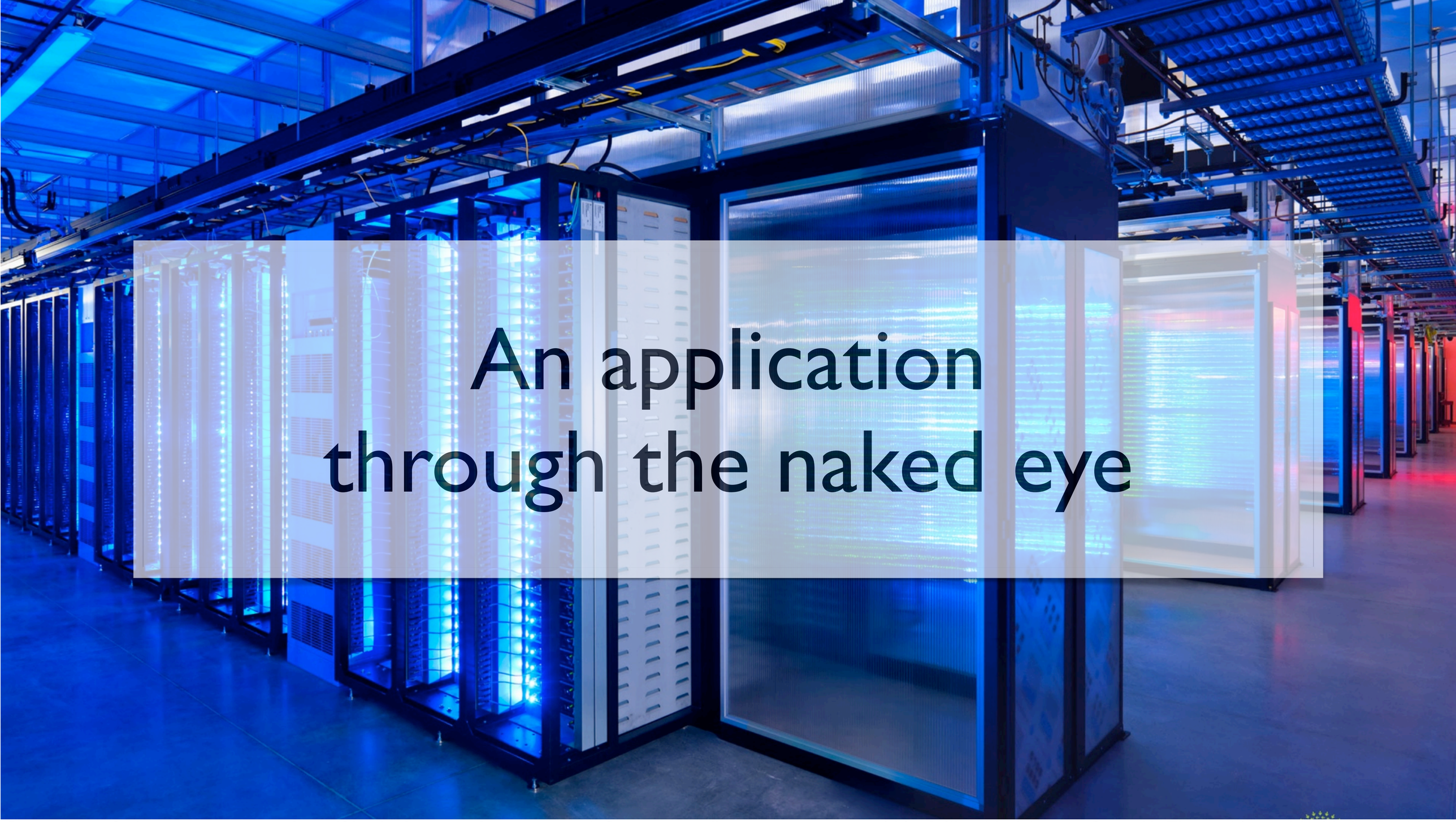
Effective Monitoring with

DEVOPS DAY Tokyo 2013

Thank you for having me here. I'm here to talk to you about modern monitoring. My goal is to present a blueprint of how a monitoring tool should behave.

I have spent the past 3 years building a monitoring service called Datadog, that works for companies, large and small. I have spent the past 3 years listening to our customers.

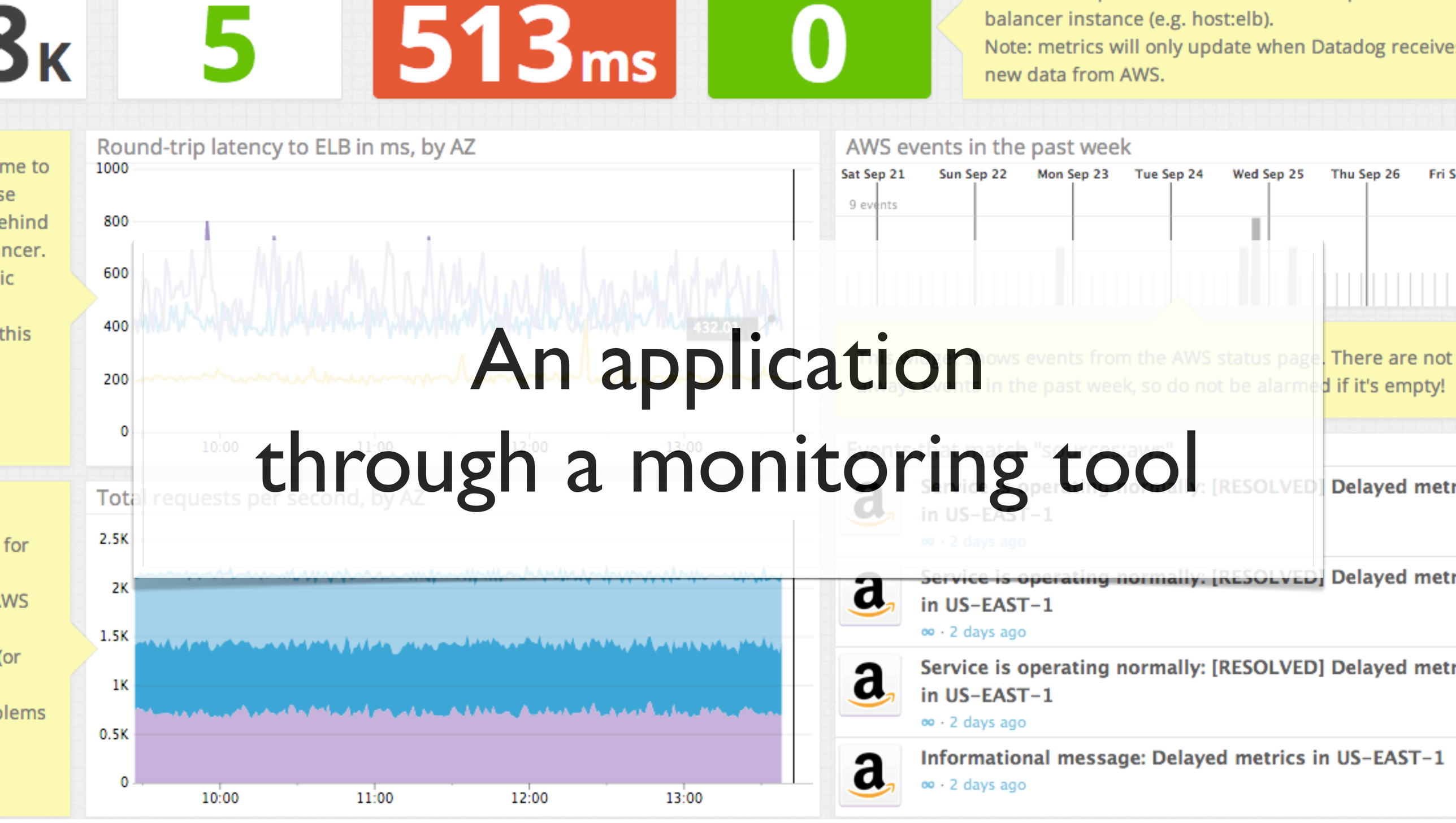
Everyone I know in our industry has deployed some kind of monitoring tool. We do it as a matter of fact yet it's good to think about why.



