An Introduction to Prometheus

Brian Brazil
Founder

Who am I?

Engineer passionate about running software reliably in production.

- Core developer of Prometheus
- Studied Computer Science in Trinity College Dublin.
- Google SRE for 7 years, working on high-scale reliable systems.
- Contributor to many open source projects, including Prometheus, Ansible, Python, Aurora and Zookeeper.
- Founder of Robust Perception, provider of commercial support and consulting for Prometheus.

Why monitor?

Know when things go wrong

To call in a human to prevent a business-level issue, or prevent an issue in advance

Be able to debug and gain insight

Trending to see changes over time, and drive technical/business decisions

To feed into other systems/processes (e.g. QA, security, automation)

Common Monitoring Challenges

Themes common among companies I've talk to and tools I've looked at:

- Monitoring tools are limited, both technically and conceptually
- Tools don't scale well and are unwieldy to manage
- Operational practices don't align with the business

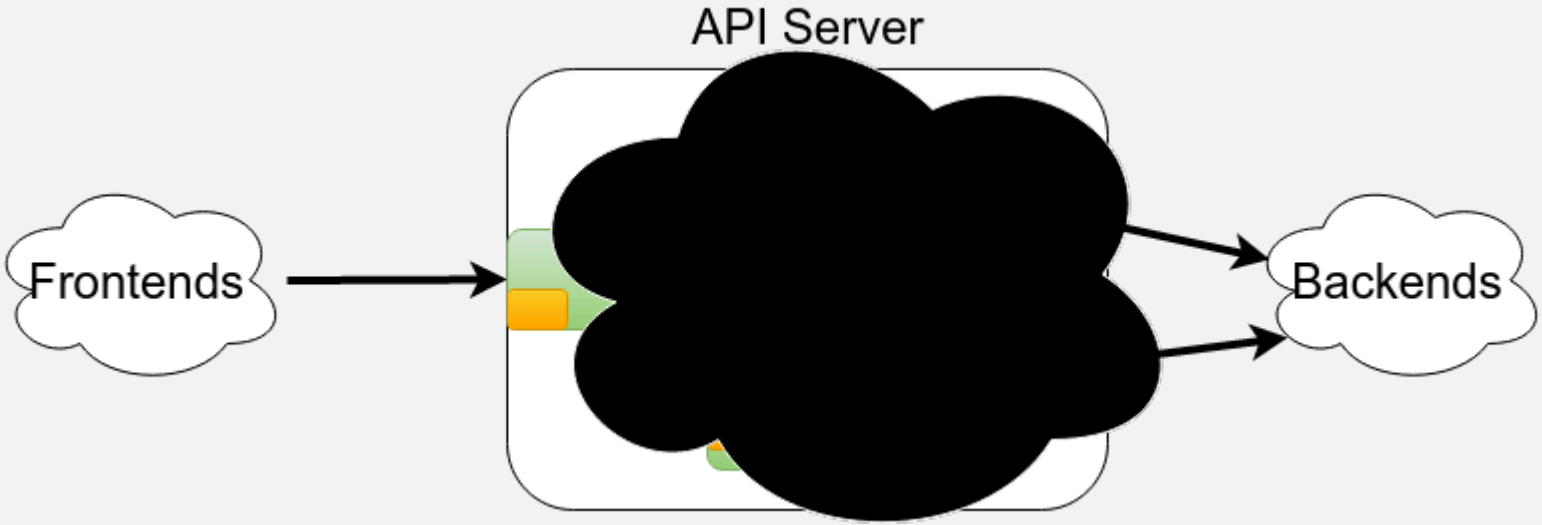
For example:

Your customers care about increased latency and it's in your SLAs. You can only alert on individual machine CPU usage.

Result: Engineers continuously woken up for non-issues, get fatigued

Robust Perception

Many Monitoring Systems Look like This



What is Prometheus?

Prometheus is a metrics-based time series database, designed for whitebox monitoring.

It supports labels (dimensions/tags).

Alerting and graphing are unified, using the same language.

Development History

Inspired by Google's Borgmon monitoring system.

