Counter, Gauge, Upper 90 - Oh my!

or, let’s learn enough to think about metrics

Amit Saha
@echorand

About me

Software Engineer at Freelancer.com

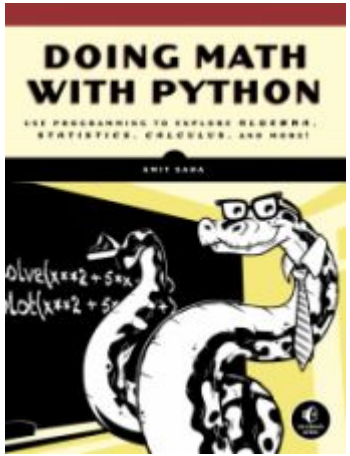
Author of “Doing Math with Python”

Past contributor to SymPy and CPython

Creator/maintainer of Fedora Scientific

Contact: @echorand, amitsaha.in@gmail.com

Blog: http://echorand.me



https://bit.ly/python-monitoring

My monitoring journey - Stage 1

Why should I monitor?

Your business needs to stay running

Understand system/application behavior

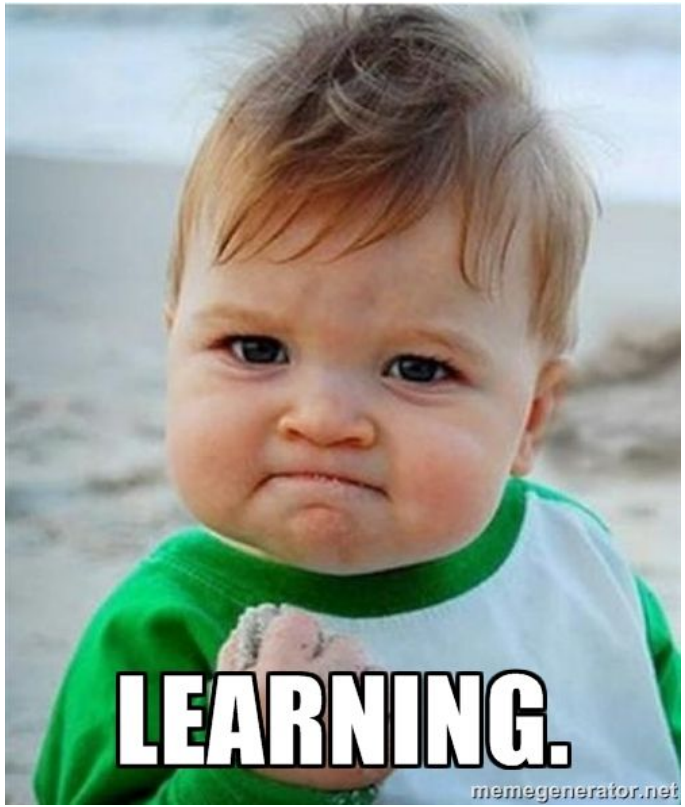
Formulate/meet Service Level Agreements (SLAs)

Capacity planning, autoscaling, hardware configuration, performance troubleshooting

Effect of software/hardware changes

Alert when abnormality occurs

My monitoring journey - Stage 2



Metric

The measure/value of a quantity at a given point of time

Source: [matplotlib examples showcase]

