

DevOps Monitoring Tools



What is Monitoring?

- Monitoring: The collection and presentation of data about the performance and stability of services and infrastructure
- Monitoring tools collect data over things such as:
 - Usage of memory
 - cpu
 - disk i/o
 - Other resources over time
 - Application logs
 - Network traffic
- The collected data is presented in various forms, such as charts and graphs, or in the form of real time notifications about problems



What does Monitoring look like?

- **Real-time notifications:**

- Performance on the website is beginning to slow down
- A monitoring tool detects that response times are growing
- An administrator is immediately notified and is able to intervene before downtime occurs

- **Postmortem analysis:**

- Something went wrong in production last night
- It's working now, but we don't know what caused it
- Luckily, monitoring tools collected a lot of data during the outage
- With that data, developers and operations engineers are able to determine the root cause (a poorly performing SQL query) and fix it

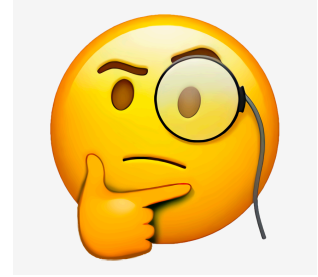


Why do Monitoring?

- **Fast recovery:** The sooner a problem is detected, the sooner it can be fixed. You want to know about a problem before your customer does!
- **Better root cause analysis** – The more data you have, the easier it is to determine the root cause of a problem.
- **Visibility across teams** – Good monitoring tools give useful data to both developers and operations people about the performance of code in production.
- **Automated response** – Monitoring data can be used alongside orchestration to provide automated responses to events, such as automated recovery from failures.

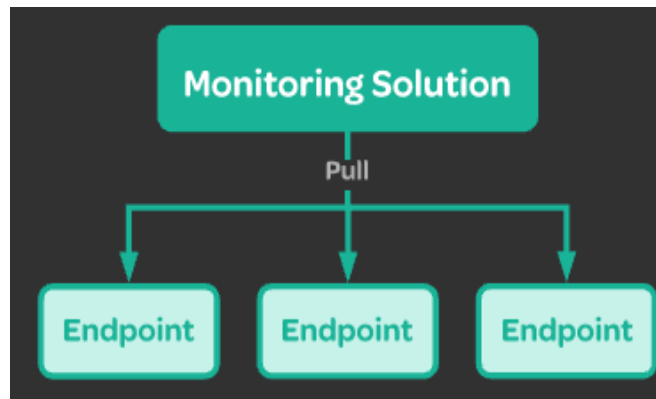


Push or Pull Monitoring Solution



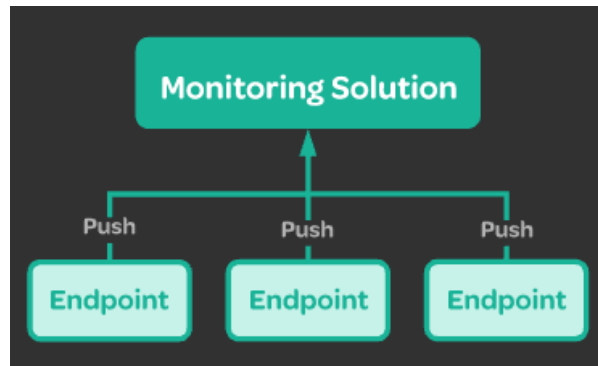
Pull-Based Monitoring

- When using a pull system to monitor your environments and applications, we're having the monitoring solution itself query our metrics endpoints.
- How the information from the endpoints , we are monitoring gets into our monitoring system?
- Monitoring solution need to scrape the endpoints itself and pulls the data to the server



Push-based monitoring

- The endpoints are the ones that push their metrics up to the monitoring application.
- Push systems are especially useful when you need event-based monitoring, and can't wait every 15 or so seconds for the data to be pulled in.
- That said, many push-based systems have greater setup requirements and overhead than pull-based ones, and the majority of the managing isn't done through only the monitoring server.



Which to Choose?

- Monitoring is not a one-person job. Every platform, every application, every container should be monitored in some way, and it should not be the job of one person to implement monitoring on everything.
- it means that we all need to be monitoring-minded. It's not good enough to place the job of monitoring on one person or one team. If we're monitoring an application, dev should be thinking about how to best monitor what's going on when the application is being used; if we're monitoring a container or virtual machine, systems needs to consider what common problems we need to look out for given that container or machine's task (is it running something that takes up a lot of memory? Do you have file system space concerns?).
- The person involved in a project from the start is going to have a better idea of what to look out for regarding that project than someone who would come in just to add monitoring.





