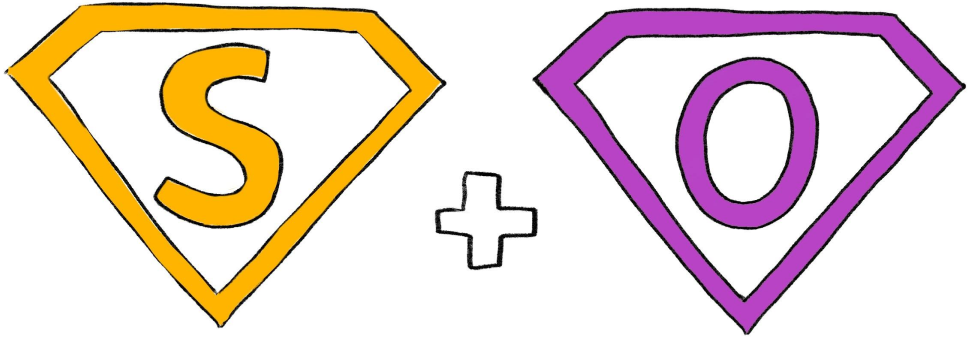
Refining Systems Data
(without losing fidelity)

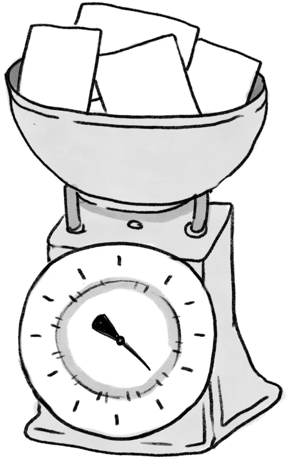
Liz Fong-Jones
@lizthegrey
#SREcon EMEA
October 3, 2019

w/ illustrations by @emilywithcurls!

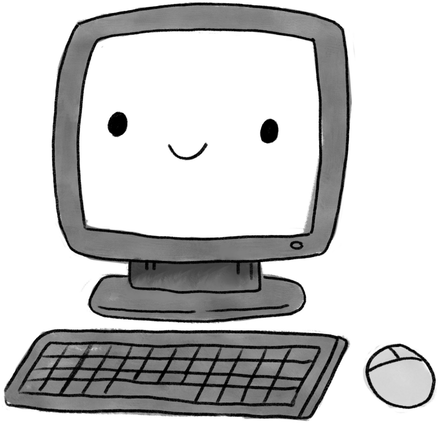
Complex systems are hard to manage.



We need SLOs and observability.



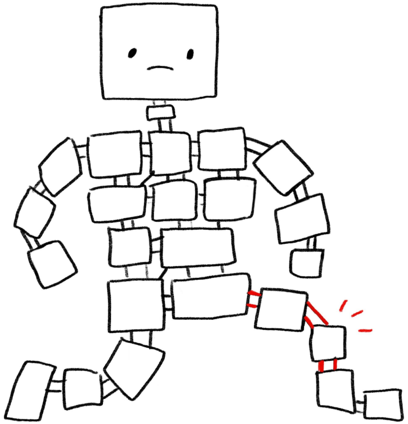
SLOs and debugging require data.



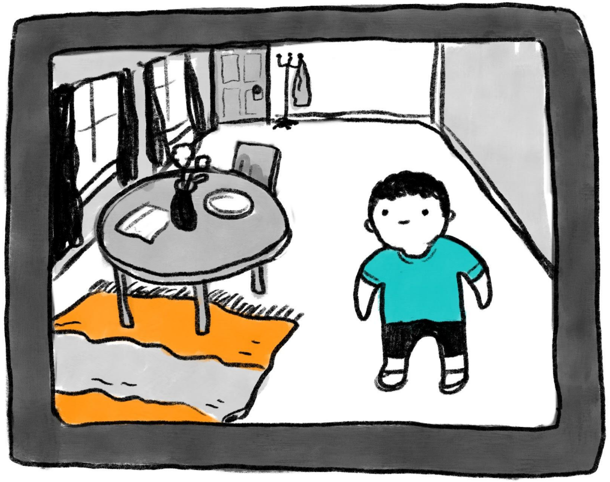
But what kind of telemetry data?

User experiences.

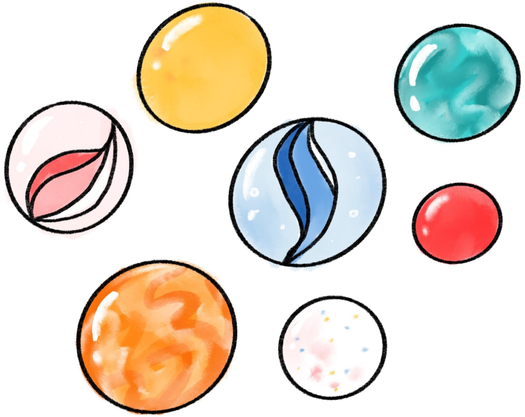
Host metrics.



But most problems aren't per-host.

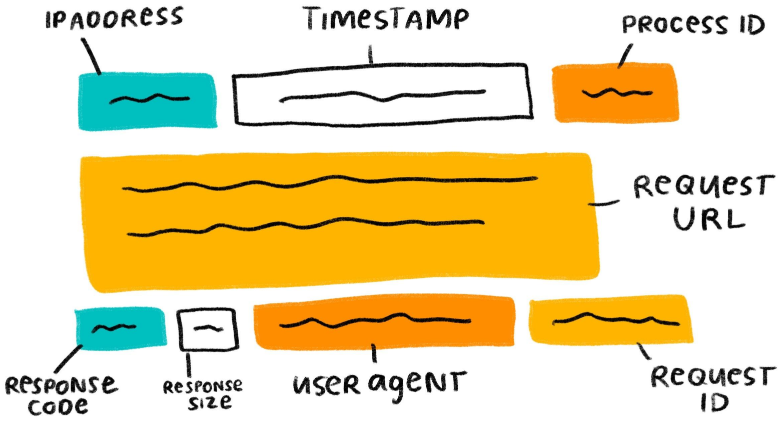


We need contextual data.

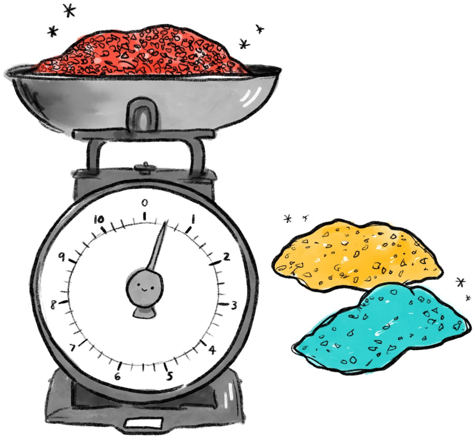


User experiences ≈ marbles.

Marbles have many properties.



