Introduction to Prometheus

An Approach to Whitebox Monitoring

Robust **Perception**

Who am I?

Engineer passionate about running software reliably in production.

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.

What is Whitebox Monitoring?

Blackbox monitoring

Monitoring from the outside

No knowledge of how the application works internally

Examples: ping, HTTP request, inserting data and waiting for it to appear on dashboard

Where to use Blackbox

Blackbox monitoring should be treated similarly to smoke tests.

It’s good for finding when things have badly broken in an obvious way, and testing from outside your network.

Not so good for knowing what’s going on inside a system.

Nor should it be treated like regression testing and try to test every single feature.

Tend to be flaky, as they either pass or fail.

Whitebox Monitoring

Complementary to blackbox monitoring.

Works with information from inside your systems.

Can be simple things like CPU usage, down to the number of requests triggering a particular obscure codepath.

Prometheus

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. Version 1.0 released in 2016. Incubating with the CNCF.

500+ companies using it including Digital Ocean, Ericsson, Weave and CoreOS.

Robust **Perception**

What is Monitoring For?

Why monitor?

Know when things go wrong

Be able to debug and gain insight

- Trending to see changes over time
- Plumbing data to other systems/processes

Knowing when things go wrong

The first thing people think of you say monitoring is alerting.

What is the wrongness we want to detect and alert on?

A blip with no real consequence, or a latency issue affecting users?

Symptoms vs Causes

Humans are limited in what they can handle.

If you alert on every single thing that might be a problem, you'll get overwhelmed and suffer from alert fatigue.

Key problem: You care about things like user facing latency. There are hundreds of things that could cause that.

Alerting on every possible cause is a Sisyphean task, but alerting on the symptom of high latency is just one alert.

Example: CPU usage

Some monitoring systems don't allow you to alert on the latency of your servers.

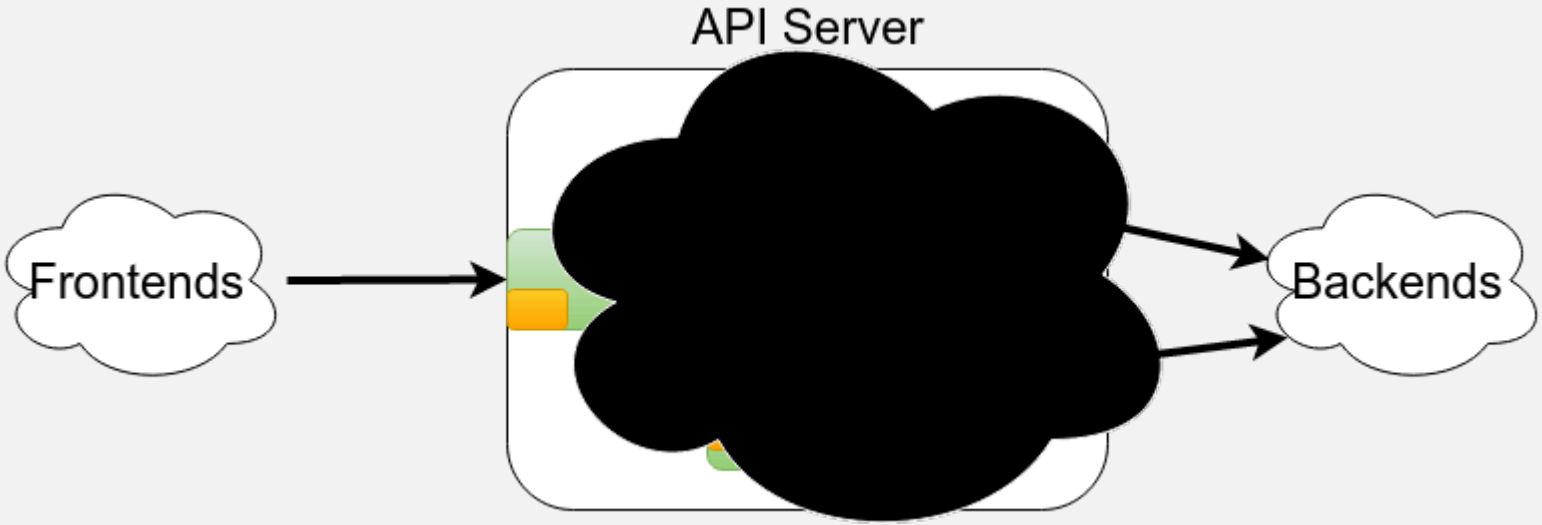
The closest you can get is CPU usage.