Prometheus



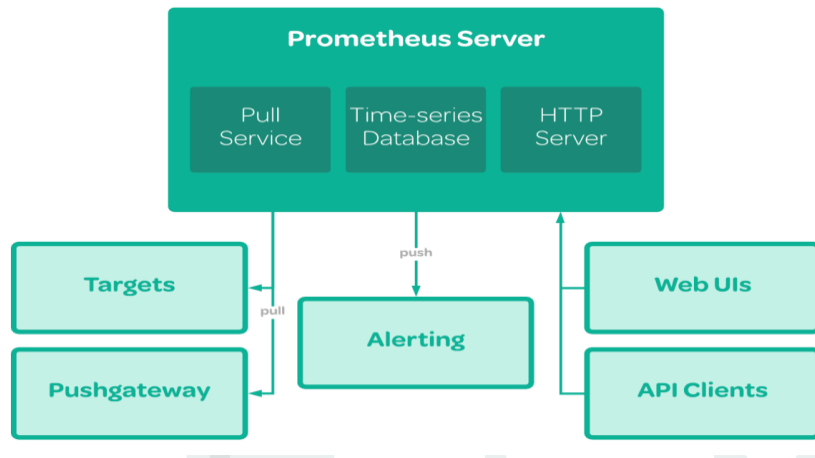
What is Prometheus?

- Prometheus is an open-source system monitoring originally built at [SoundCloud](#) in 2012.
- It records real-time metrics in a time series database built using a HTTP pull model, with flexible queries and real-time alerting.
- It is now a standalone open source project and maintained independently of any company.
- The project is written in Go with source code available on GitHub.
- Prometheus is designed for reliability, to be the system you go to during an outage to allow you to quickly diagnose problems.



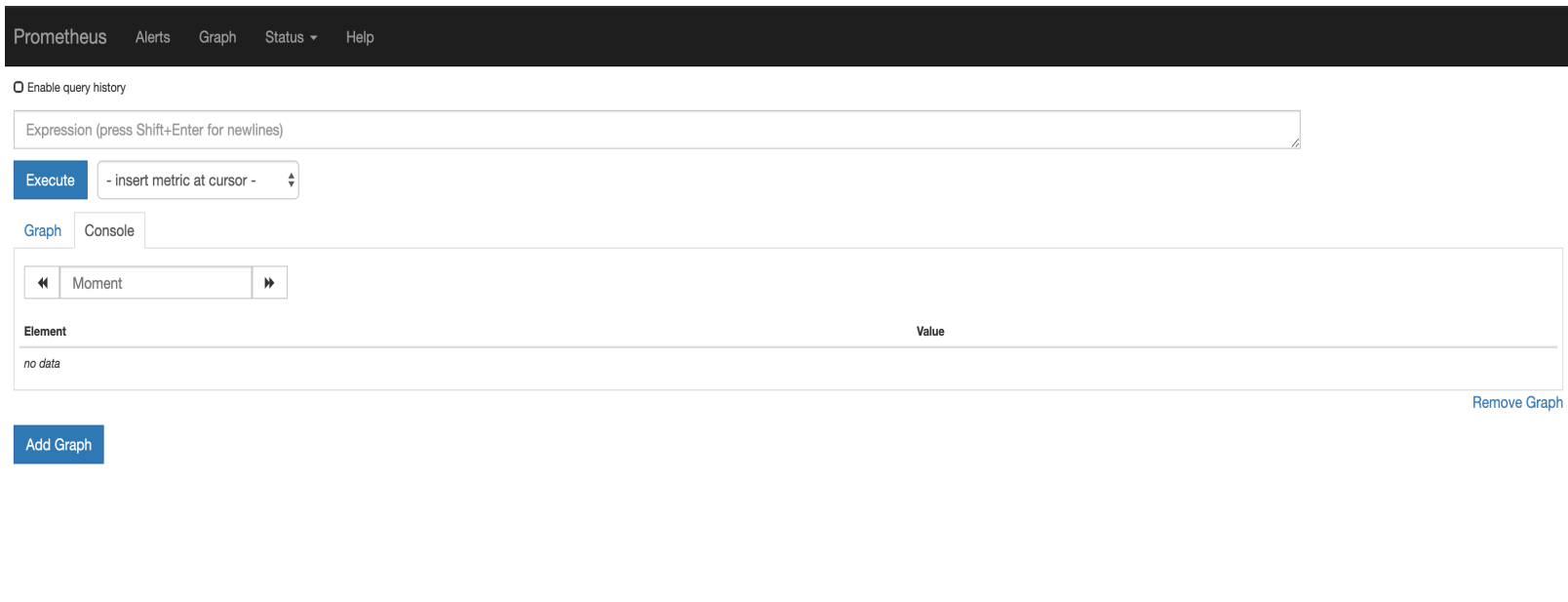
Cont'd:

- Prometheus works by scraping metrics across our system. We'll get into how these metrics are provided.
- Prometheus itself is made up of what is called the “Prometheus server”: Which is Prometheus itself, a provided time-series database that stores our pulled metrics, and an HTTP server that provides us with the web UI.
- Prometheus is able to push notifications to other applications — most usually an alerting application such as Alertmanager



Prometheus Installation:

- We are going to install Prometheus as a service



The screenshot displays the Prometheus web interface. At the top is a dark navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below this, there is a checkbox for 'Enable query history'. The main area features a text input for 'Expression (press Shift+Enter for newlines)', an 'Execute' button, and a dropdown menu currently showing '- insert metric at cursor -'. Below the input is a tabbed interface with 'Graph' and 'Console' tabs. The 'Graph' tab is active, showing a time range selector set to 'Moment' with navigation arrows. Below the selector is a table with two columns: 'Element' and 'Value'. The table currently contains the text 'no data'. In the bottom right corner of the graph area, there is a 'Remove Graph' link. At the bottom left, there is an 'Add Graph' button.