Metric Types

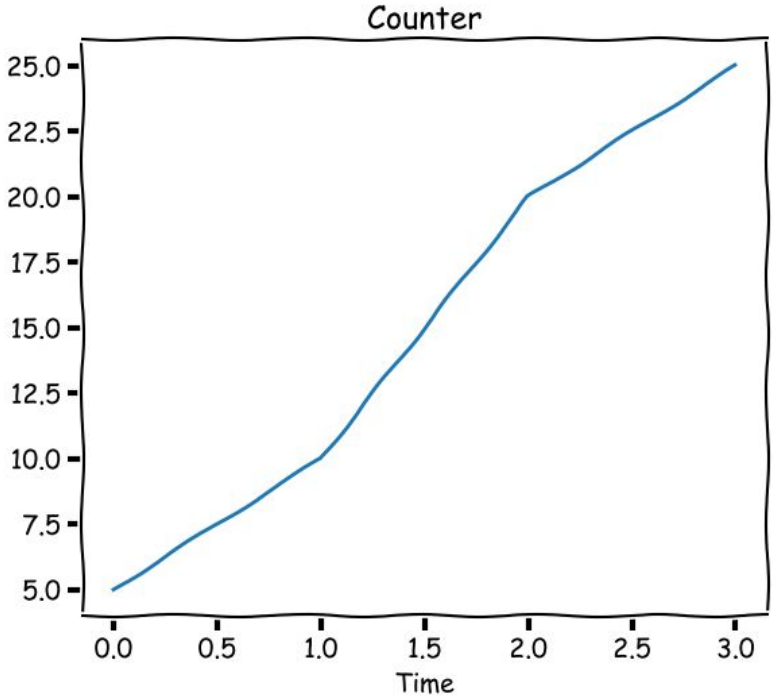
Counter

A metric whose value increases during

the lifetime of a process/system

statsd - the monitoring system is an excepti

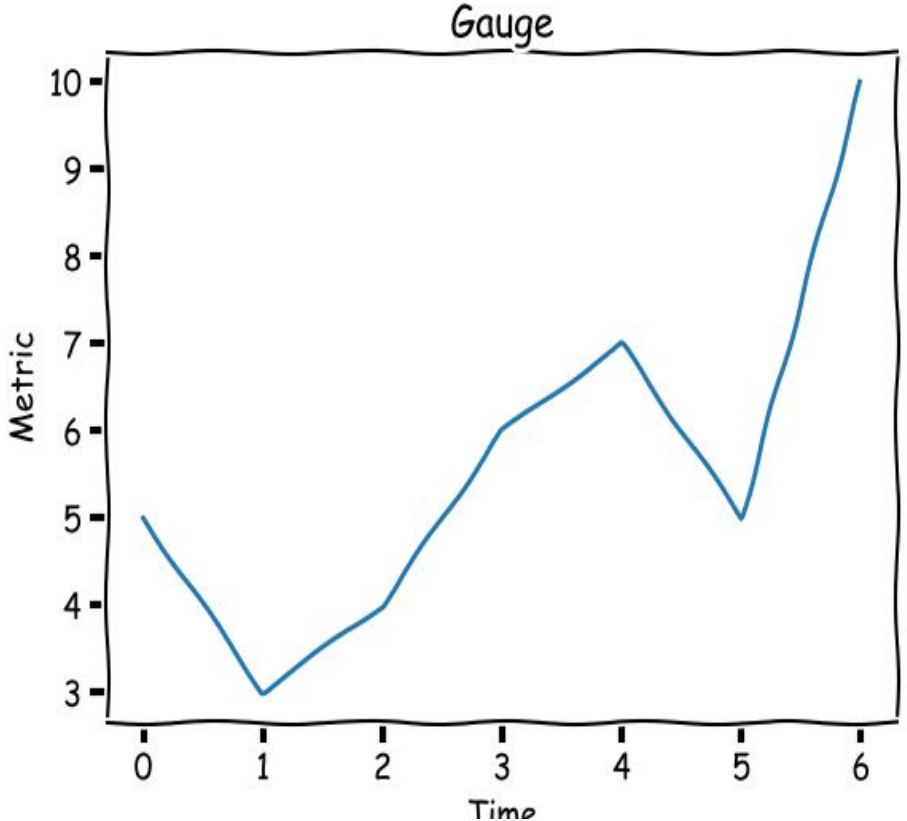
since you can *decrease* a counter too



Gauge

A metric whose value can go up or down arbitrarily - usually with a floor and ceiling

