# MONITORING WITH  Prometheus

# Agenda

- Monitoring Basics
  - Basic of Monitoring
  - Logging vs Monitoring
  - Good Monitoring Summary
  - Monitoring Mechanics : Probing and Instrumentation
  - Push vs Pull Approaches
  - What is Metrics  and type of Metrics : Gauge, Counter, etc
  - Monitoring Methodologies : USE and Four Golden Signals
  - Alert and Notifications
- Prometheus - Demo
  - Overview
  - Running Prometheus
  - Gathering Metrics : Host and Containers
  - Query
  - Alert Manager
  - Visualization : Grafana

# Monitoring

- Monitoring is the tools and processes by which you measure and manage your technology systems. Monitoring provides the translation to business value from metrics generated by your systems and applications.
- Your monitoring system translates those metrics into a measure of user experience. That measure provides feedback to the business to help ensure it's delivering what customers want.
- The measure also provides feedback to technology, as we'll define below, to indicate what isn't working and what's delivering an insufficient quality of service. A monitoring system has two customers: Technology and The business

# Logging !=Monitoring

- Logging : Recording to diagnose a system
- Monitoring : Observation, Checking and Recording

# Why Monitoring?

- Know when things go wrong or not as expected
- Be able to debug and gain insight
- Detect changes over time and drive technical/business decisions
- Feed into other systems/processes, eq : security, automation)

# Good Monitoring Summary

Good monitoring should provide:

- The state of the world, from the top (the business) down.
- Assistance in fault diagnostics.
- A source of information for infrastructure, application development, and business folks.
- Built into design and the life cycle of application development and deployment.
- Automated and provided as self-service, where possible

# Probing vs Instrumentation

- Probing monitoring probes the outside of an application. You query the external characteristics of an application: does it respond to a poll on an open port and return the correct data or response code?
- Instrumentation monitoring looks at what's inside the application. The application is instrumented and returns measurements of its state, the state of internal components, or the performance of transactions or events.

# Push vs Pull Approach

- There are two approaches to how monitoring checks are executed that are worth briefly discussing. These are the pull versus push approaches.
- Pull-based systems scrape or check a remote application—for example, an endpoint containing metrics or, as from our probing example, a check using ICMP. In push-based systems, applications emit events that are received by the monitoring system.
- The downsides for pull approach are that all metric endpoints have to be reachable for the Prometheus poller, implying a more elaborate secure network configuration
- Scaling becomes an issue in large deployments. Prometheus advises a push-based approach for collecting metrics for short-lived jobs.

# Metrics and Types

- Metrics are measures of properties of components of software or hardware. To make a metric useful we keep track of its state, generally recording data points over time. Those data points are called observations. An observation consists of the value, a timestamp, and sometimes a series of properties that describe the observation, such as a source or tags. A collection of observations is called a time series.
- Metric Types : Gauge, Counter, Histogram

# USE Method (1)

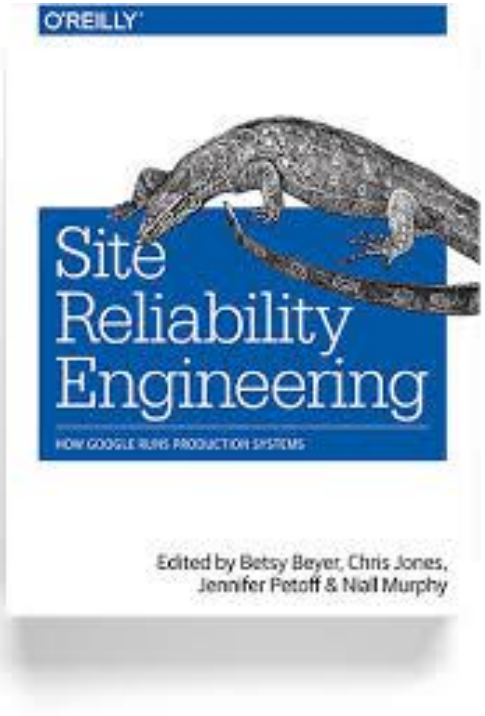
- The USE, or Utilization Saturation and Errors, Method was developed by Brendan Gregg, a kernel and performance engineer at Netflix.
- The methodology proposes creating a checklist for server analysis that allows the fast identification of issues. You work down the checklist to identify common performance issues, making use of data collected from your environment.
- The USE Method can be summarized as: For every resource, check utilization, saturation, and errors. The method is most effective for the monitoring of resources that suffer performance issues under high utilization or saturation