Started in 2012 by ex-Googlers working in Soundcloud as an open source project, mainly written in Go. Publically launched in early 2015, 1.0 released in July 2016.

It continues to be independent of any one company, and is incubating with the CNCF.

Prometheus Community

Prometheus has a very active community.

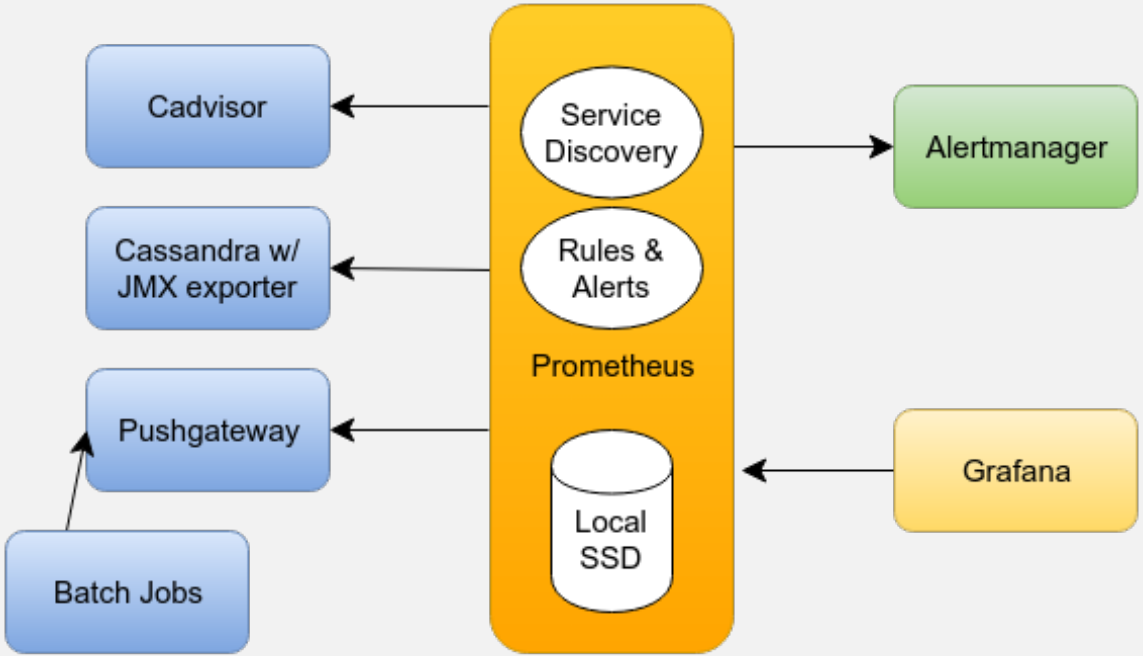
Over 250 people have contributed to official repos.

There over 100 3rd party integrations.

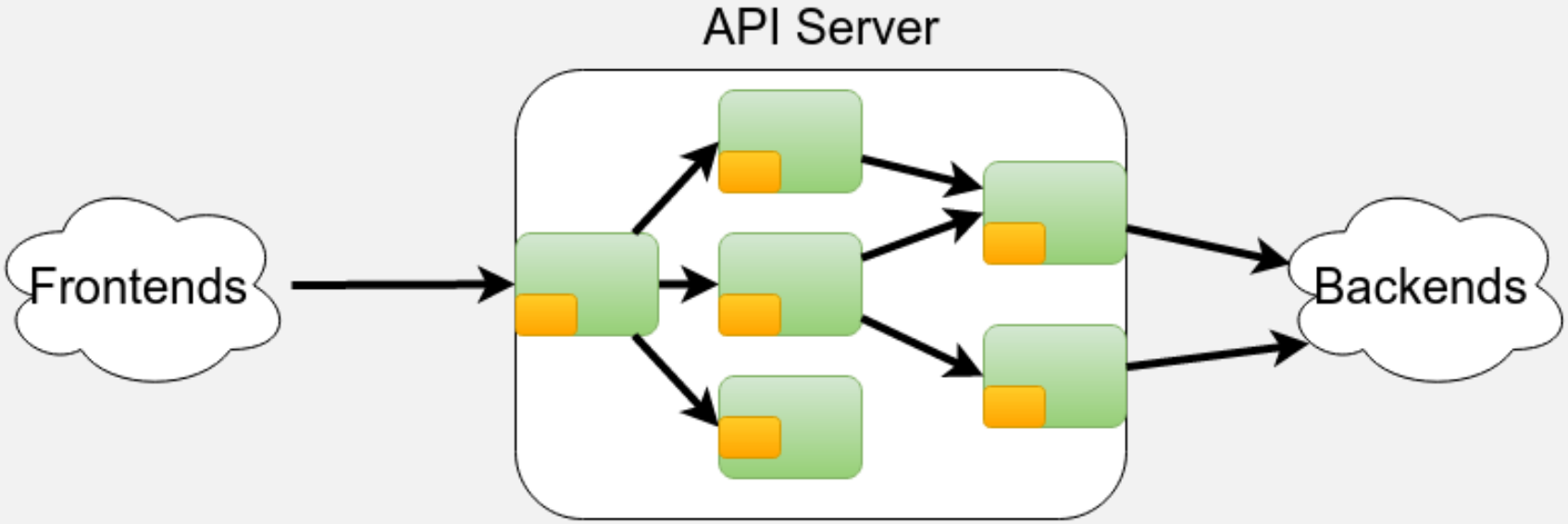
Over 200 articles, talks and blog posts have been written about it.