False positives due to e.g. logrotate running too long.

False negatives due to deadlocks.

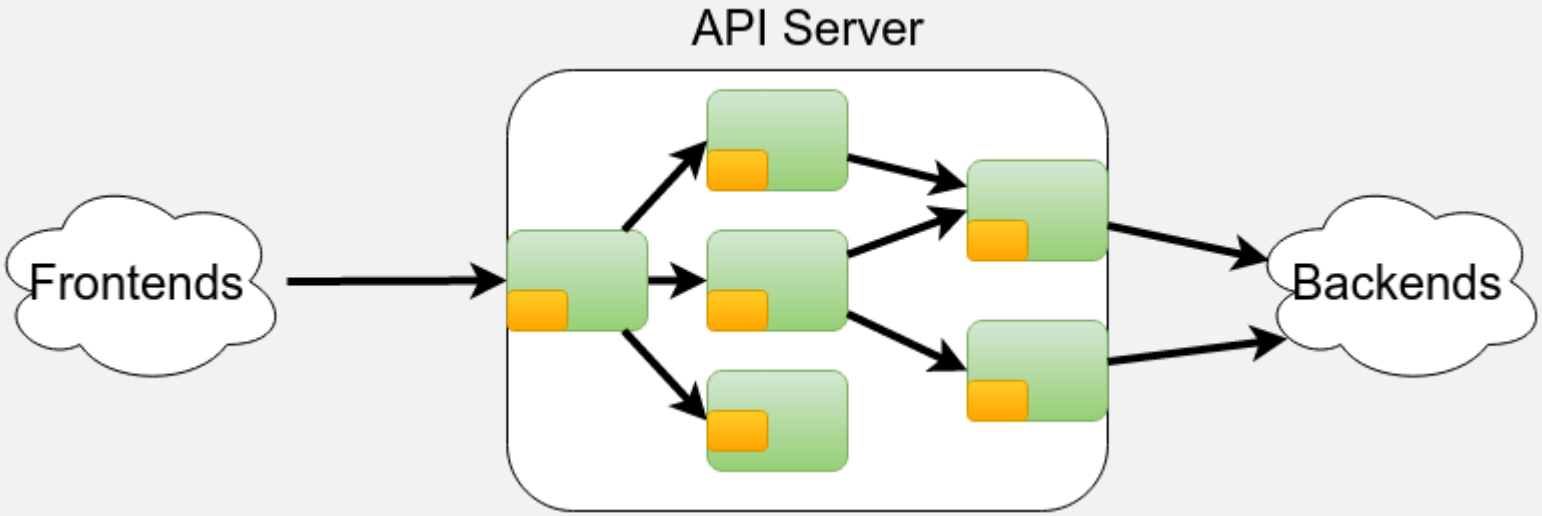
End result: Spammy alerts which operators learn to ignore, missing real problems.