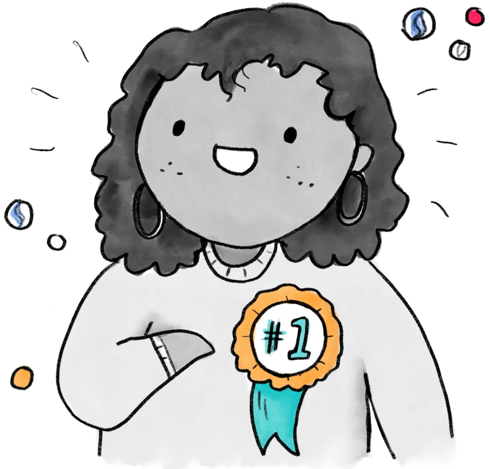
So do events in our systems.



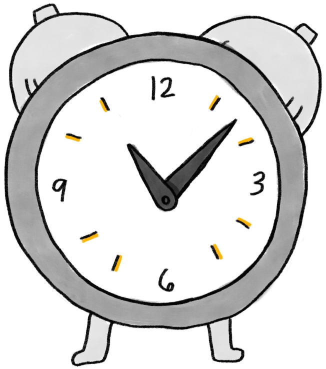
How many/how much?

GUESS
how many
MARBLES
are in the
JAR!

Have you ever played this game?



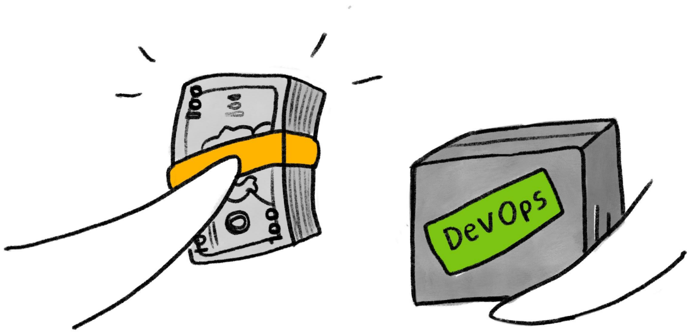
How can we win the game?



(without spending all day)

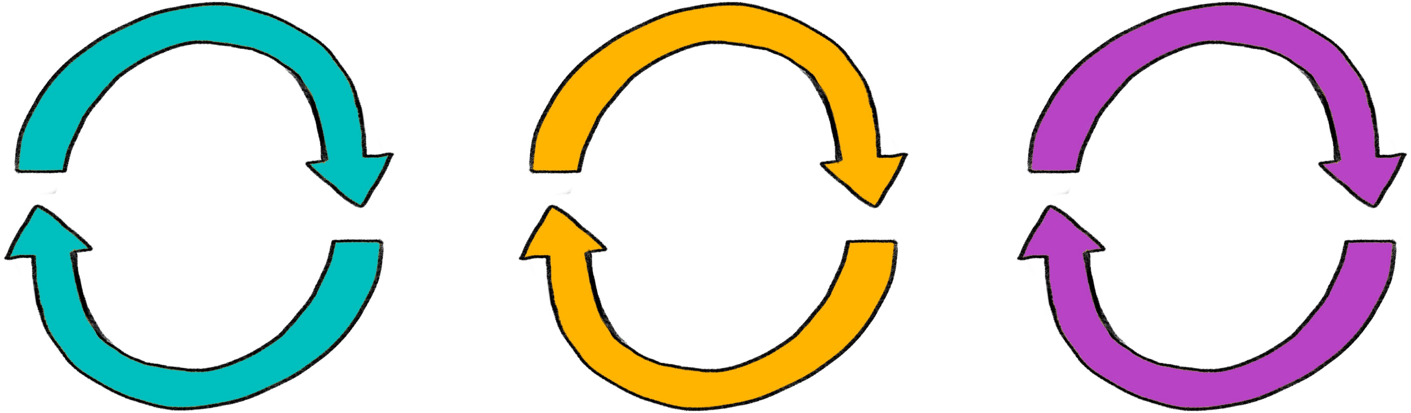
and what about these variations?

How can we debug our systems...

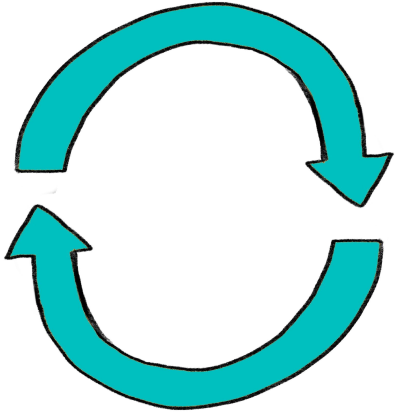


without breaking the bank?

Three strategies for taming the spew.

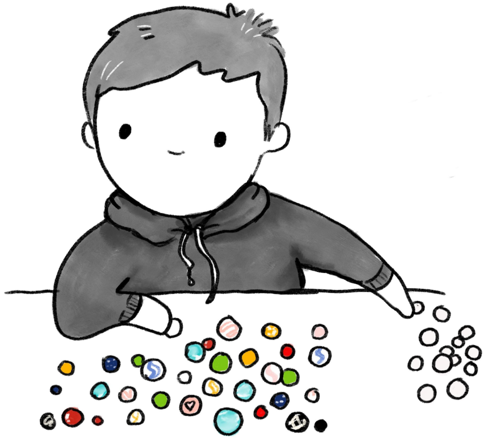


Reduce. Reuse. Recycle.



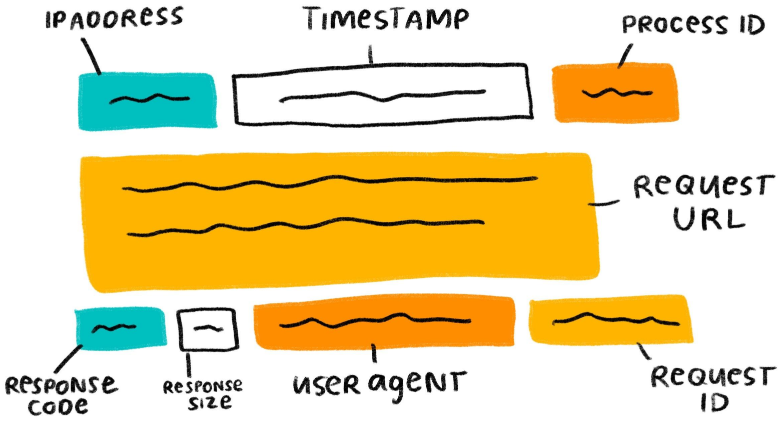
(1) Store less data.

Back to the marble analogy...

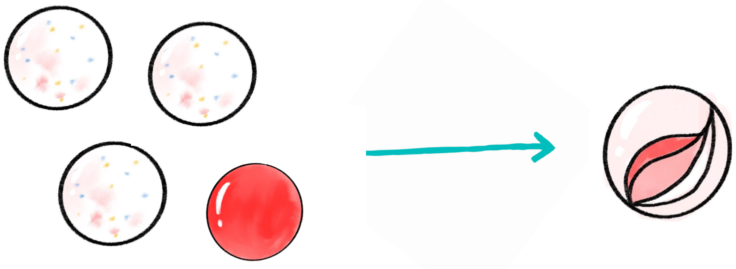


Reduce what we need to count.