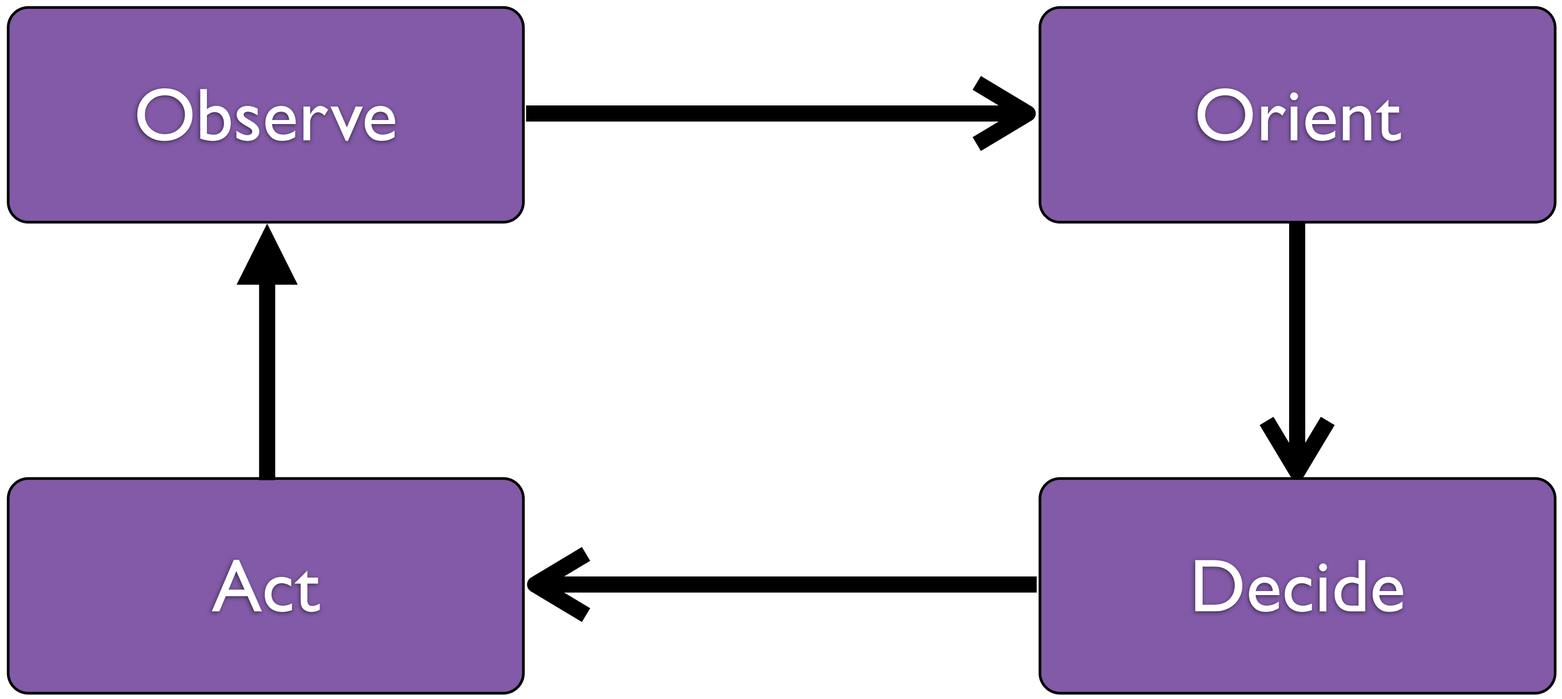
Regardless of what your company does, if I asked you to show me your infrastructure, I'm pretty sure it would look like this (picture of a rack). The problem with this is that it's not really telling me much about the application it is serving: it's just a bunch of lights flickering randomly. The real thing is happening inside these metal boxes.

So we need tools that can interpret what's happening inside and turn that into signals we, as people, can interpret. Your monitoring tool is **really** the primary interface to your application. So it is crucial to get it right.

