
- *Unification*: Single codebase, deployable as an agent or collector with support for traces, metrics, and logs.



Open Telemetry Collector

Key Features:

1. Receives Data:

1. Supports **OTLP**, **Jaeger**, **Prometheus**, **Zipkin**, **FluentBit**, etc.
2. Handles **metrics**, **traces**, and **logs**.

2. Processes Data:

1. Add attributes, filter, batch, transform telemetry.
2. Supports pipelines with **processors** (e.g., sampling, filtering, batch).

3. Exports Data:

1. To backend systems like **Prometheus**, **Grafana Mimir**, **Jaeger**, **Datadog**, **New Relic**, **Elasticsearch**, etc.

4. Extensible:

1. You can add custom receivers, processors, or exporters.
2. Distributions like **otel-contrib** include community-contributed components.



Open Telemetry Collector

Core Components:

- **Receivers**: Ingest telemetry data.
- **Processors**: Modify/enrich telemetry.
- **Exporters**: Send telemetry to observability platforms.
- **Extensions**: Support services like health checks, pprof, zpages.



Open Telemetry Collector

Collector Modes:

1. **Agent**: Runs as a **sidecar** or on the host; close to the app.
2. **Gateway**: Runs as a **central service**; receives from multiple agents or apps.



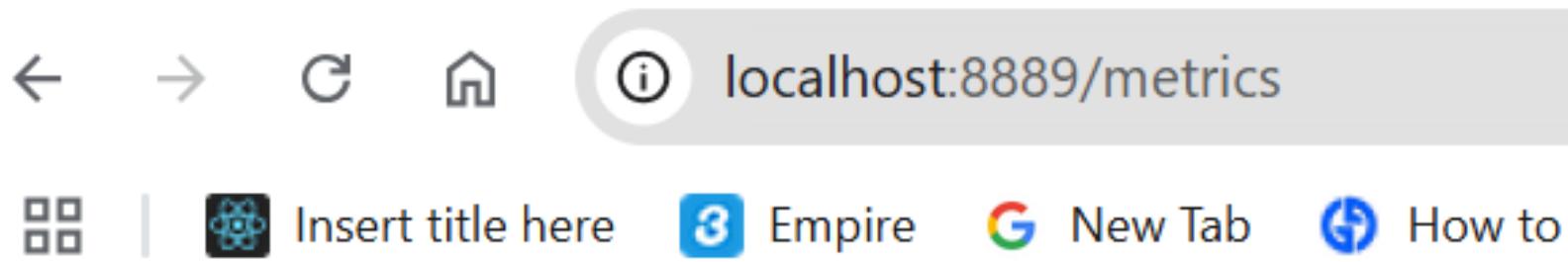
Open Telemetry Collector

Common Use Cases:

- Collect **traces** from microservices and export to **Jaeger**.
- Collect **metrics** from applications and export to **Prometheus**.
- Process logs and send them to **Elasticsearch**.



Open Telemetry Collector



```
# HELP sample_requests_total Number of requests
# TYPE sample_requests_total counter
sample_requests_total{job="sample-flask-app",route="/"}
```

Grafana Mimir



Overview:

- Grafana Mimir is a **highly scalable, long-term storage backend for Prometheus metrics**, optimized for large-scale environments.



Grafana Mimir

Key Features:

- **Horizontal Scalability:** Handle massive amounts of Prometheus metrics.
- **Multi-Tenancy:** Supports isolated metric storage for different users/teams.
- **High Availability:** Supports sharding and replication.
- **Integration:** Works seamlessly with Grafana for visualization.

Use Case:

- Store and query time-series metrics for long-term analysis.
- Backend for Prometheus setups needing high availability and scalability.
- Centralized metric storage across multiple Prometheus instances.



Integration Workflow Example

1. Instrumentation:

1. Use **OpenTelemetry SDK** in applications to capture traces and metrics.
2. Use **Telegraf** to gather system-level metrics.

2. Data Collection:

1. Send telemetry data to **OpenTelemetry Collector**.
2. Telegraf sends metrics directly to **Prometheus** or **OTel Collector**.

3. Storage:

1. OpenTelemetry Collector exports metrics to **Prometheus Remote Write** or **Grafana Mimir**.
2. Telegraf can also push to **Mimir** (via Prometheus output plugin).

4. Visualization:

1. **Grafana** reads from **Mimir** to visualize metrics and traces.



Comparison of Roles

Tool	Primary Role	Data Type
OpenTelemetry	Collects app-level telemetry (metrics/traces/logs)	Metrics, Traces, Logs
Telegraf	Collects system/infrastructure metrics	Metrics
Grafana Mimir	Stores Prometheus metrics at scale	Metrics



Real-World Scenario Example

- Microservice App:
 - Uses Open Telemetry to generate traces and metrics.
- Host Server:
 - Runs Telegraf to collect CPU, memory, and disk metrics.
- Both push data to Open Telemetry Collector.
- Collector exports data to Grafana Mimir.
- Metrics and traces are visualized in **Grafana Dashboards**.

Questions