It is estimated that over 500 companies use Prometheus in production.

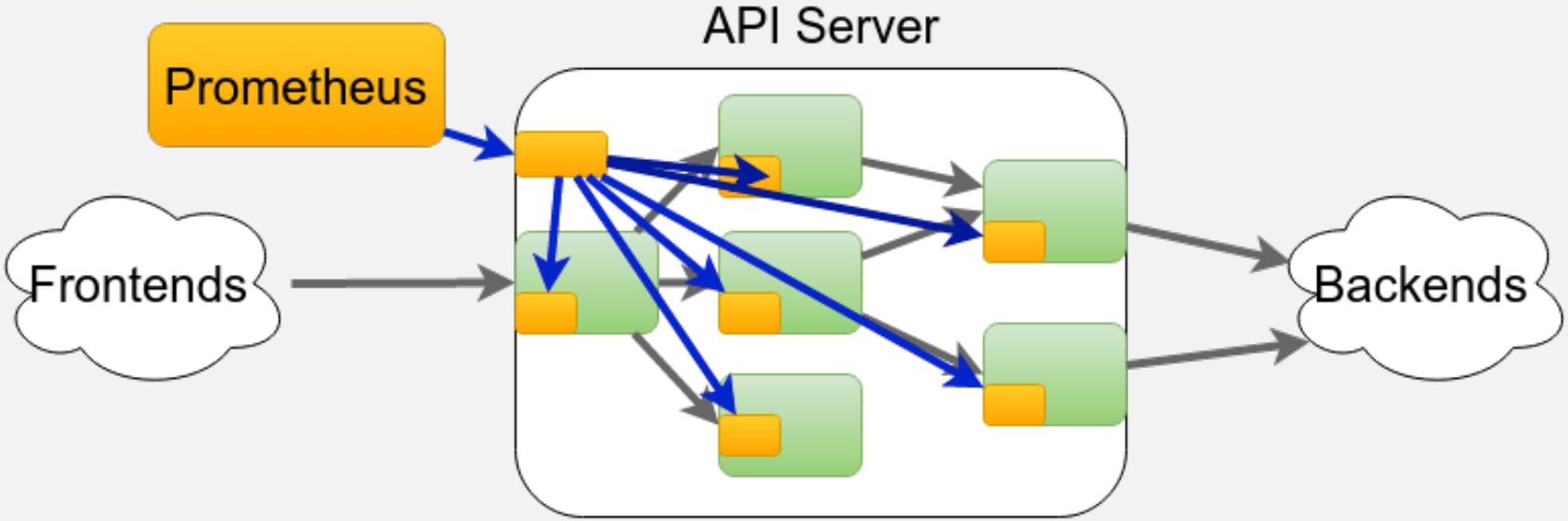
Architecture



Your Services have Internals



Monitor the Internals



Instrumentation Made Easy

```
pip install prometheus_client
```

```
from prometheus_client import Summary, start_http_server
REQUEST_DURATION = Summary('request_duration_seconds',
    'Request duration in seconds')

REQUEST_DURATION.observe(7)

@REQUEST_DURATION.time()
def my_handler(request):
  pass   # Your code here

start_http_server(8000)
```