An application through a monitoring tool

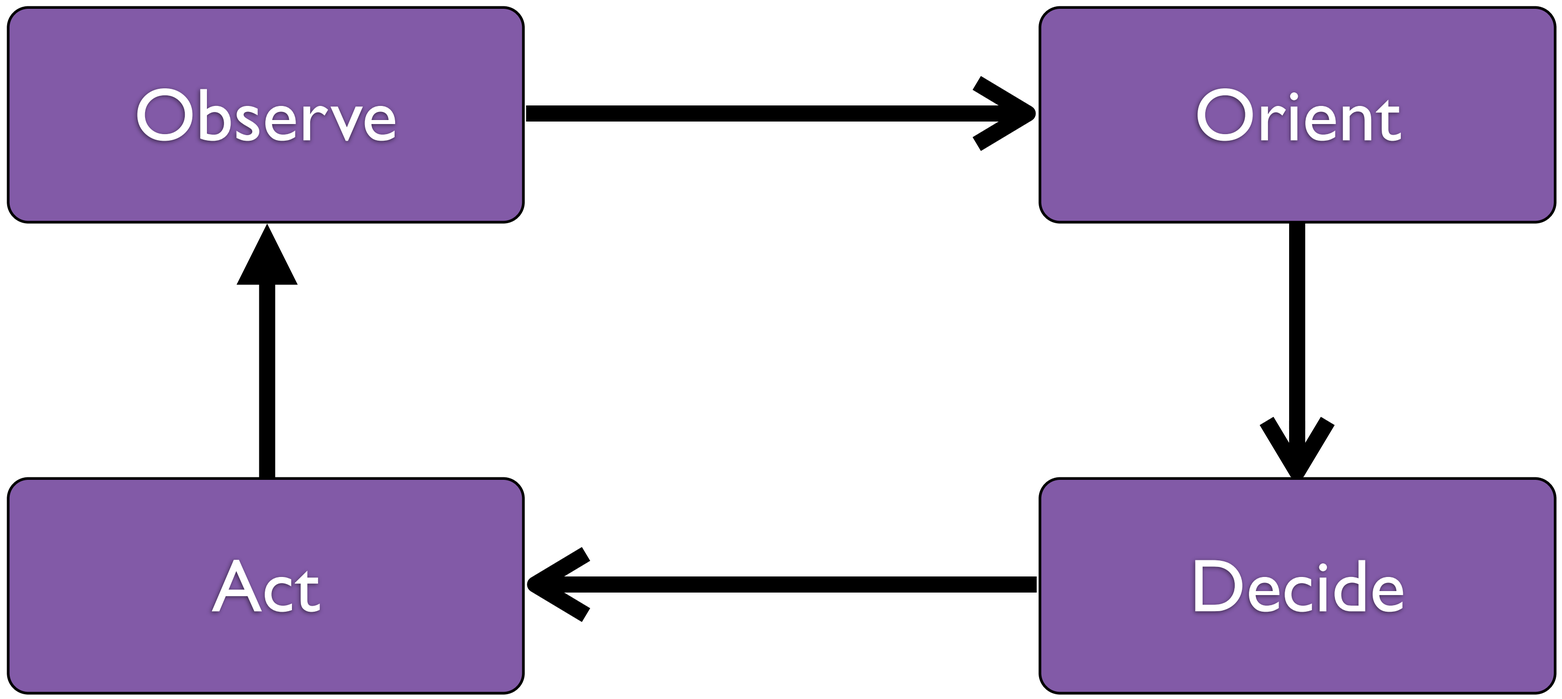


OODA Loop (simplified)



You may be familiar with the OODA loop as a model for behavior in operations: observe, orient, decide and act. (slide with OODA loop).

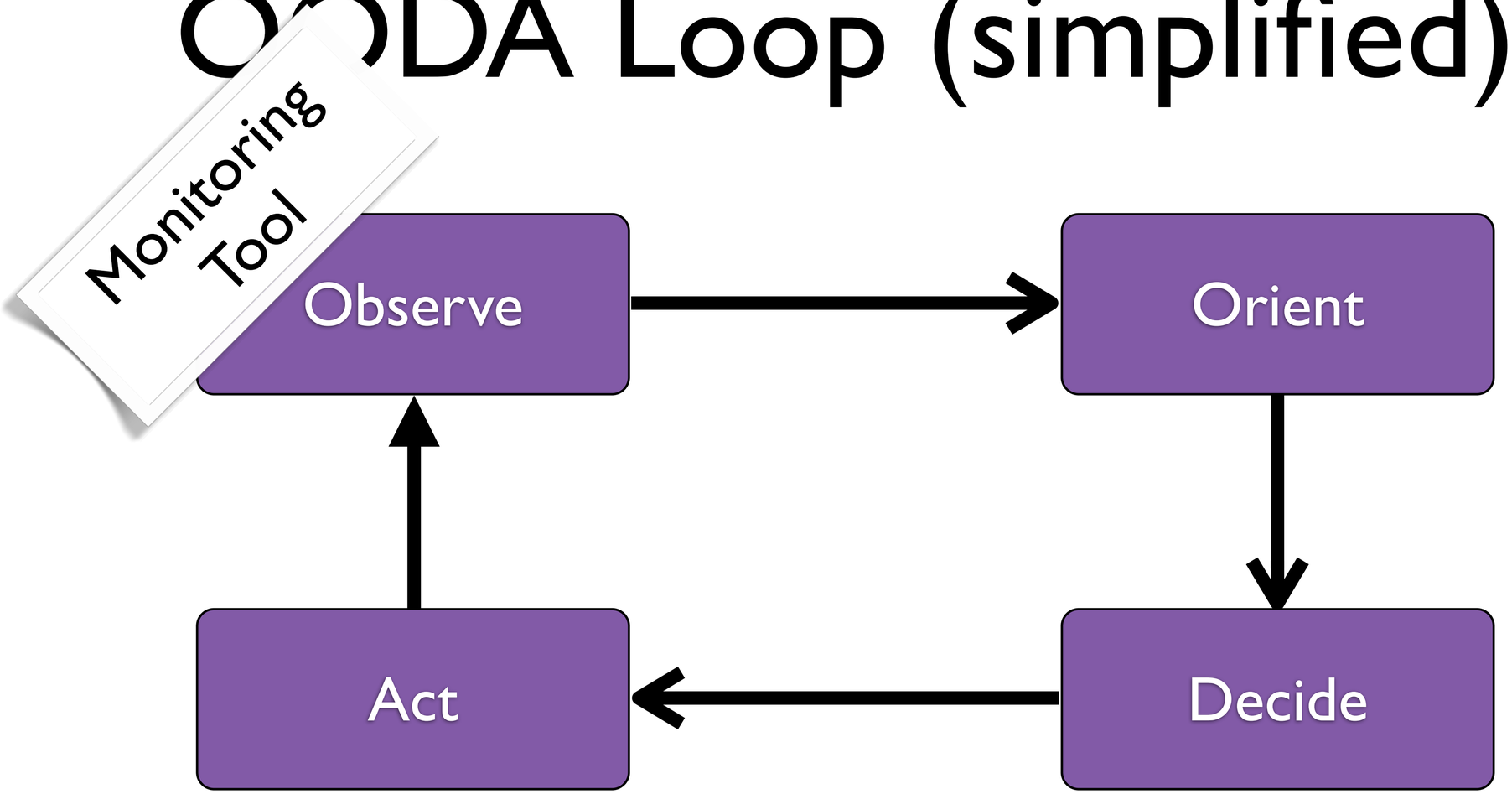
OODA Loop (simplified)



Monitoring falls mostly in the first stage "Observe". It is the base of a healthy OODA cycle. Or for that matter any scientific, empirical approach.

The rest of the loop is basically you, using your knowledge and your reason to get the best outcome possible..