Many Approaches have Limited Visibility

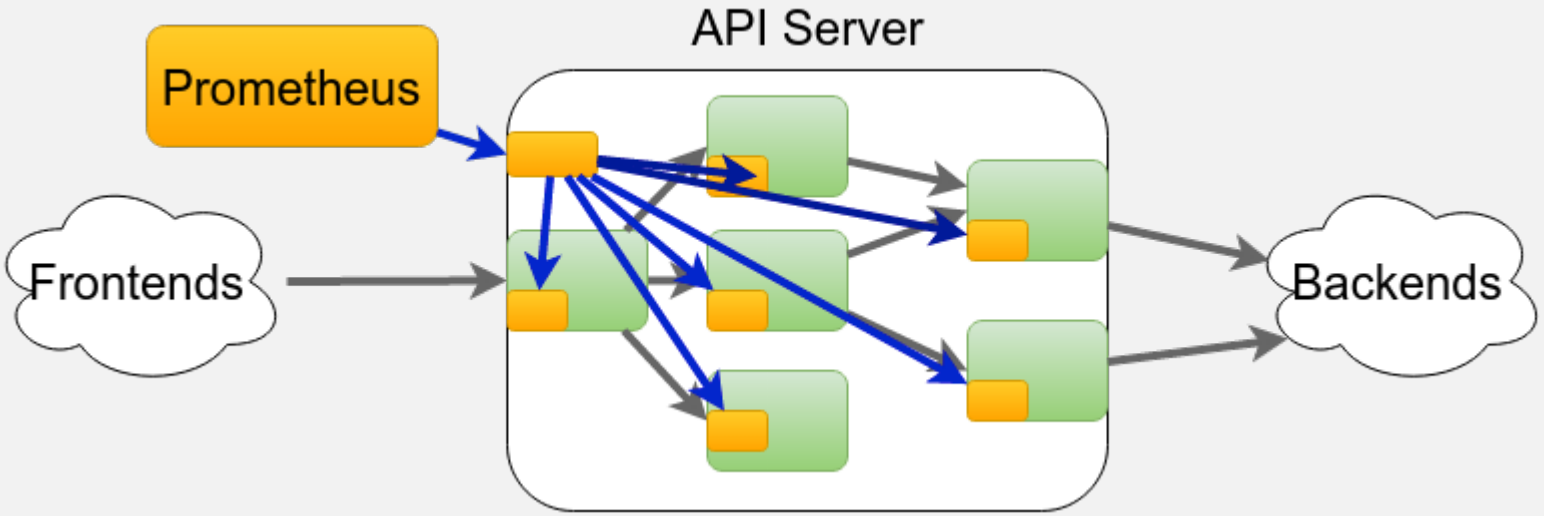


Robust Perception

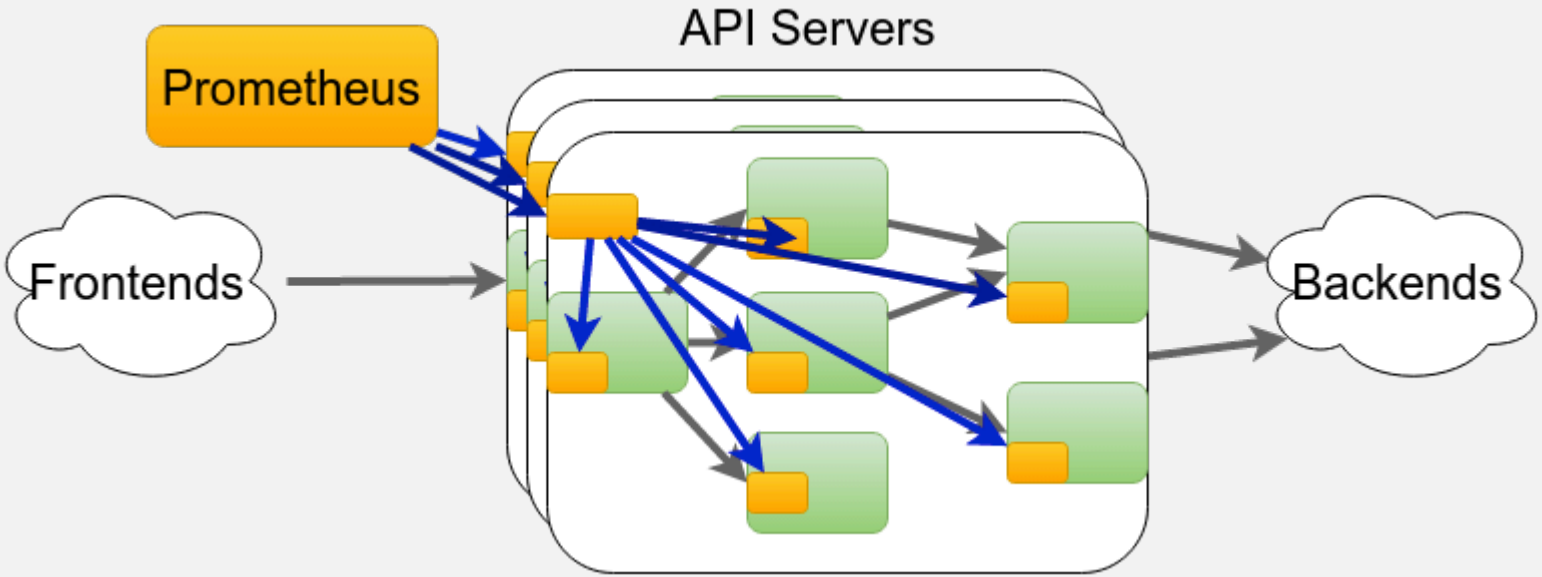
Services have Internals



Monitor the Internals



Monitor as a Service, not as Machines



Robust Perception

Freedom for Alerting