# USE Method (2)

- A resource - A component of a system. In Gregg's definition of the model it's traditionally a physical server component like CPUs, disks, etc., but many folks also include software resources in the definition.
- Utilization - The average time the resource is busy doing work. It's usually expressed as a percentage over time.
- Saturation - The measure of queued work for a resource, work it can't process yet. This is usually expressed as queue length.
- Errors - The scalar count of error events for a resource.

# 4 Golden Signals (1)

The Google Four Golden Signals come out of the Google SRE book. They take a similar approach to the USE Method, specifying a series of general metric types to monitor. Rather than being system-level-focused time series, the metric types in this methodology are more application or user-facing



# 4 Golden Signals (2)

- Latency - The time taken to service a request, distinguishing between the latency of successful and failed requests. A failed request, for example, might return with very low latency skewing your results.
- Traffic - The demand on your system—for example, HTTP requests per second or transactions for a database system.
- Errors - The rate that requests fail, whether explicit failures like HTTP 500 errors, implicit failures like wrong or invalid content being returned, or policy-based failures—for instance if you've mandated that failures over 30ms should be considered errors.
- Saturation - The "fullness" of your application or the resources that are constraining it—for example, memory or IO. This also includes impending saturation, such as a rapidly filling disk.

# Alert and Notifications

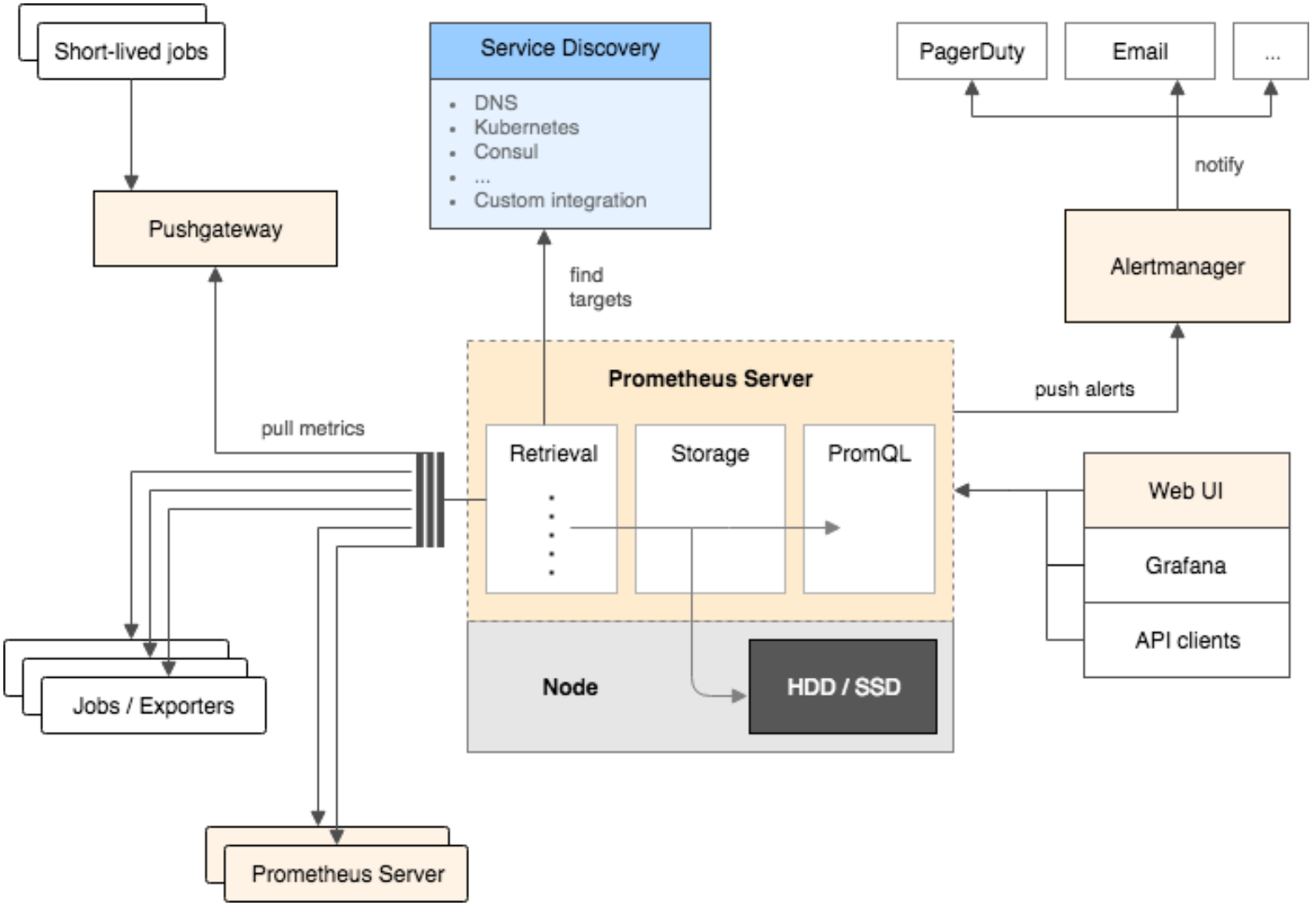
- An alert is raised when something happens—for example, when a threshold is reached. This, however, doesn't mean anyone's been told about the event. That's where notifications come in. A notification takes the alert and tells someone or something about it: an email is sent, an SMS is triggered, a ticket is opened, or the like
- To build a good notification system you need to consider the basics of:
  - What problems to notify on.
  - Who to tell about a problem.
  - How to tell them.
  - How often to tell them.
  - When to stop telling them, do something else, or escalate to someone else.