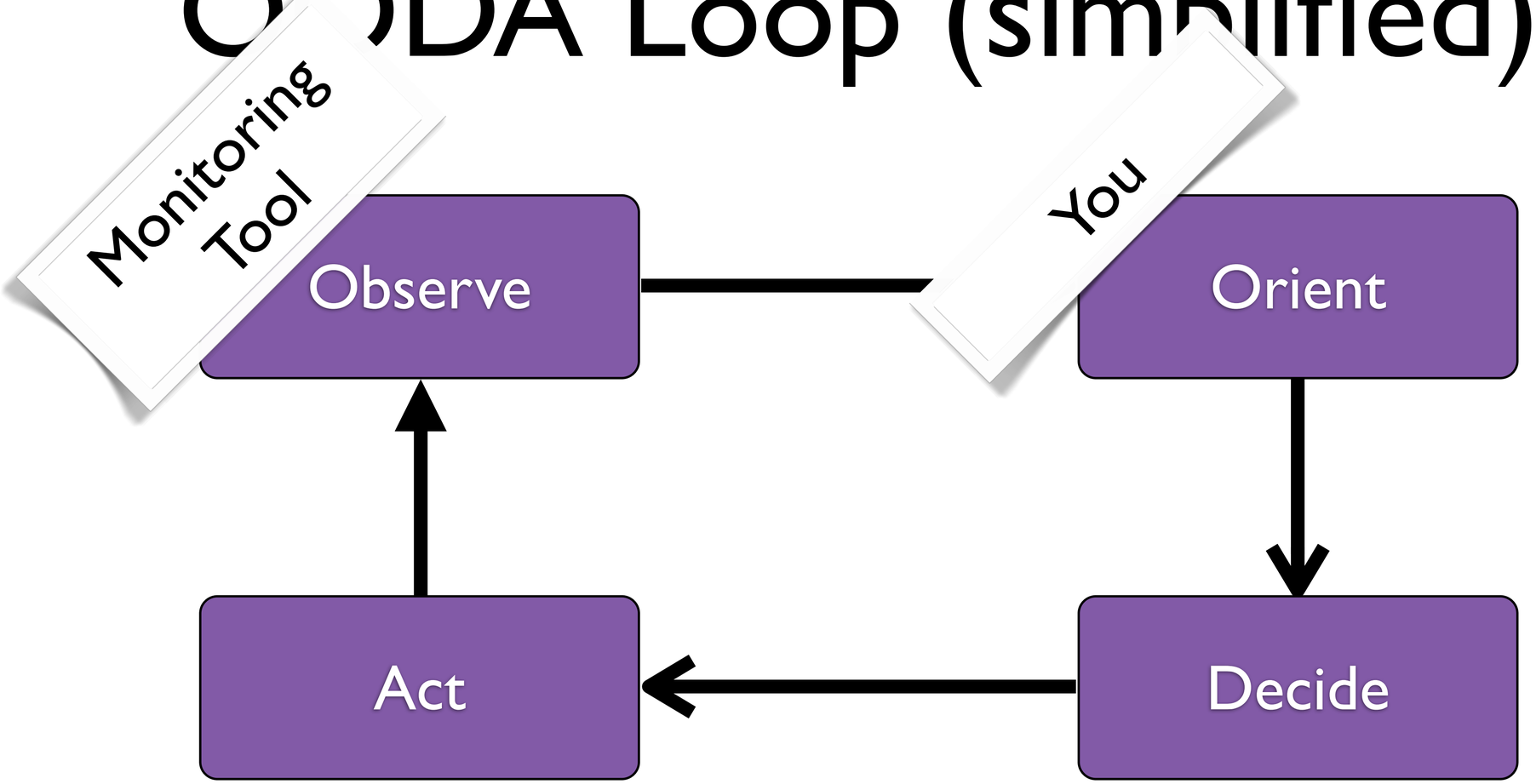
OODA Loop (simplified)



Monitoring falls mostly in the first stage "Observe". It is the base of a healthy OODA cycle. Or for that matter any scientific, empirical approach.

The rest of the loop is basically you, using your knowledge and your reason to get the best outcome possible..

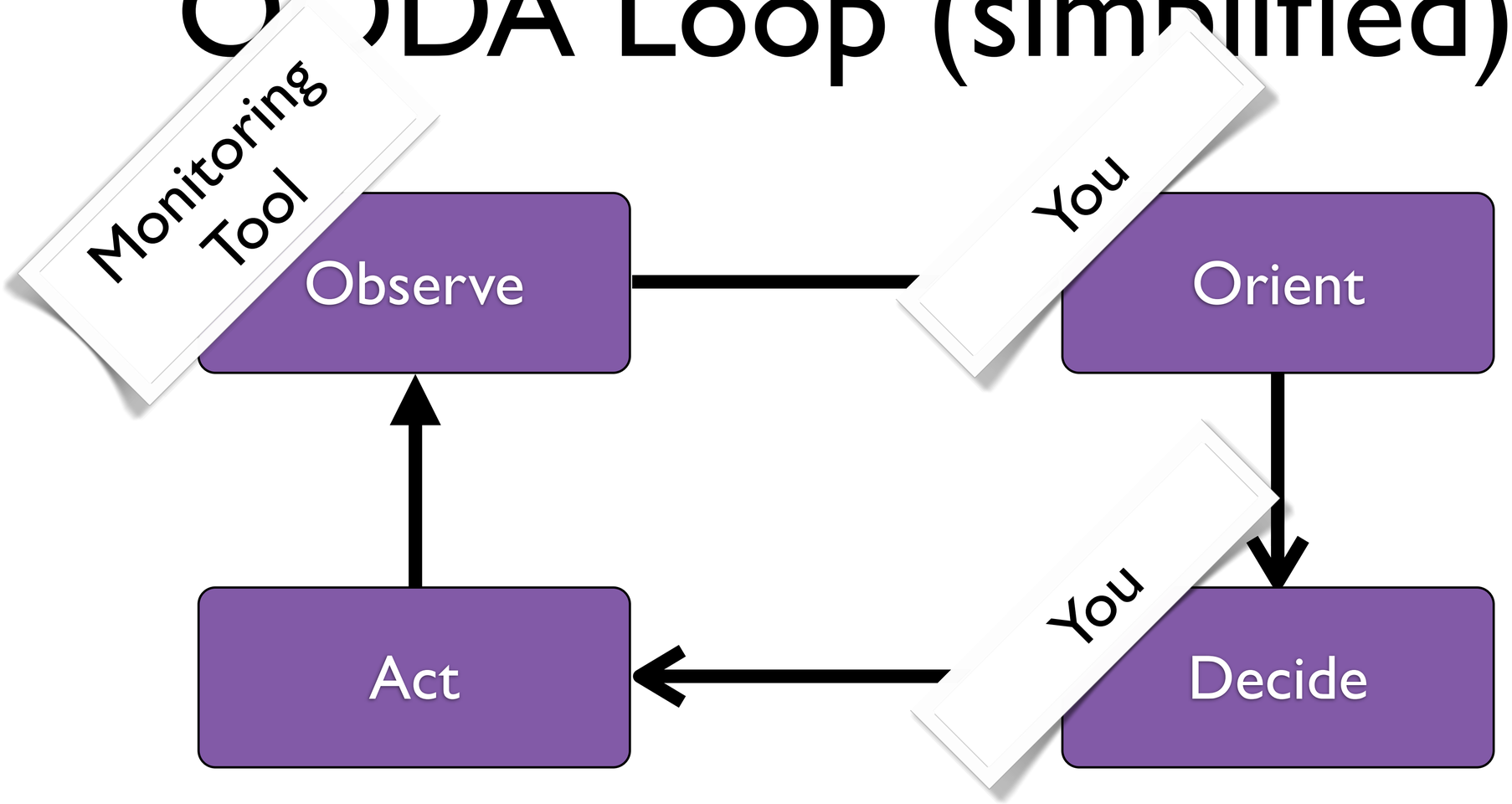
OODA Loop (simplified)