Robust Perception

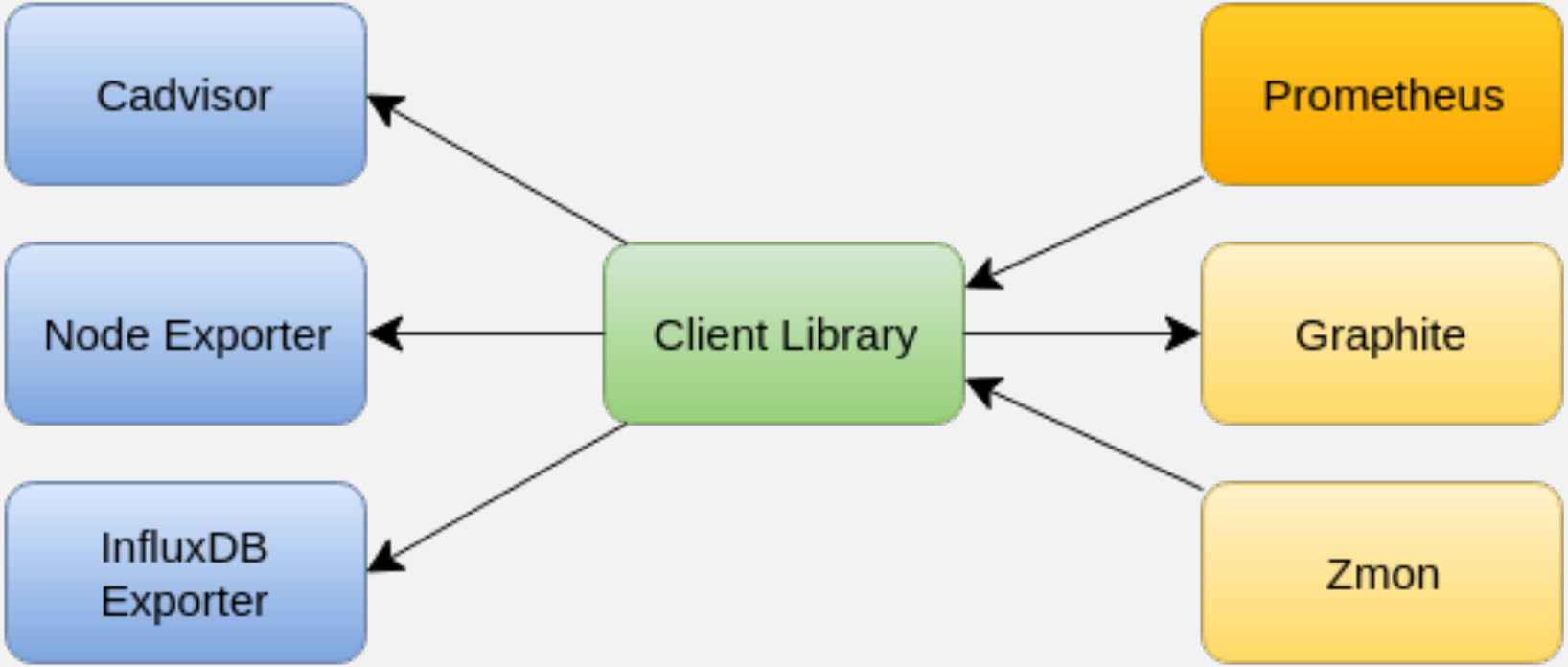
Open ecosystem

Prometheus client libraries don’t tie you into Prometheus.

```
# Expose with Prometheus text format over HTTP
start_http_server(8000)


# Push to Graphite every 10 seconds
gb = GraphiteBridge(('graphite.your.org', 2003))
gb.start(10.0)
```

Prometheus Clients as a Clearinghouse



Power of Labels

Prometheus doesn’t use dotted.strings like `metric.osf.yokohama`.

