A Tale of Two Histograms

It was the best of response times, it was the worst of response times

Adrian Cockcroft

OrionX.net - @adrianco@mastodon.social

All photos used as distracting eye candy in this presentation were taken by Adrian Cockcroft

An Apology

I submitted a geeky talk proposal that explores some obscure statistical methods

Jason decided to inflict it on you first thing on Monday morning…

You may need more coffee…

@adrianco@mastodon.social

This talk is an extended version of my blog post from Jan 29, 2023

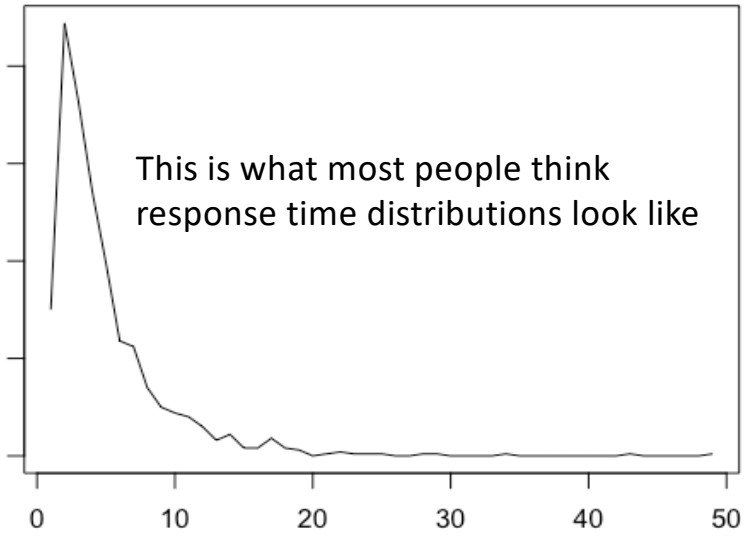
https://medium.com/@adrianco/percentiles-dont-work-analyzing-the-distribution-of-response-times-for-web-services-ace36a6a2a19

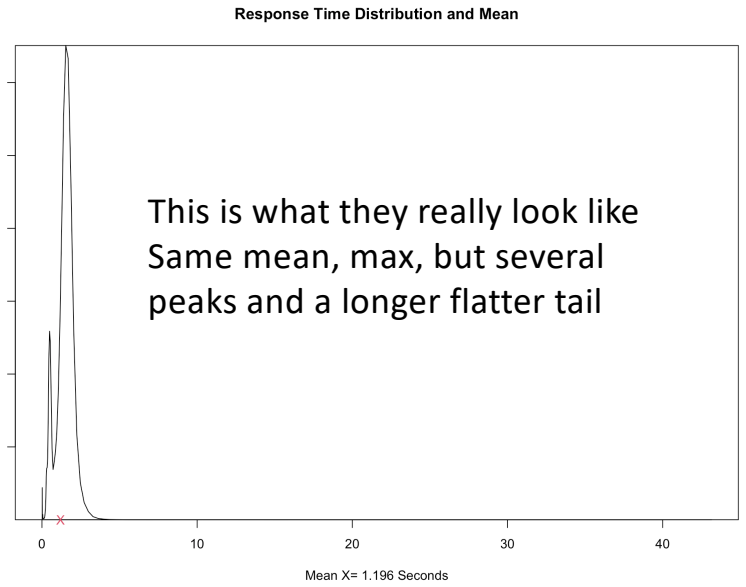
@adrianco@mastodon.social


What’s wrong with response times?
@adrianco@mastodon.social

What's wrong with response times?

This is a tale of these two histograms





@adrianco@mastodon.social

What’s wrong with response times?

Small changes in load may lead to large changes in response time

Averages aren’t useful, percentiles are better

@adrianco@mastodon.social

What's wrong with percentiles?

Small changes in load may lead to unpredictable large changes in percentiles

You can't summarize percentiles over time

Given 1 minute percentiles for an hour, you can't average them together to get the percentile for the hour

You get a number, that may be useful, but it won't be a percentile

@adrianco@mastodon.social

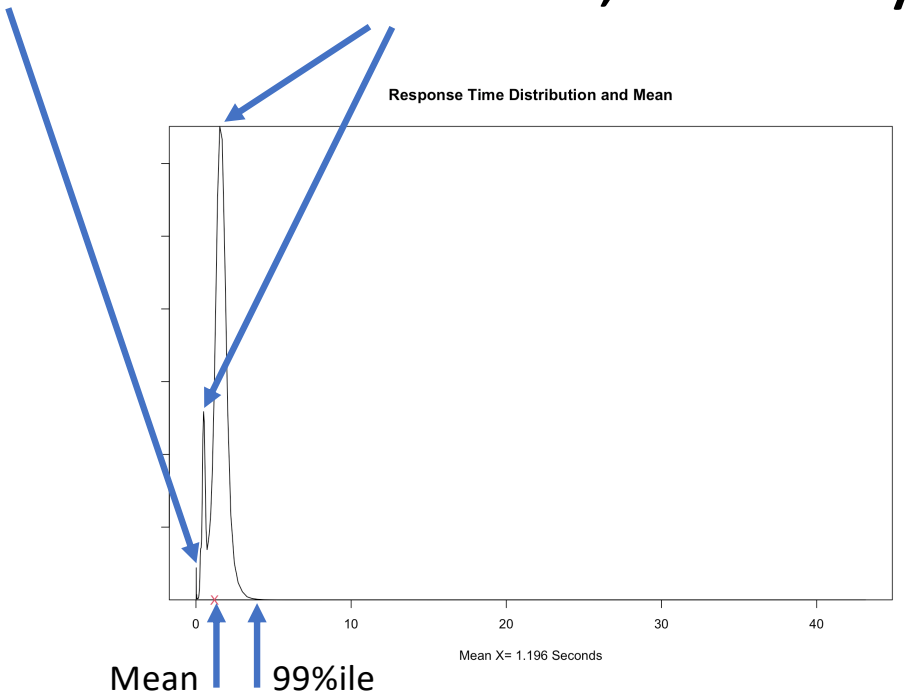
What's wrong with percentiles?

There's a lot of interesting structure in the response time distribution for real systems, that percentiles can't capture

@adrianco@mastodon.social

What should I really care about?

Where are the modes, did they change?



Remember high school stats?
Mean, Median and Mode

Mean is the average
Median is the 50th percentile
Mode is the most likely

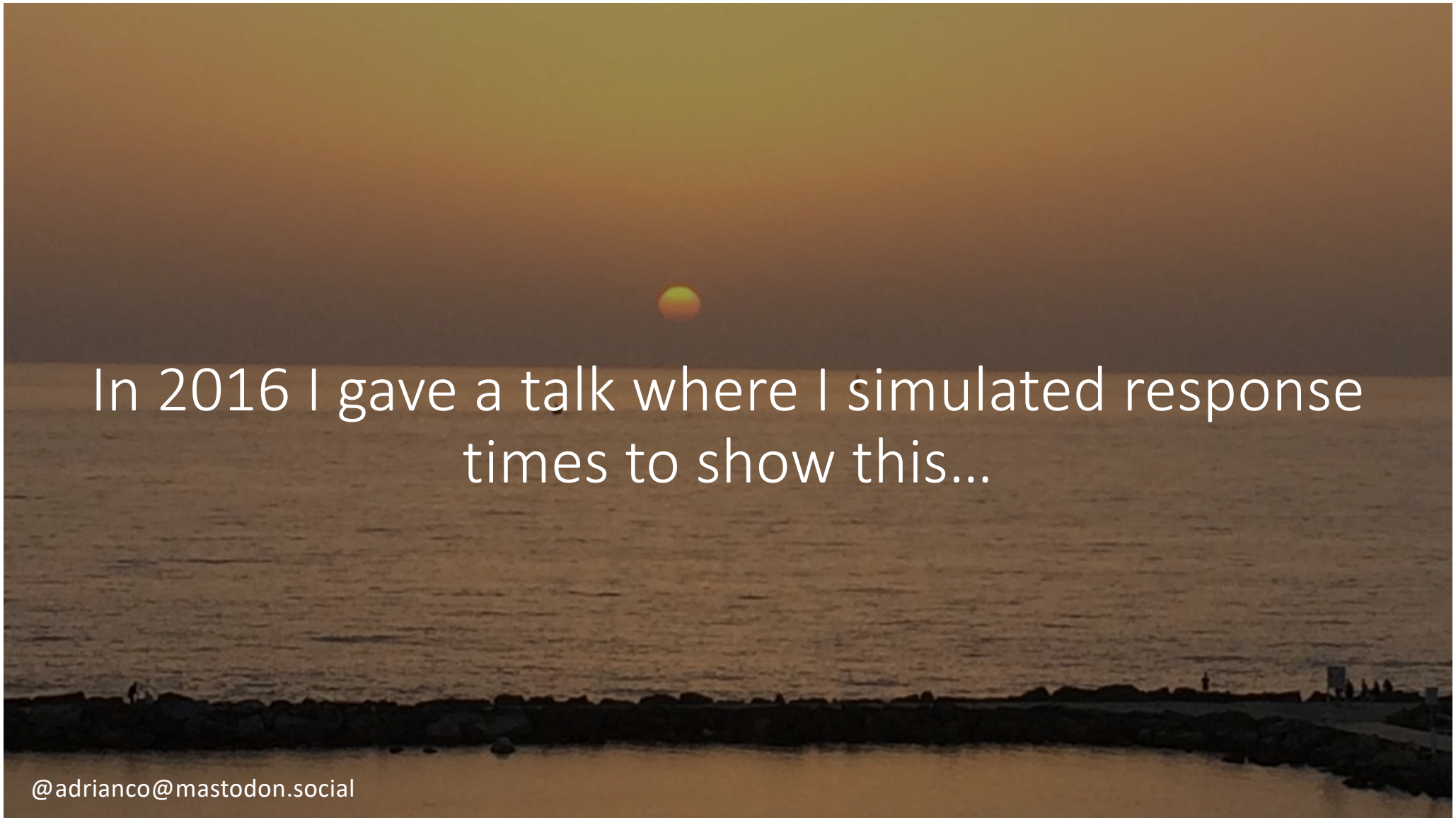
So why don't we use the modes?