A system like Prometheus gives you the freedom to alert on whatever you like.

Alerting on error ratio across all the machines in a datacenter? No problem.

Alerting on 95th percentile latency for the service being <200ms? No problem.

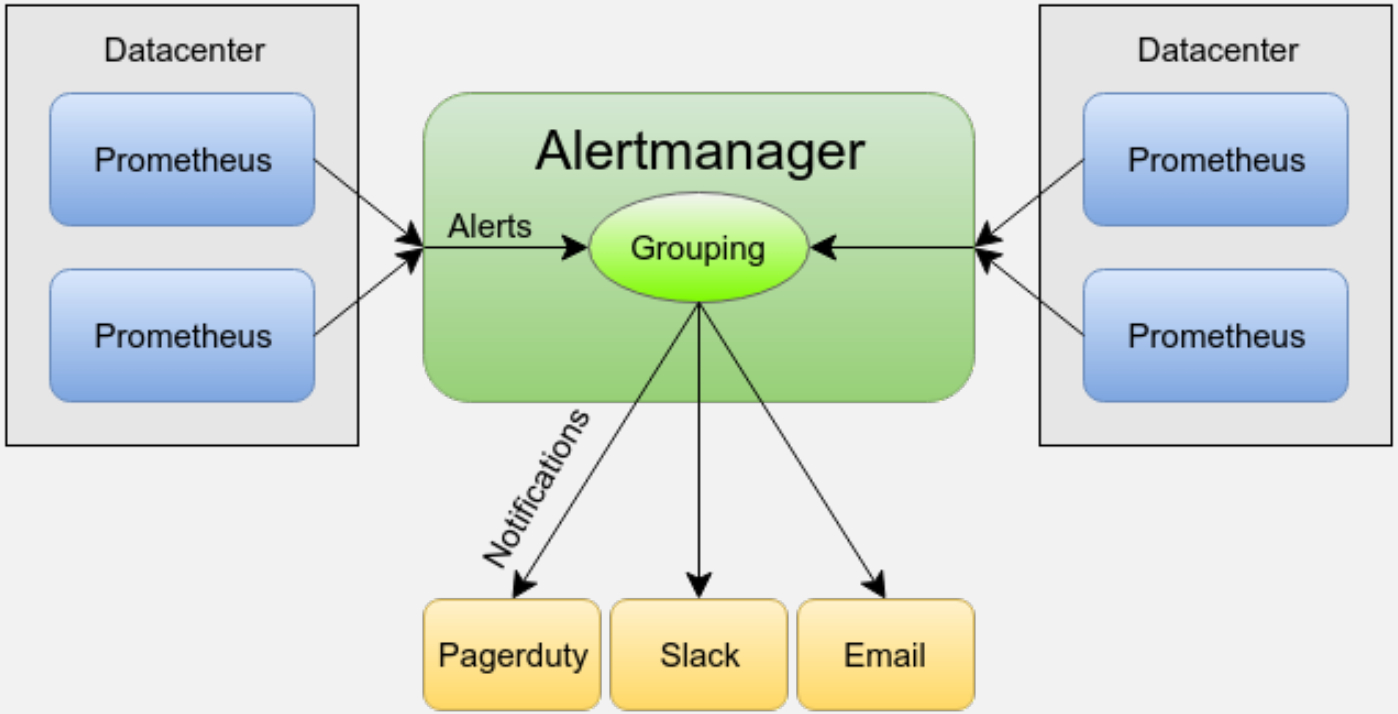
Alerting on data taking too long to get through your pipeline? No problem.