Monitoring falls mostly in the first stage "Observe". It is the base of a healthy OODA cycle. Or for that matter any scientific, empirical approach.

The rest of the loop is basically you, using your knowledge and your reason to get the best outcome possible..

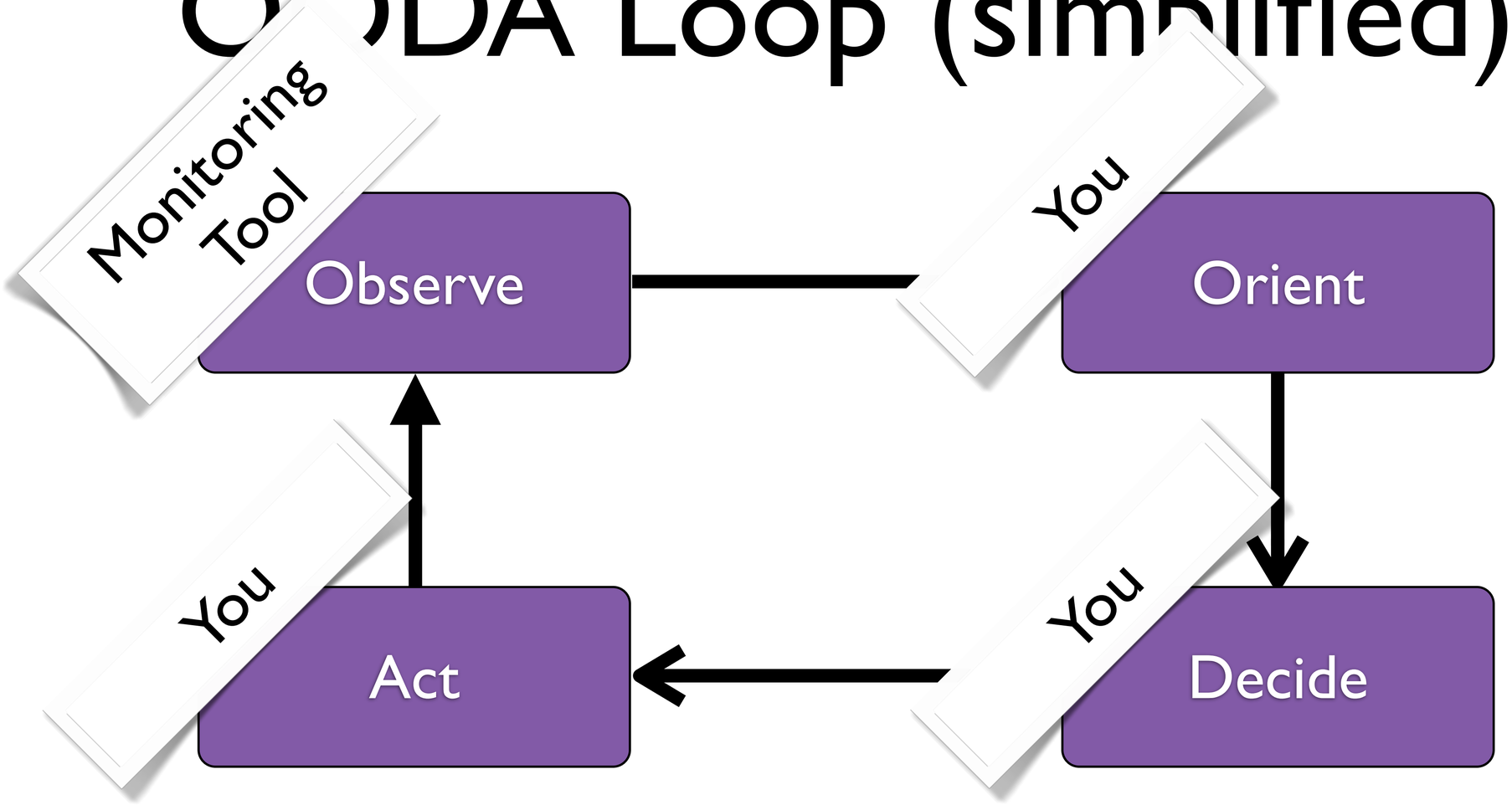
OODA Loop (simplified)



Monitoring falls mostly in the first stage "Observe". It is the base of a healthy OODA cycle. Or for that matter any scientific, empirical approach.

The rest of the loop is basically you, using your knowledge and your reason to get the best outcome possible..

OODA Loop (simplified)



Monitoring falls mostly in the first stage "Observe". It is the base of a healthy OODA cycle. Or for that matter any scientific, empirical approach.

The rest of the loop is basically you, using your knowledge and your reason to get the best outcome possible..

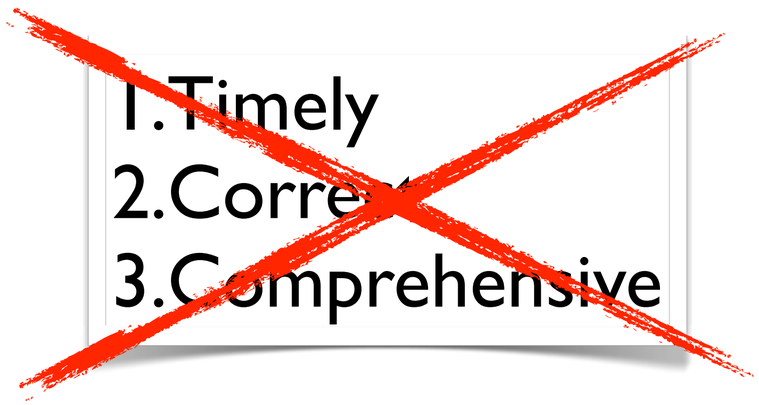
Observations need to be...

1. Timely
2. Correct
3. Comprehensive

Your observations from your monitoring tool need to be timely, correct and comprehensive.