@adrianco@mastodon.social

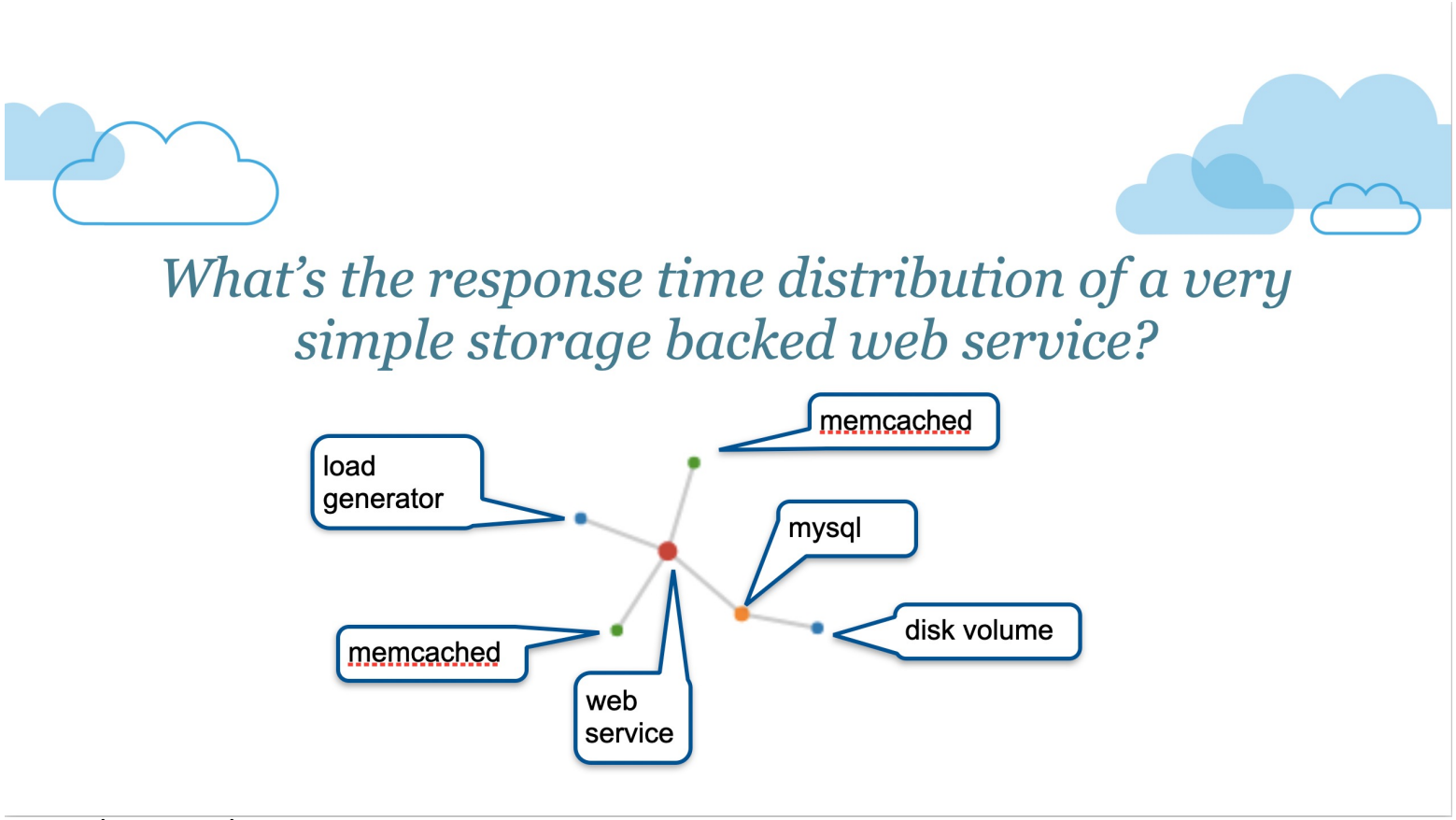

Why is there structure in response times?
@adrianco@mastodon.social

Requests do different work and take many different paths through the system

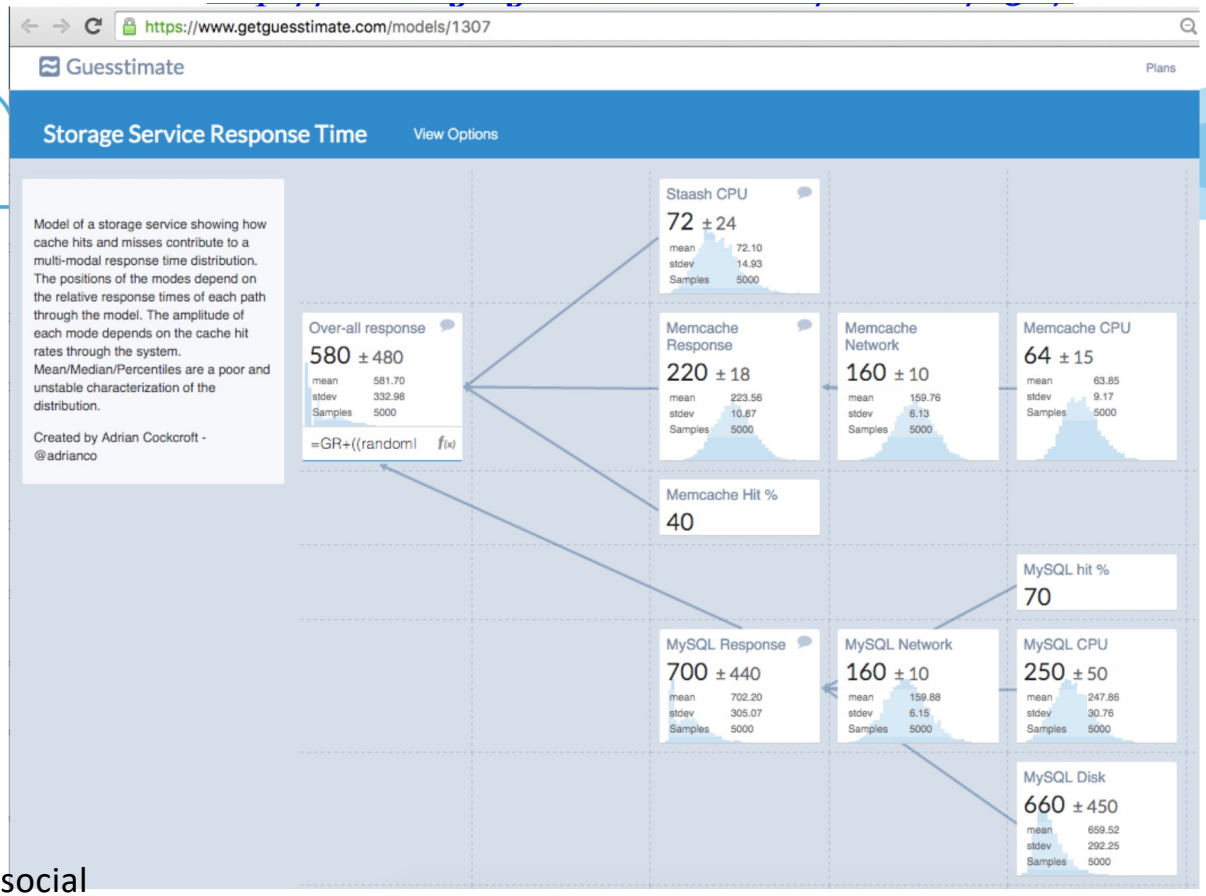
The response time distribution is a **mixture** of several underlying distributions


In 2016 I gave a talk where I simulated response times to show this…
@adrianco@mastodon.social

Slides from MicroXchg talk and Microservices Tutorial 2016
see: github.com/adrianco/slides

