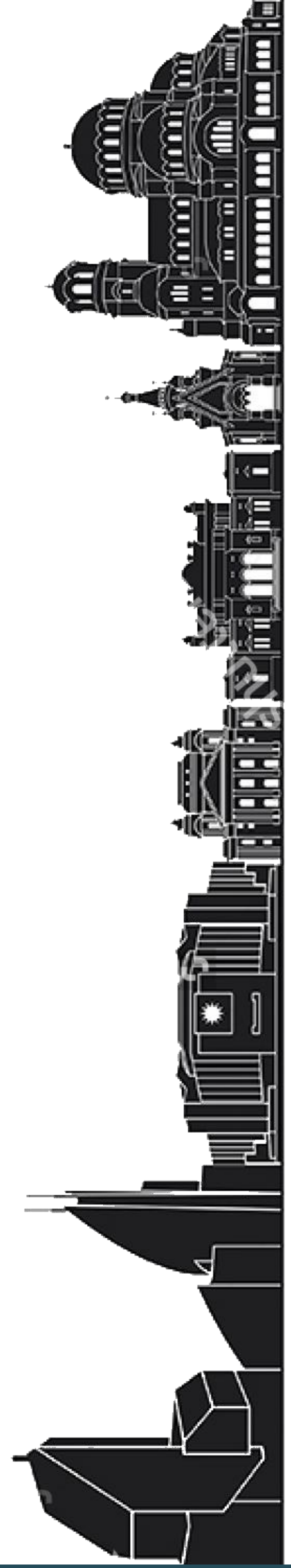
Performance Monitoring for the Cloud

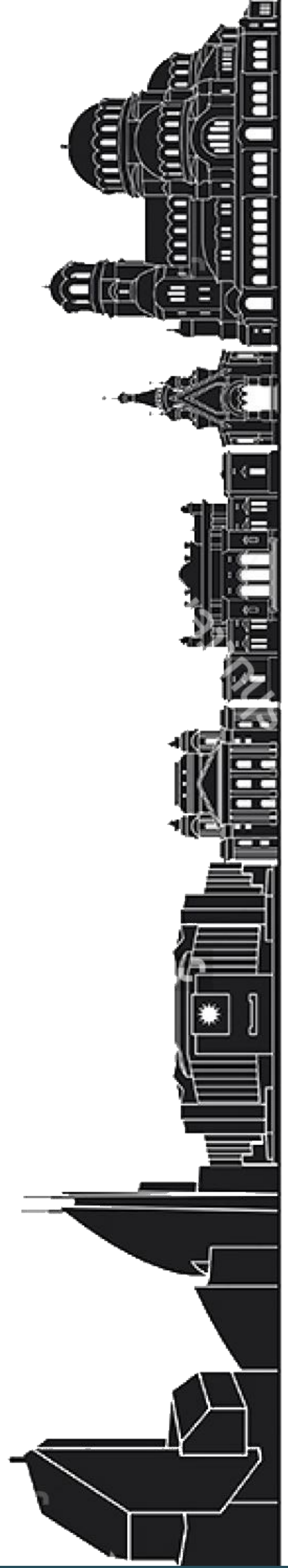
Werner Keil
JSR 363 Maintenance Lead
@wernerkeil

October 18, 2017



Agenda

1. Introduction
2. Performance Co-Pilot
3. Dropwizard Metrics
4. MicroProfile Metrics
5. Prometheus
6. StatsD
7. Demo



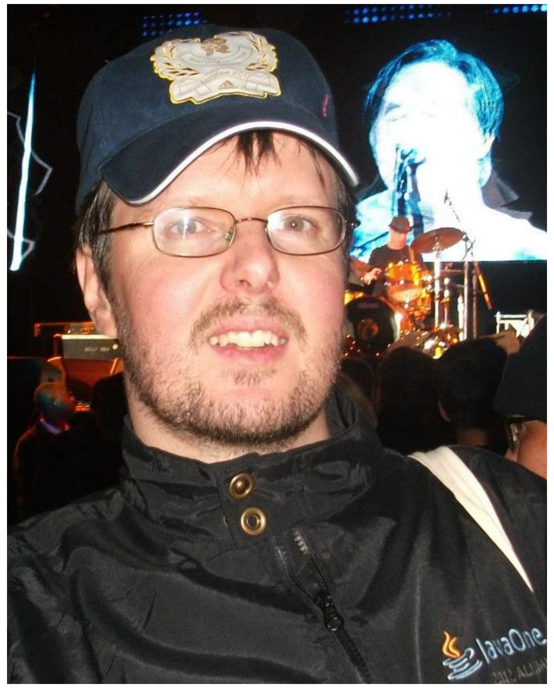
Who am I?

Werner Keil

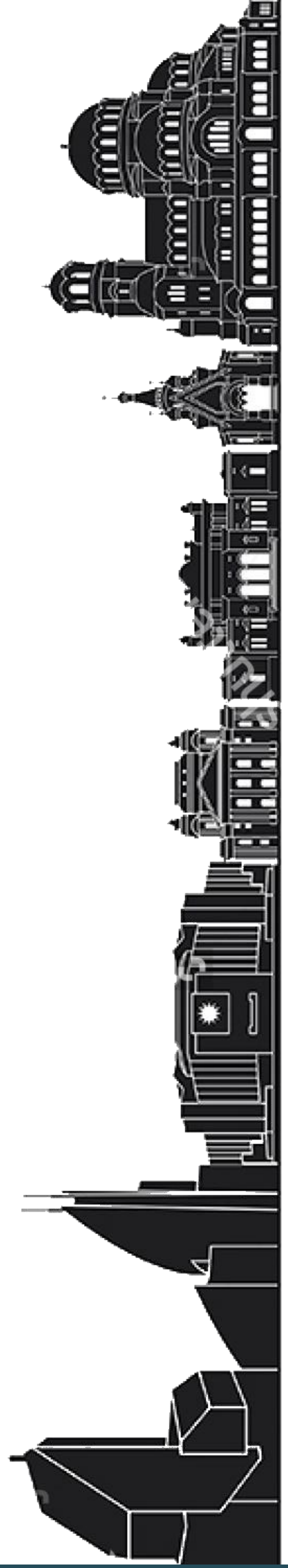
[www.linkedin.com/in/catmedia]

- Consultant – Coach
- Creative Cosmopolitan
- Open Source Evangelist
- Software Architect
- Spec Lead – JSR363
- Individual JCP Executive Committee Member

Twitter @wernerkeil



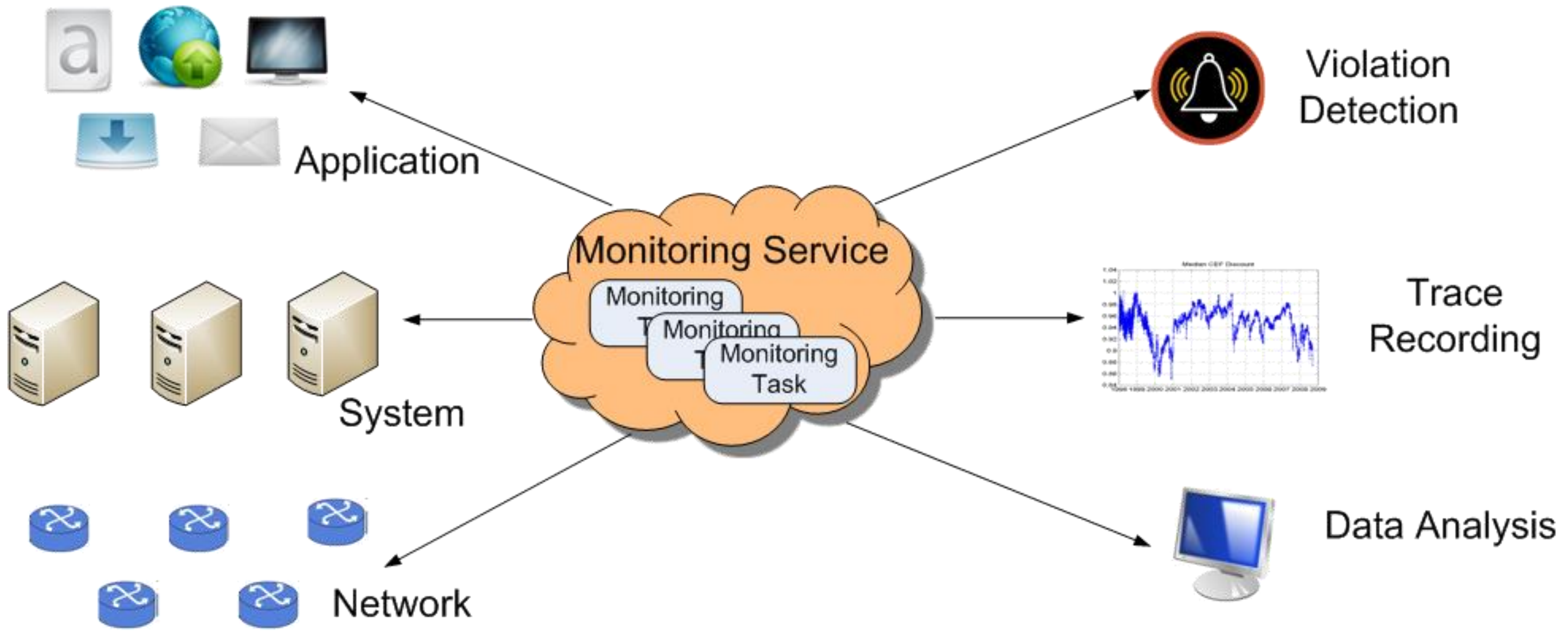
What is Monitoring?

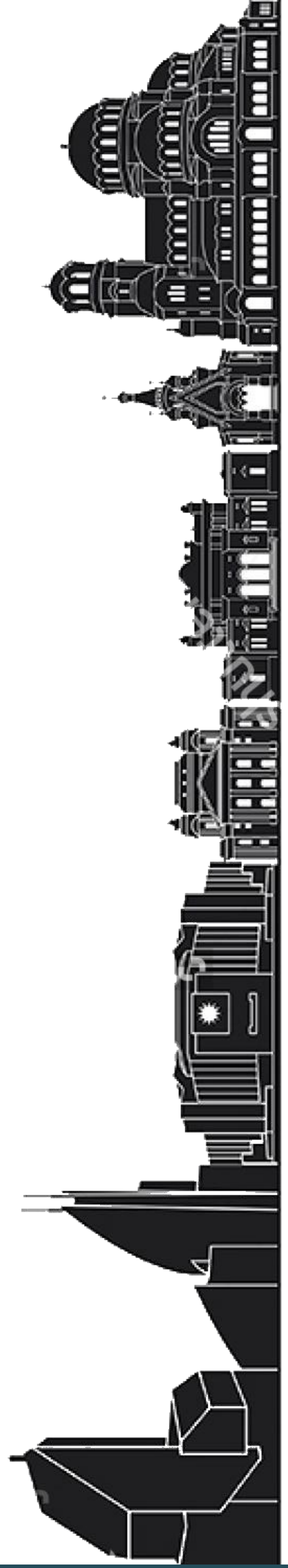


Monitoring applications is observing, analyzing and manipulating the execution of these applications, which gives information about threads, CPU usage, memory usage, as well as other information like methods and classes being used.

A particular case is the monitoring of distributed applications, aka the Cloud where an the performance analysis of nodes and communication between them pose additional challenges.

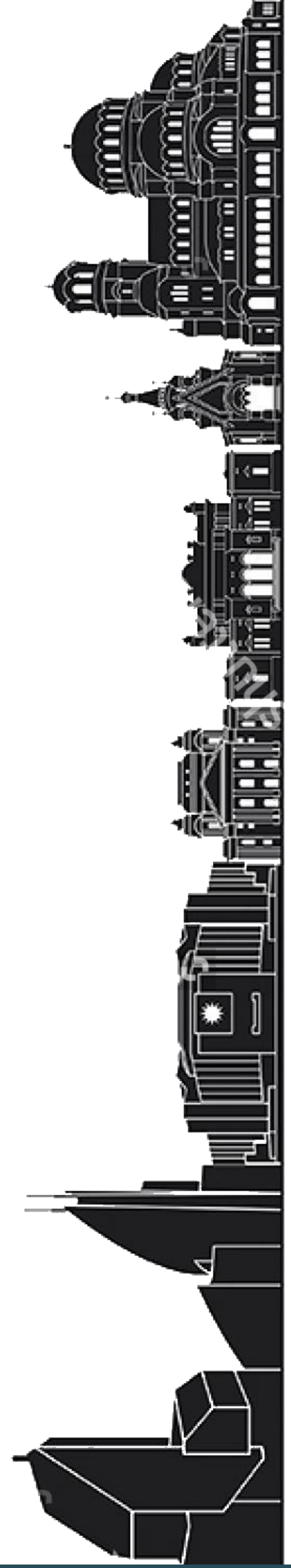
A high-level view of Cloud Monitoring





Challenges at System Level

- **Efficient Scalability**
  - **Supporting tens of thousands of monitoring tasks**
  - **Cost effective: minimize resource usage**
- **Monitoring QoS**
  - **Multi-tenancy environment**
  - **Minimize resource contention between monitoring tasks**
- **Implication of Multi-Tenancy**
  - **Monitoring tasks: adding, removing**
  - **Resource contention between monitoring tasks**