# Let us discuss about the main topic

# Prometheus History

- Prometheus owes its inspiration to Google's Borgmon. It was originally developed by Matt T. Proud, an ex-Google SRE, as a research project. After Proud joined SoundCloud, he teamed up with another engineer, Julius Volz, to develop Prometheus in earnest. Other developers joined the effort, and it continued development internally at SoundCloud, culminating in a public release in January 2015
- Prometheus was primarily designed to provide near real-time introspection monitoring of dynamic cloud- and container-based microservices, services, and applications
- Prometheus is written in Go, open source, and licensed under the Apache 2.0 license. It is incubated under the Cloud Native Computing Foundation.

# Prometheus Architecture



# How it Works

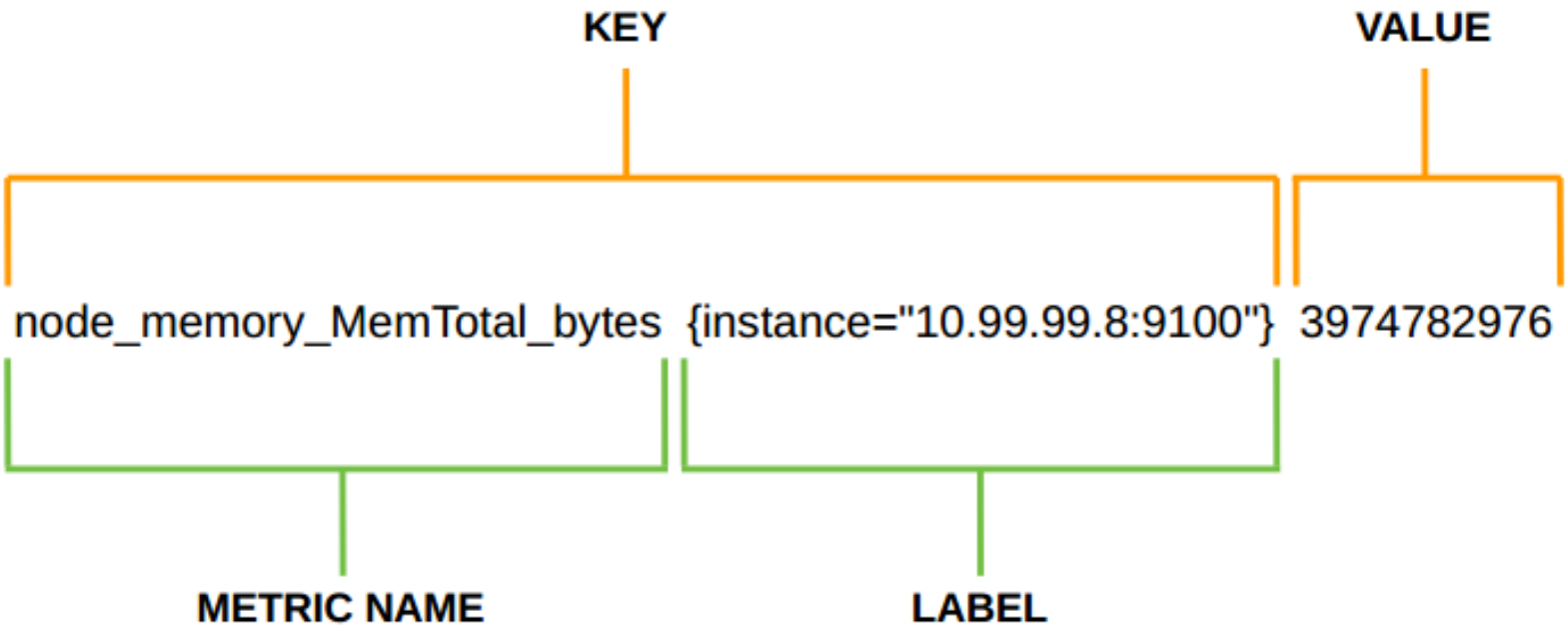
- Prometheus works by scraping or pulling time series data exposed from applications. The time series data is exposed by the applications themselves often via client libraries or via proxies called exporters, as HTTP endpoints.
- Exporters and client libraries exist for many languages, frameworks, and open-source applications—for example, for web servers like Apache and databases like MySQL.
- Prometheus also has a push gateway you can use to receive small volumes of data, for example, data from targets that can’t be pulled, like transient jobs or targets behind frewalls.

# Exporters

A) Databases: MySQL, MongoDB & PostgreSQL
B) Hardware: Node & Ubiquiti UniFi
C) Messaging: RabbitMQ & Kafka
D) Storage: Ceph, Gluster & Hadoop
E) HTTP: Apache, HAProxy, Nginx, & Varnish

* https://prometheus.io/docs/instrumenting/exporters/

# Metrics


KEY
VALUE
node_memory_MemTotal_bytes {instance="10.99.99.8:9100"} 3974782976
METRIC NAME
LABEL

# Metrics Collection

- Prometheus calls the source of metrics it can scrape endpoints. An endpoint usually corresponds to a single process, host, service, or application. To scrape an endpoint, Prometheus defines configuration called a target. This is the information required to perform the scrape—for example, how to connect to it, what metadata to apply, any authentication required to connect, or other information that defines how the scrape will occur.
- Groups of targets are called jobs. Jobs are usually groups of targets with the same role, for example : cluster of Apache servers behind a load balancer. That is, they're effectively a group of like processes.