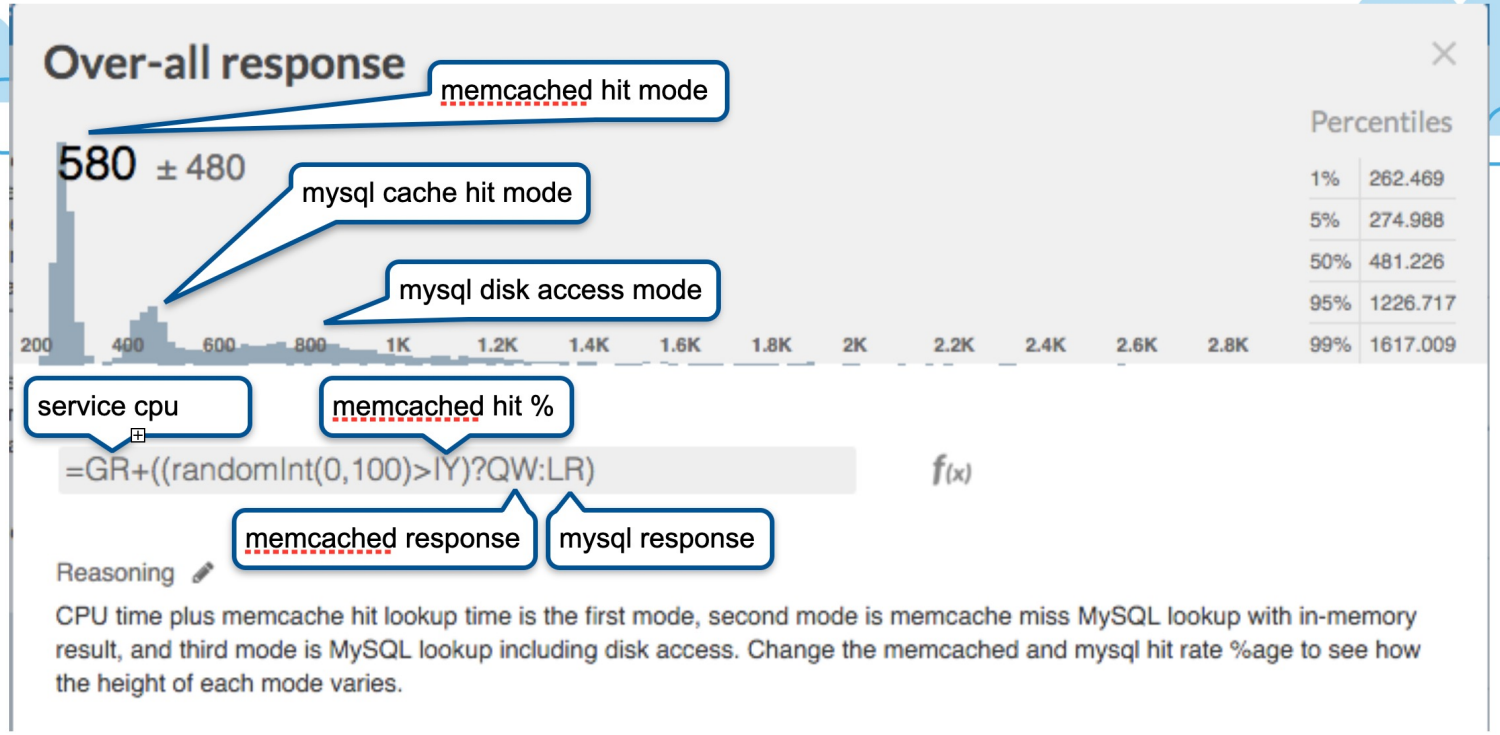


Guesstimate is a Monte-Carlo simulator spreadsheet that runs in your browser see: http://www.getguesstimate.com/models/1307



Response times for each component are combined together

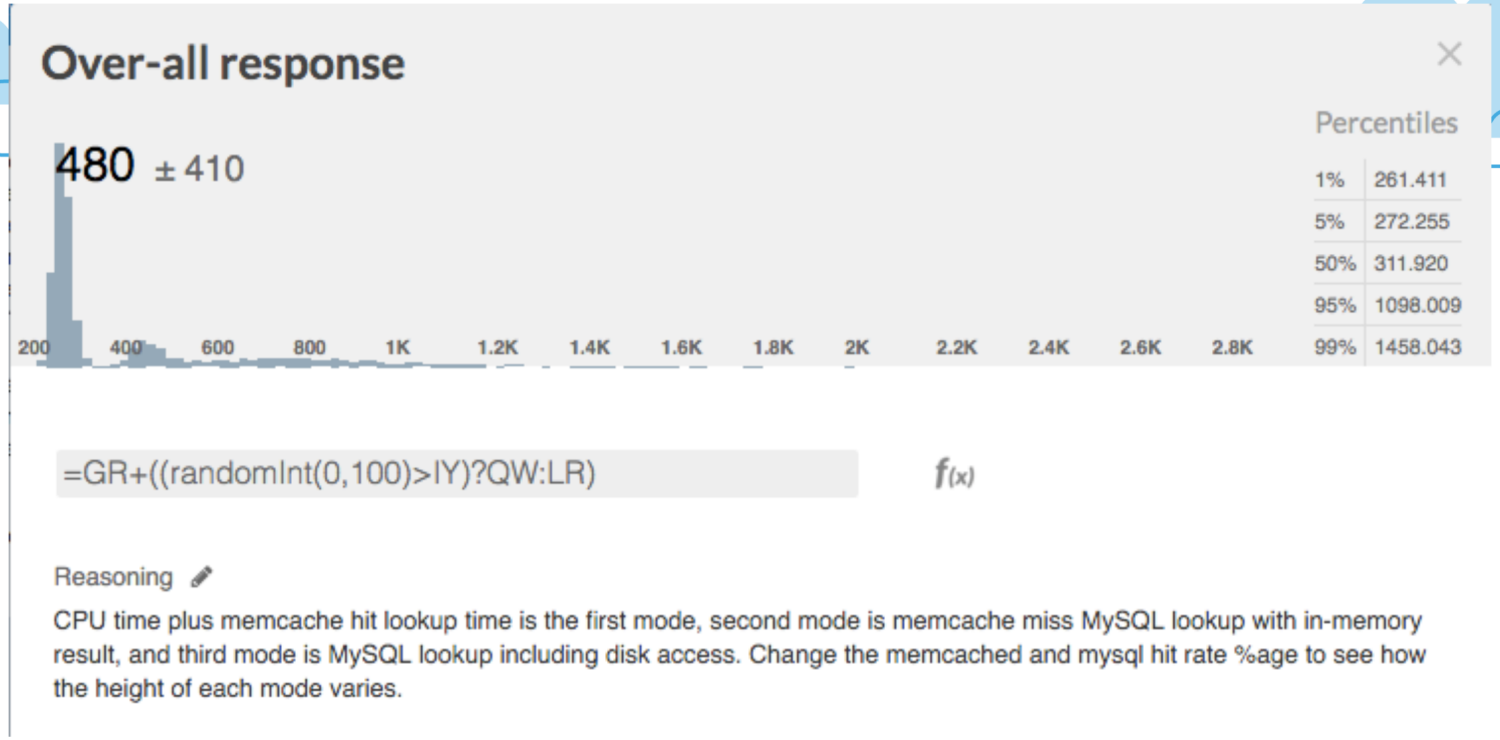
Hit rates: memcached 40% mysql 70%



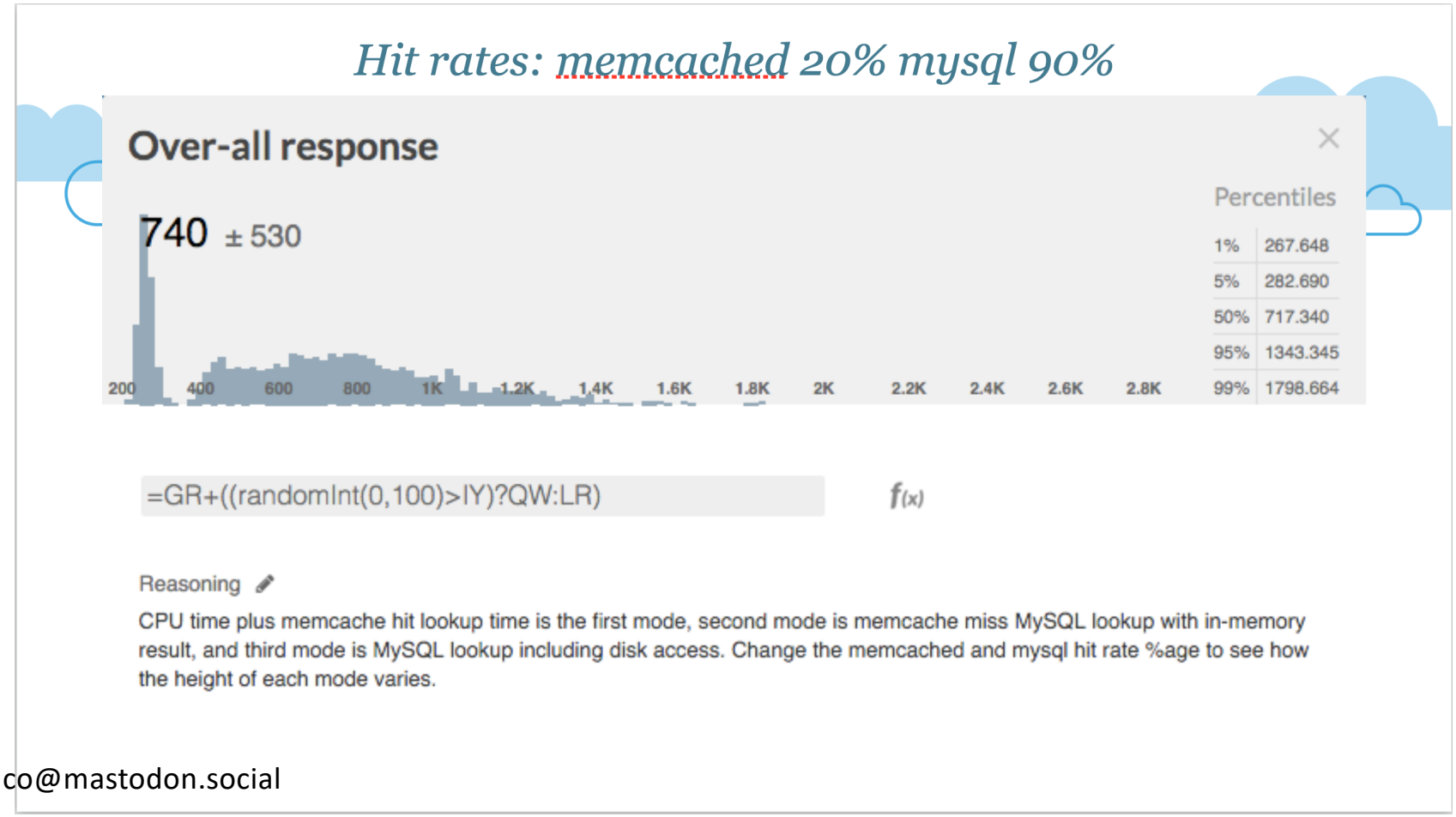
@adrianco@mastodon.social

Response times for each component are combined together

Hit rates: memcached 60% mysql 70%



Response times for each component are combined together



@adrianco@mastodon.social


What we need is an algorithm that can separate out the mixture
@adrianco@mastodon.social

It took me several years before I figured out what this statistical technique is called, so I could find some code that does it (using the R language)

During a discussion with Donny Berkholtz on Mastodon in early 2023, Ed Borasky piped up and said “I think you are looking for Mixed Methods, there’s an R package called mixtools”. Thanks Ed!