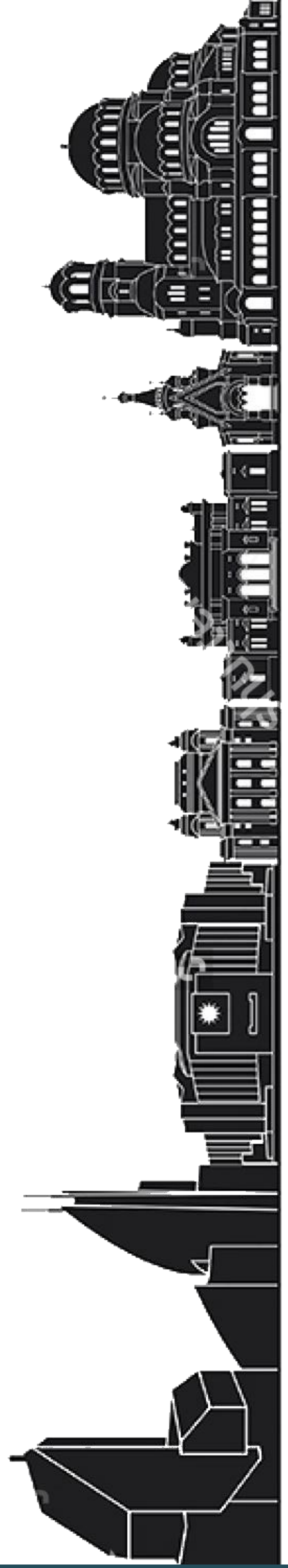
Performance vs Number of Hosts

60 items per host, update frequency once per minute

Number of hosts	Performance (values per second)
100	100
1000	1000
10000	10000

600 items per host, update frequency once per minute

Number of hosts	Performance (values per second)
100	1000
1000	10000
10000	100000



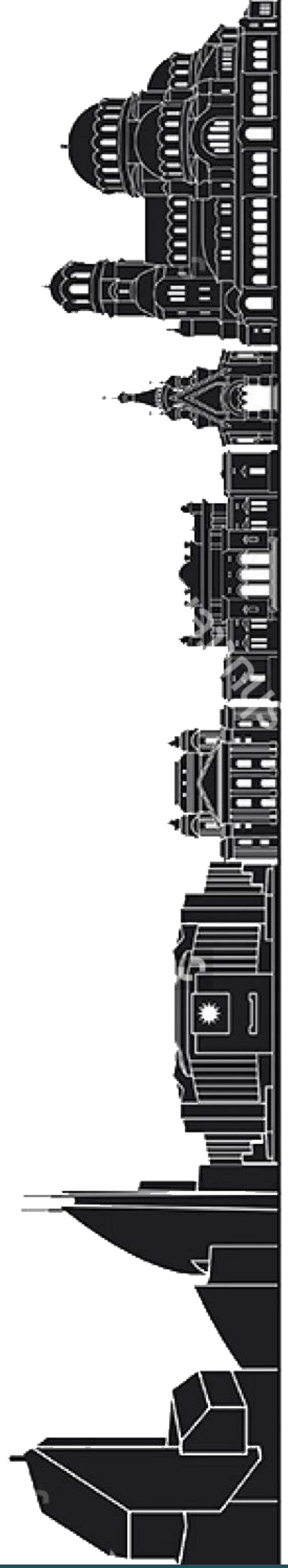
Monitoring Tips

- Regularly apply “Little’s Law” to all data... generic (queueing theory) form:

  $Q = \lambda R$

- Length = Arrival Rate x Response Time
  - e.g. 10 MB = 2 MB/sec x 5 sec

- Utilization = Arrival Rate x Service Time
  - e.g. 20% = 0.2 = 100 msec/sec x 2 sec



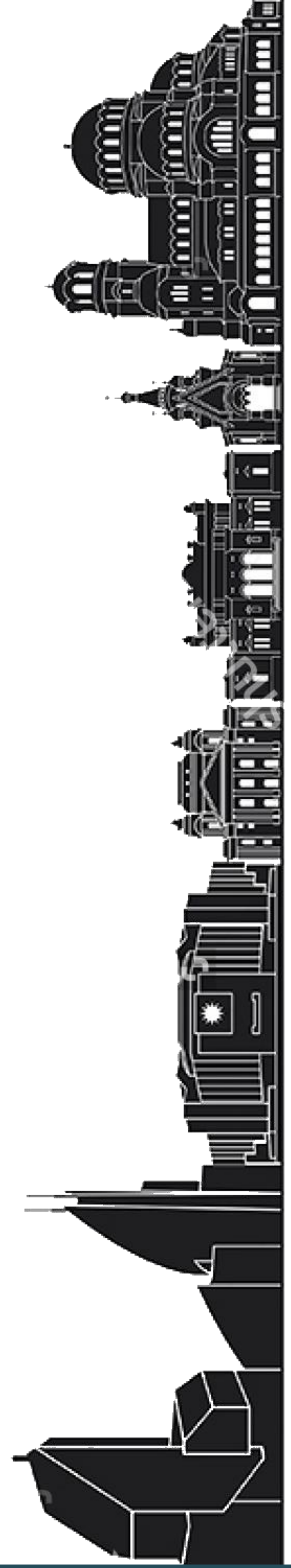
Types of Monitoring

Monitoring Logs

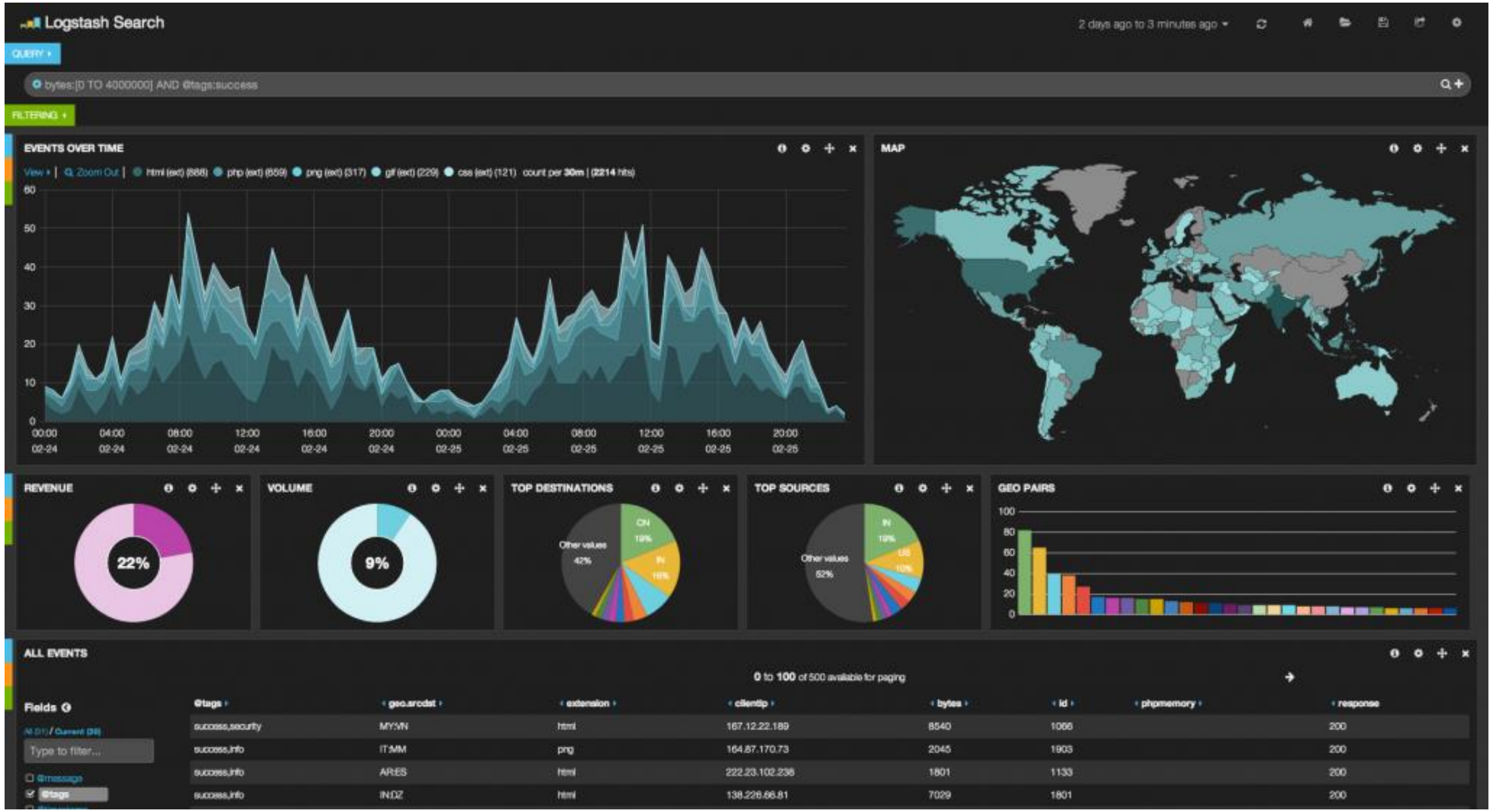
- Logstash
- Redis
- Elasticsearch
- Kibana Dashboard

## Monitoring Performance

- Collectd
- Statsd
- PCP
- Graphite
- Database (eg: PSQL)
- Grafana Dashboard

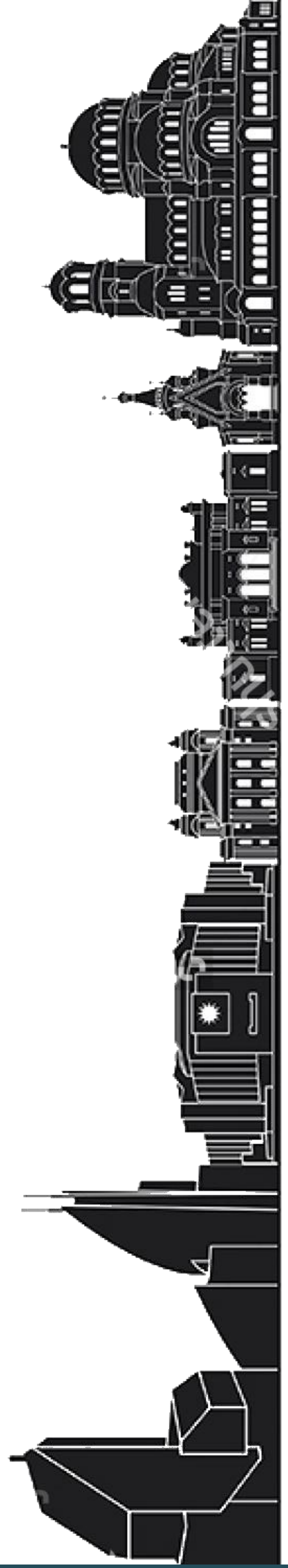


Monitoring Logs – Kibana Dashboard



bytes:[0 TO 4000000] AND @tags:success

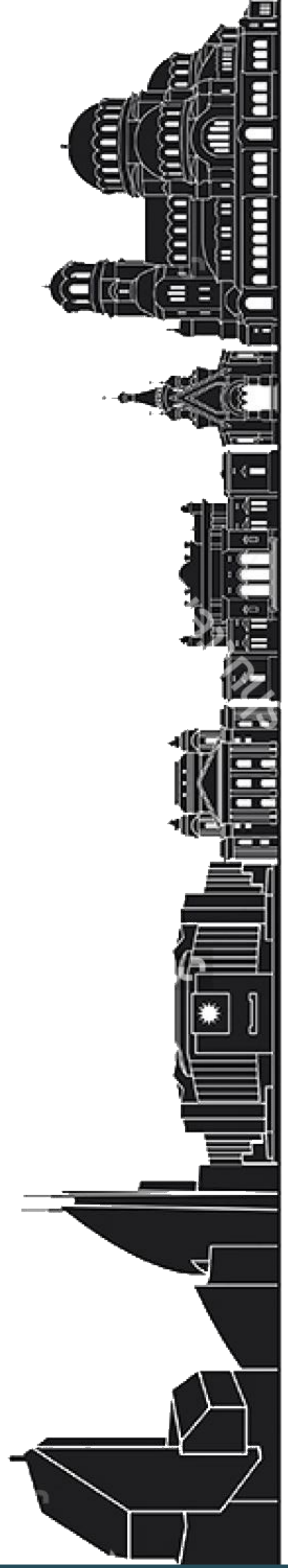
@tags	geo.srcdst	extension	clientip	bytes	id	phpmemory	response
success,security	MY:VN	html	167.12.22.189	8540	1066		200
success,info	IT:MM	png	164.87.170.73	2045	1903		200
success,info	AR:ES	html	222.23.102.236	1801	1133		200
success,info	IN:DZ	html	138.226.66.81	7029	1801		200



# Monitoring Performance

How is this traditionally done?