# How about visualization??

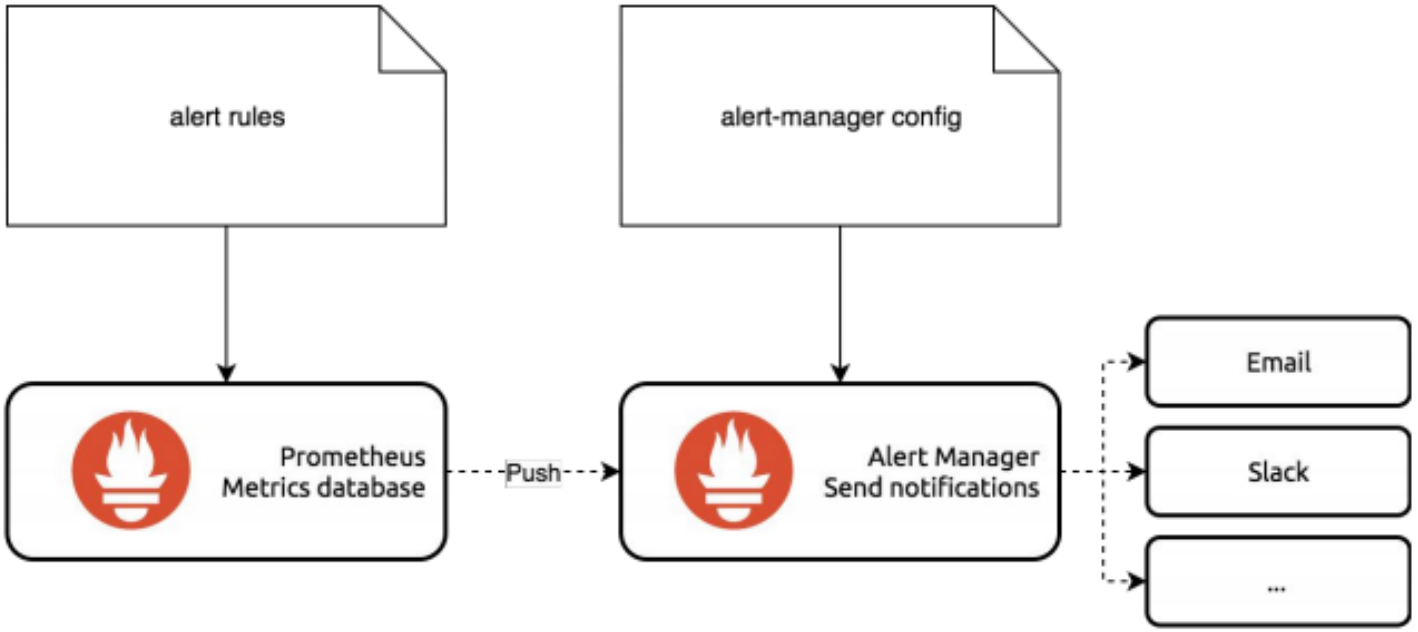
Visualization of Metrics can be made in 3 ways:

1) Expression Browser
2) Console Templates
3) Grafana

# Alert and Notification

Alerting with Prometheus is separated into two parts:
1. Alerting Rules
2. AlertManager

Sending out notifications via email, Telegram, PagerDuty, HipChat, Slack, and etc.