@adrianco@mastodon.social

I told you you would need some coffee

Univariate Mixture of Normals: MCMC and EM algorithms

by Hedibert Freitas Lopes
Applied Econometrics, Spring 2005
Graduate School of Business
University of Chicago

The observations $y_1, \ldots, y_n$ form a sample from the following finite mixture of normal distributions:

$$p(y_i|\theta) = \sum_{j=1}^{k} w_j p_N(y_i|\mu_j, \sigma_j^2)$$

where $\theta = (\mu, \sigma^2, w)$, $\mu = (\mu_1, \ldots, \mu_k)'$, $\sigma^2 = (\sigma_1^2, \ldots, \sigma_k^2)'$, $w = (w_1, \ldots, w_k)'$, and and $p_N(y|\mu, \sigma^2)$ is the density of a normal distribution with mean μ and variance σ^2 evaluated at y. Therefore,

$$p(y|\theta) = \prod_{i=1}^{n} \left[\sum_{j=1}^{k} w_j p_N(y_i|\mu_j, \sigma_j^2) \right]$$

Using latent indicators $z_1, \ldots, z_n$, such that $z_i \in \{1, \ldots, k\}$ and $p(z_i = j|\theta) = w_j$, the augmented model for (y, z) has the following joint density:

$$p(y, z|\theta) = p(y|z, \theta)p(z|\theta) = \left[\prod_{j=1}^{k} \prod_{i \in I_j} p_N(y_i|\mu_j, \sigma_j^2) \right] \prod_{i=1}^{n} p(z_i|\theta)$$

where $I_j = \{i : z_i = j\}$.

Maximum Likelihood Inference (EM)

The Expectation-Maximization (EM) algorithm finds $\hat{\theta}$ that maximizes the (incomplete) log-likelihood, ie.

$$\hat{\theta} \equiv \arg\max_{\theta} l(\theta|y)$$

where

$$l(\theta|y) = \sum_{i=1}^{n} \log \left[\sum_{j=1}^{k} w_j (2\pi\sigma_j^2)^{-1/2} \exp\left\{ \frac{1}{2\sigma_j^2}(y_i - \mu_j)^2 \right\} \right]$$

by iteratively cycling through the following two steps:

- **E-step:** Compute the integral $Q(\theta, \theta^{(l)}) = \int \log\{p(y, z|\theta)\} p(z|y, \theta^{(l)}) dz$
- **M-step:** Find $\theta^{(l+1)}$ such that $\theta^{(l+1)} = \arg\max_{\theta} Q(\theta, \theta^{(l)})$

The EM algorithm for the mixture of normal model case, with $\theta^{(0)}$ as starting value, cycles through $l = 1, \ldots, L$ as follows.

For $i = 1, \ldots, n$ and $j = 1, \ldots, k$ compute

$$\delta_{ij} = p(z_i = j|y_i, \theta^{(l)}) = \frac{w_j^{(l)} p_N(y_i|\mu_j^{(l)}, \sigma_j^{2(l)})}{p(y_i|\theta^{(l)})}$$

For $j = 1, \ldots, k$, compute

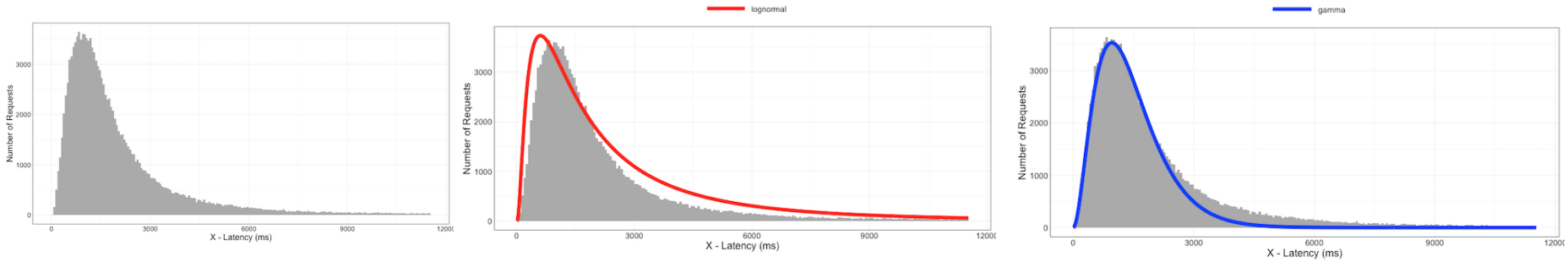
$$\begin{aligned} w_j^{(l+1)} &= n^{-1} \sum_{i=1}^{n} \delta_{ij} \\ \mu_j^{(l+1)} &= \frac{\sum_{i=1}^{n} y_i \delta_{ij}}{n w_j^{(l+1)}} \\ \sigma_j^{2(l+1)} &= \frac{\sum_{i=1}^{n} (y_i - \mu_j^{(l)})^2 \delta_{ij}}{n w_j^{(l+1)}} \end{aligned}$$

It can be shown that the sequence $\{\theta^{(1)}, \theta^{(2)}, \ldots\}$ converges to $\hat{\theta} = \arg\max_{\theta} l(\theta|y)$ as $l \to \infty$ (for more details about the EM algorithm, see Dempster, Laird and Rubin, 1977).

@adrianco@mastodon.social


What are the underlying distributions that are mixed together?
@adrianco@mastodon.social

Bill Kaiser of New Relic discussed this, and his data didn't really fit lognormal or gamma

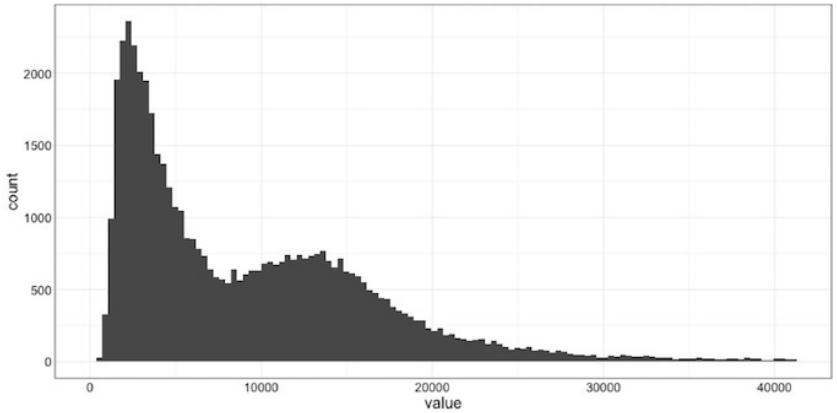


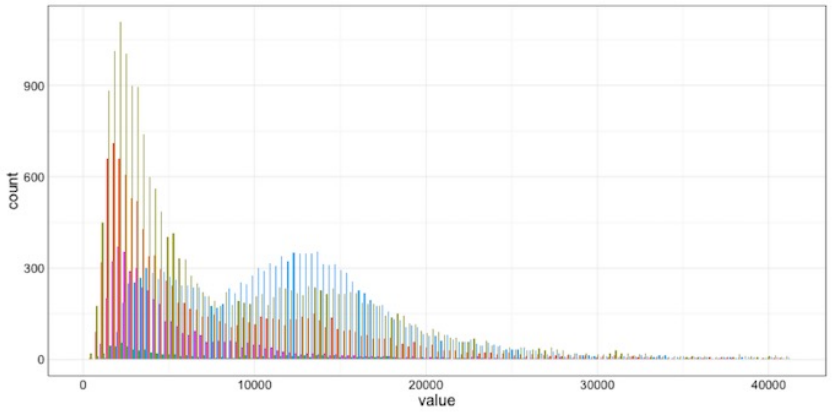
https://newrelic.com/blog/best-practices/expected-distributions-website-response-times

@adrianco@mastodon.social

He also showed a multi-modal distribution

(but didn't say how to figure out the modes)





https://newrelic.com/blog/best-practices/expected-distributions-website-response-times

@adrianco@mastodon.social

Time for even more coffee

What distribution should we expect?

Normal assumes variation adds or subtracts from the mean

Lognormal assumes variation multiplies or divides from the mean

Gamma assumes random arrivals queuing on a single path

@adrianco@mastodon.social

Lognormal makes sense to me…

If a CPU slows down for any reason, then all the requests that flow through it will be proportionally slowed down, which leads to a mixture of underlying lognormal distributions

@adrianco@mastodon.social

This is the key “new” idea I’m proposing

Taking the logarithm of response times transforms to a mixture of normal distributions

I can make any distribution shape by mixing together normals!