To deep dive in Prometheus Archticture let's define the following:

- Service discovery
- Node Exporter
- CPU Metrics
- Monitor Containers
- Client Library



Service discovery

- Service discovery is simply the way Prometheus detects our metrics pages for any of our desired targets.
- if we wanted Prometheus to discover a service in server, we would have to add that information to our Prometheus configuration.
- The configuration is in /etc/prometheus/prometheus.yml under scrape_configs.
- To get Prometheus to discover these new targets, all we have to do is restart Prometheus service.

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

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



Node Exporter:

- We use it to monitor Infrastructure things like CPU, Memory, Disk usage.....
- We use Node exporters to gather metrics for other servers and creates endpoints that prometheus can scrape.
- The Prometheus Node Exporter is for *nix systems, there is a WMI exporter for Windows systems.



CPU Metrics

- Metrics involving the CPU are prefixed with `node_cpu`.
- `node_cpu_seconds_total` is what is known as a counter — that is, it keeps a running total of the amount of time the CPU is in each defined state, in seconds. As such, you'll only see these numbers going up.
- This metric measures the amount of time of each of the following CPU states:
 - idle: Time the CPU is doing nothing
 - iowait: Time the CPU is waiting for I/O
 - irq: Time spent on fixing interrupts
 - nice: Time spent on user-level processes with a positive nice value
 - softirq: Time spent fixing interrupted
 - steal: In running a virtual machine, time spent with other VMs “stealing” CPU
 - guest: On servers that host VMs, this value will be guest and contain the amount of CPU usage of the VMs hosted
 - system: Time spent in the kernel
 - user: Time spent in userland



Cont'd

- Prometheus and the majority of monitoring programs receive their data about CPU from one source: the `/proc/stat` file on the host itself.
- Prometheus uses a language called PromQL that lets us run queries against this metric data, performing tasks like calculating averages, means, and other mathematical functions.
- Examples:
 - Return all time series with the metric:
`node_cpu_seconds_total`
 - Return all time series with the metric `node_cpu_seconds_total` and the given job and mode labels:
`node_cpu_seconds_total{job="node-exporter", mode="idle"}`
 - Return a whole range of time (in this case 5 minutes) for the same vector, making it a range vector:
`node_cpu_seconds_total{job="node-exporter", mode="idle"}[5m]`
 - Query job that end with `-exporter`:
`node_cpu_seconds_total{job=~".*-exporter"}`



Monitor containers

- Google's **cAdvisor** is an open-source solution that works out of the box with most container platforms, including Docker. And once we have cAdvisor installed, we can see much of the same metrics we see for our host on our container, only these are provided to us through the prefix container.
- **cAdvisor** also monitors *all* our containers automatically. That means when we view a metric, we're seeing it for everything that cAdvisor monitors.
- The image name is google/cadvisor:latest



Client Library

- when we write custom applications, we want to be able to monitor those, too, This is where Prometheus client libraries come in
- Prometheus itself supports four client libraries — Go, Java, Python, and Ruby — but third-party client libraries are provided for a number of other languages.
- we're going to need to add this library the Node.js prom-client is not restricted to just allowing us to write new metrics; it also includes some default application metrics we can enable, mostly centered around the application's use of memory.
- Of course, adding this library isn't enough in and of itself: We also need to make sure we have a /metrics endpoint generated that Prometheus can scrape, which we'll be creating using the Express framework our application already utilizes.



Rules

- We need to locate the `rule_files` section of the configuration “Prometheus.yml”, This is where we’re going to define the file that will contain our alerting rules.
- Rules are written in YAML and are always organized in groups, with a group name that cannot be reused across the rules.
- Rules in a group are always run together.
- We're first going to define a *recording* rule, which will keep track of the results of a PromQL expression, without performing any kind of alerting then we will define alerting rules.

```
groups:
- name: uptime
  rules:
    - record: job:nodeapp
      expr: avg without (instance) (up{job="node-app"})
    - alert: ForethoughtApplicationDown
      expr: job:nodeapp < 0.1
```



Alert Manager

- We will use alert manager to send alerts and control them.
- First we need to install it as a service also.
- We need to add it to prometheus.yml config file.
- We need to add the configuration of firing alerts in /etc/alertmanager/alertmanager.yml

```
global:
  resolve_timeout: 5m

route:
  group_by: ['alertname']
  group_wait: 10s
  group_interval: 10s
  repeat_interval: 1h
  receiver: 'slack'
  routes:
    - match:
        severity: critical
receivers:
- name: 'slack'
  slack_configs:
    - channel: "#prometheus"
      api_url: https://hooks.slack.com/services/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
inhibit_rules:
- source_match:
    severity: 'critical'
  target_match:
    severity: 'warning'
  equal: ['alertname', 'dev', 'instance']
```