- rsyslog/syslog-ng/journald
- top/iostat/vmstat/ps
- Mixture of scripting languages (bash/perl/python)
- Specific tools vary per platform
- Proper analysis requires more context



Performance Co-Pilot

PCP

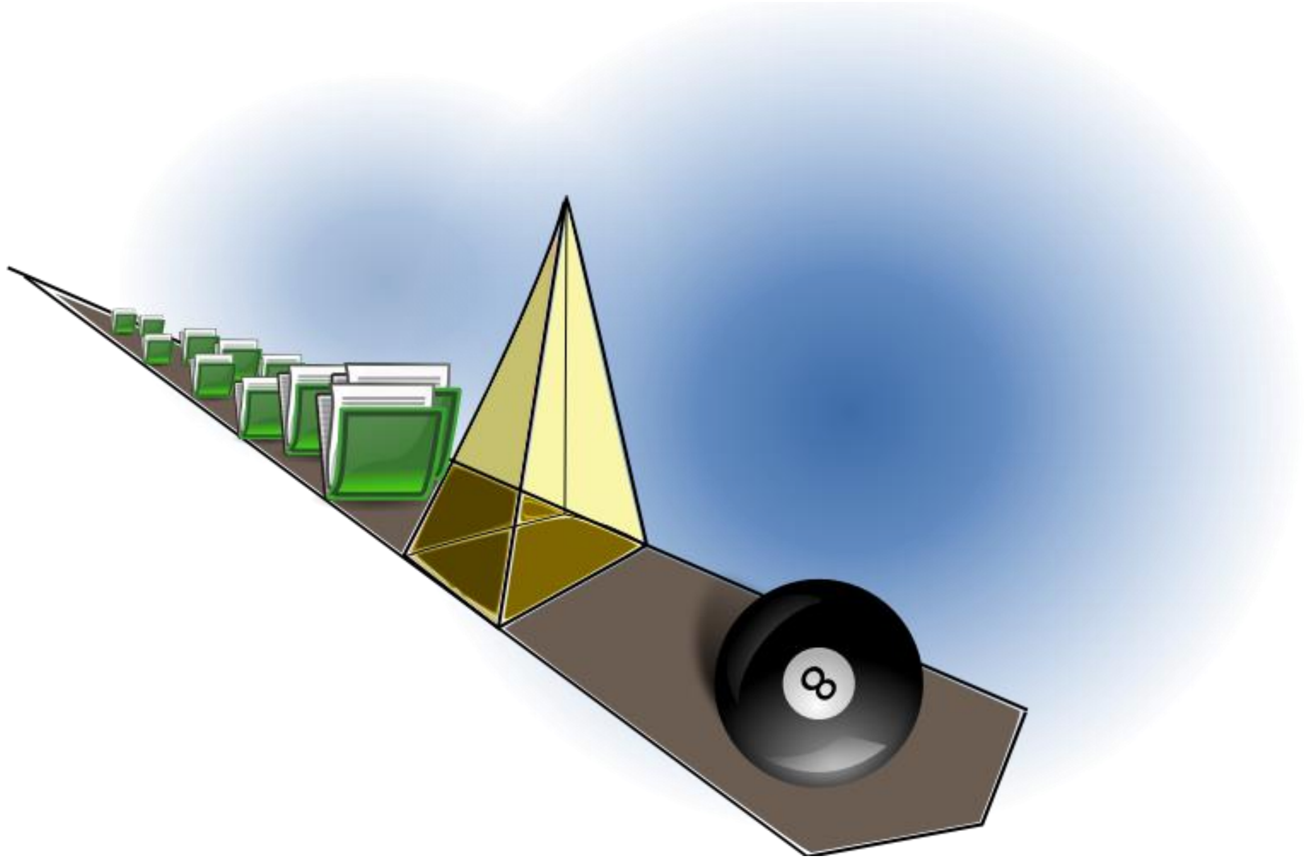
http://www.pcp.io

GitHub

https://github.com/performancecopilot

What is PCP?

- Open source toolkit
- System-level analysis
- Live and historical
- Extensible (monitors, collectors)
- Distributed
- Unix-like component design
- Cross platform
- Ubiquitous units of measurement



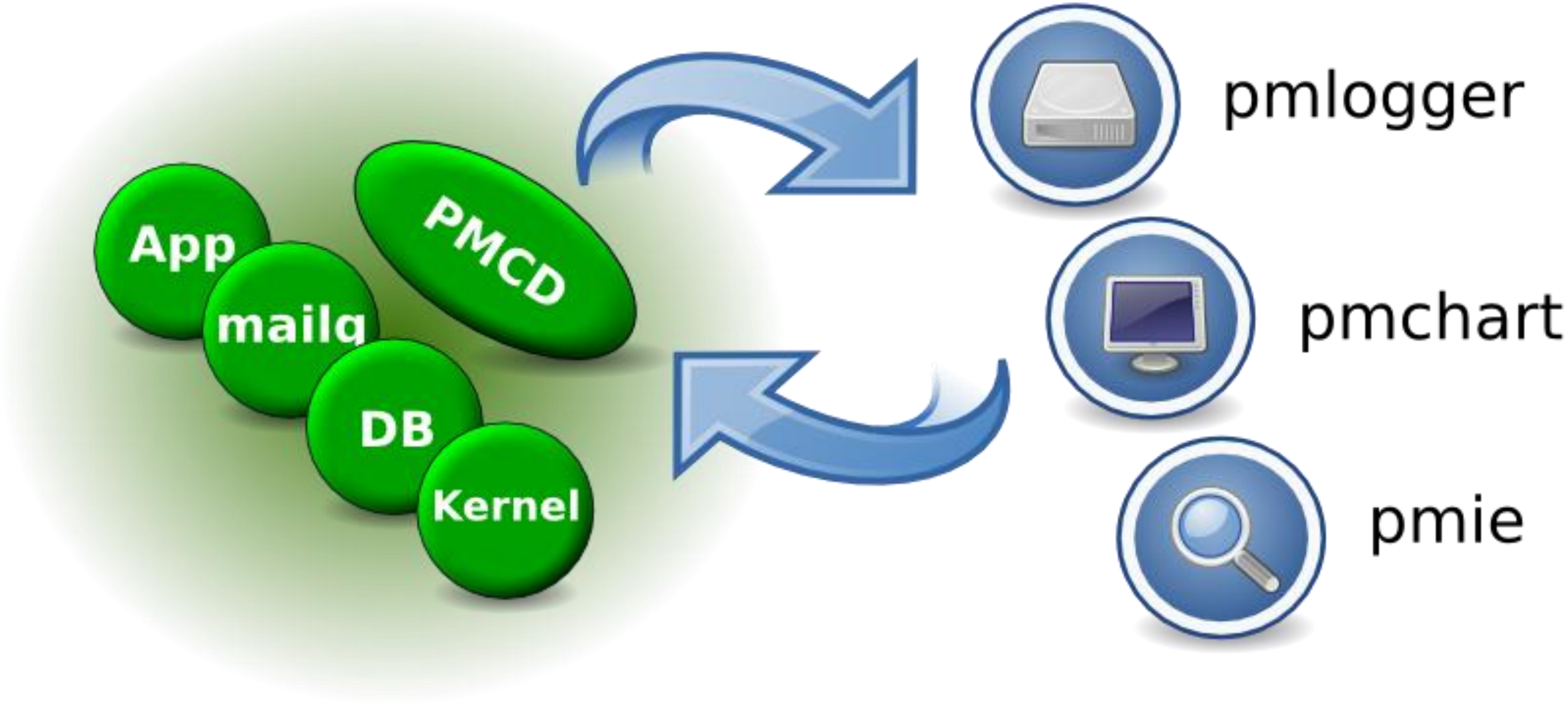
PCP Basics

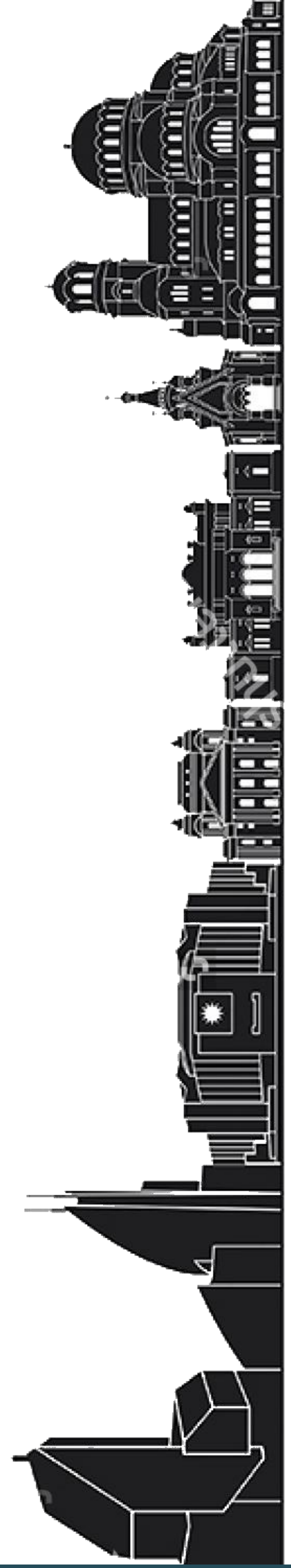
Agents and Daemons

At the core we have two basic components:

1. Performance Metric Domain Agents
   - Agents
2. Performance Metric Collection Daemon
   - PMCD

PCP Architecture

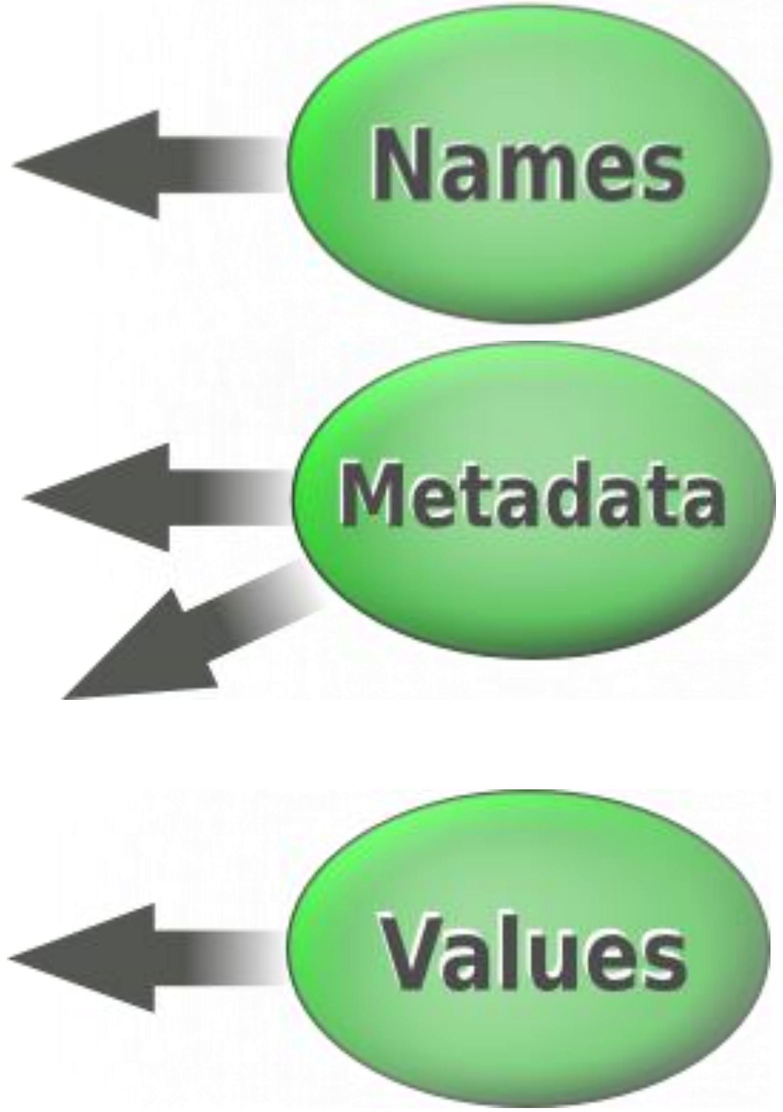




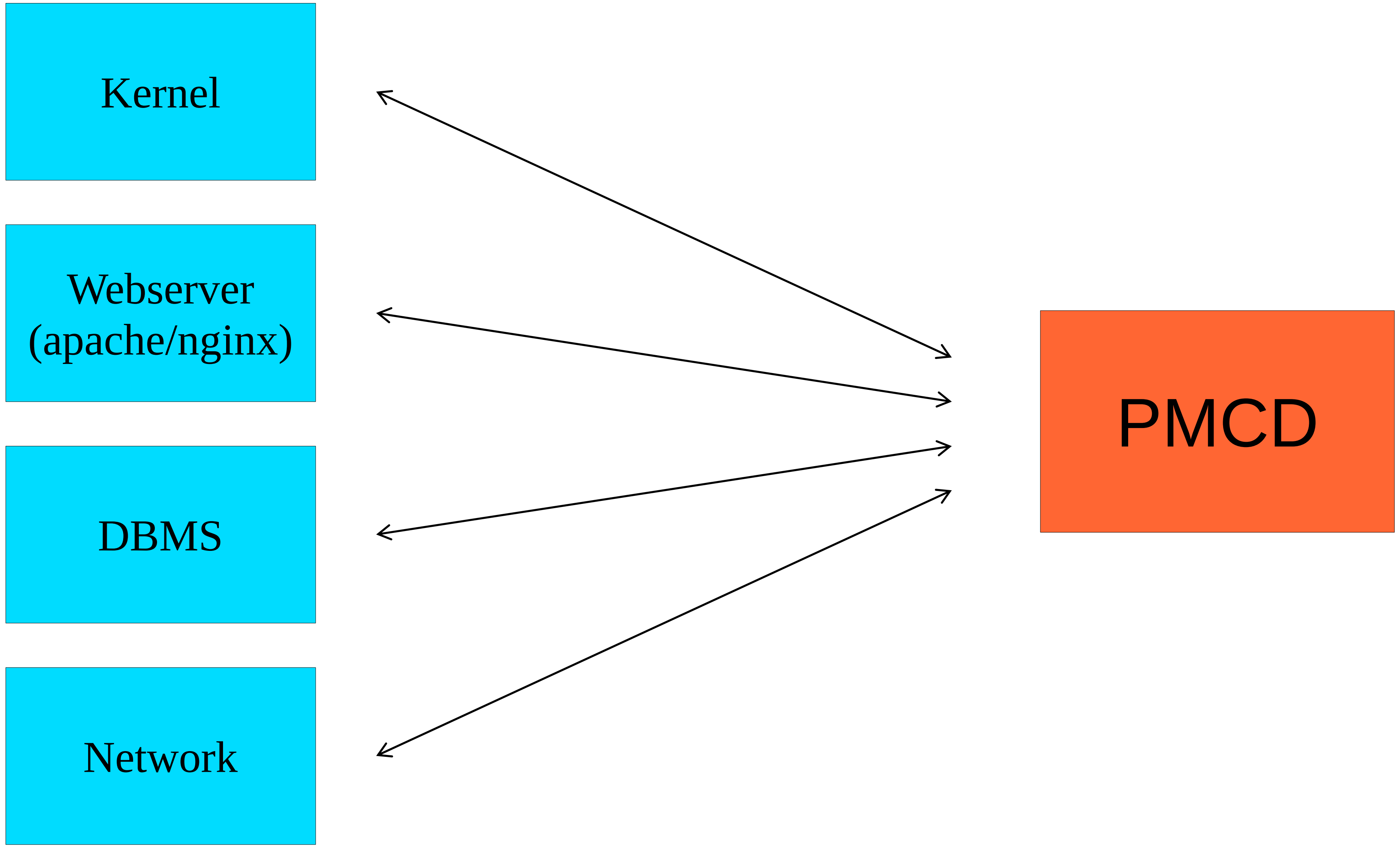
PCP Metrics

- **pminfo** --desc -tT --fetch ***disk.dev.read***

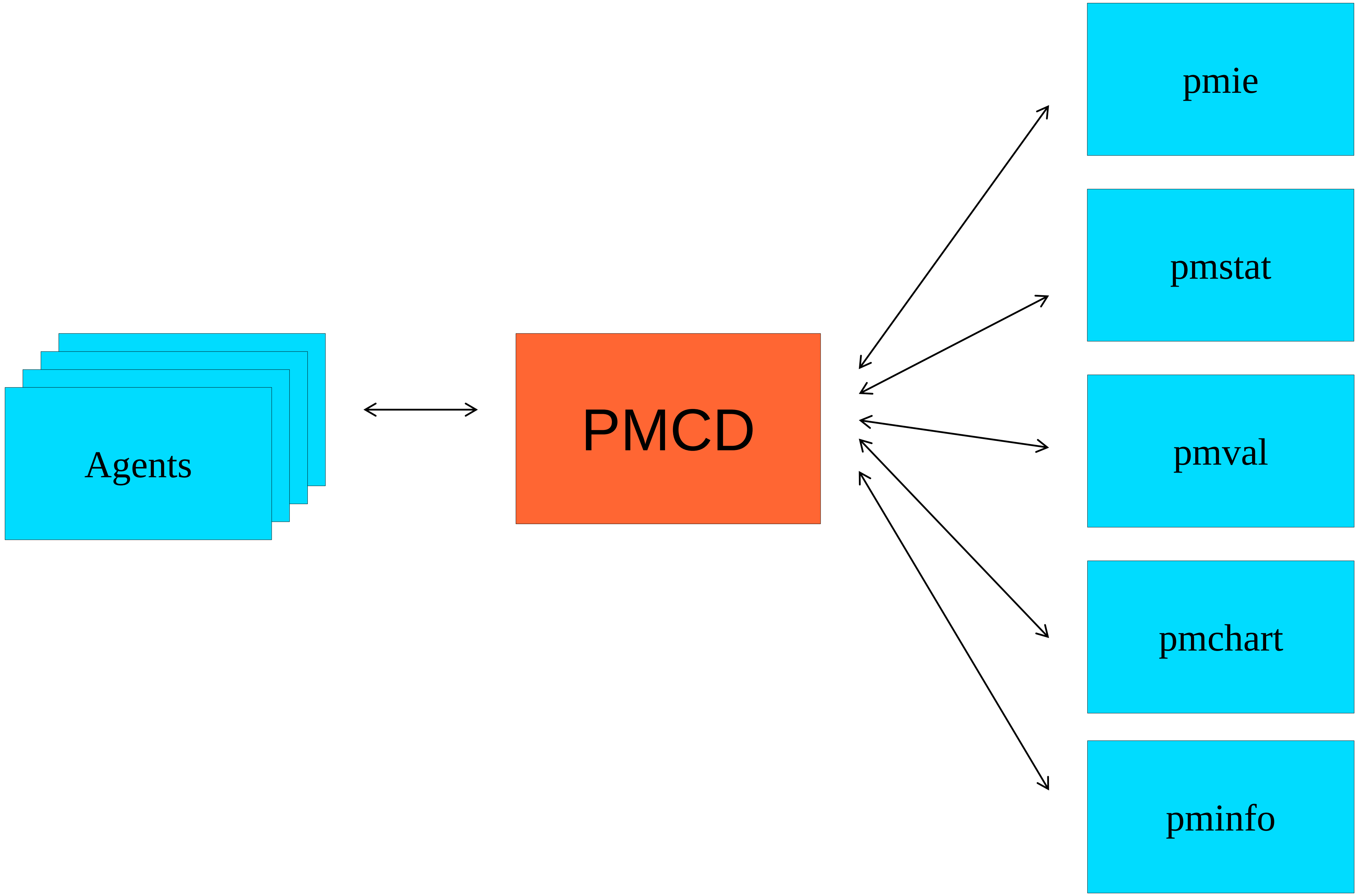
```
disk.dev.read [per-disk read operations]
Data Type: 32-bit unsigned int  InDom: 60.1
Semantics: counter  Units: count
Help: Cumulative count of disk reads since
boot time
Values:
    inst [0 or "sda"] value 3382299
    inst [1 or "sdb"] value 178421
```



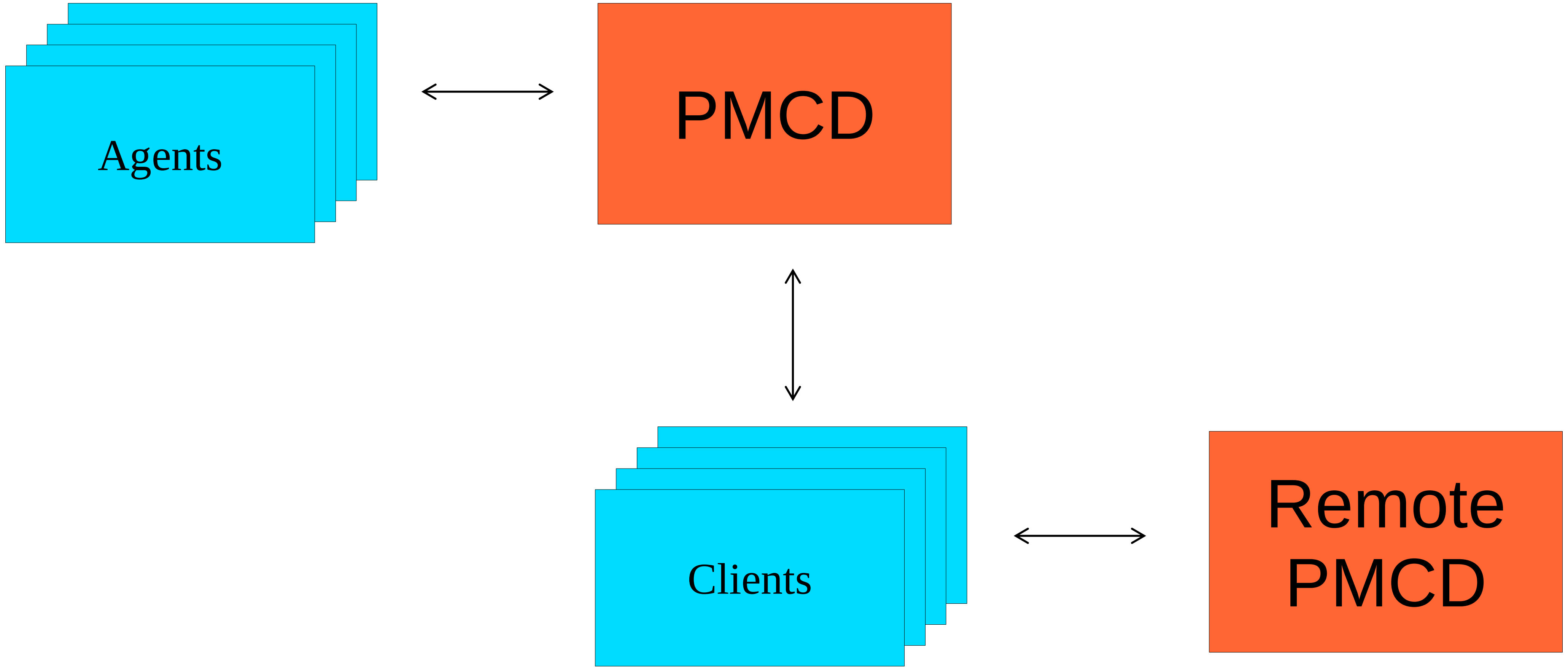
PCP Agents

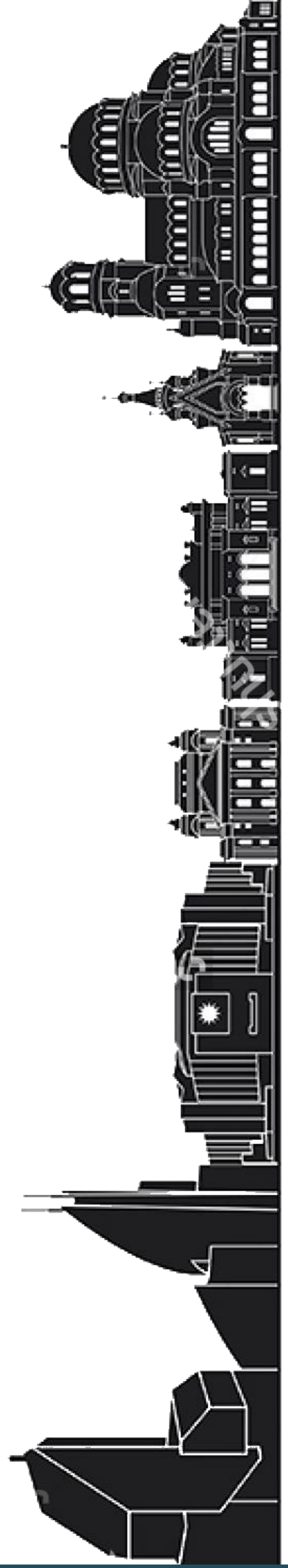


PCP Clients



PCP Remote Clients



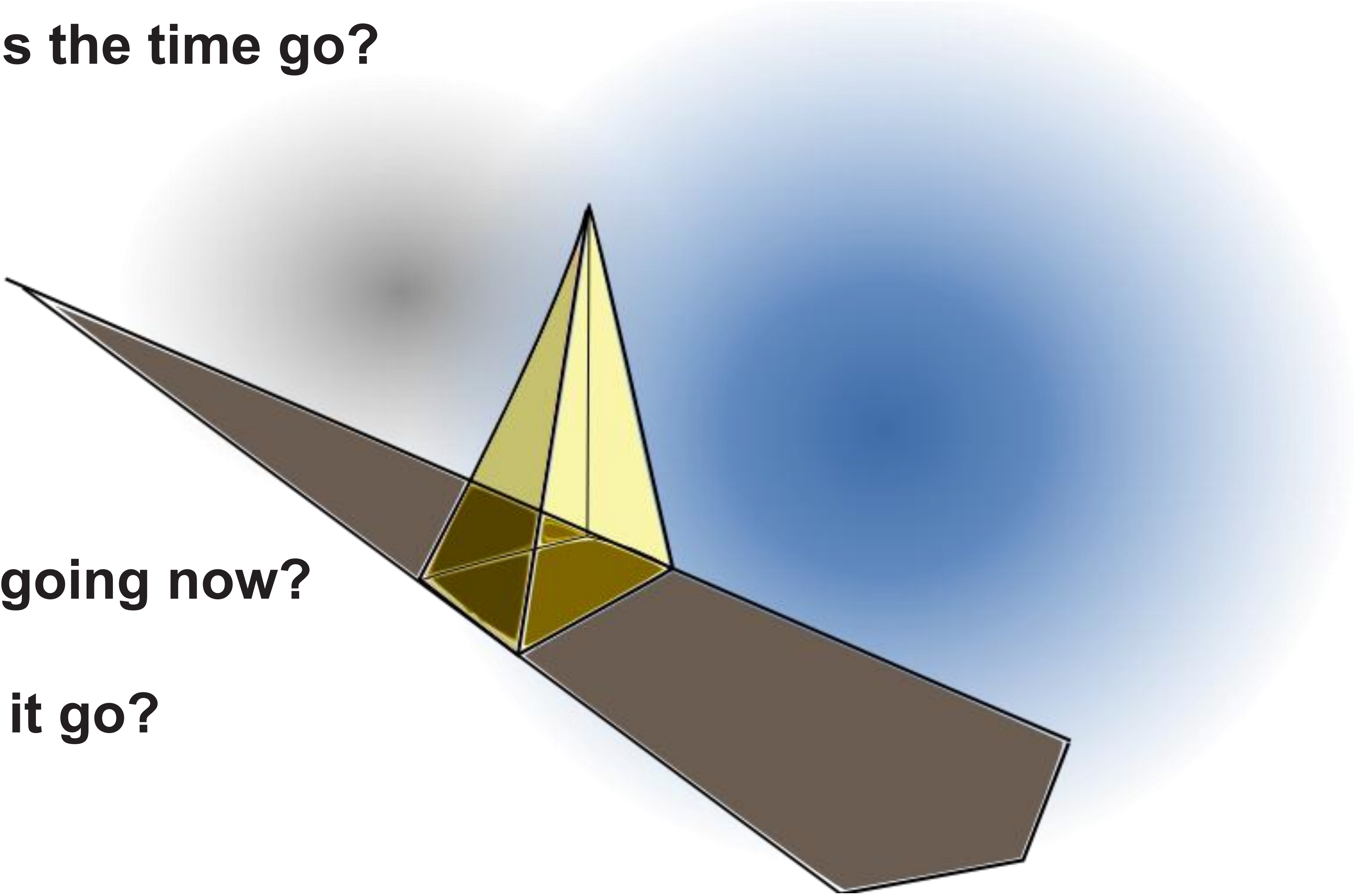


PCP Data Model

- Metrics come from one source (host / archive)
- Source can be queried at any interval by any monitor tool
- Hierarchical metric names
  e.g. disk.dev.read and aconex.response_time.avg
- Metrics are singular or set-valued (“instance domain”)
- Metadata associated with every metric
  - Data type (int32, uint64, double, ...)
  - Data semantics (units, scale, ...)
  - Instance domain