Alerting on your VIP not giving the right HTTP response codes? No problem.

Produce alerts that require intelligent human action!

Robust **Perception**

Alerting Architecture



Robust Perception

Debugging to Gain Insight

After you receive an alert notification you need to investigate it.

How do you work from a high level symptom alert such as increased latency?

You drill down through your stack with dashboards to find the subsystem that's the cause!

Dashboards

Robust Perception

Metrics from All Levels of the Stack

Many existing integrations: Java, JMX, Python, Go, Ruby, .Net, Machine, Cloudwatch, EC2, MySQL, PostgreSQL, Haskell, Bash, Node.js, SNMP, Consul, HAProxy, Mesos, Bind, CouchDB, Django, Mtail, Heka, Memcached, RabbitMQ, Redis, RethinkDB, Rsyslog, Meteor.js, Minecraft and Factorio.

Graphite, Statsd, Collectd, Scollector, Munin, Nagios integrations aid transition.

It’s so easy, most of the above were written without the core team even knowing about them!

Metrics are just one Tool

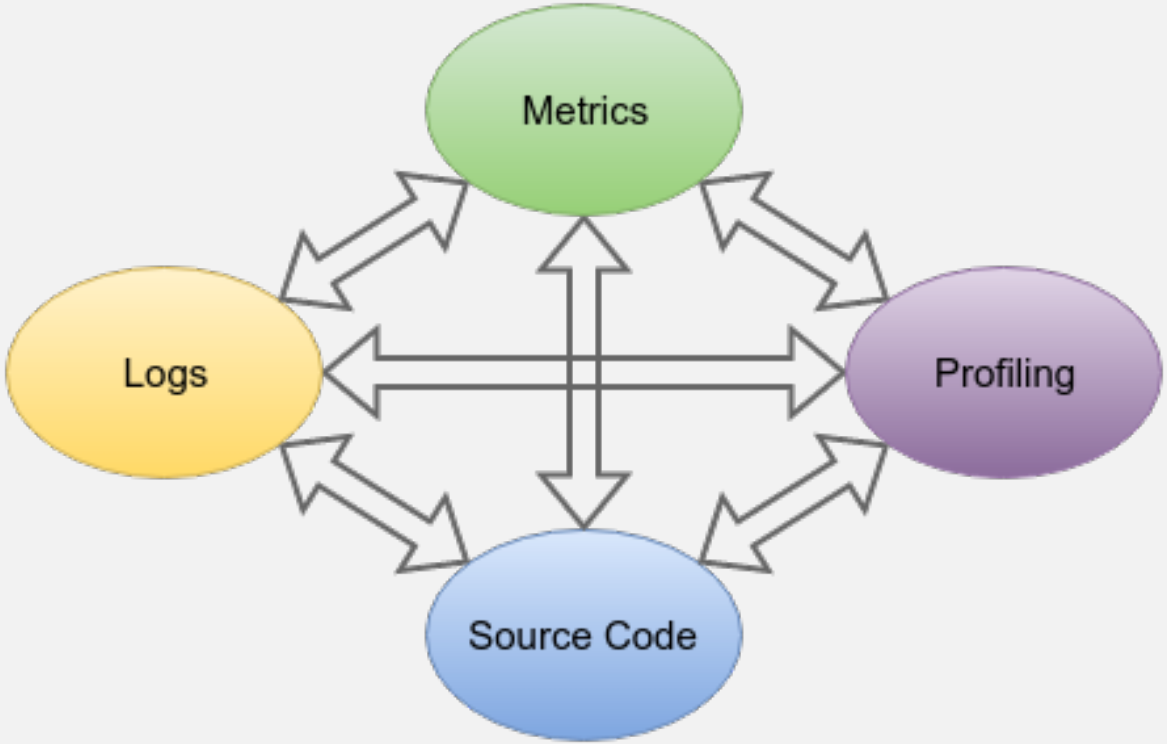
Metrics are good for alerting on issues and letting you drill down the focus of your debugging.