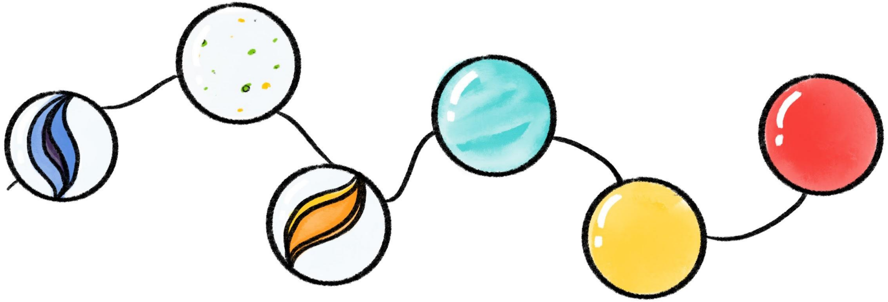
Stop writing read-never data.



First, structure your data.

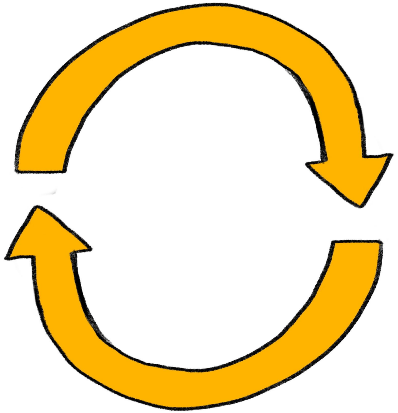


One event per transaction.

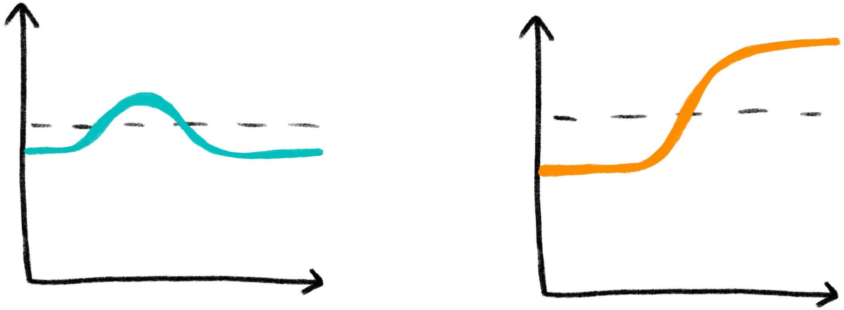


Use tracing for linked events.

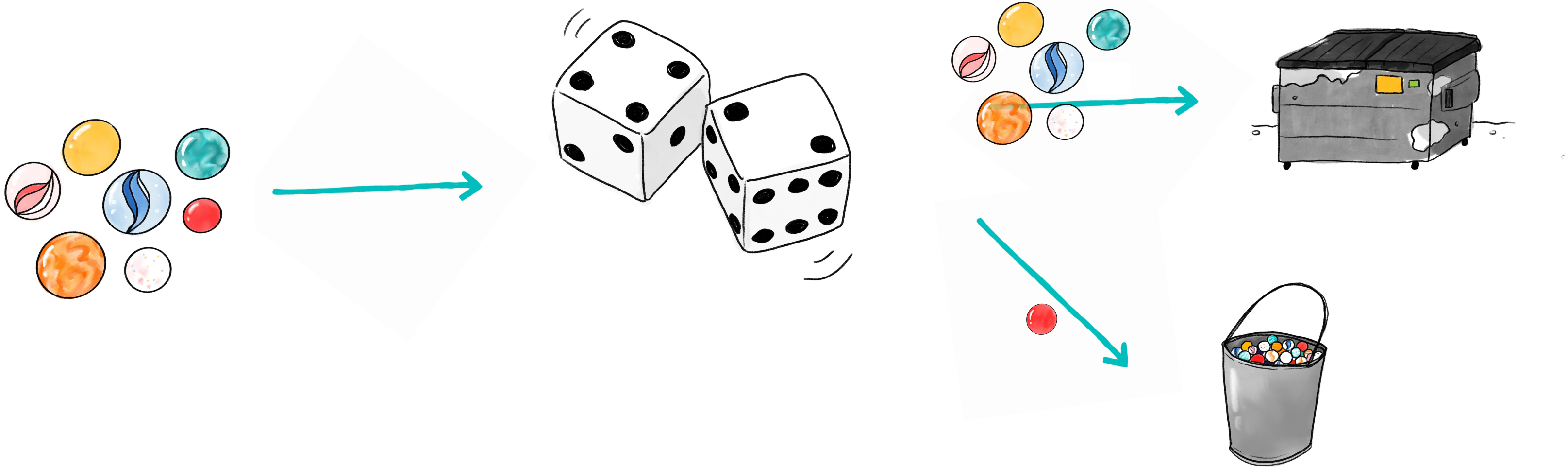
Often, trimming isn't enough.



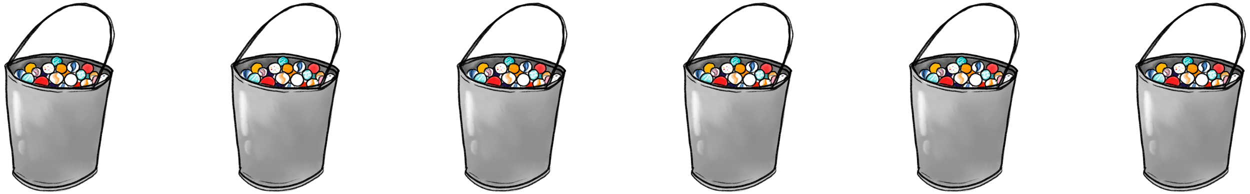
(2) Sample your data.