Performance Timeline

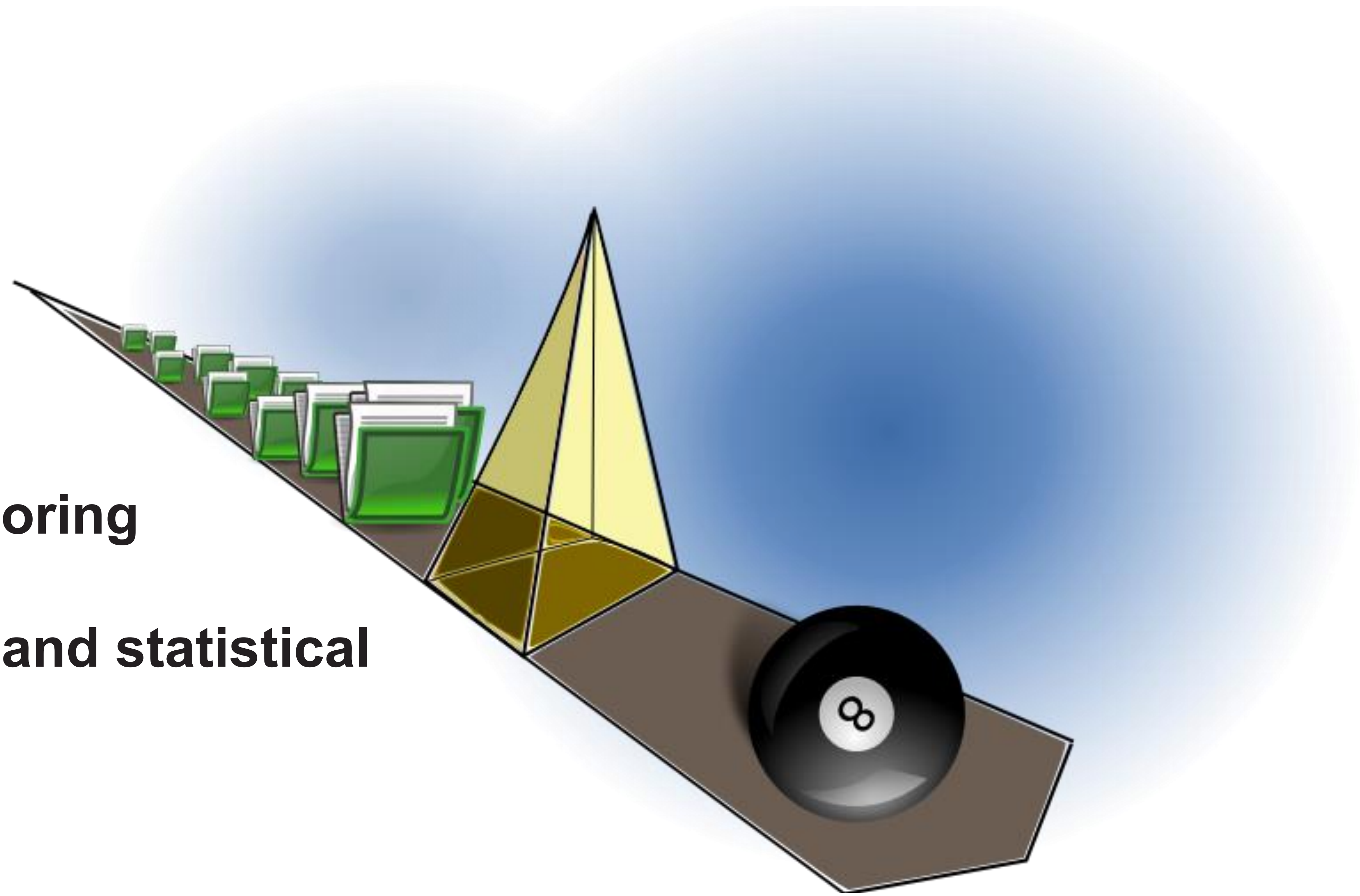
- Where does the time go?



- Where’s it going now?
- Where will it go?

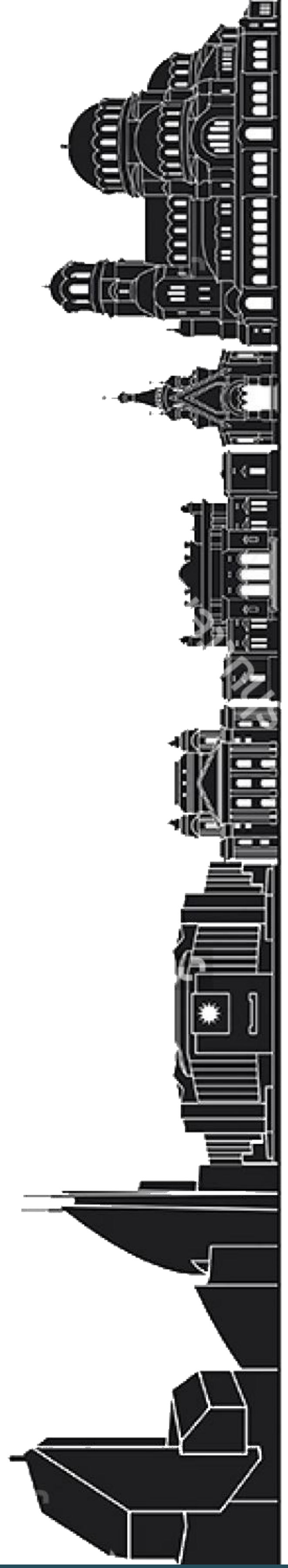
Performance Timeline – Toolkit

- Archives



- Live Monitoring

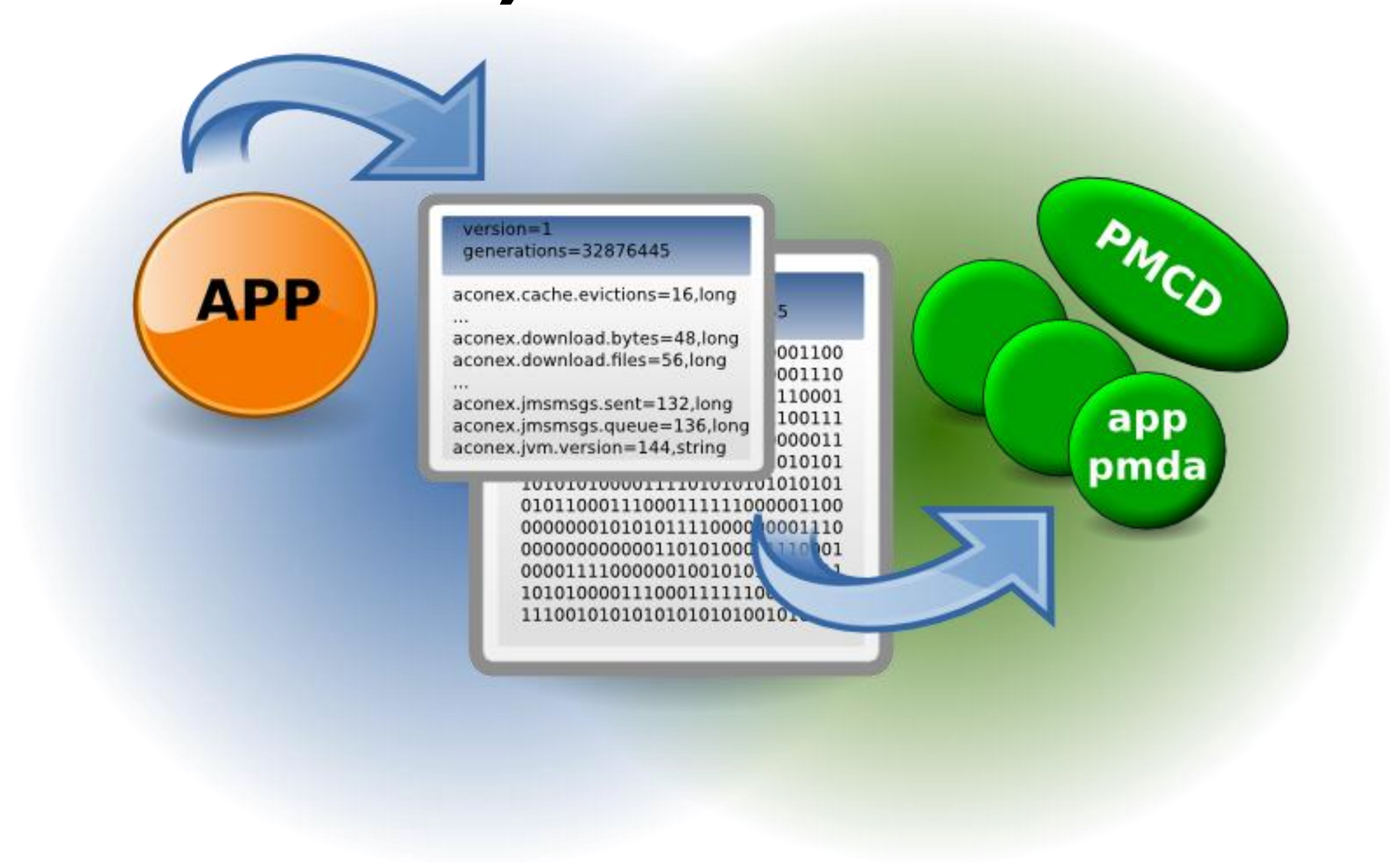
- Modelling and statistical prediction



Performance Timeline – PCP Toolkit

- Yesterday, last week, last month, ...
- All starts with pmlogger
  - Arbitrary metrics, intervals
  - One instance produces one PCP archive for one host
  - An archive consists of 3 files
    - Metadata, temporal index, data volume(s)
- pmlogger_daily, pmlogger_check
  - Ensure the data keeps flowing
- pmlogsummary, pmwtf, pmdumptext
- pmlogextract, pmlogreduce

Custom Instrumentation (Applications)



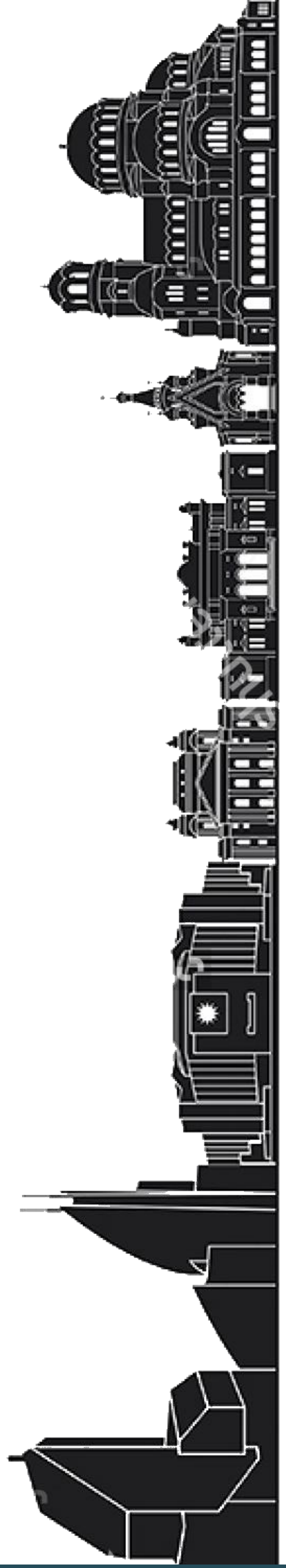
```
version=1
generations=32876445

aconex.cache.evictions=16,long
...
aconex.download.bytes=48,long
aconex.download.files=56,long
...
aconex.jmsmsgs.sent=132,long
aconex.jmsmsgs.queue=136,long
aconex.jvm.version=144,string
```

PCP – Parfait

Parfait has 4 main parts (for now)

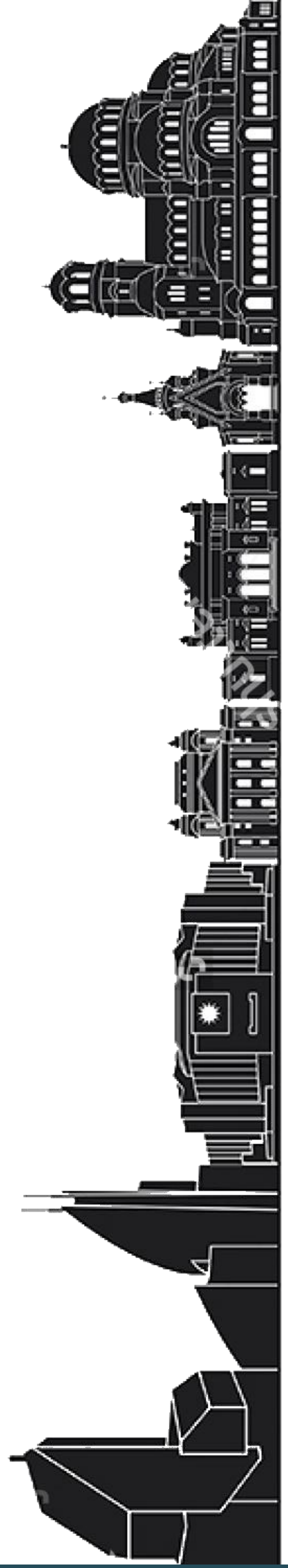
- Monitoring
- DXM
- Timing
- Requests

parfait
java performance
framework
by custardsource



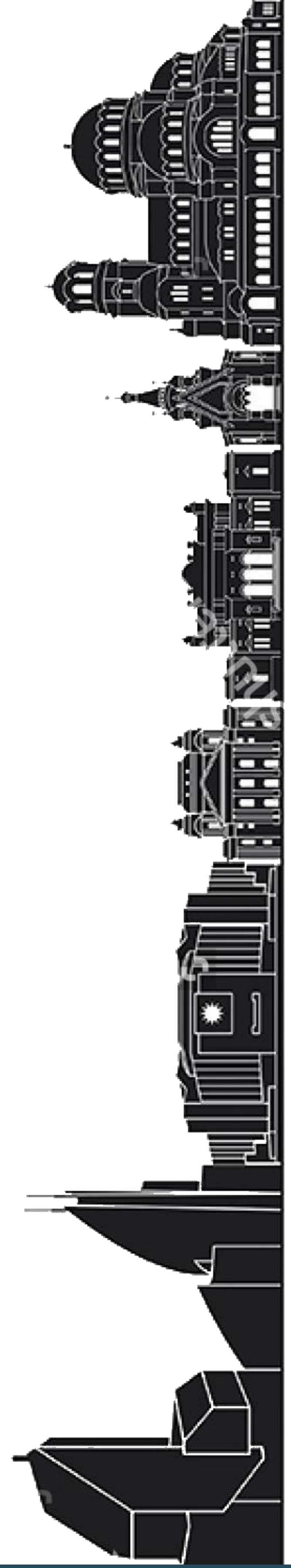
Parfait – Monitoring

- This is the ‘original’ PCP bridge metrics (heavily modified)
- Simple Java objects (MonitoredValues) which wrap a value (e.g. AtomicLong, String)
- MonitoredValues register themselves with a registry (container)
- When values changes, observers notice and output accordingly
    - PCP
    - JMX
    - Other (Custom/Extended)
- Very simple to use
- ‘Default registry’ (legacy concept)