We have metrics like:

```
metric{event=”osf”,city=”yokohama”}
```

Full UTF-8 support - don't need to worry about values containing dots.

Can aggregate, cut, and slice along them.

Great for Aggregation

```
topk(5,
  sum by (image)(
   rate(container_cpu_usage_seconds_total{
    id=~"/system.slice/docker.*"}[5m]
   )
  )
)
```

Great for Alerting

Alert on any machine being down:

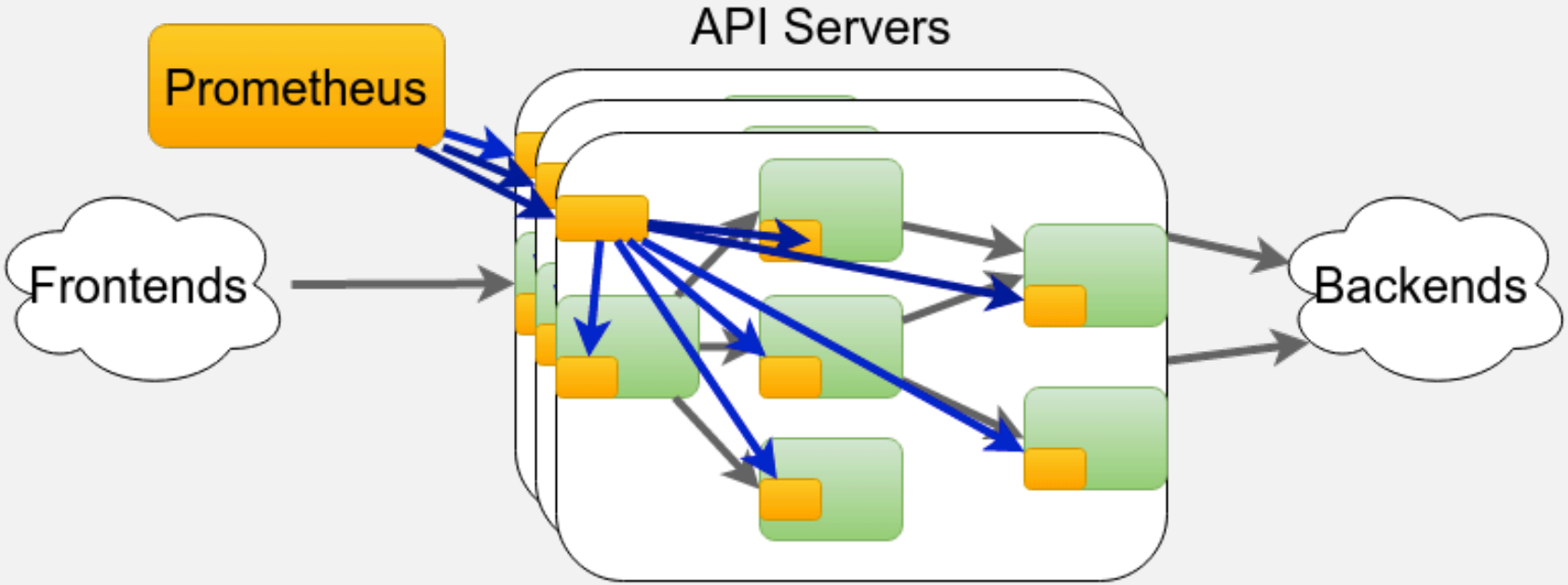
```
ALERT InstanceDown
  IF up{job="node"} == 0
  FOR 10m
```

Alert on 25% of machines being down:

```
ALERT ManyInstancesDown
  IF avg by(job)(up{job="node"}) < .75
  FOR 10m
```

Robust Perception

Monitor as a Service, not as Machines



Efficient

A single Prometheus server can handle 800K samples/s

New varbit encoding uses only 1.3 bytes/sample

Node exporter produces ~700 time series, so even with a 10s scrape interval a single Prometheus could handle over 10,000 machines!

This efficiency means that the vast majority of users never need to worry about scaling.