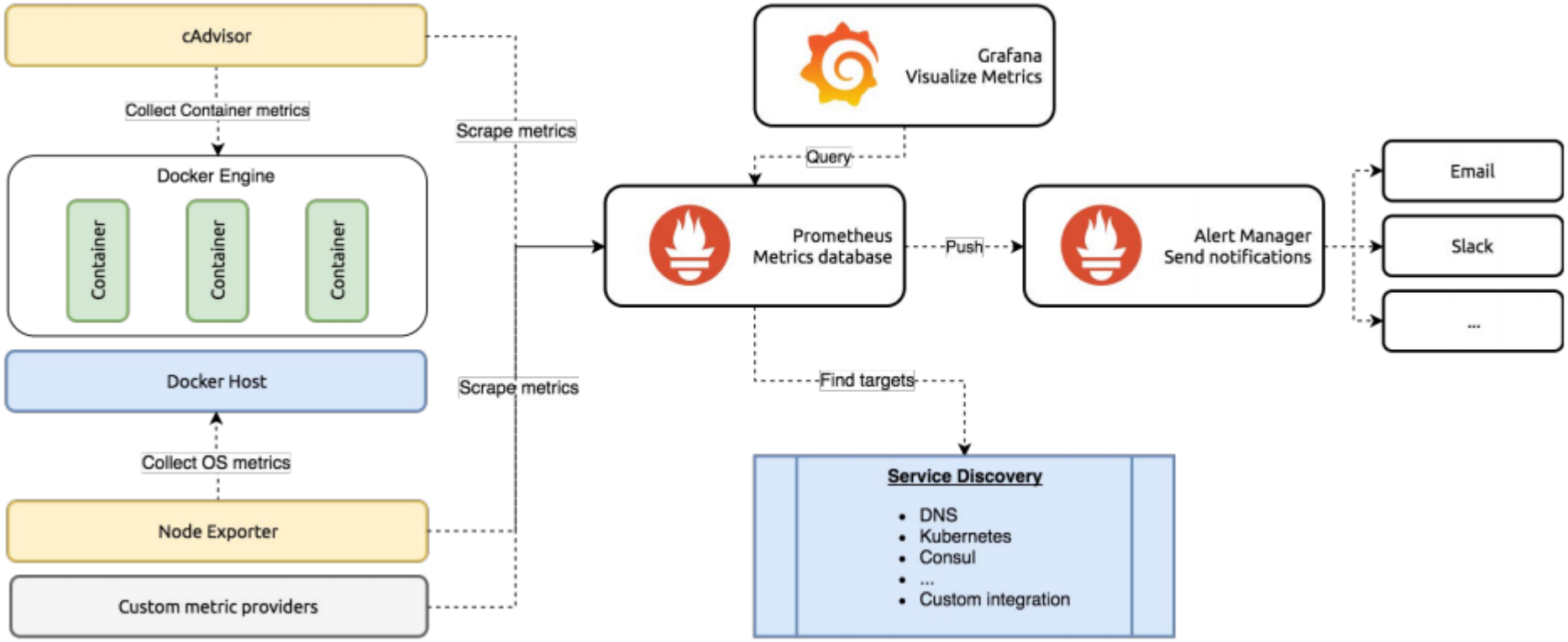


# Alert State

An alert can have three potential states:

- Inactive - The alert is not active.
- Pending - The alert has met the test expression but is still waiting for the duration specified in the for clause to fire.
- Firing - The alert has met the test expression and has been Pending for longer than the duration of the for clause.

# Demo


cAdvisor
Collect Container metrics
Docker Engine
Container
Container
Container
Docker Host
Collect OS metrics
Node Exporter
Custom metric providers
Scrape metrics
Scrape metrics
Grafana
Visualize Metrics
Query
Prometheus
Metrics database
Push
Alert Manager
Send notifications
Email
Slack
...
Find targets
Service Discovery
DNS
Kubernetes
Consul
...
Custom integration

# Prometheus Config File

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

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

rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

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

# Node Rule Config File

```
groups:
- name: node.rules
  rules:
  - alert: Down
    expr: up == 0
    for: 5m
    annotations:
      summary: "Instance {{ $labels.instance }} down"
      description: "Instance {{ $labels.instance }} of job {{ $labels.job }} down for more than 5 minutes."
```

```
groups:
- name: node_alerts
  rules:
  - alert: HighNodeCPU
    expr: instance:node_cpu:avg_rate5m > 80
    for: 60m
    labels:
      severity: warning
    annotations:
      summary: High Node CPU for 1 hour
      console: You might want to check the Node Dashboard at http://monitoring.com
```

# Alert Manager Config File - Email

```
global:
  resolve_timeout: 5m

route:
  group_by: EmailAlert
  receiver: email-me

receivers:
- name: email-me
  email_configs:
  - to: "to@email.com"
    from: "from@email.com"
    smarthost: smtp.email.com:587
    auth_username: "auth"
    auth_identity: "pass"
    auth_password: "XXXXXXXX"
    send_resolved: True
```

# Whats Next?

- Scaling Prometheus : Cortex, Thanos, Vulcan
- Instrumenting : Go, Java, Python ,etc
- Target/Service Dynamic Discovery : DNS, Consul, etc

# Q & A