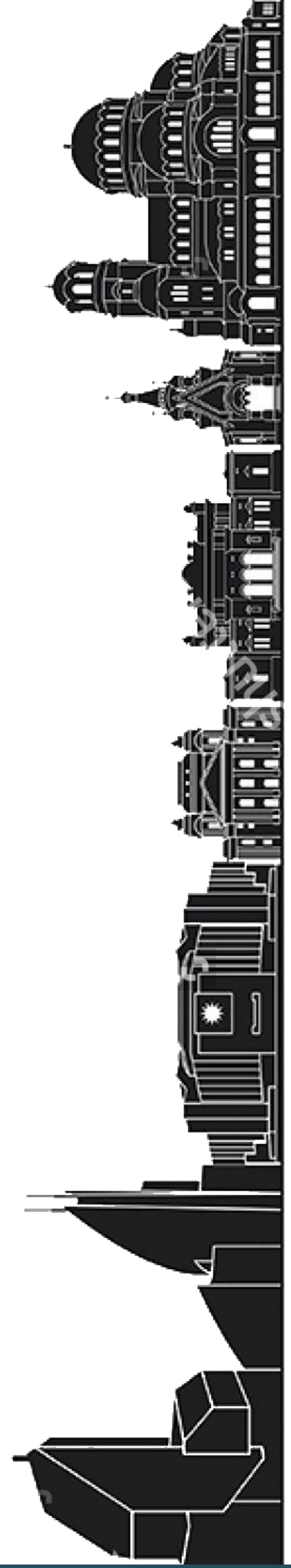
Parfait – Timing

- Logs the resources consumed by a request (an individual user action)
- Relies on a single request being thread-bound (and threads being used exclusively)
- Basically needs a Map<Thread, Value>
- Take the value for a Thread at the start, and at the end
- Delta is the ‘cost’ of that request



Parfait – Timing Example

```
[2010-09-22 15:02:13,466 INFO ][ait.timing.Log4jSink][http-8080-Processor3
gedq93kl][192.168.7.132][20][] Top  taskssummaryfeatures:tasks
   taskssummaryfeatures:tasks      Elapsed time: own 380.146316 ms, total
380.14688 ms Total CPU: own 150.0 ms, total 150.0 ms   User CPU: own 140.0 ms,
total 140.0 ms  System CPU: own 10.0 ms, total 10.0 ms Blocked count: own 40,
total 40  Blocked time: own 22 ms, total 22 ms   Wait count: own 2, total 2
   Wait time: own 8 ms, total 8 ms  Database execution time: own 57 ms, total 57
ms Database execution count: own 11, total 11Database logical read count: own
0, total 0   Database physical read count: own 0, total 0  Database CPU time:
own 0 ms, total 0 ms   Database received bytes: own 26188 By, total 26188 By
   Database sent bytes: own 24868 By, total 24868 By   Error Pages: own 0, total
0  Bobo execution time: own 40.742124 ms, total 40.742124 ms  Bobo execution
count: own 2, total 2  Bytes transferred via bobo search: own 0 By, total 0 By
   Super search entity count: own 0, total 0 Super search count: own 0, total 0
   Bytes transferred via super search: own 0 By, total 0 By   Elapsed time
during super search: own 0 ms, total 0 ms
```



Parfait – Requests

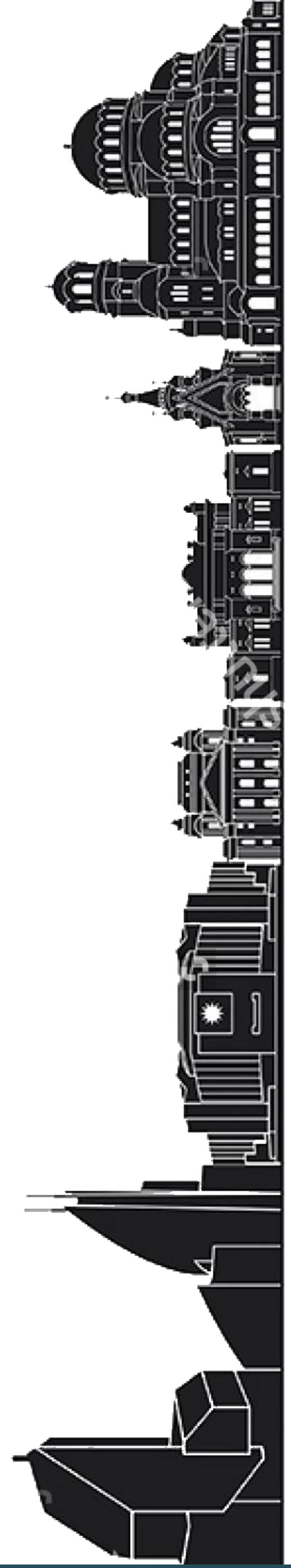
- As well as snapshotting requests after completion, for many metrics we can see meaningful ‘in-progress’ values
- Simple JMX bean which ‘walks’ in-progress requests
- Tie in with ThreadContext (MDC abstraction)
- Include UserID
- ThreadID

PCP – Speed

Golang implementation of the PCP instrumentation API

There are 3 main components in the library

- **Client**
- **Registry**
- **Metric**

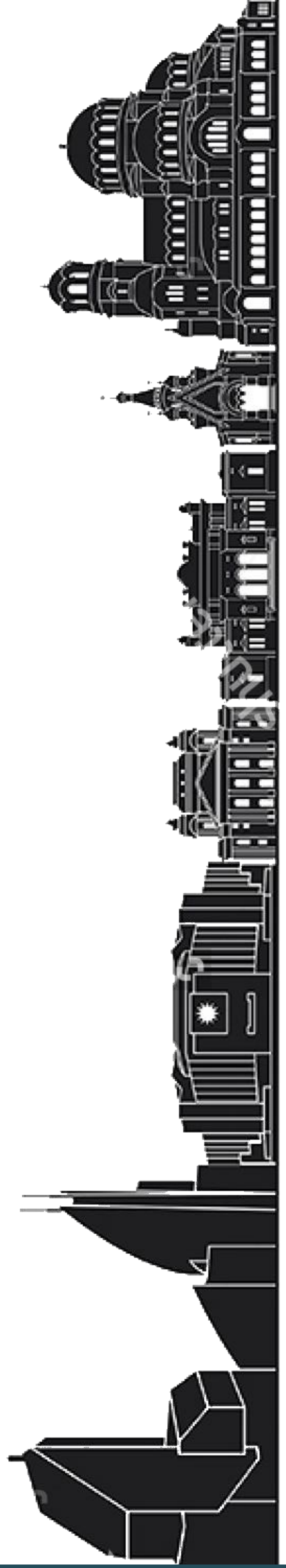


PCP – Speed Metric

- **SingletonMetric**
  - **This type defines a metric with no instance domain and only one value. It requires type, semantics and unit for construction, and optionally takes a couple of description strings.**
    A simple construction

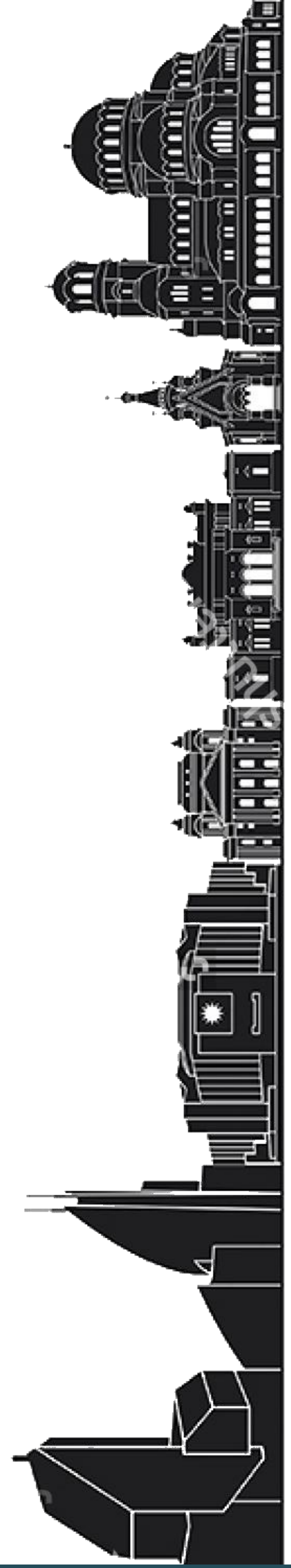
```
metric, err := speed.NewPCPSingletonMetric(
    42,                                                        // initial value
    "simple.counter",                                          // name
    speed.Int32Type,                                           // type
    speed.CounterSemantics,                                   // semantics
    speed.OneUnit,                                            // unit
    "A Simple Metric",                                        // short description
    "This is a simple counter metric to demonstrate the speed API", // long desc
)
```

PCP for Containers



PCP for Containers – Cgroup Accounting

- [subsys].stat files below /sys/fs/cgroup
- individual cgroup or summed over children
- blkio
- IOPs/bytes, service/wait time – aggregate/per-dev
- Split up by read/write, sync/async
- cpuacct
- Processor use per-cgroup - aggregate/per-CPU
- memory
- mapped anon pages, page cache, writeback, swap, active/inactive LRU state

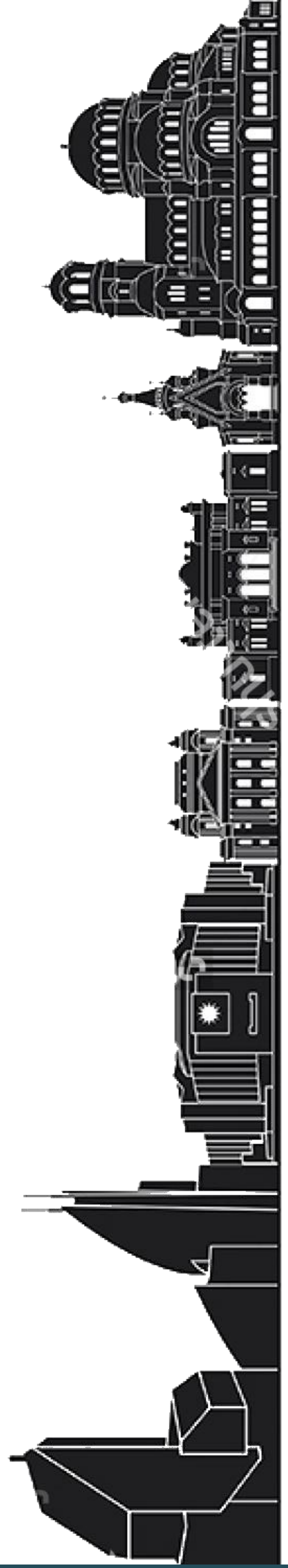


PCP for Containers – Namespaces

- **Example: cat /proc/net/dev**
- **Contents differ inside vs outside a container**

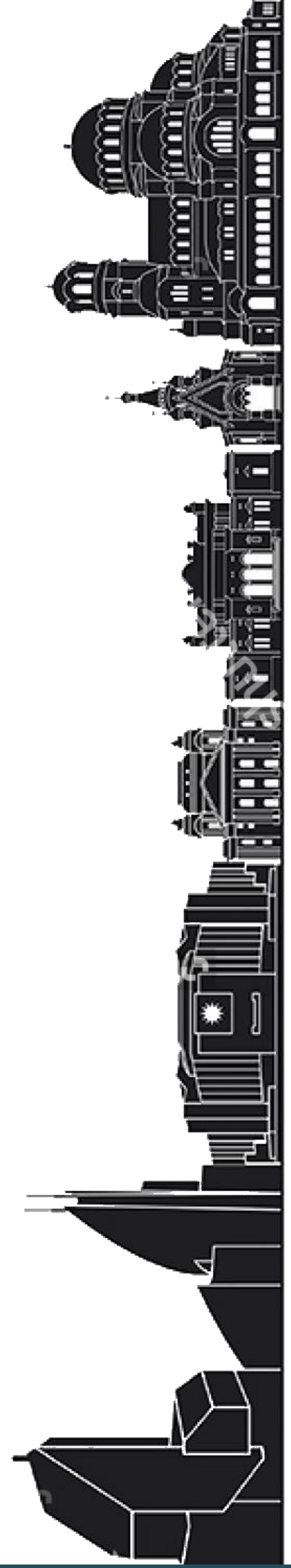
- **Processes (e.g. cat) in containers run in different network, ipc, process, uts, mount namespaces**

- **Namespaces are inherited across fork/clone**
- **Processes within a container share common view**



PCP Container Analysis – Goals

- Allow targeting of individual containers
- e.g. /proc/net/dev
- pminfo --fetch network
- vs
- pminfo –fetch –container=crank network
- Zero installation inside containers required
- Simplify your life (dev_t auto-mapping)
- Data reduction (proc.*, cgroup.*)



PCP Container Analysis – Mechanisms

- pminfo -f –host=acme.com –container=crank network
- Wire protocol extension
- Inform interested PCP collector agents
- Resolving container names, mapping names to cgroups, PIDs, etc.
- setns(2)
- Runs on the board, plenty of work remains
- New monitor tools with container awareness

What is Metrics?