Without having **timely**, **correct** and **comprehensive** observation, orientation will be based on low-quality data. Low-quality observation leads to low-quality decisions. In other words it’ll be garbage in, garbage out.

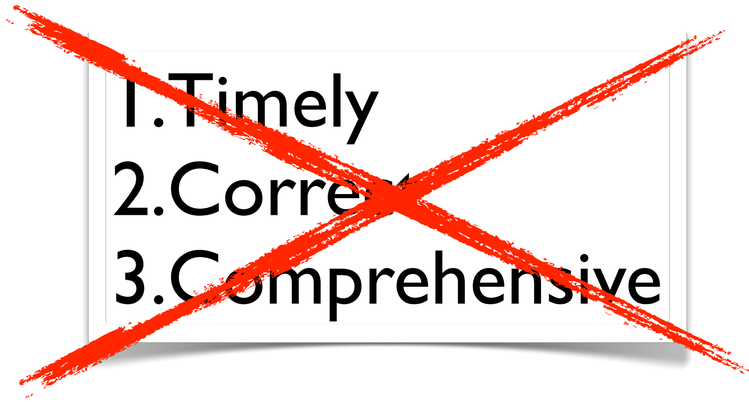
Observations need to be...



Your observations from your monitoring tool need to be timely, correct and comprehensive.

Without having **timely**, **correct** and **comprehensive** observation, orientation will be based on low-quality data. Low-quality observation leads to low-quality decisions. In other words it’ll be garbage in, garbage out.

Observations need to be...



Else

Your observations from your monitoring tool need to be timely, correct and comprehensive.

Without having **timely**, **correct** and **comprehensive** observation, orientation will be based on low-quality data. Low-quality observation leads to low-quality decisions. In other words it’ll be garbage in, garbage out.

Observations need to be...

1. Timely
2. Correct
3. Comprehensive

Else

Garbage In,
Garbage Out

Your observations from your monitoring tool need to be timely, correct and comprehensive.

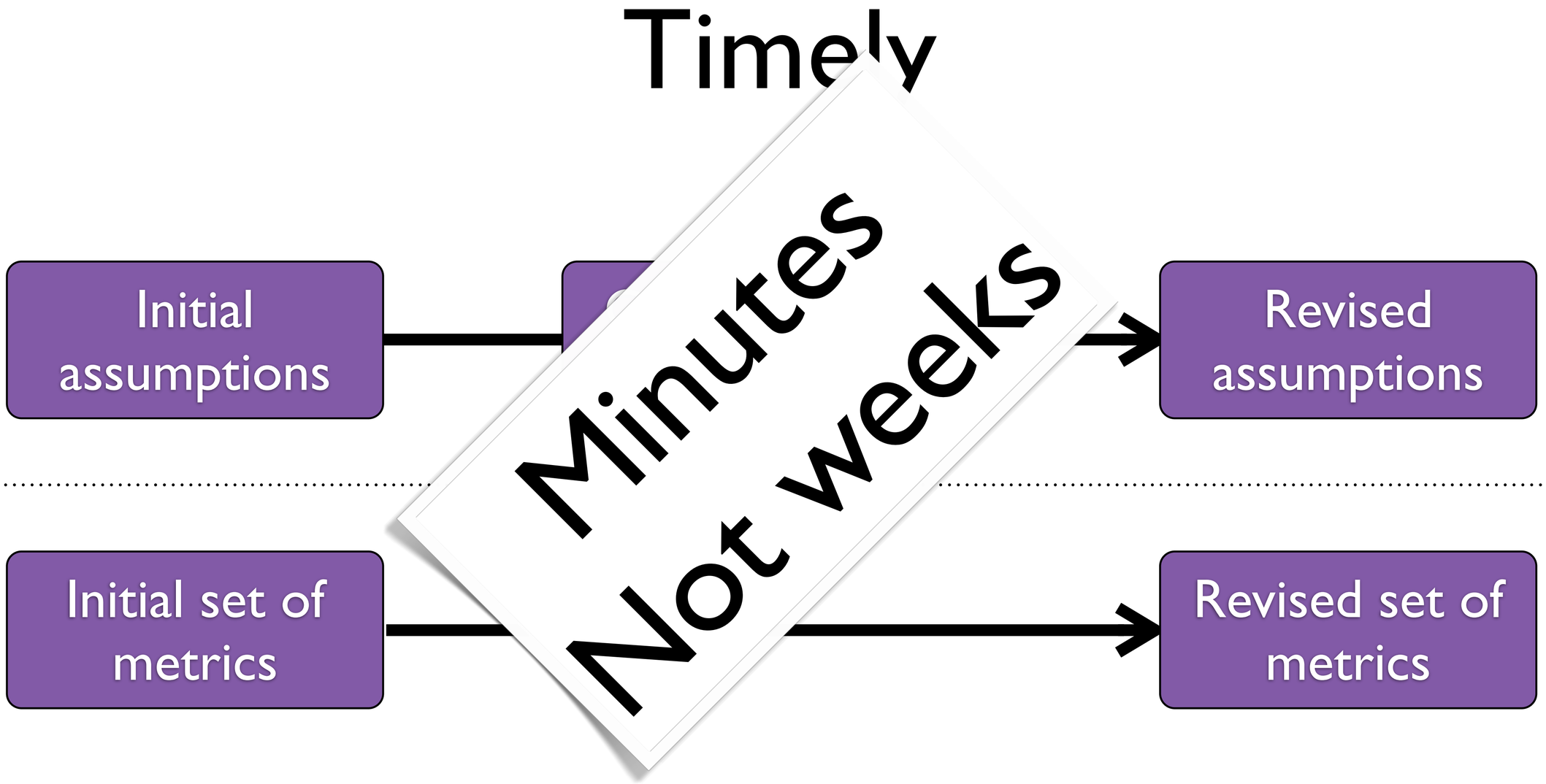
Without having **timely**, **correct** and **comprehensive** observation, orientation will be based on low-quality data. Low-quality observation leads to low-quality decisions. In other words it’ll be garbage in, garbage out.

Timely

Out of the 3 qualities of monitoring I would like to spend more time on timeliness and comprehensiveness, leaving correctness for another time.

First, *timeliness*. When you run a new application in production, you go with an initial set of metrics that you will use for monitoring. The initial set represents the assumptions held by the people who wrote the application before it was running in production. In my own experience that set is almost always incomplete. And that's OK. We are working with software, we want to move fast and fix things as they occur, rather than not move at all.

So for me, having timely monitoring really means being able to add the metrics and monitors you want with minimal delay. If you can measure that delay in minutes, you're fine. If the delay is in weeks, you're in trouble.



Out of the 3 qualities of monitoring I would like to spend more time on timeliness and comprehensiveness, leaving correctness for another time.

