Martyrs On Film: learning to hate the #oncallselfie
by Alice Goldfuss
Published May 22, 2017 in Technology
Why *does* `#monitoring(still)suck`?

The cost of maintenance must scale sublinearly with the growth of the service. The service size includes e.g. queries, storage footprint, cores used, watts.

Why does `#monitoring(still)suck`?

The cost of maintenance must scale sublinearly with the growth of the service.
Symptom Based Alerting

https://www.flickr.com/photos/chris-warren-photos/2220257496/ CC-BY-NC 2.0
Is this a symptom
What makes this a symptom?

Engineering Tolerance

Availability “Tolerance”
**SLAs, SLOs, SLIs**

- **SLI → Indicator:** a measurement
  - distribution of response time over 10 minutes
  - response error ratios over 10 minutes
- **SLO → Objective:** a goal
  - 99.9th percentile response latency below 5ms
  - lower than 1% rate of errors
- **SLA → Agreement:** economic incentives
  - or we get paged
“As a mechanical engineer in an R&D lab I frequently ask myself, what is a reasonable tolerance to set on this part?”

https://engineerdog.com/2017/12/02/engineering-guidelines-for-selecting-mechanical-design-tolerances/
Does your service have an SLO?

The answer may surprise you!

Posted on May 31, 2018, at 1:53 p.m.

Jamie Wilkinson
Site Reliability Engineer
A symptom is anything that can be measured by the SLO.

A symptom-based alert is an alert when the SLO is in danger of being missed.
For availability SLAs we often talk about *system uptime*:

\[
\text{availability} = \frac{\text{uptime}}{\text{uptime} + \text{downtime}}
\]

How do you measure uptime of a distributed system?
Another way to calculate this is with a request success rate:

\[
\text{availability} = \frac{\text{successful requests}}{\text{total requests}}
\]

Defining SLOs in terms of request success rate makes it easier to measure an error budget.
```go
var responses = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Name: "responses",
        Help: "total errors served"},
    []string{"code", "user"})
...

responses.WithLabelValues(
    http.StatusText(400),
    GetUser(req)).Add(1)
```
1/qps sample density

queries per second

time, 1 second samples
record: error_ratio_by_user
expr: sum by (job, user)(
    rate(responses{code!~"200"}[10s]))
    / on (job, user)
    sum by (job, user)(rate(responses[10s]))

alert: ErrorRatioTooHigh
expr: error_ratio_by_user > 0.01
SLO Burn 🔥

cumulative errors vs. time

- alerting window
- error rate threshold
- scaled error budget
Burn rate maths

Average QPS rate: 1000
SLO: 99% over 1 week
= 604,800,000 total queries
= 6,048,000 permissible errors
Fast Burn alert

Take 1 hour moving average of errors Page if error budget is going to be exhausted in less than 24 hours

= 6,048,000 errors consumed per day
= 70 err/s = 252,000 errors in 1 hour
Page if 15m rate over 70.
expr: delta(errors[1h]) > (expected_events * error_budget / burn_period)
=
expr: delta(errors[1h]) > ((1000 qps * 7d) * 0.01 / 24h)
=
expr: delta(errors[1h]) > 70
Slow SLO Burn

Alerting window = ??

Scaled error budget = ??

Error rate page threshold = ?? err/s

SLO error rate = 10 err/s
The pillar(s) of observability?

1. Logs (preformatted events)
2. Metrics (prefiltered and preaggregated events)
3. Traces (events in a tree structure)
4. Exceptions/stack traces, core files (mass extinction events)
“one of the most powerful context-sensitive incredibly adaptive anomaly-detecting and responding agents in the world”

-- John Allspaw, Monitorama 2013
What questions do you have?
Source code for the demo

https://github.com/jaqx0r/blts
Backup Slides
1. Symptom-based alerts are good for your health
2. SLO is defined by you, customers, and system
3. SLO implies error budget, informs engineering tolerance
4. Page only on SLO risk, because that's what matters
import "github.com/golang/glog"

func HandleUrl(...) {
    ...
    glog.Infof("url %s returned %s took %d seconds", url, code, duration)
}

/url (.*) returned (.*.) took (\d+) seconds/ {
    responses[$1][$2]++
    latency[$1][$2][floor($3/1000.)*1000]++
}
import "github.com/prometheus/client_golang/prometheus"

var response_latency =
    prometheus.NewHistogram(prometheus.HistogramOpts{
        Name: "http_responses",
        Buckets: prometheus.LinearBuckets(20, 5, 5),
    }, []string{"url", "code"})

func HandleUrl(...) {

    ...
    
    response_latency.WithLabelValues(url, code).Observe(duration)
}

sum by(url, code)(irate(http_responses[5m]))
import "go.opencensus.io/trace"

func HandleUrl(...) {
    ctx, span := trace.StartSpan(ctx, url)
    defer span.End()
    ...
    span.Annotate([]trace.Attribute{
        trace.StringAttribute("code", code),
        trace.Int64Attribute("duration", duration),
    } "handler return")
}

select * from traces where duration > 5000
import "github.com/honeycombio/libhoney-go"

func HandleUrl(...) {
    ...
    event := libhoney.NewEvent()
    event.AddField("code", code)
    event.AddField("duration", duration)
    event.AddField("url", url)
}
glog.Infof("url %s returned %s took %d seconds", url, code, duration)

{
  'formatStr': 'url %s returned ...
  'url': url,
  'code': code,
  'duration': duration,
}

c.f. Nanolog: [github.com/Platformlab/NanoLog]
glog.Infof("url %s returned %s took %d seconds", url, code, duration)

span.Annotate([]trace.Attribute{
    trace.StringAttribute("url", url),
    trace.StringAttribute("code", code),
    trace.Int64Attribute("duration", duration),
} "handler return")

event := libhoney.NewEvent()
eventaddField("code", code)
event.AddField("duration", duration)
event.AddField("url", url)
glog.Infof("\textbf{request_id} %d url %s returned %s took %d seconds", uid, url, code, duration)

ctx, span := trace.StartSpan(ctx, url)
defer span.End()
span.Annotate([]trace.Attribute{
    trace.StringAttribute("code", code),
    trace.Int64Attribute("duration", duration),
} "handler return")

event := libhoney.NewEvent()
eventaddField("code", code)
event.AddField("duration", duration)
event.AddField("url", url)

rainbow unicorn emoji

ned %s took %d seconds", url, code, duration)
response_latency.WithLabelValues(url, code).Observe(duration)

event := libhoney.NewEvent()
event.AddField("code", code)
event.AddField("duration", duration)
event.AddField("url", url)