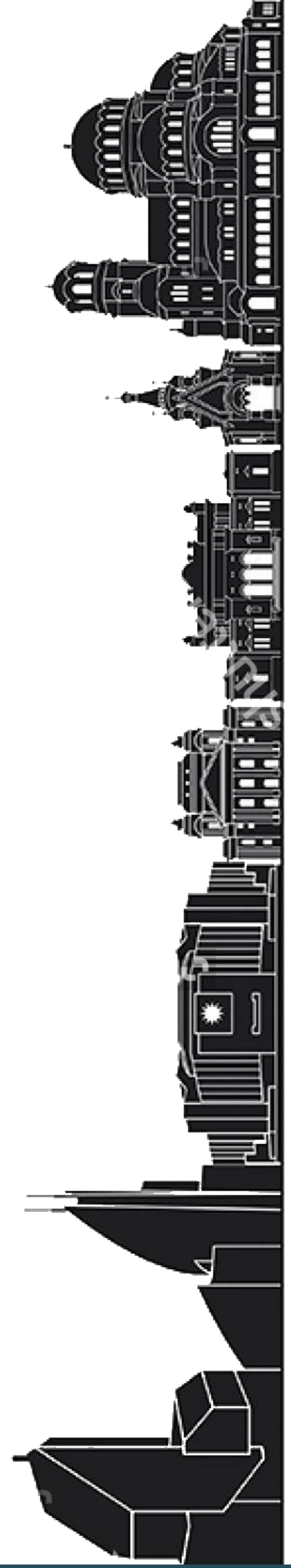
- Code instrumentation
- Meters
- Gauges
- Counters
- Histograms
- Web app instrumentation
- Web app health check

Metrics Reporters

- Reporters
  - Console
  - CSV
  - Slf4j
  - JMX
- Advanced reporters
  - Graphite
  - Ganglia

Metrics 3rd Party Libraries

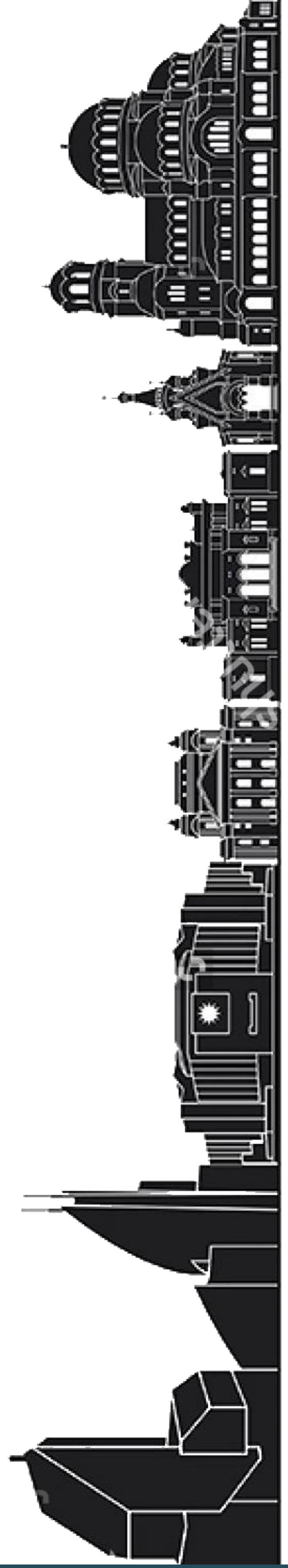
- AspectJ
- InfluxDB
- StatsD
- Cassandra
- Spring



Metrics Basics

- MetricsRegistry
  - A collection of all the metrics for your application
  - Usually one instance per JVM
  - Use more in multi WAR deployment
- Names
  - Each metric has a unique name
  - Registry has helper methods for creating names

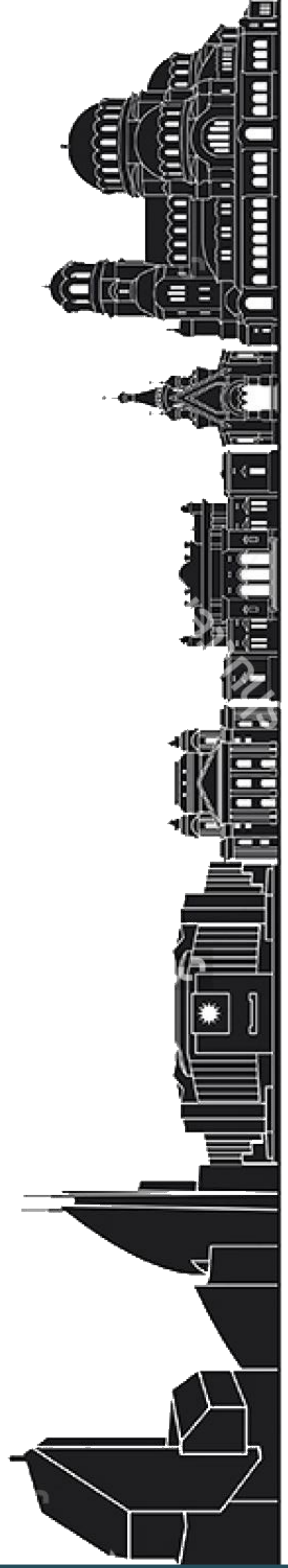
```
MetricRegistry.name(Queue.class, "items", "total")
//com.example.queue.items.total
MetricRegistry.name(Queue.class, "size", "byte")
//com.example.queue.size.byte
```



Metrics Elements

- **Gauges**
  - **The simplest metric type: it just returns a *value***

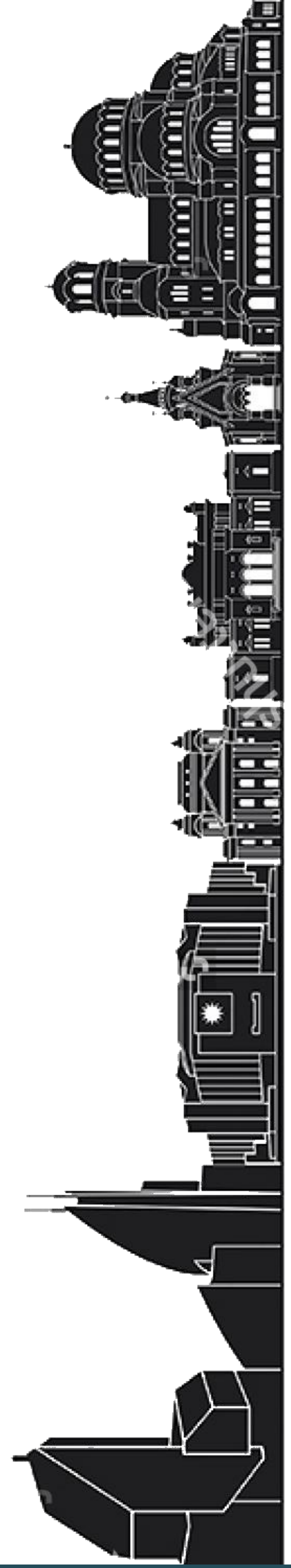
```
final Map<String, String> keys = new HashMap<>();
  registry.register(MetricRegistry.name("gauge", "keys"), new
   Gauge<Integer>() {

@Override
public Integer getValue() {
    return keys.keySet().size();
  }
});
```



Metrics Elements (2)

- **Counters**
  - **Incrementing and decrementing 64.bit integer**

```
final Counter counter= registry.counter(MetricRegistry.name("counter",
    "inserted"));
counter.inc();
```



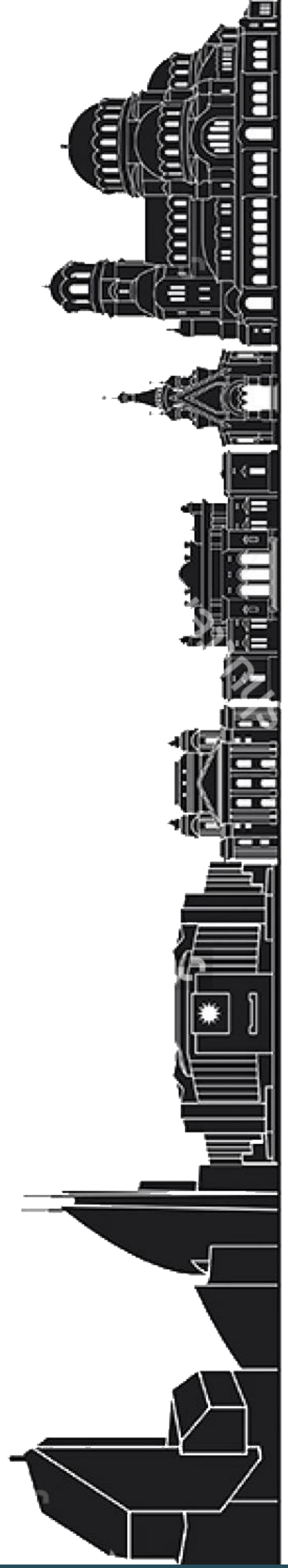
Metrics Elements (3)

- **Histograms**
  - **Measures the distribution of values in a stream of data**

```
final Histogram resultCounts = registry.histogram(name(ProductDAO.class,
"result-counts");
resultCounts.update(results.size());
```

- **Meters**
  - **Measures the rate at which a set of events occur**

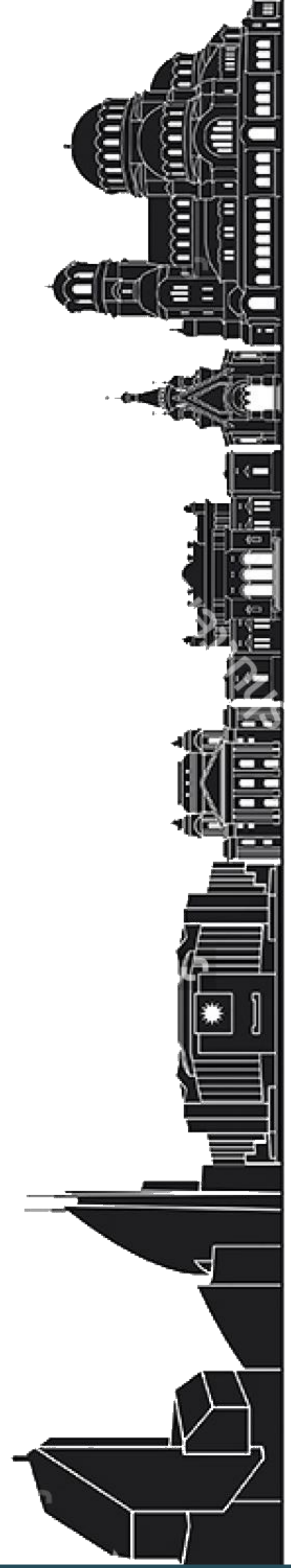
```
final Meter meter = registry.meter(MetricRegistry.name("meter", "inserted"));
meter.mark();
```



Metrics Elements (4)

- **Timers**
  - **A histogram of the duration of a type of event and a meter of the rate of its occurrence**

```
Timer timer = registry.timer(MetricRegistry.name("timer", "inserted"));
Context context = timer.time();
//timed ops
context.stop();
```