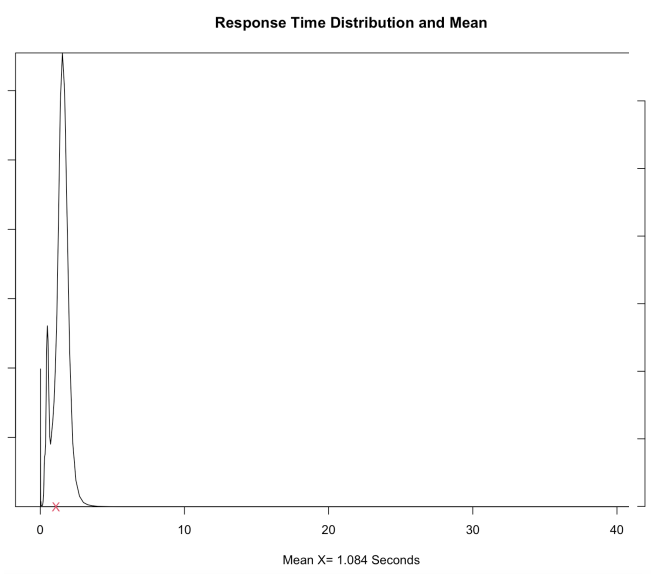
The math is far easier for normal distributions…

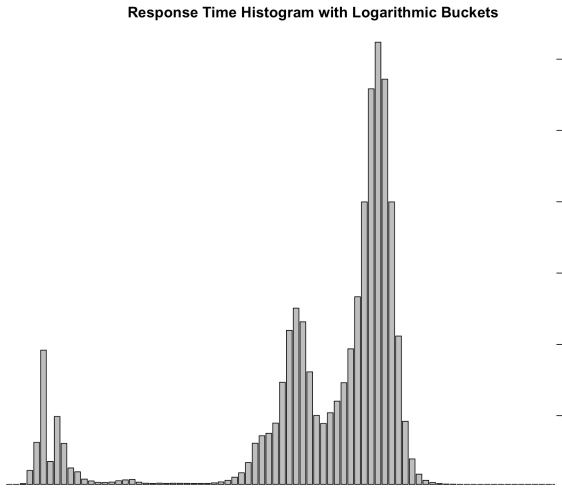
There are no new ideas, just ideas someone else figured out that you hadn’t heard of yet…

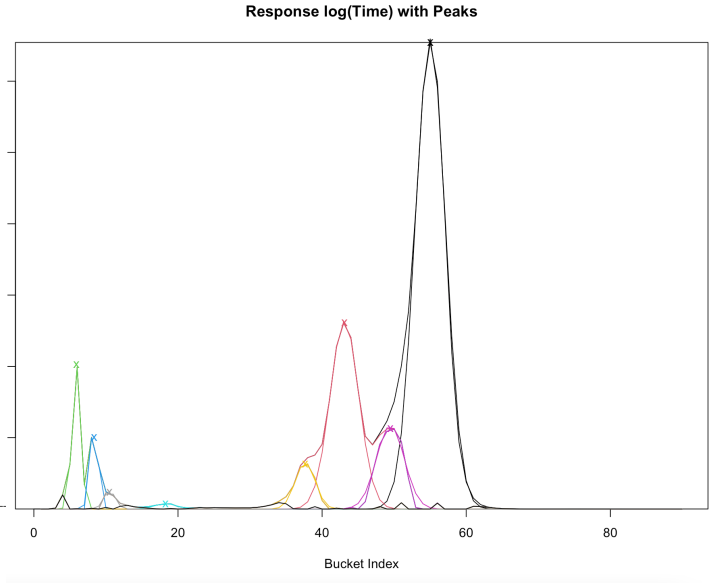
@adrianco@mastodon.social

What does real data look like?

This data was collected using hdrhistogram (high dynamic range histogram) where the bins increase logarithmically, and normal peaks were subtracted out to leave residual normal peaks







Turns out this data (from a web frontend) does seem to fit the “lognormal” pattern

@adrianco@mastodon.social

Problems…

The statistical tools expect to process all the underlying data
But we may have millions of data points summarized in a histogram…

The statistical tools like to have an initial guess of the peaks
But we only need a rough guess of the peaks anyway…

There is no algorithm to get an exact answer
So we will ignore all the Greek and brute force a solution!

@adrianco@mastodon.social

Input data…

Start with a histogram, with exponentially increasing buckets

or a log file containing timestamp, latency, and a query tag per line
then use hist(log(latency)) to make a histogram for a chunk of time

Code in R that takes a histogram and returns peaks: as.peaks()

```
# Written by Adrian Cockcroft (@adrianco@mastodon.social) - 2023 - Apache 2.0 License
# Thanks to Donnie Berkholz (@dberkholz@hostux.social) and Ed Borasky (@AlgoCompSynth@ravenation.club) for guidance
#
# Take a histogram and find the peaks in it, interpolating between bins
# approximate finite mixed model decomposition starting with histograms of data
# Intended for use on response time histograms with logarithmic bins like hdrhistogram
# so that each (approximately) lognormal component is interpreted as a symmetric normal peak

# h - is either a data frame from a csv containing Values and Counts or an R histogram object
# where the Value/break for each bucket is assumed to be exponential - i.e. hist(log(values))
# time is an optional timestamp that is added as a column for this set of peaks
# plots - optionally shown
# normalize - divides down the counts to a probability density
# peakcount=0 does a quick estimate of all the peaks, otherwise it does up to the specified number of dnorm fits
# epsilon=0.005 - sets a minimum interesting peak size to 0.5% of the largest peak, the ones that are visible in the plot.
# the algorithm is quite sensitive to epsilon so try tinkering with it
# printdebug turns on helpful intermediate print output to see what's going on

# Returns a data frame with one row per peak, sorted biggest peak first
# row names show the order the peaks were discovered in
# PeakDensity is the histogram count or normalized density
# PeakBucket is the center bucket for the peak
# PeakMin and PeakMax are the start and end buckets for the peak
# if a gaussian fit was obtained, then PeakAmplitude is non-zero, along with updated PeakMean and PeakSD
# The Value of the peak is interpolated to estimate PeakLatency

as.peaks <- function(h, time=0, plots=FALSE, normalize=FALSE, epsilon=0.005, peakcount=0, printdebug=F)
```

@adrianco@mastodon.social

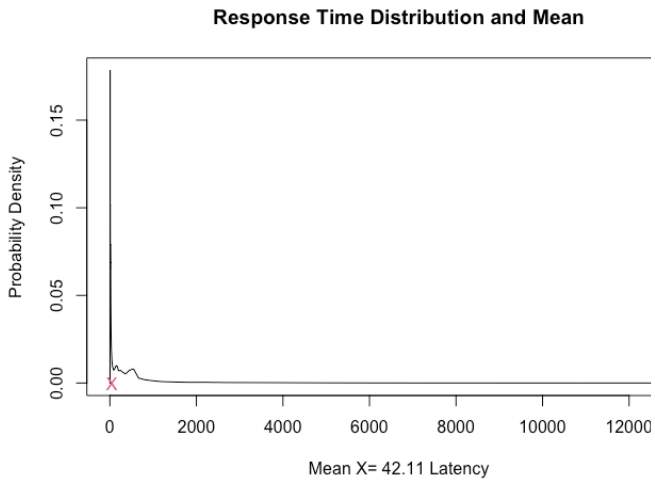
See the code here: https://github.com/adrianco/responsetime-distribution-analysis
as.peaks.R is 120 or so lines of commented R code… This is what it produces:

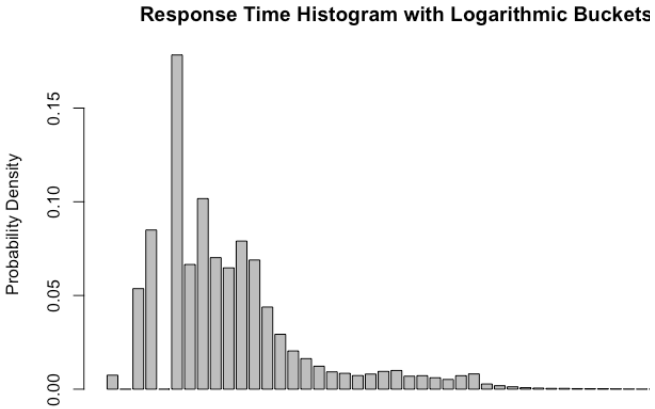
```
Source data is a logfile containing latency for requests in milliseconds, the first 200,000 values are
transformed via logarithm, a histogram is made with 40 breaks, and as.peaks is called looking for 12
peaks, generating plots, and normalizing the data so that the counts are normalized into PeakDensity.

PeakBucket is the histogram bucket that has a peak, PeakMin and PeakMax are the width of that peak
PeakMean is the center of the peak, within the bucket, PeakSD is the standard deviation if a normal
fit was attempted, PeakAmplitude is reported if a normal could be fitted. PeakLatency is the actual
latency of that peak in milliseconds. Row number shows what order they were found in, biggest first.

> as.peaks(hist(log(cr1g.trt$latency[1:200000]),breaks=40),peakcount=12,plots=T,normalize=T)
   PeakDensity PeakBucket PeakMin PeakMax  PeakMean    PeakSD PeakAmplitude PeakLatency
7  0.007474812          1       0       2  1.000000 0.0000000    0.00000000    2.013753
3  0.084946639          4       2       5  3.612864 0.4870953    0.00000000    3.395913
1  0.178329678          6       5       7  6.271748 0.4448606    0.00000000    5.779687
9  0.066543823          7       6       8  7.000000 0.0000000    0.00000000    6.685894
2  0.101696280          8       7      10  8.439103 1.0546903    0.00000000    8.915760
10 0.064739068         10       9      11 10.000000 0.0000000    0.00000000   12.182494
4  0.079049212         11      10      20 11.358128 1.0374243    0.00000000   15.984597
11 0.043839912         13      12      20 13.749582 0.7900827    0.00000000   25.788185
12 0.007977967         21      20      22 20.523371 0.4994535    0.00000000   99.950422
5  0.009994837         23      20      24 22.457629 1.0574215    0.00000000  147.160803
8  0.007200072         25      24      27 24.574692 1.2056374    0.02295962  224.738724
6  0.008137976         29      27      35 28.347701 1.3427718    0.02729880  477.966245
```

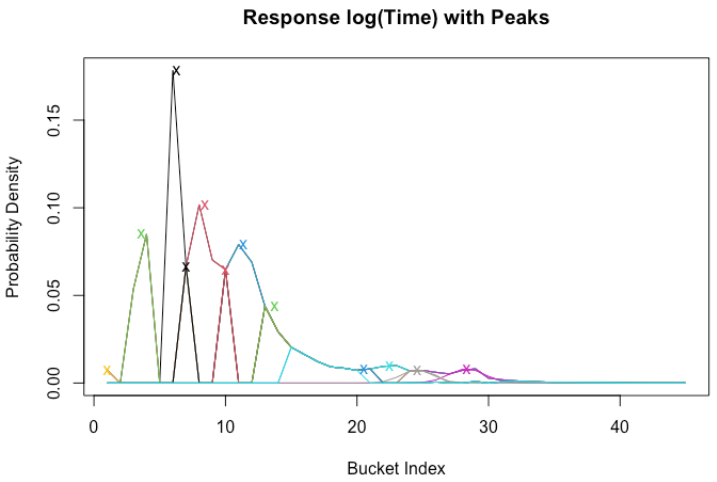
@adrianco@mastodon.social

Microservice calling patterns

This shows how overlapping peaks are extracted from data that was rounded to the nearest millisecond
The algorithm can't fit a normal distribution in some cases so it uses a triangle approximation







@adrianco@mastodon.social

How do the peaks change over time?