E.g. Number of pending requests in a fixed-size queue

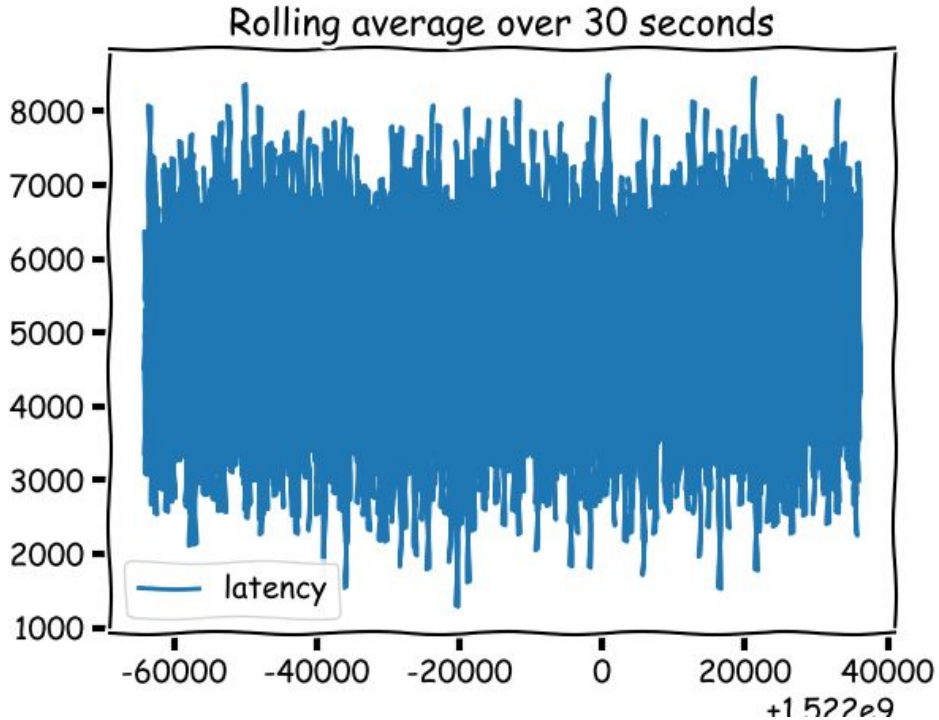


Histogram/Timer

A metric to track *observations* -

request latency for example,

Observations can be sampled



Demo time
(Demo 1)

Demo 1: What did we see?

Using flask *middleware* to calculate/report metrics

Lots of metrics generated

We need to be able to *summarize* the data

No characteristics in the metrics - which endpoint? What response status?..

Statistics

Mean and Median

Mean

Mean of 5, 8, 3 = (5+8+3)/3 = 5.33....

Median: a better average

Median of 5, 8, 3 is 5

Percentile, Quantile and Upper X

The *percentile* is a measure which gives us a measure below which a certain, k percentage of the numbers lie.

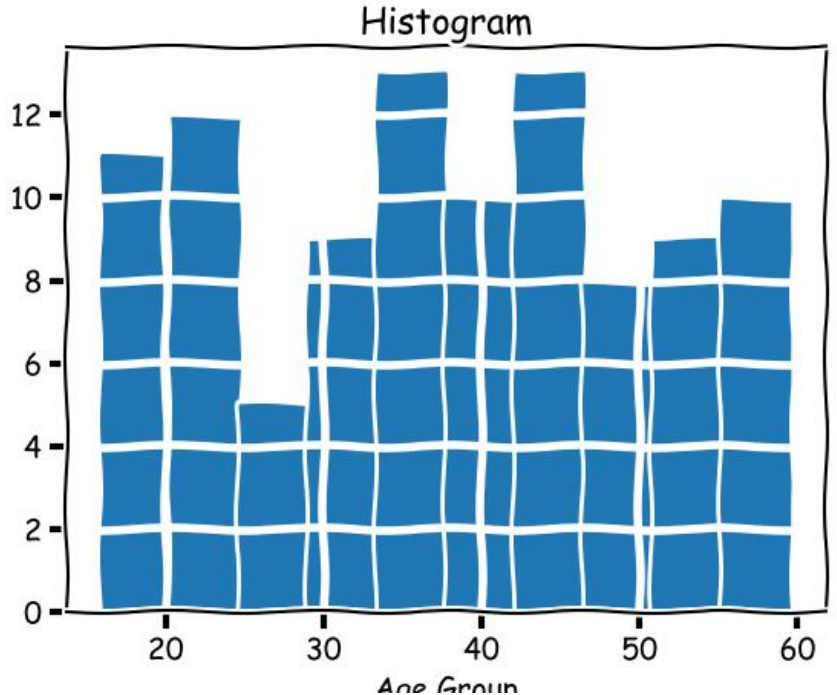
Some monitoring systems refer to the percentile measure as *upper_X* where *X* is the percentile.