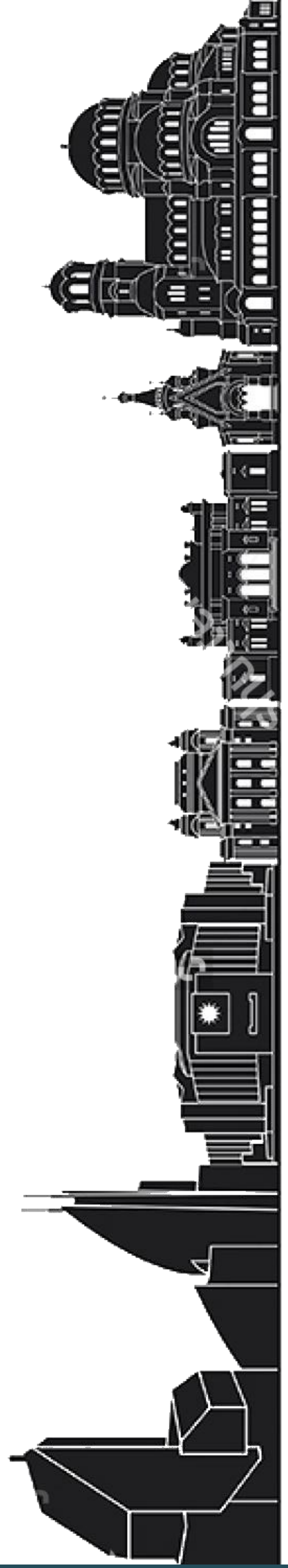


Metrics – Graphite Reporter

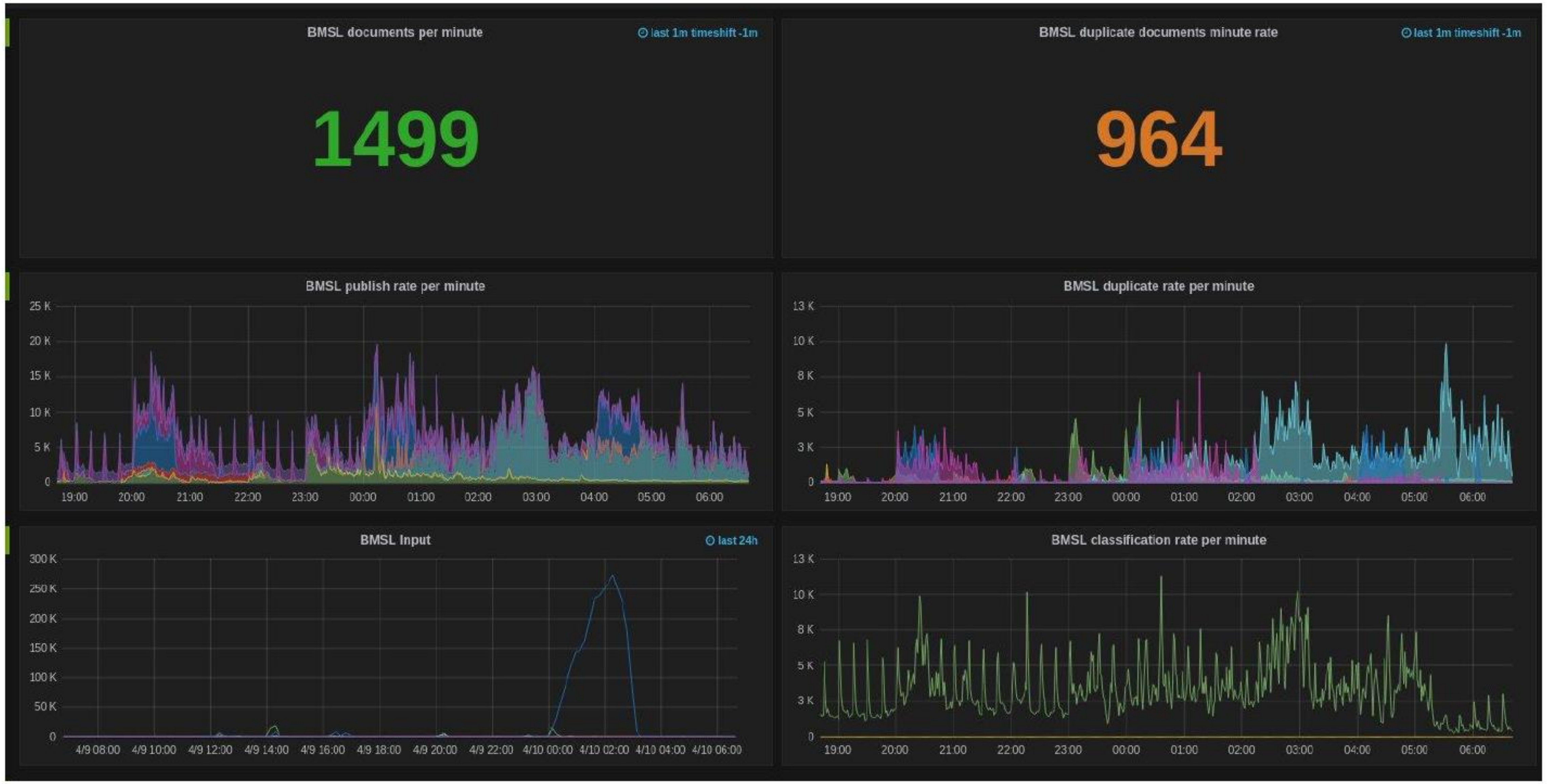
```
final Graphite graphite = new Graphite(new
InetSocketAddress("graphite.example.com", 2003));
final GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
.prefixedWith("web1.example.com")
.convertRatesTo(TimeUnit.SECONDS)
.convertDurationsTo(TimeUnit.MILLISECONDS)
.filter(MetricFilter.ALL)
.build(graphite);
reporter.start(1, TimeUnit.MINUTES);
```

Metrics can be prefixed

Useful to divide environment metrics: prod, test



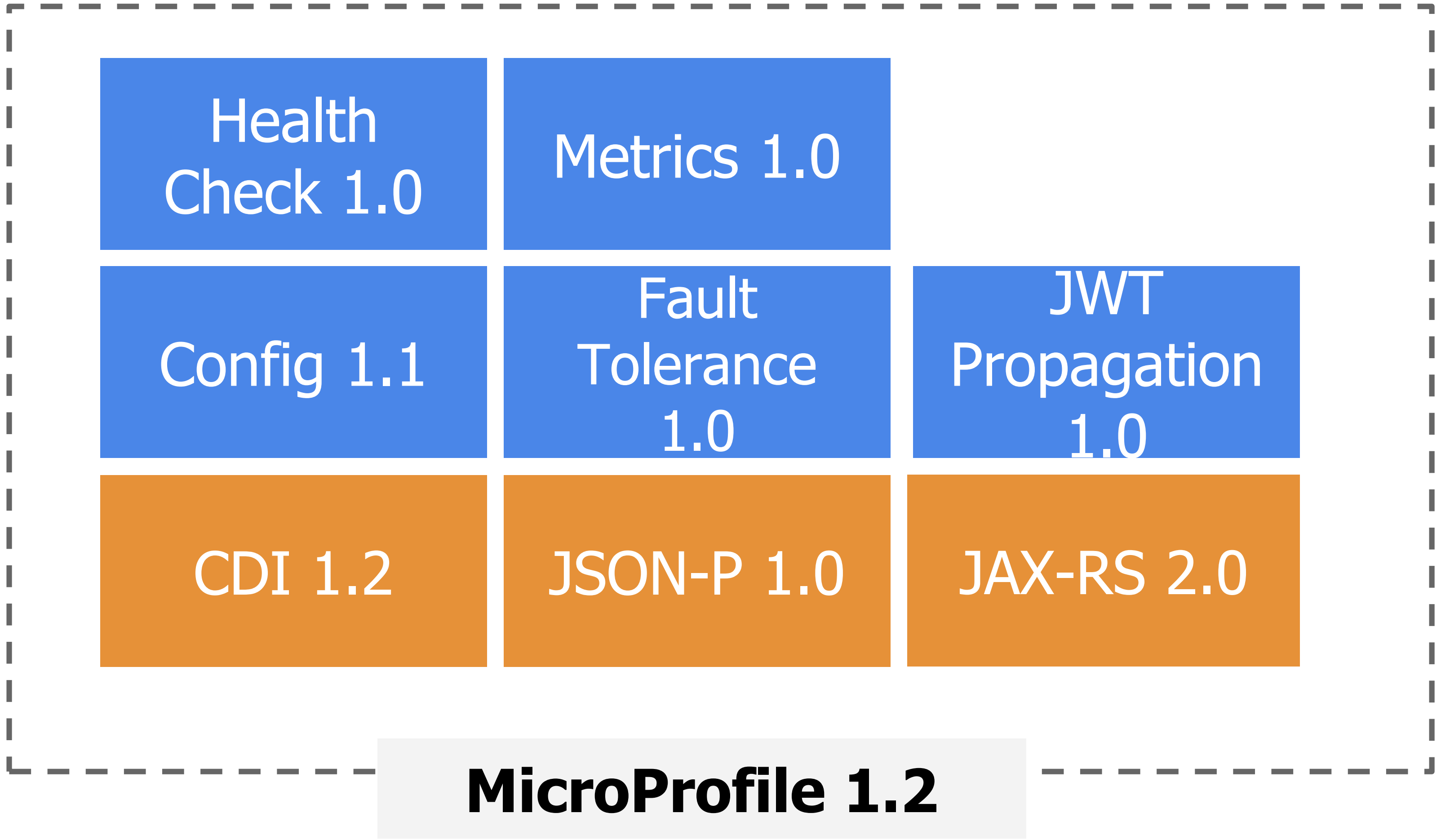
Metrics – Grafana Application Overview



What is Eclipse MicroProfile?

- Eclipse MicroProfile is an open-source community specification for Enterprise Java microservices
- A community of individuals, organizations, and vendors collaborating within an open source (Eclipse) project to bring microservices to the Enterprise Java community

Specifications 1.2



Health Check 1.0

Metrics 1.0

Config 1.1

Fault Tolerance 1.0

JWT Propagation 1.0

CDI 1.2

JSON-P 1.0

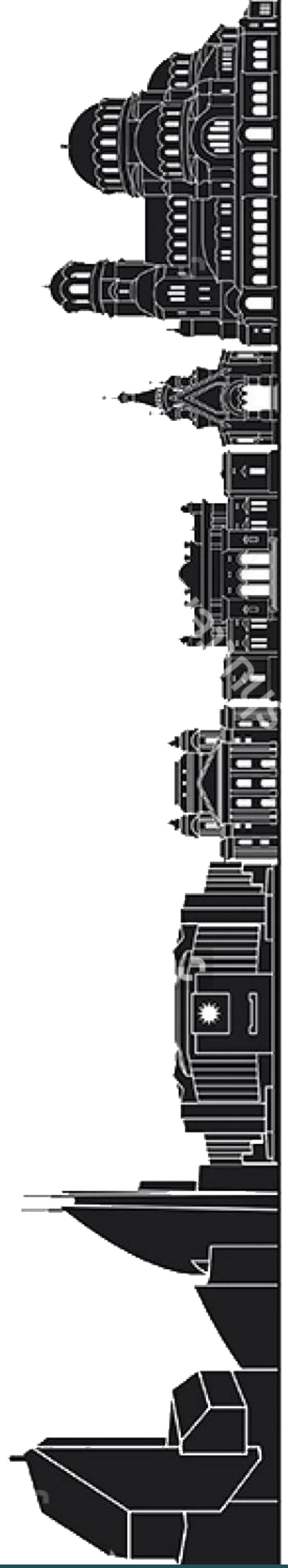
JAX-RS 2.0

MicroProfile 1.2

🟦 = New

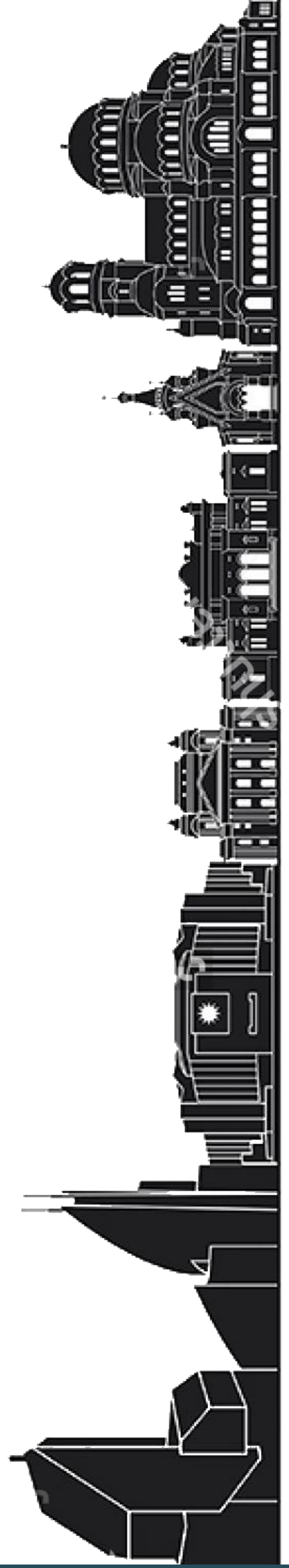
🟧 = No change from last release

Prometheus



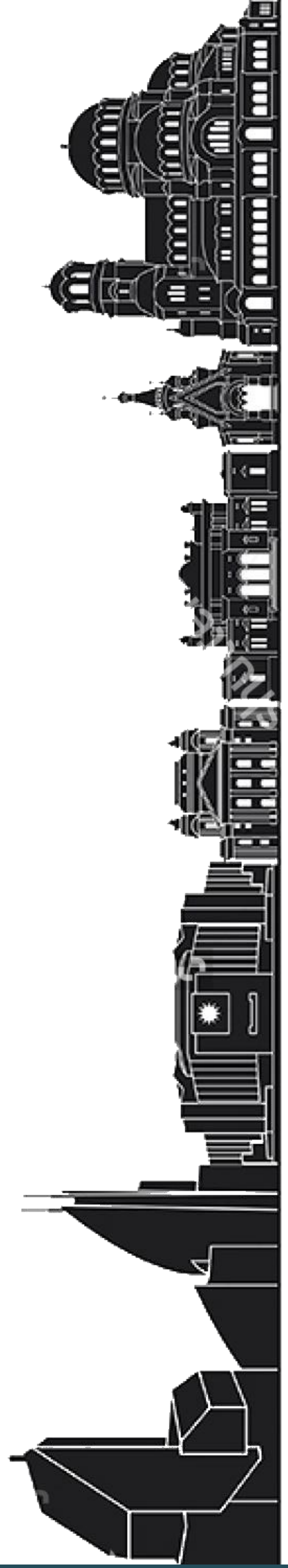
What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. It is now a standalone open source project and maintained independently of any company.



Prometheus Components

- The main Prometheus server which scrapes and stores time series data
- Client libraries for instrumenting application code
- A push gateway for supporting short-lived jobs
- Special-purpose exporters (for HAProxy, StatsD, Graphite, etc.)
- An alertmanager
- Various support tools
- WhiteBox Monitoring instead of probing (aka BlackBox Monitoring)



What is StatsD?

A network daemon that runs on the Node.js platform and listens for statistics, like counters and timers, sent over UDP or TCP and sends aggregates to one or more pluggable backend services (e.g., Graphite).

StatsD was inspired (heavily) by the project (of the same name) at Flickr.

Action!

Images: Nu Image / Millennium Films

Links

Performance Co-Pilot
- http://www.pcp.io

Dropwizard Metrics
- http://metrics.dropwizard.io

Eclipse MicroProfile
- http://microprofile.io

Prometheus
- http://prometheus.io

StatsD
- https://github.com/etsy/statsd/wiki

Q & A