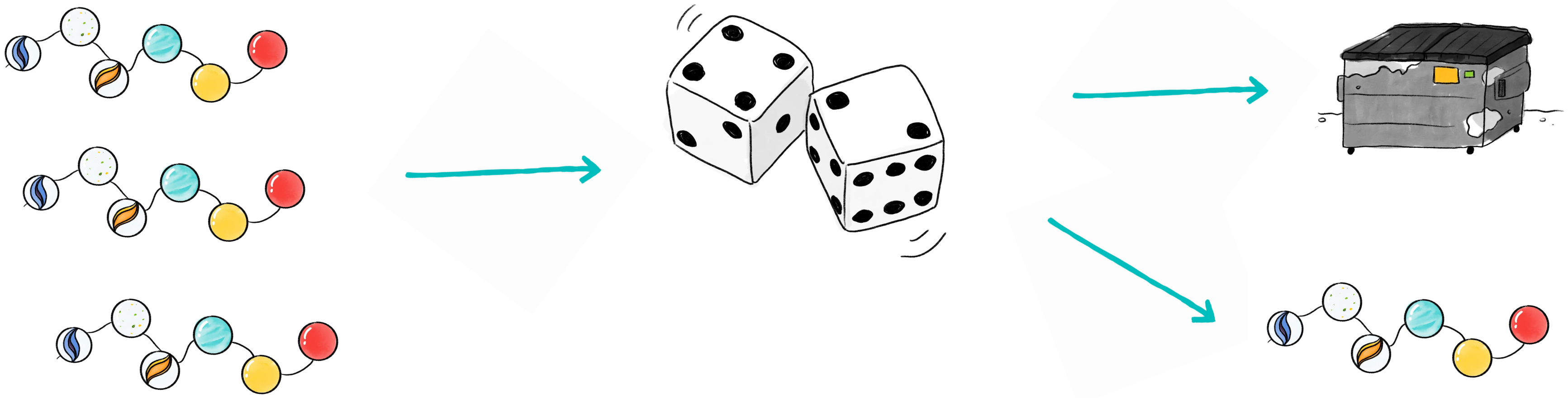
Statistics to the rescue!



Count 1/N events.



Weight the results by N afterwards.



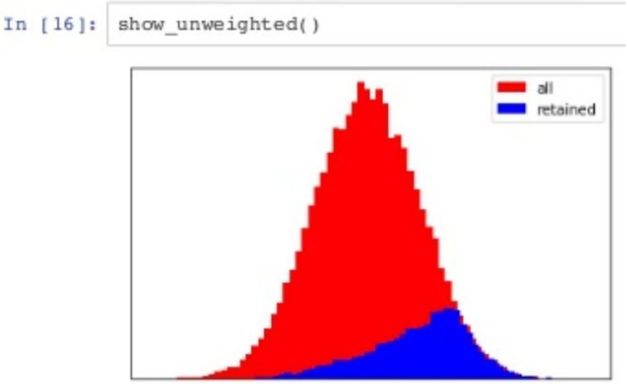
Count traces together.

```go
var sampleRate = flag.Int("sampleRate", 1000, "Service's sample rate")

func handler(resp http.ResponseWriter, req *http.Request) {
    // Use an upstream-generated random sampling ID if it exists.
    // otherwise we're a root span. generate & pass down a random ID.
    var r float64
    if r, err := floatFromHexBytes(req.Header.Get("Sampling-ID")); err != nil {
        r = rand.Float64()
    }

    start := time.Now()
    // Propagate the Sampling-ID when creating a child span
    i, err := callAnotherService(r)
    resp.Write(i)

    if r < 1.0 / *sampleRate {
        RecordEvent(req, *sampleRate, start, err)
    }
}
```

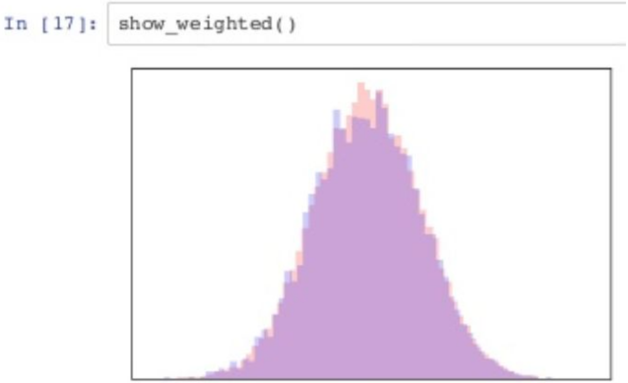
11x

Don't be afraid of sample rates.

Distinct samples ~> accuracy.



```
In [16]: show_unweighted()
```



```
In [17]: show_weighted()
```

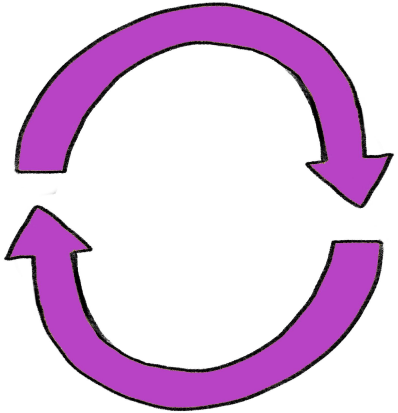
Don't believe me? Ask a data scientist.

Ross, Joe (SignalFx). "Statistical Aspects of Distributed Tracing" at Monitorama Portland 2019

https://www.slideshare.net/secret/INwmsyntwaBbx7

i-Quantiles & SLOs are sample-safe*!

* caveats at Heinrich Hartmann's Statistics for Engineers

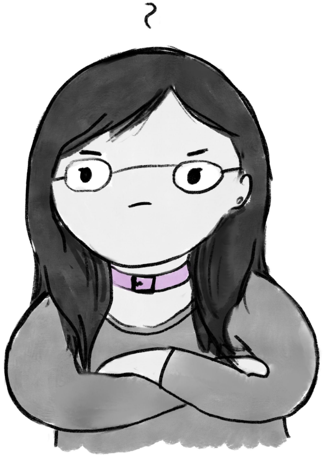


(3) Aggregate data.

Aggregation destroys cardinality.

This has mixed results.

Cheap to answer known queries.

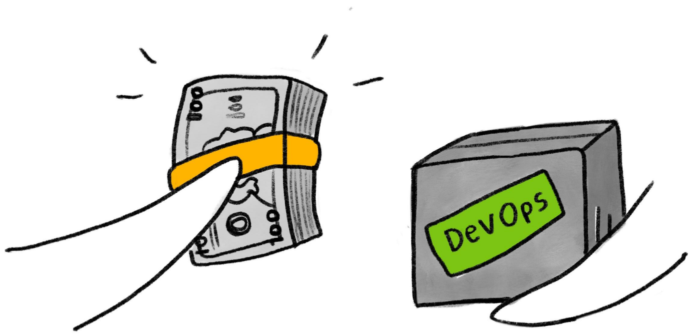


But inflexible for new questions.

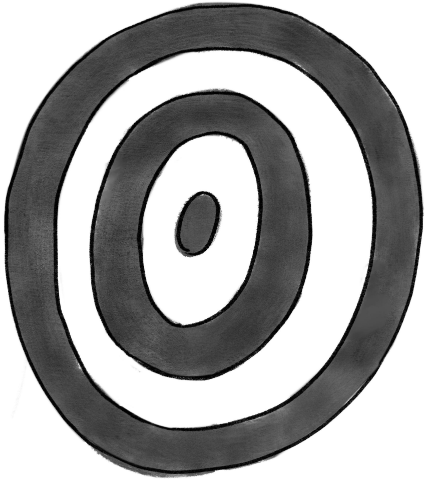
Temporal correlation is weak.

Math on quantiles is misleading.

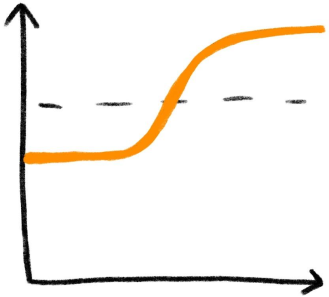
Aggregation is a last resort.