You may be collecting histograms via Prometheus, or have a big logfile that you can chop into chunks

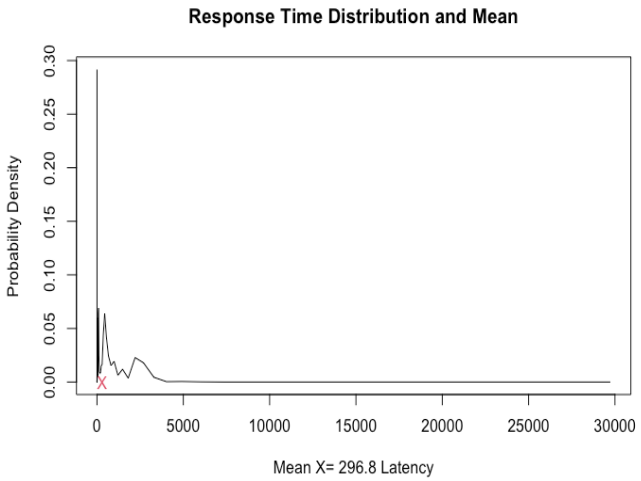
as.peaks is a compact way to process each chunk

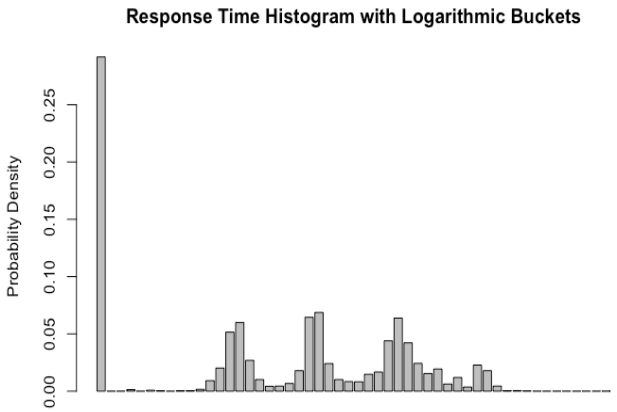
We should see peaks form clusters over time…

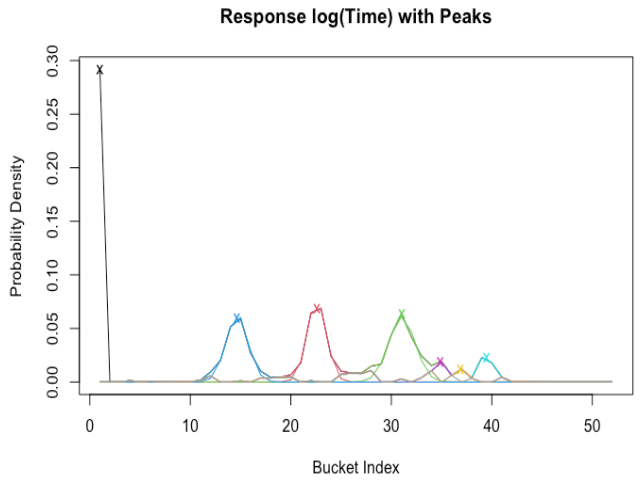
@adrianco@mastodon.social

API frontend calling patterns

This shows calling patterns into a GraphQL API front end with response time data that was rounded to the nearest millisecond. There's a lot of ~1ms responses from a config lookup call…



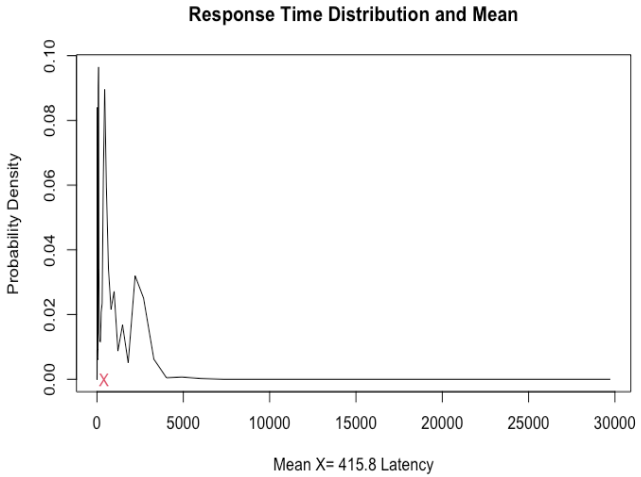


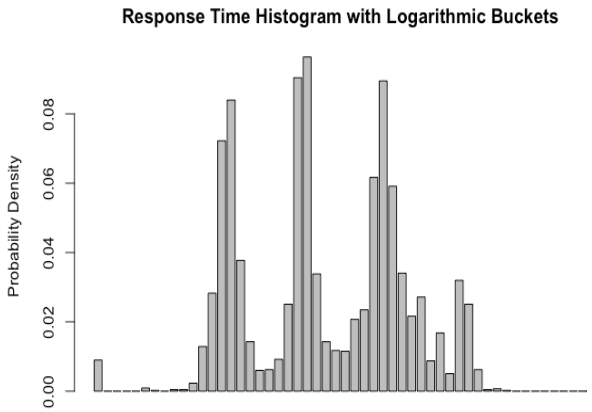


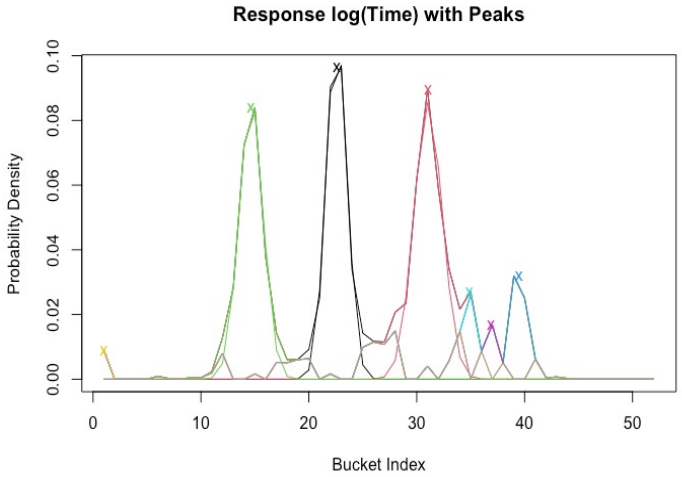
@adrianco@mastodon.social

API frontend calling patterns

If we exclude the config lookup call it looks more interesting…



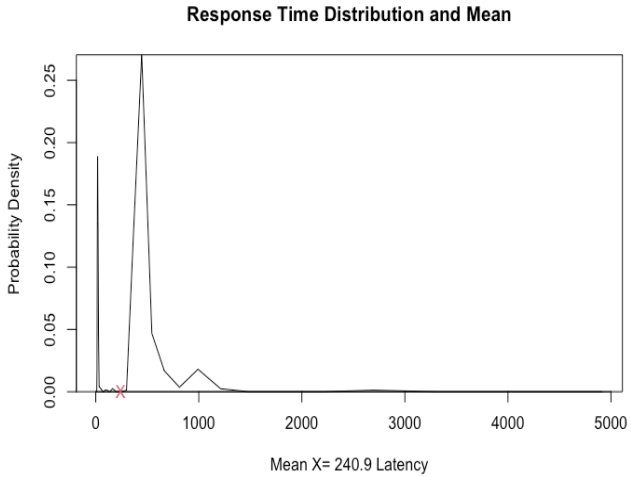


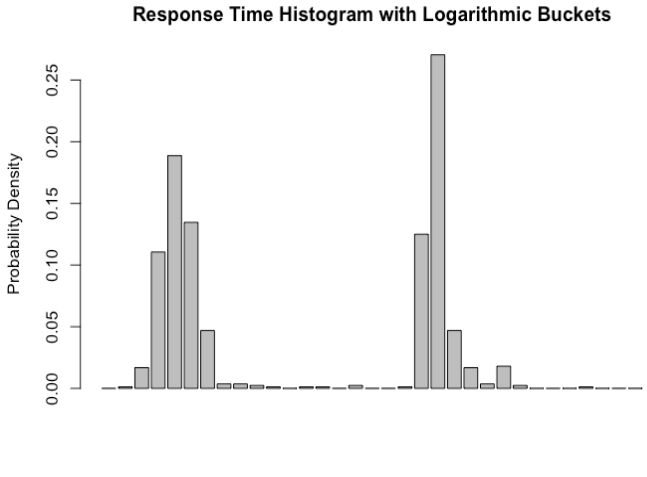


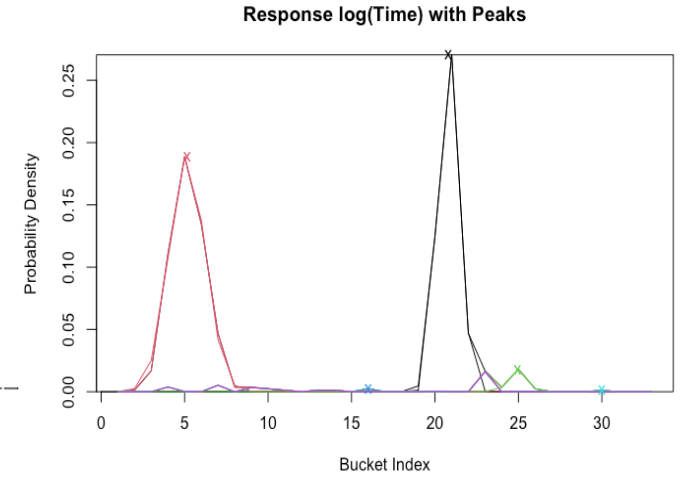
@adrianco@mastodon.social

API frontend calling patterns

Drilling in, these plots show the data for the single most common business query type…