Not a panacea though, as with all approaches fundamental limitations on data volumes.

For successful debugging of complex problems you need a mix of logs, profiling and source code analysis.

Robust **Perception**

Complementary Debugging Tools



Robust Perception

Trending and Reporting

Alerting and debugging is short term.

Trending is medium to long term.

How is cache hit rate changing over time?

Is anyone still using that obscure feature?

With Prometheus you can do analysis beyond this.

Powerful Query Language

Can multiply, add, aggregate, join, predict, take quantiles across many metrics in the same query. Can evaluate right now, and graph back in time.

Answer questions like:

What's the 95th percentile latency in each datacenter over the past month?

How full will the disks be in 4 days?

Which services are the top 5 users of CPU?

Example: Top 5 Docker images by CPU

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

Structured Data: Labels

Prometheus doesn’t use dotted.strings like metric.grafnacon.nyc.

Multi-dimensional labels instead like
metric{event=”grafanacon”,aircraft_carrier_location=”nyc”}

Can aggregate, cut, and slice along them.

Can come from instrumentation, or be added based on the service you are monitoring.

Example: Labels from Node Exporter

Element	Value
node_network_receive_bytes{device="docker0",instance="192.168.1.73:9100",job="node"}	15158034
node_network_receive_bytes{device="eth0",instance="192.168.1.73:9100",job="node"}	130671075
node_network_receive_bytes{device="lo",instance="192.168.1.73:9100",job="node"}	18044208
node_network_receive_bytes{device="veth4a58094",instance="192.168.1.73:9100",job="node"}	3740912
node_network_receive_bytes{device="veth65f741e",instance="192.168.1.73:9100",job="node"}	1508181
node_network_receive_bytes{device="vethc9db2fc",instance="192.168.1.73:9100",job="node"}	1548023
node_network_receive_bytes{device="wlan0",instance="192.168.1.73:9100",job="node"}	1028688187

Python Instrumentation: An example

```
pip install prometheus_client
```

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

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

start_http_server(8000)
```

Robust Perception

Adding Dimensions (No Evil Twins Please)

```
from prometheus_client import Counter
REQUESTS = Counter('requests_total',
    'Total requests', ['method'])