First, *timeliness*. When you run a new application in production, you go with an initial set of metrics that you will use for monitoring. The initial set represents the assumptions held by the people who wrote the application before it was running in production. In my own experience that set is almost always incomplete. And that's OK. We are working with software, we want to move fast and fix things as they occur, rather than not move at all.

So for me, having timely monitoring really means being able to add the metrics and monitors you want with minimal delay. If you can measure that delay in minutes, you're fine. If the delay is in weeks, you're in trouble.

Comprehensive

A comprehensive monitoring is one where all the important metrics are captured. The important metrics reflect the amount of **work** done by your application. The amount of **work** that your application produces is what makes it valuable. Note that most metrics that you can collect out of the box, are **resource** metrics. How much memory I have, how much compute I am using, etc.

If all you have available are **resource** metrics, it makes the "orientation" stage of the OODA loop extremely difficult because you have to derive all the **work** metrics from the **resource** metrics.

One analogy would be to measure the success of a building construction by how many stones have been laid out, how much concrete has been poured. Or how much money has been spent.

So for me, comprehensive monitoring has both **work** and **resource** metrics. And since I won't have all the metrics ready from the get-go I must be able to add them quickly as I discover what I need.

Comprehensive

Easy to collect
generic
but **not actionable**

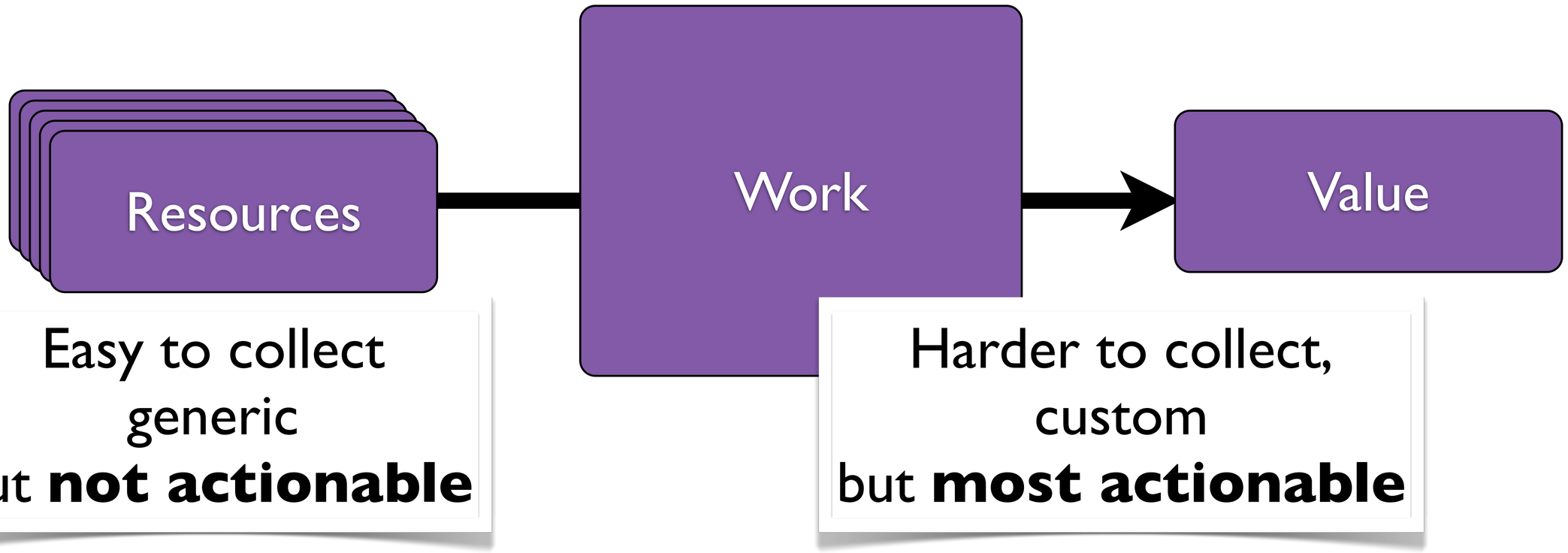
A comprehensive monitoring is one where all the important metrics are captured. The important metrics reflect the amount of **work** done by your application. The amount of **work** that your application produces is what makes it valuable. Note that most metrics that you can collect out of the box, are **resource** metrics. How much memory I have, how much compute I am using, etc.

If all you have available are **resource** metrics, it makes the "orientation" stage of the OODA loop extremely difficult because you have to derive all the **work** metrics from the **resource** metrics.

One analogy would be to measure the success of a building construction by how many stones have been laid out, how much concrete has been poured. Or how much money has been spent.

So for me, comprehensive monitoring has both **work** and **resource** metrics. And since I won't have all the metrics ready from the get-go I must be able to add them quickly as I discover what I need.

Comprehensive



A comprehensive monitoring is one where all the important metrics are captured. The important metrics reflect the amount of **work** done by your application. The amount of **work** that your application produces is what makes it valuable. Note that most metrics that you can collect out of the box, are **resource** metrics. How much memory I have, how much compute I am using, etc.

If all you have available are **resource** metrics, it makes the "orientation" stage of the OODA loop extremely difficult because you have to derive all the **work** metrics from the **resource** metrics.

One analogy would be to measure the success of a building construction by how many stones have been laid out, how much concrete has been poured. Or how much money has been spent.

So for me, comprehensive monitoring has both **work** and **resource** metrics. And since I won't have all the metrics ready from the get-go I must be able to add them quickly as I discover what I need.

statsD

```python
# Track the run time of the database query.
start_time = time.time()
results = db.query()
duration = time.time() - start_time
statsd.histogram('database.query.time', duration)

# We can also use the `timed` decorator as a short-hand for timing functions.
@statsd.timed('database.query.time')
def get_data():
    return db.query()
```

Easy

To get timely and comprehensive monitoring I have found that statsD is an excellent tool to rely on.

It offers **timeliness** because it is very quick for developers or DevOps folks to add to pretty much any application (in most languages), without a lot of setup beforehand and without introducing the risk to bring the application down thanks to UDP.

By the same token it allows for **comprehensive** monitoring because I can add more metrics until I have what I need to feed the rest of the OODA loop.

It's also easy-to-use and has broad language coverage.

statsD

```
# Track the run time of the database query.
start_time = time.time()
results = db.query()
duration = time.time() - start_time
statsd.histogram('database.query.time', duration)

# We can also use the `timed` decorator as a short-hand for timing functions.
@statsd.timed('database.query.time')
query()
```

Easy

Timely

To get timely and comprehensive monitoring I have found that statsD is an excellent tool to rely on.

It offers **timeliness** because it is very quick for developers or DevOps folks to add to pretty much any application (in most languages), without a lot of setup beforehand and without introducing the risk to bring the application down thanks to UDP.