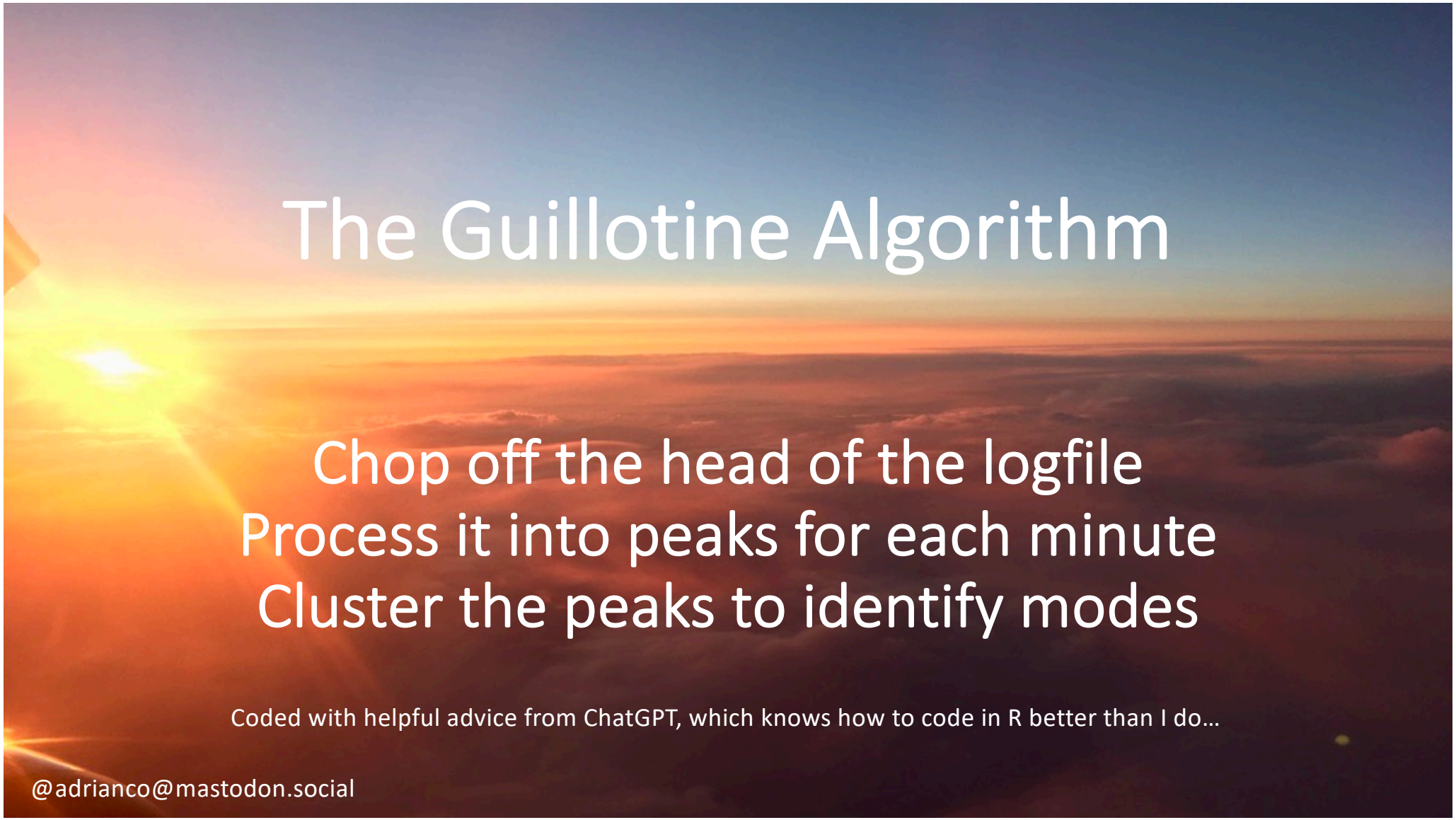






Min.: 8.0 Median: 40.0 Mean: 240.9 95%: 611.0. 99%: 1047.0 99.9%: 1489.625 Max.:4658.0

@adrianco@mastodon.social


The Guillotine Algorithm
Chop off the head of the logfile
Process it into peaks for each minute
Cluster the peaks to identify modes
Coded with helpful advice from ChatGPT, which knows how to code in R better than I do…
@adrianco@mastodon.social

The two main peaks measured over time

	PeakDensity	PeakBucket	PeakMin	PeakMax	PeakMean	PeakSD	PeakAmplitude	**PeakLatency**	Time
2	0.1814214	5	1	8	4.894839	1.0395188	0.5035725	17.79589	2023-06-23 23:10:36
21	0.2157005	5	1	8	4.945127	0.9147616	0.4939367	17.97578	2023-06-23 23:11:00
22	0.2145231	5	2	8	4.969180	0.9340081	0.5040957	18.06246	2023-06-23 23:12:00
23	0.2123629	5	1	12	4.896886	0.9163511	0.4947977	17.80318	2023-06-23 23:13:00
24	0.1977819	5	1	8	4.873739	0.9816810	0.5051812	17.72096	2023-06-23 23:14:00

…

	PeakDensity	PeakBucket	PeakMin	PeakMax	PeakMean	PeakSD	PeakAmplitude	**PeakLatency**	Time
1	0.2300499	21	18	27	20.84743	0.7202185	0.4249293	432.4581	2023-06-23 23:10:36
2	0.2541063	21	19	24	20.79214	0.6671278	0.4468025	427.7029	2023-06-23 23:11:00
3	0.2408591	21	19	24	20.75011	0.6747557	0.4377622	424.1224	2023-06-23 23:12:00
4	0.2442672	21	19	24	20.88990	0.7107443	0.4399665	436.1469	2023-06-23 23:13:00
5	0.2429364	21	19	24	20.86698	0.7005555	0.4344144	434.1524	2023-06-23 23:14:00

…

The two main modes are about 18ms and about 430ms

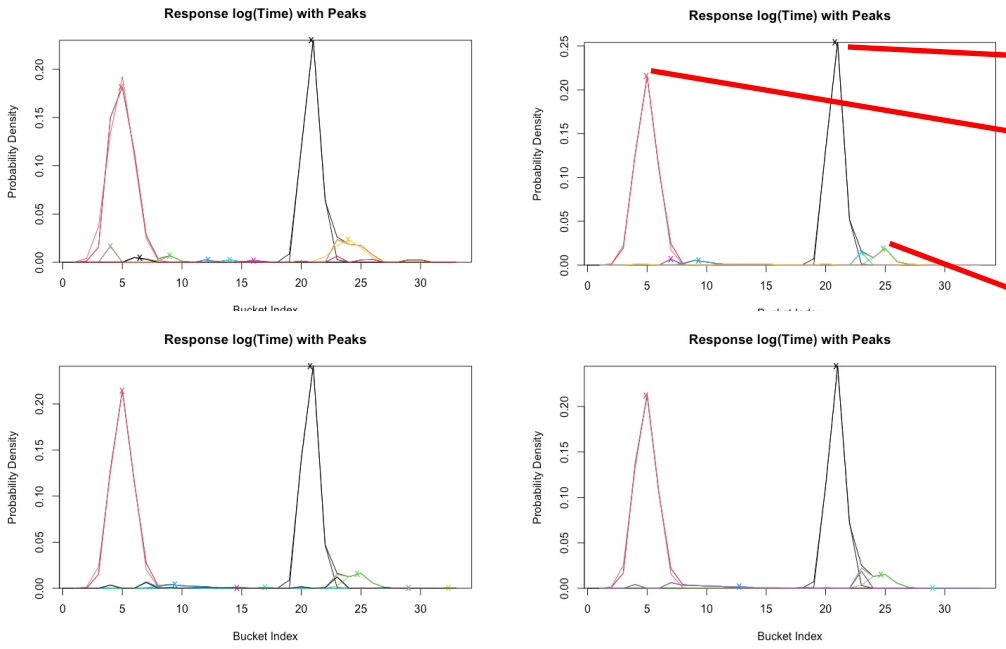
As the ratio of these modes varies the “usual metrics” below will move about…

Min.: 8.0 Median: 40.0 Mean: 240.9 95%: 611.0. 99%: 1047.0 99.9%: 1489.625 Max.:4658.0

