Engineer

Prometheus

a database that scrapes and stores metrics

What it is

Prometheus is an open-source monitoring system with its own time-series database at the centre: every few seconds it asks the programs it watches for their current numbers over HTTP, stores each value with a timestamp, and lets you query, graph and alert on what it has kept. The name covers the server on its own and, just as often, the small family of components that has grown up around it.

It was built at SoundCloud, where work on it started in 2012, and it now belongs to the Cloud Native Computing Foundation. It joined in 2016 as the foundation's second hosted project, after Kubernetes, and graduated in 2018. Prometheus 3.0, announced in November 2024, was its first major release in seven years. The code is written mostly in Go and published under the Apache 2.0 licence.

Everything it stores is a time series, named by a metric and a set of labels: key and value pairs that say which instance of a thing was measured, as in `api_http_requests_total{method="POST", handler="/messages"}`. Each sample in that series is a 64-bit floating-point value, or a native histogram, with a timestamp in milliseconds. A different value for any label is a different series. That is what makes the data easy to slice by host, route or status code, and also what makes it easy to overload.

Programs expose their numbers on an HTTP endpoint, usually `/metrics`, through a client library compiled into the code or through an exporter: a small adapter that translates a system Prometheus cannot instrument directly, such as a Linux host or a database. The server finds its targets through service discovery or a static list and scrapes them on a schedule. A batch job that ends before the next scrape can push to a separate push gateway instead. Alerting rules run on

the server, and firing alerts go to Alertmanager, which deduplicates and groups them, applies silences and routes them to email, PagerDuty, OpsGenie and similar receivers.

It is usually weighed against push-based collectors in the Graphite and StatsD tradition, against general time-series databases such as InfluxDB, and against hosted metrics services. Grafana is the usual face in front of it. It stores numbers rather than events, so a log line or a trace belongs somewhere else, and the project is plain that it favours reliability over completeness: per-request billing, where every count has to be exact, is a job for a different system.

Use the concept in a decision

Put “What is Prometheus?” to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

The decision Prometheus sits in is whether to run your own metrics database or rent one, and if you run it, how large it has to be.

Running it carries no licence fee. The cost is a machine and the hours of whoever keeps it healthy. A hosted service takes that work away and replaces it with a charge that grows with the data sent to it, which turns every new label into a line on an invoice. For a few dozen machines the self-run server is usually the cheaper and simpler path. The sum shifts when the series count runs into the millions and the team would rather pay than staff it.

The sizing arithmetic is kinder than most people expect. The project’s storage documentation puts the average at one to two bytes per sample. A thousand series scraped every 15 seconds and kept for the default retention of 15 days come to 86.4 million samples, roughly 86–173 MB on disk. Disk is seldom what runs out. Memory is: the block currently being filled lives in memory, guarded against a crash by a write-ahead log, and every active series costs something there whether or not anyone ever queries it.

Each server is deliberately self-contained. It leans on no network storage, so it keeps answering while the infrastructure it watches is falling over, which is the moment it is needed most. The price is that local storage is neither clustered nor replicated. High availability means two identical servers scraping the same targets, and history longer than a few weeks means sending a copy with `remote_write` to a store built to keep it.

Pulling rather than receiving has a practical payoff too. Because the server does the asking, it notices when a target stops answering and records that as `up` dropping to zero. And an engineer can open the same `/metrics` page in a browser to see exactly what the server sees, which makes a disagreement between the graph and the application quick to settle.

What it looks like in practice

In the founder's work Prometheus turns up as the collector behind a dashboard: once on a fleet of database servers, and once as the second version of a view the whole office watched.

The database case was a fleet of ten Ubuntu 20.04 LTS virtual servers carrying PostgreSQL 14 and MS SQL Server 2019. Prometheus and Grafana tracked CPU, memory, disk I/O and query performance as they happened, alongside automated backups, firewall rules and custom scripts for log analysis. Across all ten servers, uptime held at 99.9%. Database hosts suit scraping well, because the numbers that predict trouble there, a disk filling or a query slowing, move over hours, and a sample every few seconds sees them long before a user does.

The office view shows the step from a home-made collector to Prometheus. In an earlier role, the founder built an organisation's first observability dashboard. It began by reading CPU, RAM, disk, temperature and more from remote Linux servers over SSH, and watched Docker containers the same way. A second version was rebuilt on Prometheus and Grafana, around the metrics chosen with the DevOps and engineering teams: CPU utilisation, memory usage, service uptime and API latency, with alerts on threshold breaches. With the dashboard in place, teams detected and resolved incidents 40% faster. Polling over SSH is a fine start for a handful of machines; a scraper with labels and a query language is what lets the same view keep growing.

Where it goes wrong

Memory climbs after a release that changed nothing important. Someone added a label whose values are unbounded: a user ID, an email address, a full URL path, a request ID. Each new value became a new series, and the server now holds millions where it held thousands. The naming guide warns against exactly this. Keep label values to small, known sets, send per-request detail to logs or traces, and find the worst offenders with a query like this one:

```
topk(10, count by (__name__) ({__name__=~".+"}))
```

A graph shows a cliff or a spike that never happened. A counter only rises, and it falls to zero when the process restarts. Plotting the raw value, or subtracting by hand, turns every restart into a crash or a surge. `rate()` understands resets, so take the rate first and add up afterwards, over a window several scrape intervals wide:

```
sum by (job) (rate(http_requests_total{status=~"5.."}[5m]))
```

A recording rule that sums a counter before the rate is taken hides the resets inside the sum and brings the false cliffs back.

A service goes down and its error alert stays green. A rule such as an error rate above a threshold is evaluated over the series that exist. When the target disappears, so do its series, the expression returns an empty result, and the alert resolves itself at the worst possible moment. Alert on `up == 0` for every scrape target, use `absent()` for any series that must always be there, and have something outside the server check that the server itself is running.

Last quarter’s numbers are gone when someone asks for them. Local storage keeps 15 days by default and is a single copy on a single disk. It works as a window on recent behaviour, and as an archive it lets people down. Raise the retention with care, send what has to last to long-term storage through `remote_write`, and keep figures that money depends on in a database of record.

Further reading

- Prometheus overview — the project's own account of what it is, its components, and where it is the wrong fit.
- Data model — metric names, labels and samples, and why each label value starts a new series.
- Metric types — counter, gauge, histogram and summary, and which questions each one can answer.
- Querying basics — PromQL selectors, range vectors and the types an expression can return.
- Storage — two-hour blocks, the write-ahead log, retention and the remote-write interface.
- Metric and label naming — base units, the `_total` suffix and the warning about high-cardinality labels.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is Prometheus?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/prometheus/