A *Q-quantile* gives us another way to find a measure, k which is at a certain position *nQ* in a set of numbers of size *n*

0.xy quantile => xy percentile

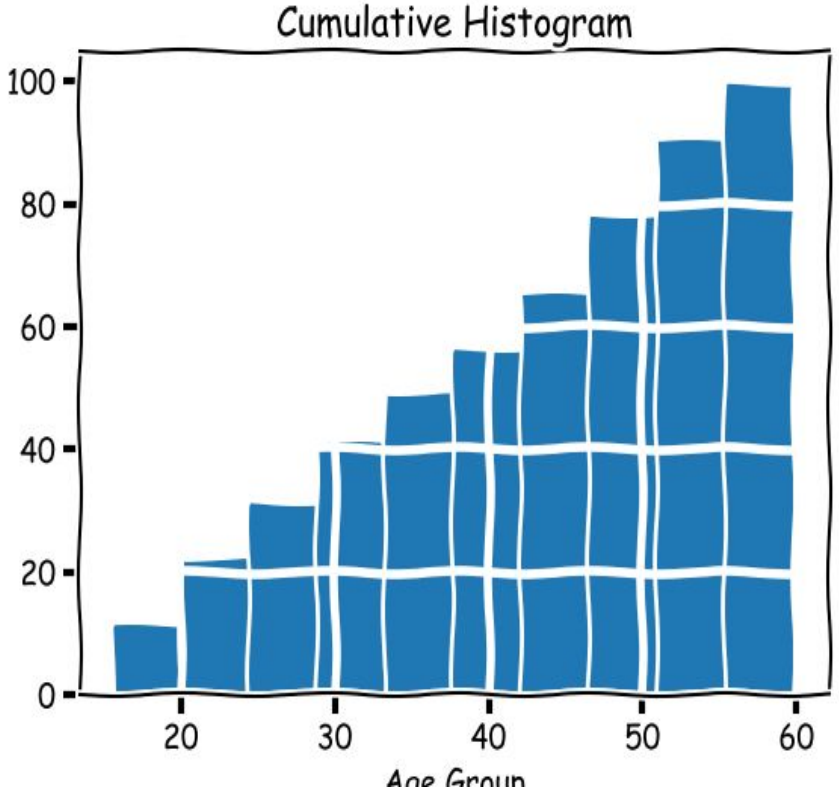
The (real) Histogram

Groups data into *buckets*



Cumulative Histogram

Groups data into buckets, but each bucket also contains the previous bucket members



Adding characteristics to metrics

Characteristics of metrics

System identifier (IP address, AWS instance ID..)

HTTP Endpoint name

HTTP Method

HTTP response status

RPC Method Name

..

Analysing metrics with *pandas*

Metrics as a *DataFrame*

The *timestamp* is the *index*

Each metric characteristic is a *column*

timestamp	app_prefix	node_id	http_endpoint	http_method	http_status	latency
1522219178	webapp1	10.1.3.4	test	GET	200	0.109911
1522219178	webapp1	10.1.3.4	test	GET	200	0.054598
1522219178	webapp1	10.1.3.4	test	GET	200	0.051498
1522219178	webapp1	10.0.1.1	test	GET	400	0.059128
1522219178	webapp1	10.1.3.4	test	GET	200	0.053644

Demo time
(Demo 2)

Monitoring your applications

1. Your application calculates the metrics

2. A monitoring system stores these

3. Human/machine queries the monitoring system

Integrating monitoring in your applications (for real)

What application metrics should I calculate?