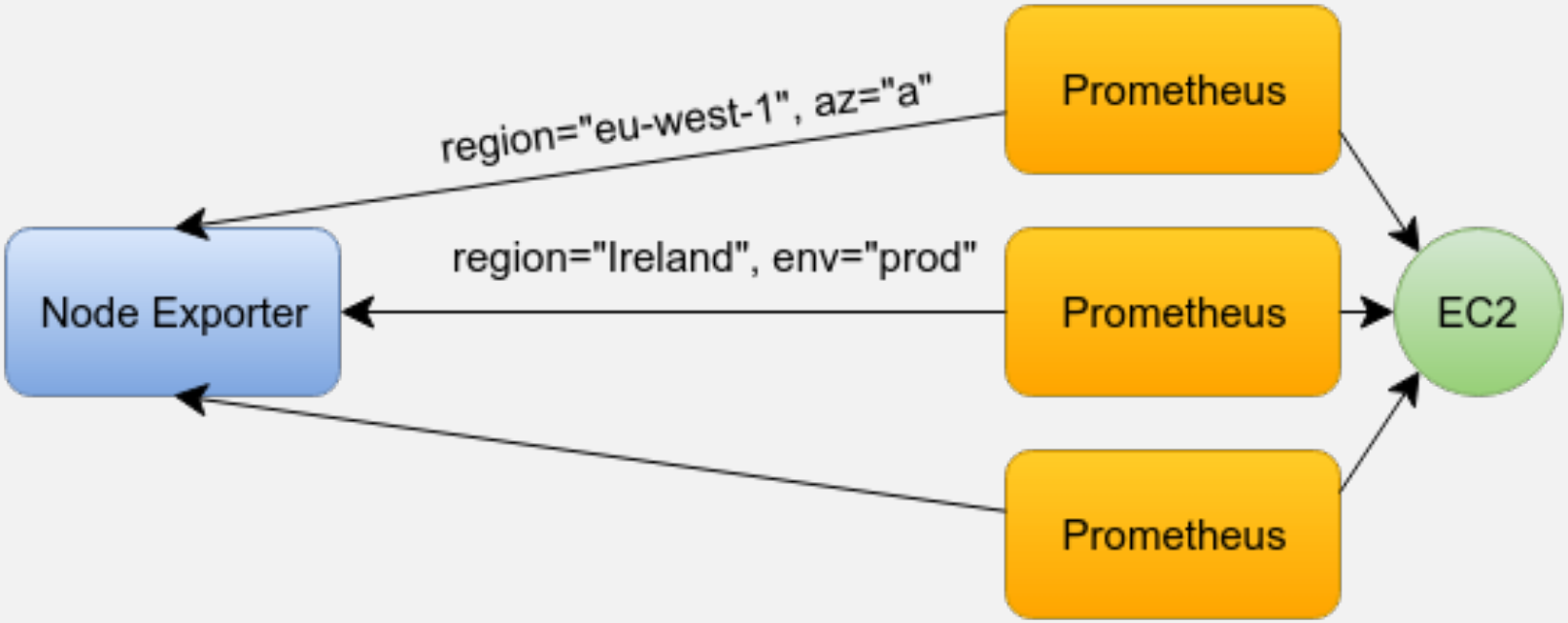
Decentralised

Pull based, so easy to on run a workstation for testing and rogue servers can't push bad metrics.

Each team can run their own Prometheus, no need for central management or talking to an operations team.

Perfect for microservices!