How can sampling be cheap enough?



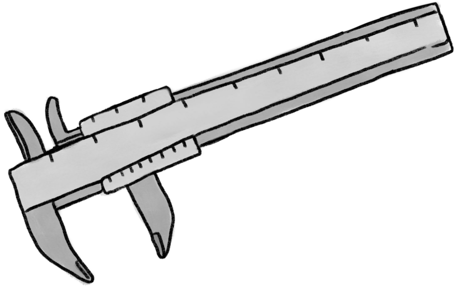
Target-rate sampling



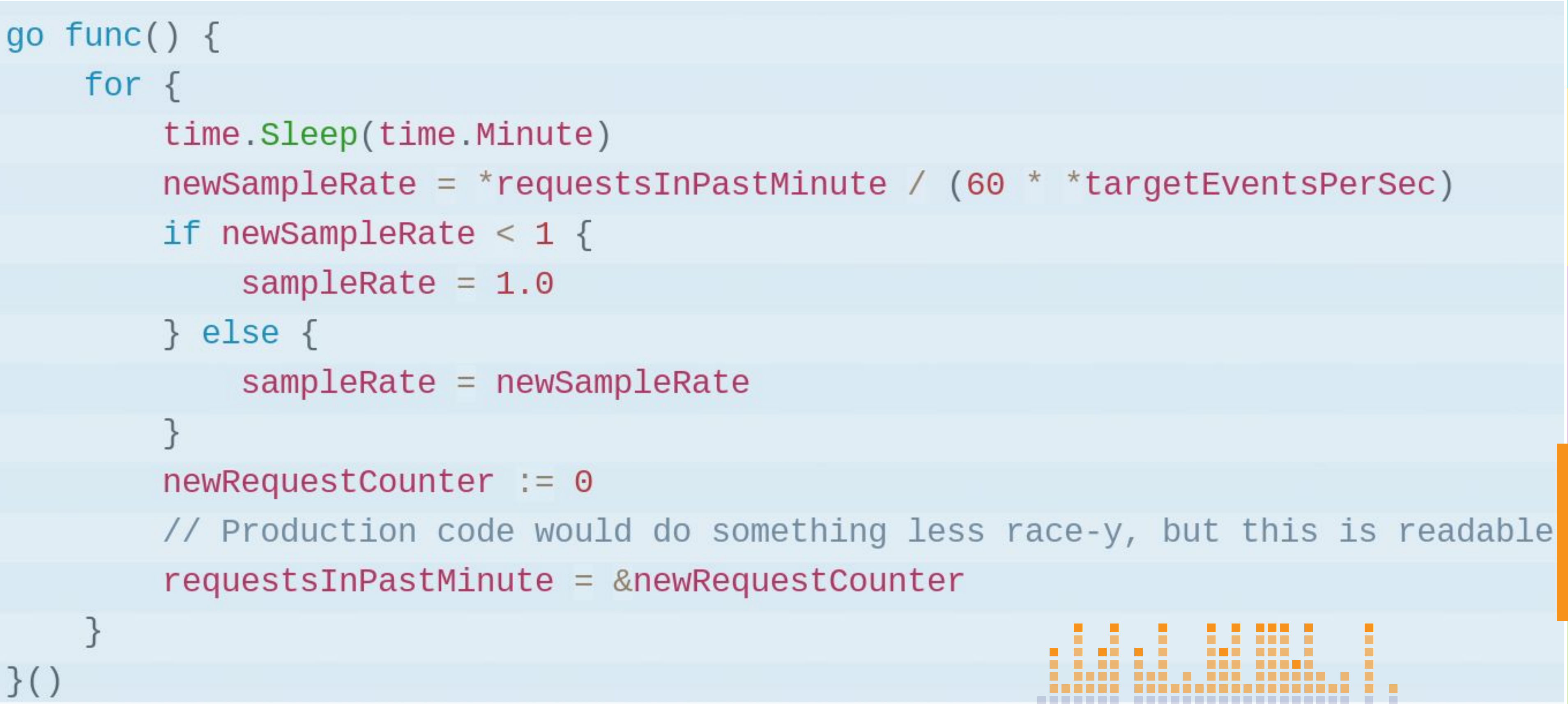
Systems scale with load.

Cost predictability matters.

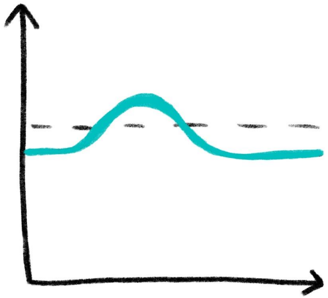
Keep enough traces to debug.



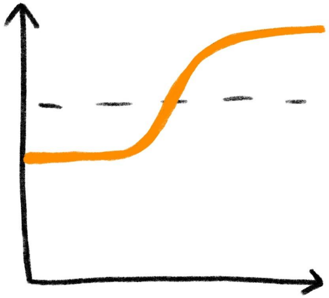
Adjust on trailing volume.



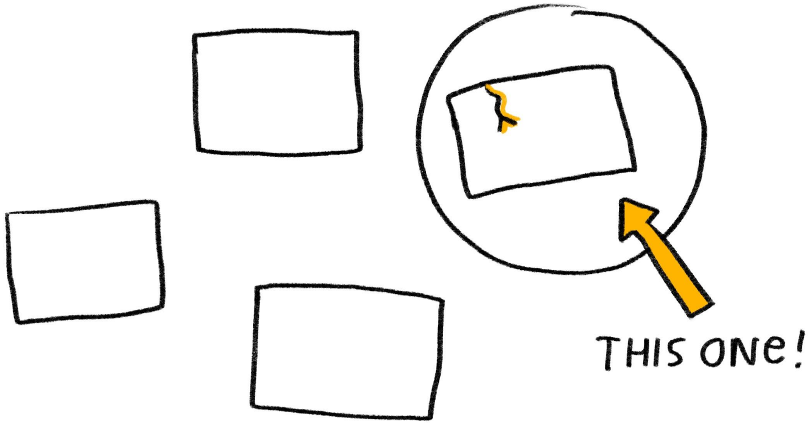
```go
go func() {
    for {
        time.Sleep(time.Minute)
        newSampleRate = *requestsInPastMinute / (60 * *targetEventsPerSec)
        if newSampleRate < 1 {
            sampleRate = 1.0
        } else {
            sampleRate = newSampleRate
        }
        newRequestCounter := 0
        // Production code would do something less race-y, but this is readable
        requestsInPastMinute = &newRequestCounter
    }
}()
```



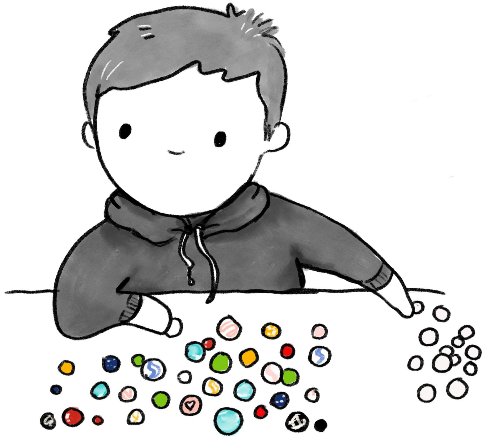
Keep a **consistent** number of events.