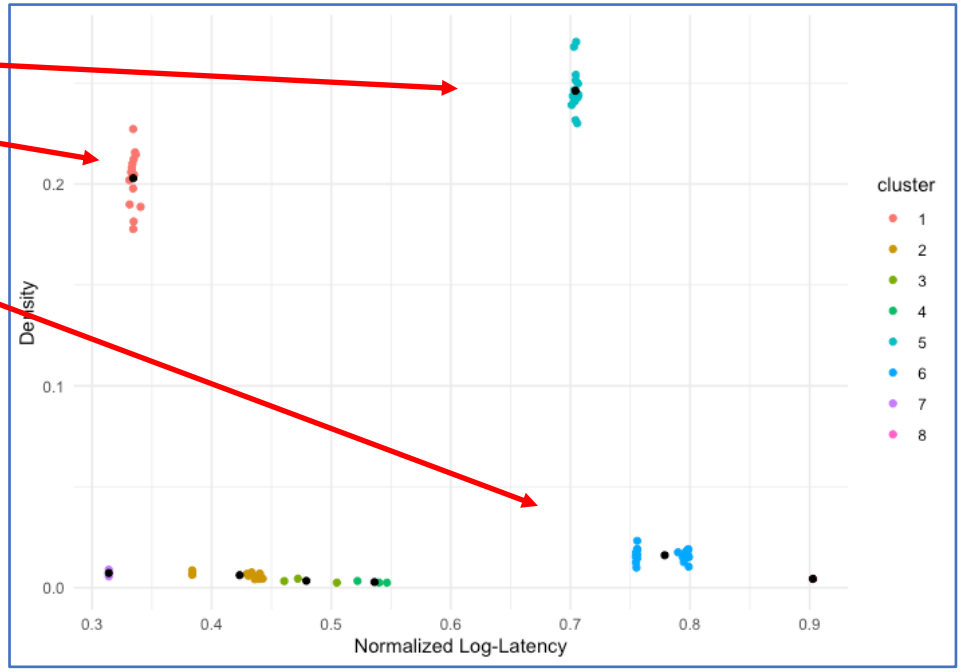
@adrianco@mastodon.social

API frontend calling patterns

A 15 minute log file is chopped into one minute sections, the peaks are obtained and clustered, and the positions of the peaks are plotted. The important ones are stable in response time, vary in density

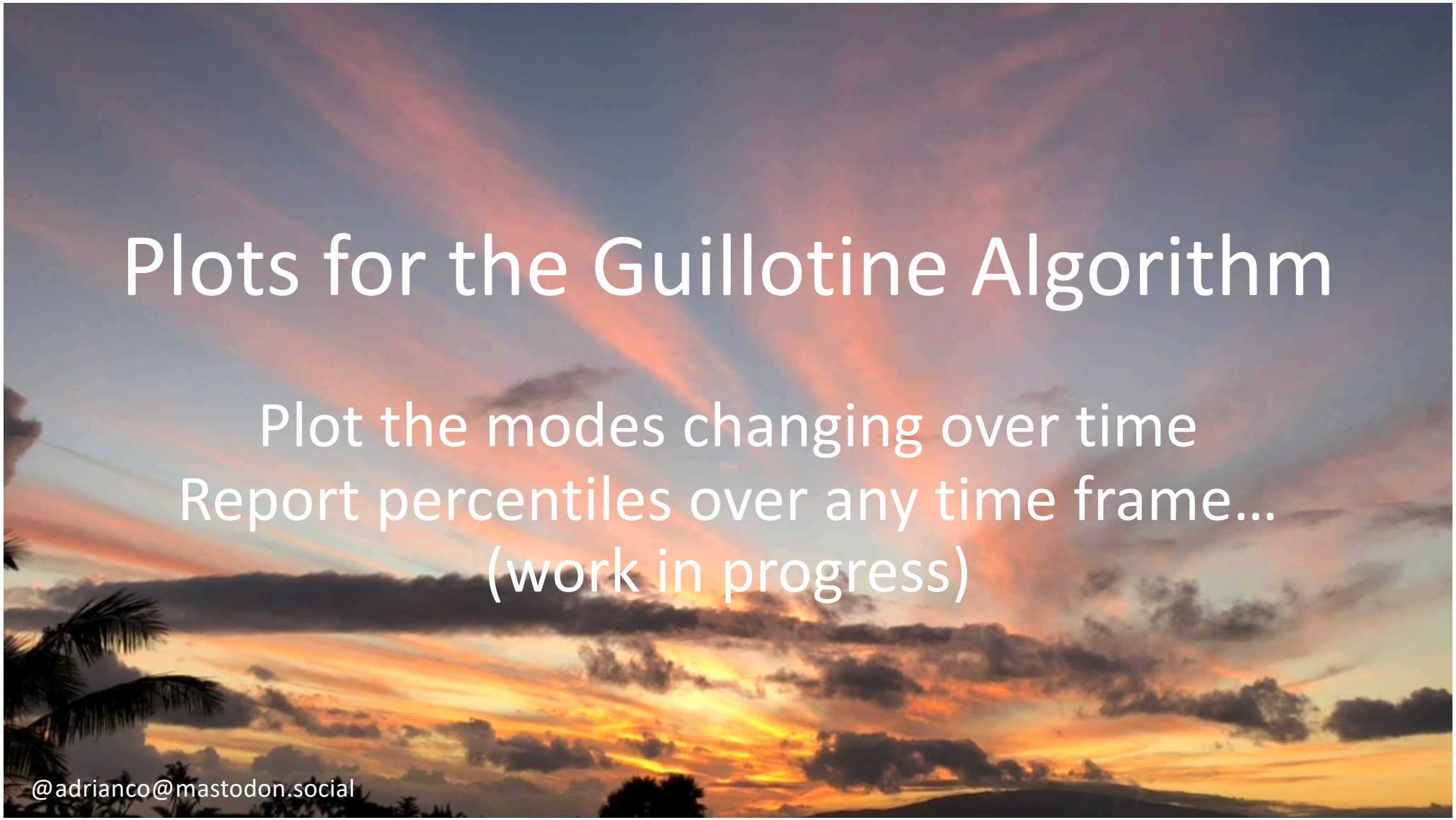


These are just the first four out of 15 peak fits



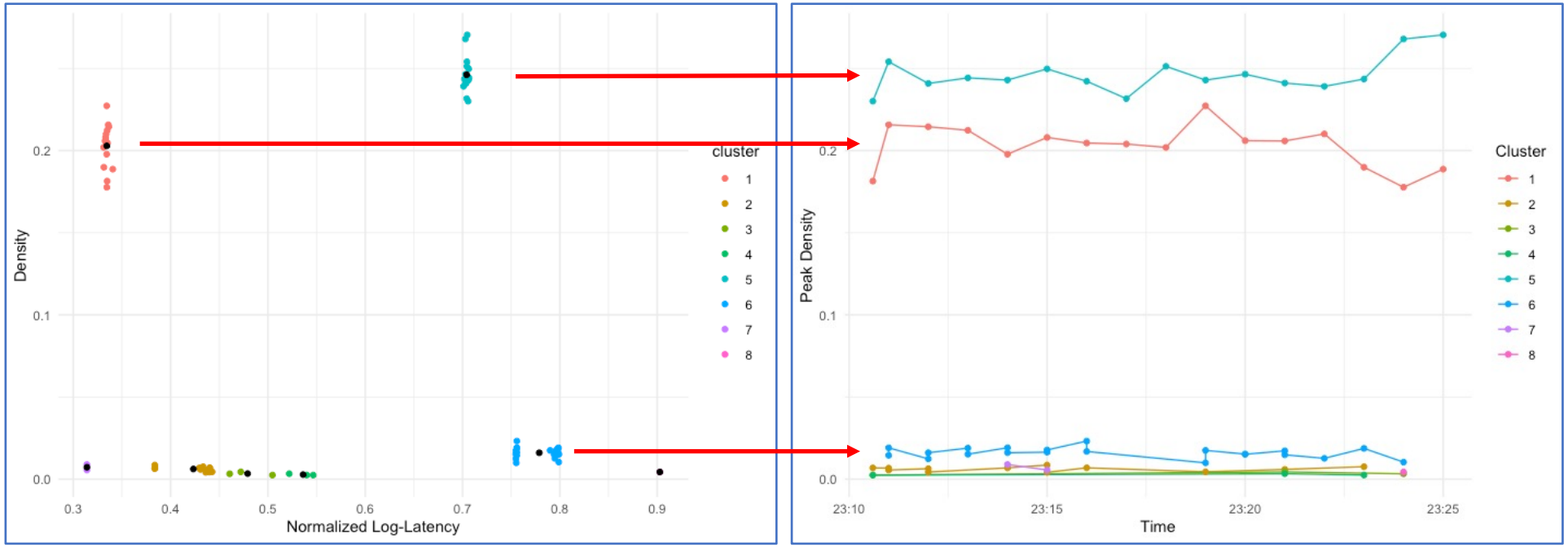
For clustering the latency values need to be normalized
Black dot is the centroid for each cluster

@adrianco@mastodon.social


Plots for the Guillotine Algorithm
Plot the modes changing over time
Report percentiles over any time frame…
(work in progress)
@adrianco@mastodon.social

API frontend calling patterns

When peaks are plotted over time, the relative probability (Peak Density) of each peak varies



For the third cluster, two nearby peaks have been clustered together

@adrianco@mastodon.social

Learnings

Histograms of response times contain interesting structure
Scale and process response times logarithmically
Peaks indicate interesting modes
Mode latencies are stable over time

When a system “slows down” sometimes all that is happening is that the proportions of modes is changing…

Source code for all of this is Apache 2.0 licensed at https://github.com/adrianco/responsetime-distribution-analysis

@adrianco@mastodon.social

Next Steps

If anyone wants to help me develop this further or implement this in their own tools I'm happy to help

If you want it in a different language than R, try asking ChatGPT to translate it for you…

Source code for all of this is Apache 2.0 licensed at https://github.com/adrianco/responsetime-distribution-analysis

@adrianco@mastodon.social

A Tale of Two Histograms

"It is a far, far better thing that I do, than I have ever done; it is a far, far better REST API that I go to than I have ever known."

Adrian Cockcroft

OrionX.net - @adrianco@mastodon.social

With apologies to Charles Dickens and my high school English Literature teacher