Per-team Hierarchy of Targets



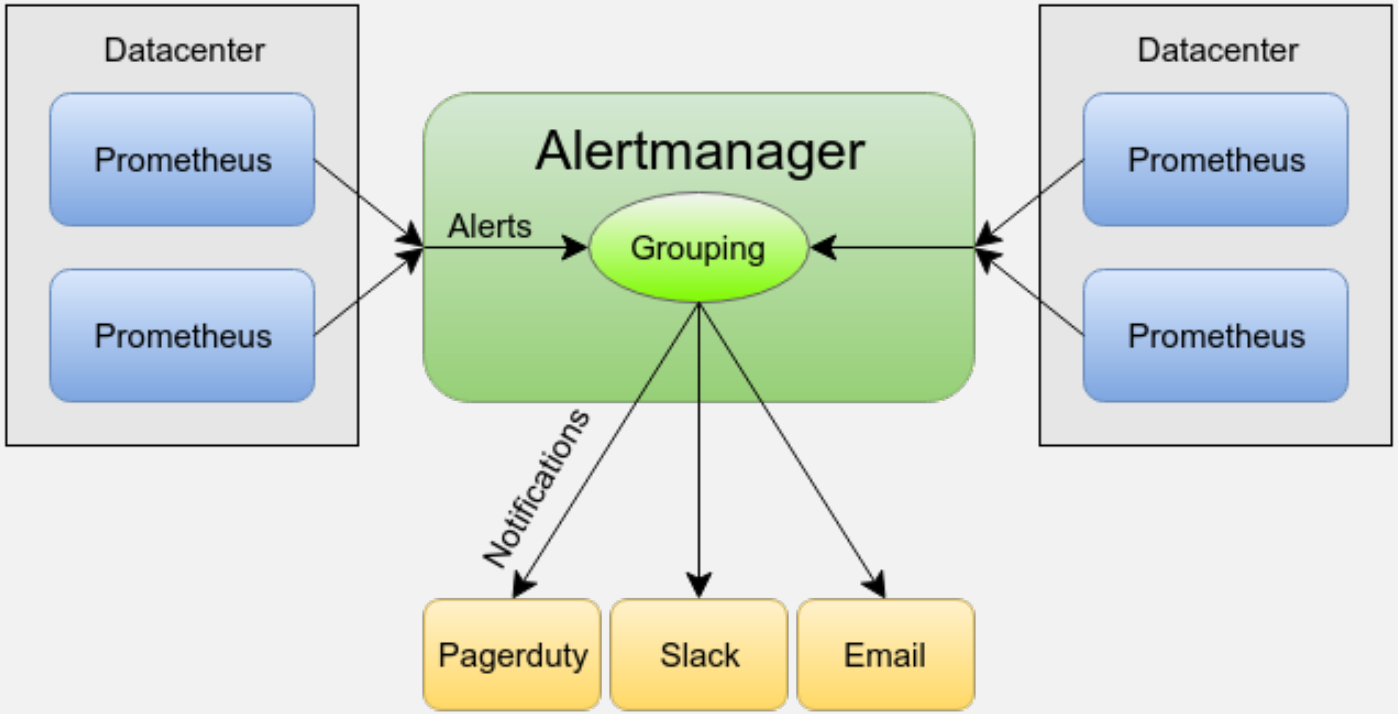
Reliable

A monitoring system should be something you trust when your network and systems are falling apart.

Prometheus has no external dependencies. It has no complex CP clustering to go wrong.

For HA we keep it simple.

Alerting Architecture



Robust Perception

Opinionated

As a project, we recognise our limitations.

We try avoid reinventing the wheel when others have already solved a problem, e.g. Grafana over Promdash.

We encourage good practices and using the right tool for the job: You could trigger a script to restart a failed process via the Alertmanager, but a supervisor such as daemontools or monit is probably a better approach.

Demo

What defines Prometheus?

Key attributes of the Prometheus monitoring system:

- Open Source
- Instrumentation Made Easy
- Powerful
- Efficient
- Decentralized
- Reliable
- Opinionated

How can you get involved?

You can help with user questions on
https://groups.google.com/forum/#!forum/prometheus-users

Or #prometheus on irc.freenode.net

Give a talk or write a blog post!

For coding, https://github.com/prometheus/prometheus is the main respository.

Many exporters, both 1st and 3rd party to improve.

Future Plans

Long term storage: Remote read path, finalization of remote write path.

HA for the alertmanager, continuing to work if there's a network partition.

Improved staleness handling in Prometheus.

More exporters and other integrations!

10 Tips for Monitoring

With potentially millions of time series across your system, can be difficult to know what is and isn't useful.

What approaches help manage this complexity?

How do you avoid getting caught out?

Here's some tips.

Robust **Perception**

#1: Choose your key statistics

Users don't care that one of your machines is short of CPU.

Users care if the service is slow or throwing errors.