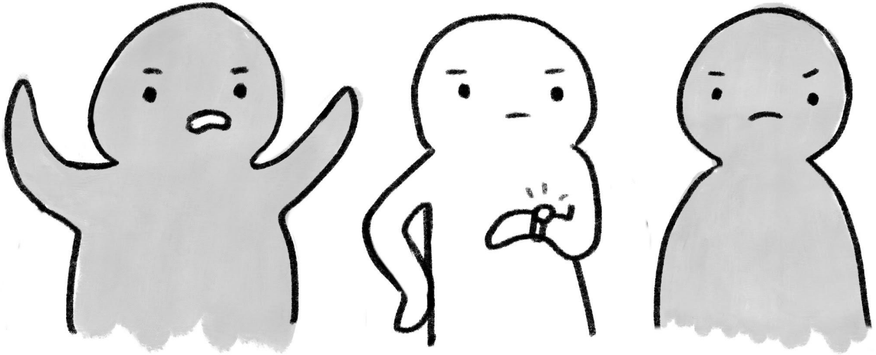
Reconcile using the sample rate.



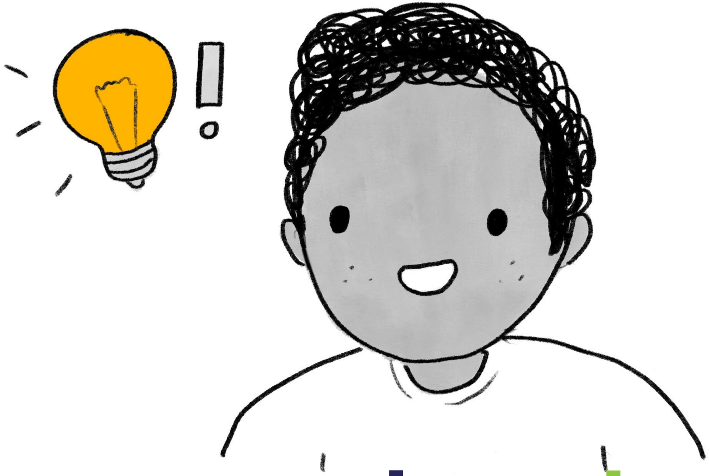
Per-key sampling



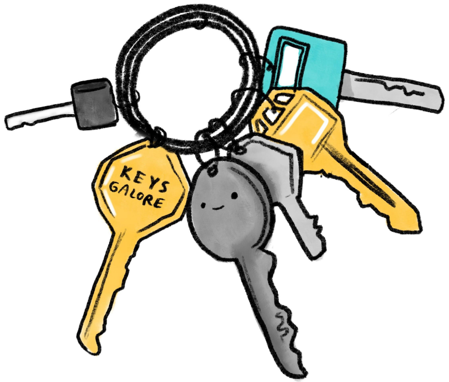
99%+ of events are low in signal.



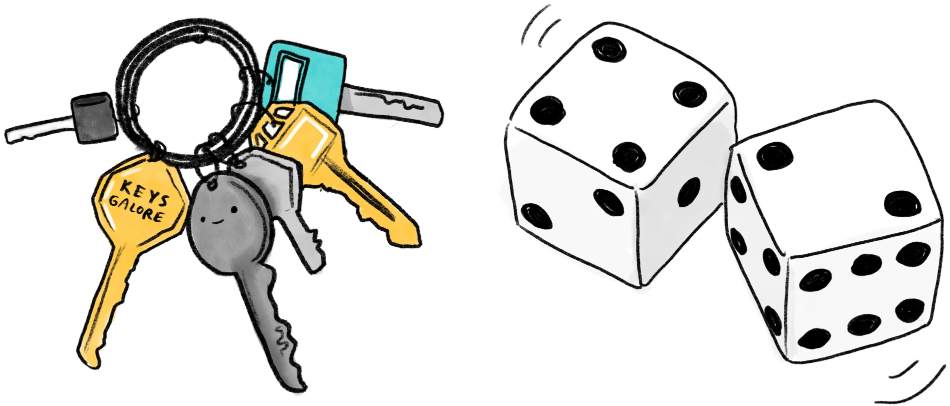
Each customer is unique.



How can we save the relevant events?



Normalize per-key.