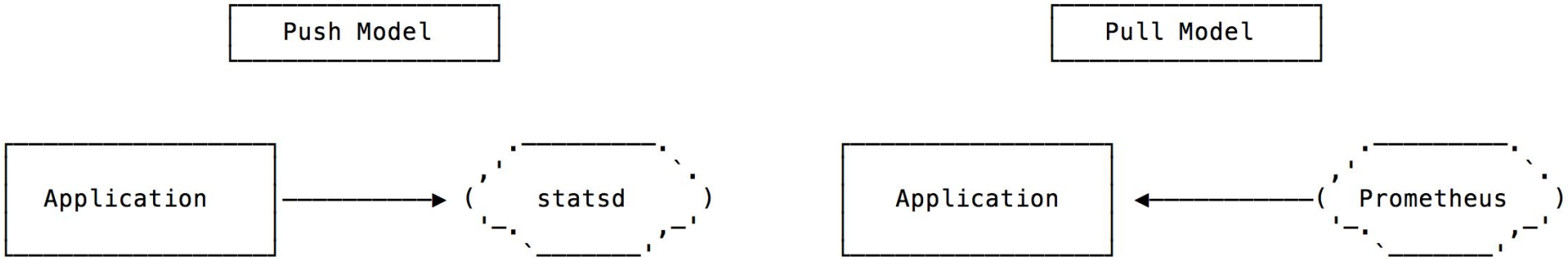
Network servers: Request latency, Queue size (if any), Exceptions, Waiting time, Worker usage

Batch jobs: Last run, latency

Consumers: Latency

Recommended: The four golden signals

Application metrics -> Monitoring System



statsd

Key statsd concepts

Application push metrics to statsd server (usually over UDP)

A metric *key* is of the form *webapp1.<key1>.<key2>...<keyN>.latency*

Each *dot* separated part of the key is a *metric characteristic/dimension* and we can perform *grouping* and *aggregation* functions over

Key statsd concepts: example keys

```
# request.<path>.<method>.http_<status_code>.latency
REQUEST_LATENCY_METRIC_KEY_PATTERN = 'instance1.{0}.{1}.http_{2}.latency'

# request.<path>.<method>.http_<status_code>
REQUEST_COUNT_METRIC_KEY_PATTERN = 'instance1.request.{0}.{1}.http_{2}.count'
```

ip-10-11-12-54.*webapp1.test_endpoint.get.http_200.latency* is a valid statsd metric name

Key statsd concepts: Pushing metrics

Key statsd concepts: Grouping and Aggregation

```
scaleToSeconds(sumSeries(stats.timers.webapp1.*.*.*.http_200.latency.count),60))
```

```
groupByNode(stats.timers.webapp1.*.*.*.upper_90, 3, 'maxSeries')
```

Prometheus

Key prometheus concepts

Application exposes a HTTP endpoint for prometheus to scrape

A metric is an descriptive name of the form *request_latency_ms*

Each metric can be associated with multiple *labels* which are the characteristics of the metric

Internally, each *metric* and *label* combination is a separate metric

Key prometheus concepts: endpoint

```
@app.route('/metrics')
def metrics():
    return Response(prometheus_client.generate_latest(), mimetype=CONTENT_TYPE_LATEST)
```

Key prometheus concepts: Metric definition

```
REQUEST_COUNT = Counter(
    'request_count_total', 'App Request Count',
    ['app_name', 'method', 'endpoint', 'http_status']
)
REQUEST_LATENCY = Histogram('request_latency_seconds', 'Request latency',
    ['app_name', 'endpoint']
)
```

Key prometheus concepts: Metric updates

```
REQUEST_LATENCY.labels('webapp', request.path).observe(resp_time)

REQUEST_COUNT.labels('webapp', request.method, request.path, response.status_code).inc()
```

Key prometheus concepts: Grouping and Aggregation

```
max(request_latency_ms{label1="value1", label2="value2", quantile="0.99"}) by  (label2)
```

Statsd or Prometheus?

Statsd or Prometheus?

Native prometheus exporting in Python has certain *gotchas*

I recommend using the statsd exporter

I have written about this topic elsewhere

Summary

Metrics are hard

Metrics needs learning to get right - lots of excellent resources

But we shouldn’t ignore them, Learning by doing

My monitoring journey - Stage 3

https://bit.ly/python-monitoring

Feedback? Questions?

@echorand
amitsaha.in@gmail.com

Thank You!

I am talking at PyCon US

Saturday 11:30 a.m.–noon in Grand Ballroom B