For your primary dashboards focus on high-level metrics that directly impact users.

#2: Use aggregations

Think about services, not machines.

Once you have more than a handful of machines, you should treat them as an amorphous blob.

Looking at the key statistics is easier for 10 services than 10 services each of which is on 10 machines

Once you have isolated a problem to one service, then can see if one machine is the problem

Robust **Perception**

#3: Avoid the Wall of Graphs

Dashboards tend to grow without bound. Worst I've seen was 600 graphs.

It might look impressive, but humans can't deal with that much data at once. (and they take forever to load)

Your services will have a rough tree structure, have a dashboard per service and talk the tree from the top when you have a problem. Similarly for each service, have dashboards per subsystem.

Rule of Thumb: Limit of 5 graphs per dashboard, and 5 lines per graph.

#4: Client-side quantiles aren't aggregatable

Many instrumentation systems calculate quantiles/percentiles inside each process, and export it to the TSDB.

It is not statistically possible to aggregate these.

If you want meaningful quantiles, you should track histogram buckets in each process, aggregate those in your monitoring system and then calculate the quantile.

This is done using histogram_quantile() and rate() in Prometheus.

#5: Averages are easy to reason about

Q: Say you have a service with two backends. If 95th percentile latency goes up due to one of the backends, what will you see in 95th percentile latency for that backend?

A: ?

#5: Averages are easy to reason about

Q: Say you have a service with two backends. If 95th percentile latency goes up due to one of the backends, what will you see in 95th percentile latency for that backend?

A: It depends, could be no change. If the latencies are strongly correlated for each request across the backends, you'll see the same latency bump.

This is tricky to reason about, especially in an emergency.

Averages don't have this problem, as they include all requests.

Robust **Perception**

#6: Costs and Benefits

1s resolution monitoring of all metrics would be handy for debugging.

But is it ten time more valuable than 10s monitoring? And sixty times more valuable than 60s monitoring?

Monitoring isn't free. It costs resources to run, and resources in the services being monitored too. Quantiles and histograms can get expensive fast.

60s resolution is generally a good balance. Reserve 1s granularity or a literal handful of key metrics.

#7: Nyquist-Shannon Sampling Theorem

To reconstruct a signal you need a resolution that's at least double it's frequency.

If you've got a 10s resolution time series, you can't reconstruct patterns that are less than 20s long.

Higher frequency patterns can cause effects like aliasing, and mislead you.

If you suspect that there's something more to the data, try a higher resolution temporarily or start profiling.

#8: Correlation is not Causation - Confirmation Bias

Humans are great at spotting patterns. Not all of them are actually there.

Always try to look for evidence that'd falsify your hypothesis.

If two metrics seem to correlate on a graph that doesn't mean that they're related.

They could be independent tasks running on the same schedule.
Or if you zoom out there plenty of times when one spikes but not the other.
Or one could be causing a slight increase in resource contention, pushing the other over the edge.

#9 Know when to use Logs and Metrics

You want a metrics time series system for your primary monitoring.

Logs have information about every event. This limits the number of fields (<100), but you have unlimited cardinality.

Metrics aggregate across events, but you can have many metrics (>10000) with limited cardinality.

Metrics help you determine where in the system the problem is. From there, logs can help you pinpoint which requests are tickling the problem.

Robust **Perception**

#10 Have a way to deal with non-critical alerts

Most alerts don't justify waking up someone at night, but someone needs to look at them sometime.

Often they're sent to a mailing list, where everyone promptly filters them away.

Better to have some form of ticketing system that'll assign a single owner for each alert.

A daily email with all firing alerts that the oncall has to process can also work.

Final Word

It's easy to get in over your head when starting to appreciate to power of of time series monitoring and Prometheus.

The one thing I'd say is to Keep it Simple.

Just like anything else, more complex doesn't automatically mean better.

Robust **Perception**

Resources

Official Project Website: prometheus.io

Official Mailing List: prometheus-developers@googlegroups.com

Demo: demo.robustperception.io

Robust Perception Blog: www.robustperception.io/blog

Queries about Consulting/Support: prometheus@robustperception.io

Robust **Perception**