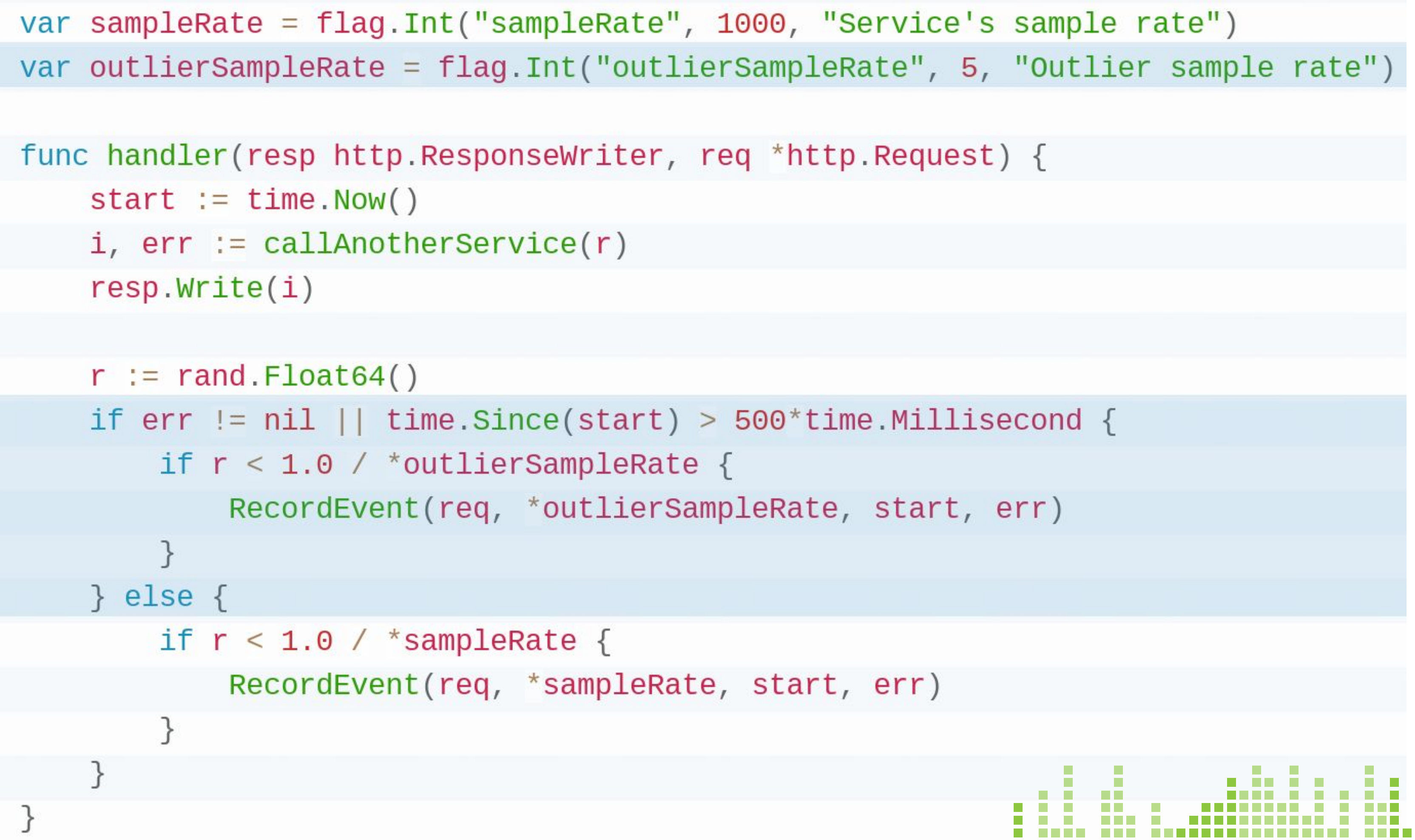


Different key, different probability.

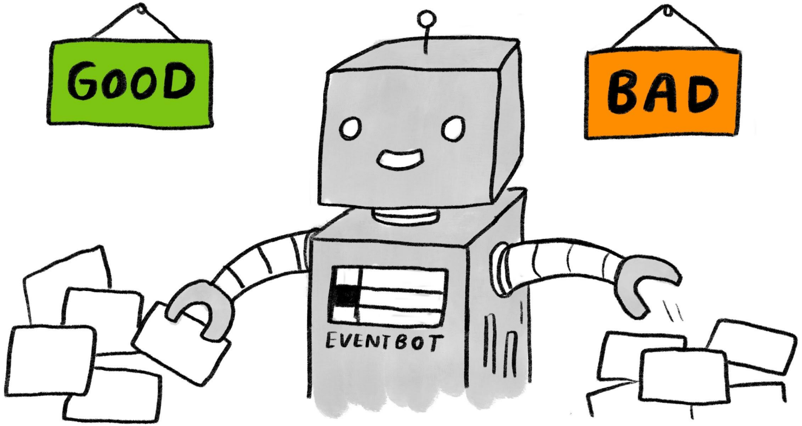
Sample down voluminous customers.

1x

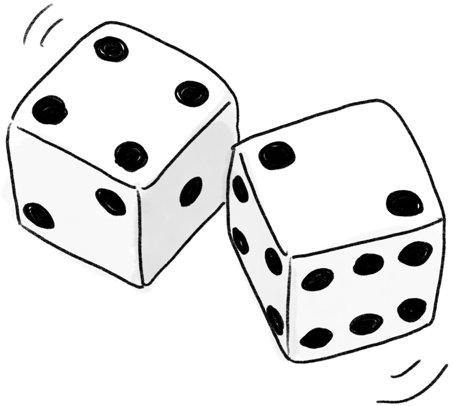
Retain errors & slow queries.

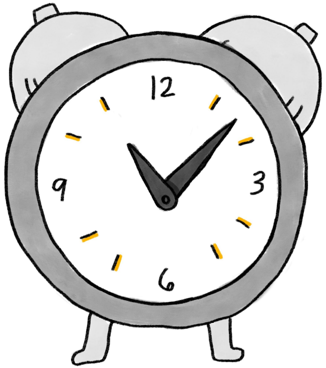


```go
var sampleRate = flag.Int("sampleRate", 1000, "Service's sample rate")
var outlierSampleRate = flag.Int("outlierSampleRate", 5, "Outlier sample rate")

func handler(resp http.ResponseWriter, req *http.Request) {
    start := time.Now()
    i, err := callAnotherService(r)
    resp.Write(i)

    r := rand.Float64()
    if err != nil || time.Since(start) > 500*time.Millisecond {
        if r < 1.0 / *outlierSampleRate {
            RecordEvent(req, *outlierSampleRate, start, err)
        }
    } else {
        if r < 1.0 / *sampleRate {
            RecordEvent(req, *sampleRate, start, err)
        }
    }
}
```