def my_handler(request):
  REQUESTS.labels(request.method).inc()
  pass // Your code here
```

Labels go beyond Prometheus

If you're using Kubernetes, Prometheus can take in your labels and annotations too.

Similar data models and mutual integrations make your life easier!

Plumbing

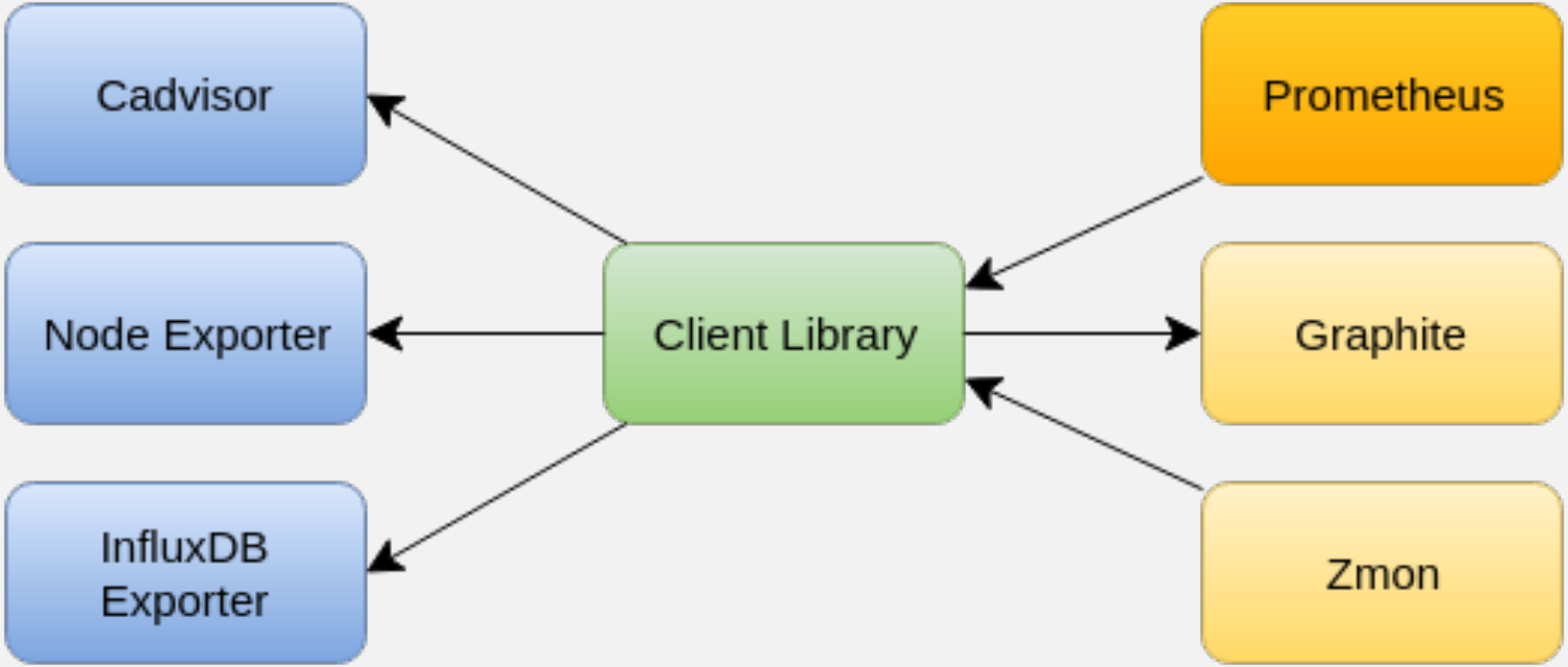
Prometheus isn't just open source, it's also an open ecosystem.

We know we can't support everything, so at every level there's a generic interface to let you get data in and/or out.

So for example if you want to run a shell script when an alert fires, you can make that happen.

Robust **Perception**

Prometheus Clients as a Clearinghouse



Live Demo!

Monitoring What Matters with Prometheus

To summarise, the key things Prometheus empowers you to build:

- Alerting on symptoms. Alerts which require intelligent human action.
- Debugging dashboards that let you drill down to where the problem is.
- The ability to run complex queries to slice and dice your data.
- Easy integration points for other systems.

These are good things to have no matter which monitoring system(s) you use.

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

#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.

#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.

#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.

Questions?

Project Website: prometheus.io

Demo: demo.robustperception.io

Company Website: www.robustperception.io