By the same token it allows for **comprehensive** monitoring because I can add more metrics until I have what I need to feed the rest of the OODA loop.

It's also easy-to-use and has broad language coverage.

statsD

```
# Track the run time of the database query.
start_time = time.time()
results = db.query()
duration = time.time() - start_time
statsd.histogram('database.query.time', duration)

# We can also use the `timed` decorator as a short-hand for timing functions.
@statsd.timed('database.query.time')
```

Easy

Timely

Comprehensive

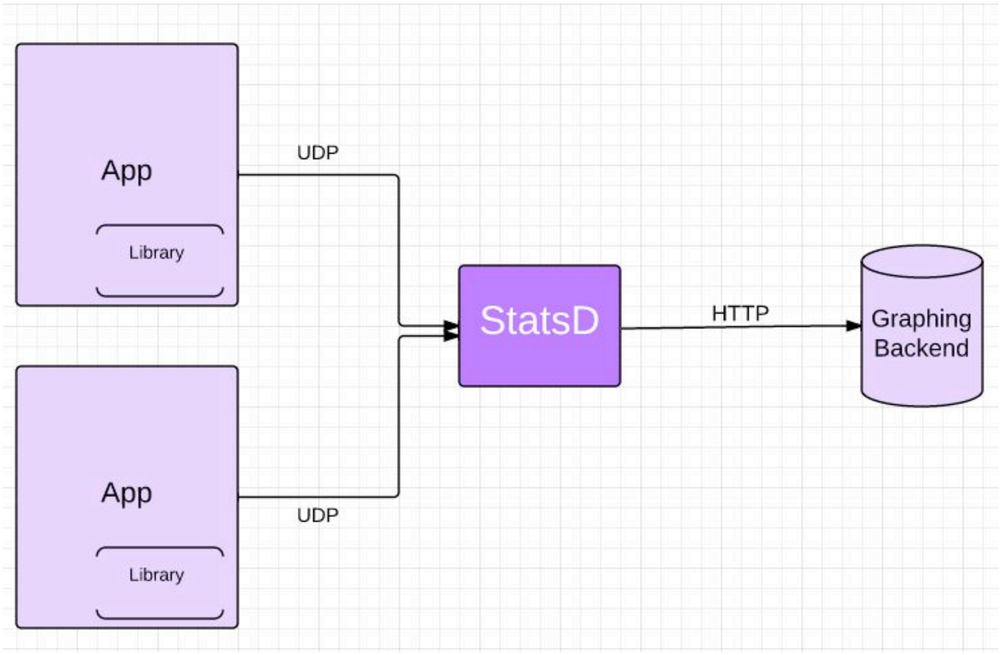
To get timely and comprehensive monitoring I have found that statsD is an excellent tool to rely on.

It offers **timeliness** because it is very quick for developers or DevOps folks to add to pretty much any application (in most languages), without a lot of setup beforehand and without introducing the risk to bring the application down thanks to UDP.

By the same token it allows for **comprehensive** monitoring because I can add more metrics until I have what I need to feed the rest of the OODA loop.

It's also easy-to-use and has broad language coverage.

How statsD works




```
pageviews:100|c@0.25
latency:320|ms
backlog:333|g
uniques:765|s
```

Client libraries talk to a simple UDP server...

...using a simple text protocol

Its principle is easy to understand: a UDP server that aggregates metrics for (usually) 10 seconds and flushes the results for persistent storage (graphite, Datadog, etc.).

The protocol itself is very simple.

statsD types

Type	*Definition*	*Example*
Gauges	Absolute values	Queue size
Counters	Per-second rates	Page views
Histograms	Gauge summary	Page Latency
Timers	Gauge distribution	Page Latency
Sets	Counters of unique things	Unique visitors

This is a simple summary of all the metric types, along with examples.

* `gauges`: absolute values like queue sizes, # of concurrent sessions, only 1 value per 10s interval; the last value wins.
* `counters`: per-second rates like page views; offers an increment/decrement interface.
* `histograms`: same as gauges but allows multiple values to be aggregated; returns summary statistics such as the median, the upper 95th percentile, etc.
* `timers`: same as histograms but bins points into fixed buckets instead of computing summary statistics
* `sets`: counter for unique things, for instance unique visitors to a page.

statsD problems

Type	*Definition*	*Problem*
Gauges	Absolute values	Latest value wins. Gauge deltas???
Counters	Per-second rates	Rates, not counts (!= `rrdtool`)
Histograms	Gauge summary	Assumes normal distribution
Timers	Gauge distribution	Can measure much more than time
Sets	Counters of unique things	:-)

Probably the biggest problem with statsD is that the metric type names are very confusing. 3 out of 5 are misnomers.

1. `gauges` only keep the last value sent. They also offer an increment/decrement interface called a gauge delta, where you can keep track of a count. So `gauges` can behave like real counters. This is really confusing.
2. `counters` returns rates, not absolute counts so you cannot count the number of time something was incremented over a day for instance
3. `histograms` send summary statistics that do not really say much about the underlying statistical distribution.
4. `timers` can measure much more than time. They are very much like histograms. Instead of sending min/max/average/median/percentiles, they group values in predefined buckets.
5. `sets` are actually fine

#1 pitfall: “Counters”

https://p.datadoghq.com/sb/487b7950d5

Counters is probably the most misunderstood metric type of statsD. Let us look at some real data for this.

It offers an increment interface (e.g. +1 page view) but sends a rate (page viewed per second).

Here is the code. And here is the result. You can see that for rare events, it gives you hard-to-read data. Here is the underlying rare events, accumulated over the past hour.

This makes counters not well-suited for metrics that are rarely incremented.

You need to use gauge deltas to represent this kind of data.

How we use statsD

https://p.datadoghq.com/sb/9b246c4ade

We use statsD internally at Datadog (our version of it) to measure about 50,000 different metrics and we could not operate our monitoring service without it.

The velocity and the depth of instrumentation it has given us, and by us, I mean every single engineer in the team, is amazing. When the time from idea to metric is measured in minutes, it makes the business a lot more agile.

Tagging

https://p.datadoghq.com/sb/823a4dafcf

The usual storage and graphing backend for statsD is **graphite**, where any metadata (e.g. version of the application) is encoded in the metric name. This is severely limiting if you want to decorate your metrics with more than 1 piece of metadata.

That's why we added **tagging** to our version of the statsD client and server (called `dogstatsd`). Tagging your metrics let you query for metrics based on interesting properties.

Here’s in an example of one metric split by various tags or dimensions. One dimension is the data center, another dimension is the host. Some dimensions are automatically imported from the environment via Chef, Puppet, AWS, etc.

ありがとうございました。

質問？@alq

Thank you very much!

Have Questions ? @alq