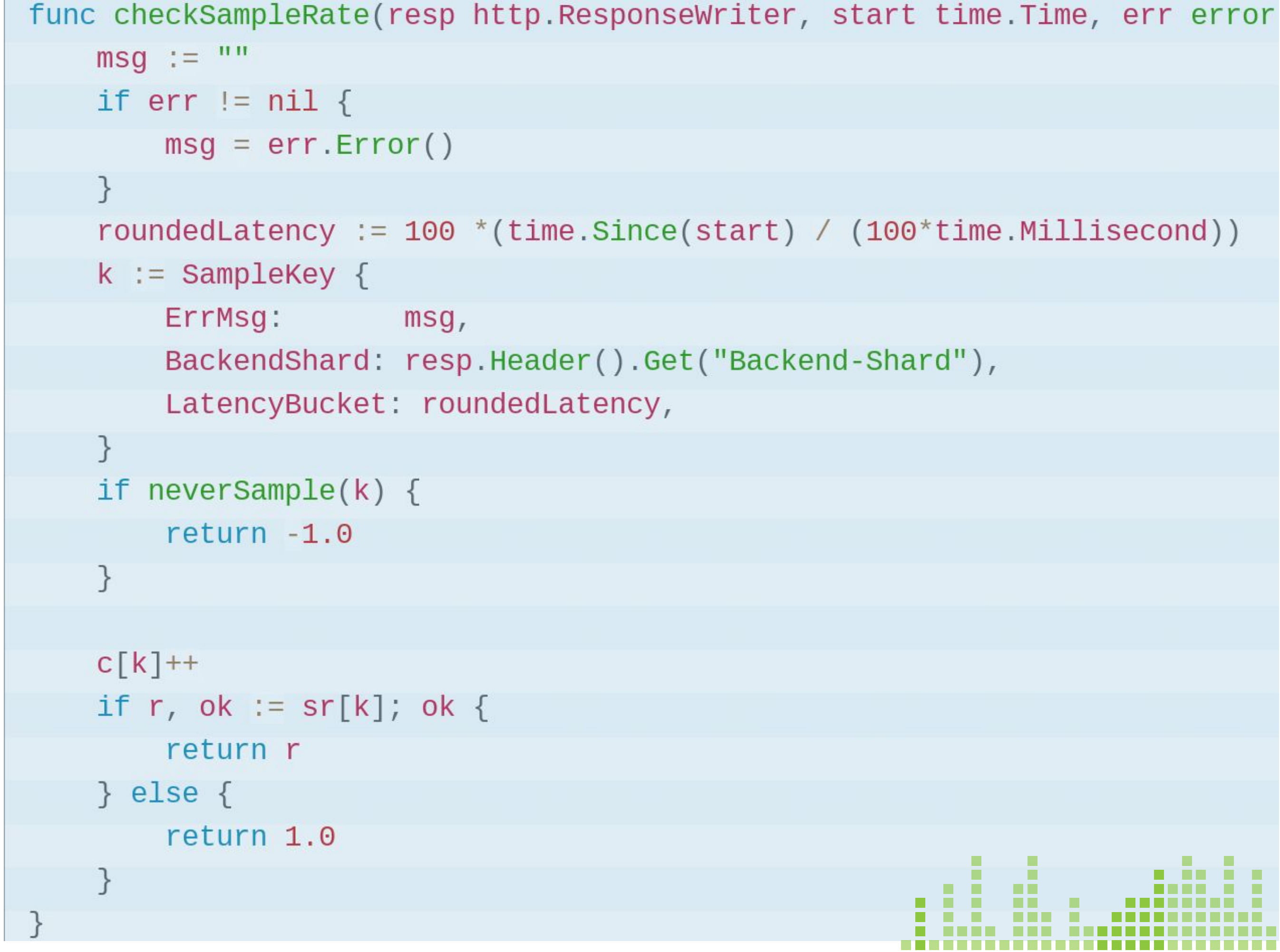


Buffering lets us choose wisely.





Let's put both together!

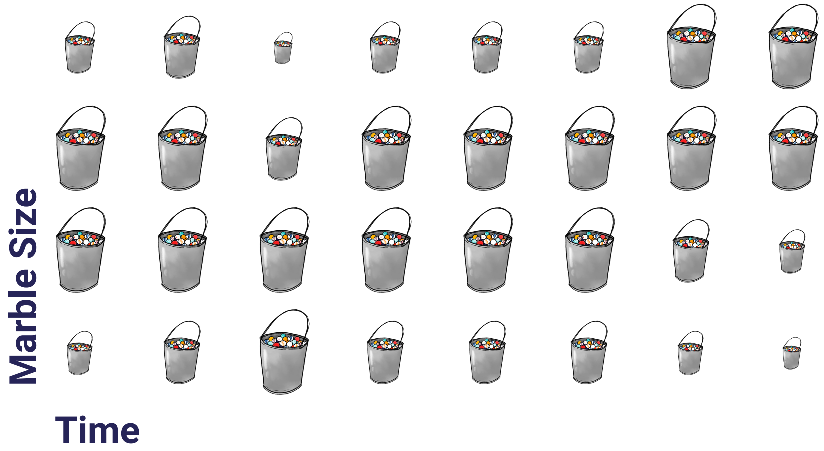


```
func checkSampleRate(resp http.ResponseWriter, start time.Time, err error
    msg := ""
    if err != nil {
        msg = err.Error()
    }
    roundedLatency := 100 *(time.Since(start) / (100*time.Millisecond))
    k := SampleKey {
        ErrMsg:       msg,
        BackendShard: resp.Header().Get("Backend-Shard"),
        LatencyBucket: roundedLatency,
    }
    if neverSample(k) {
        return -1.0
    }

    c[k]++
    if r, ok := sr[k]; ok {
        return r
    } else {
        return 1.0
    }
}
```

Low-volume data is precious.

Target-rate + Per-Key = ~~💜~~

"But I love aggregated metrics!"



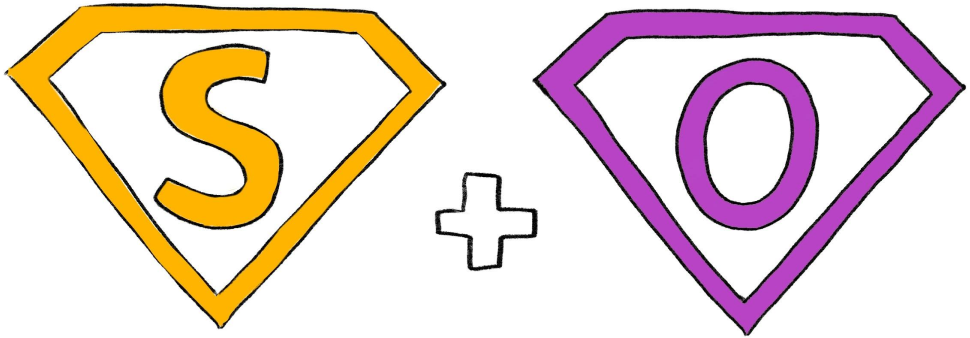
Distributions are bucket counts.

Exemplar: Distribution + Sampled Events

This is the same concept!

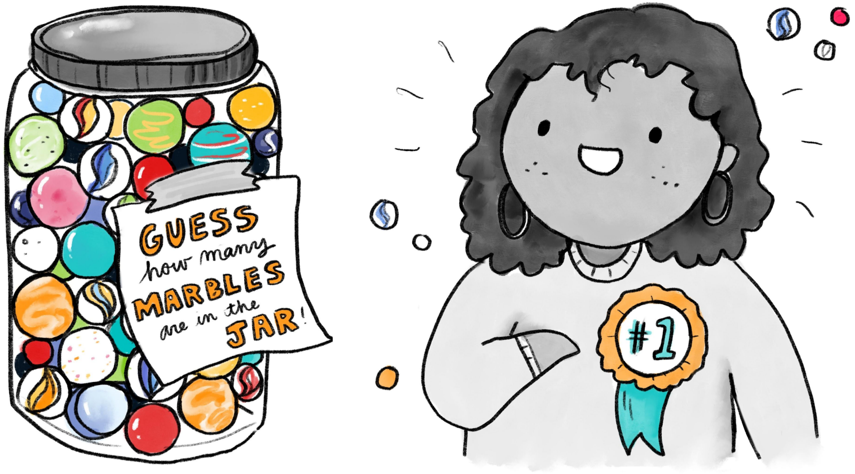
Metrics and events can be friends!

You can prevent data spew!



Get the right data. Cheaply enough.

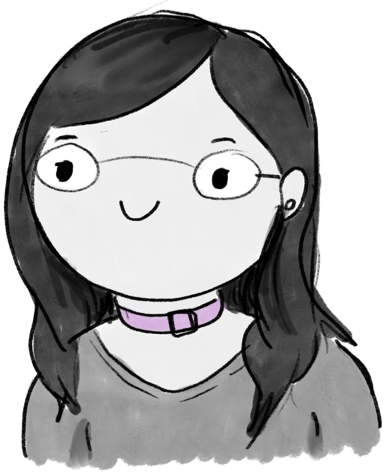
Structure. Sample. (Aggregate?)



Refine your data.

Reduce, Reuse, & Recycle. Wisely!

lizthegrey.com; @lizthegrey



Find me at the BLAMELESS booth!

lizthegrey.com; @lizthegrey