How to trade off server utilization and tail latency

SRECon 2019, Singapore Julius Plenz <plenz@google.com>

https://www.usenix.org/conference/srecon19asia/presentation/plenz

Abstract:
When running large scale systems, we strive to deliver both low tail latency and high utilization of servers. However, these two dimenions are at odds: increasing the average utilization of a system will have a detrimental impact on the tail latency. This talk provides a light-weight walkthrough of the important basics of queueing theory (avoiding unnecessary formalism), illustrates graphically several typical outcomes of this analysis, and closes with a few basic rules on how to think about utilization and tail latency.

Who am I?

- SRE at Google Sydney
- I currently work on Google Cloud Datastore and Firestore
- Worked on other Google-internal storage systems before
- I try to make visualizations to understand things!

- If you find any of this interesting, come talk to me afterwards
- Slide deck is published on the conference web site including speaker notes with additional information

Disclaimer

Queuing Theory is its own subfield in Computer Science / Statistics.

I'll gloss over many subtleties and won't use any formulas.

Instead, we'll look at graphs of numerical simulations only.

If you want a good book with a rigorous yet practical approach to queueing theory, I recommend Mor Harchol-Balter's book "**Performance Modeling and Design of Computer Systems: Queueing Theory in Action**" (http://www.cs.cmu.edu/~harchol/PerformanceModeling/book.html).

Queues are everywhere, let's talk about them!

Image: Google Datacenter Gallery

In-flight request

Queued requests

Server

Queues are everywhere:

- Web server
- Network card
- Hard drive
- Memcache
- Database
- Restaurants
- etc etc

Let's talk about queues. Queues are everywhere: in a webserver, network card, hard drive, a memcache server, a database server, etc.

Here is a typical example of a database that exposes a REST API. Requests queue up and are handled in FIFO order.

[Image derived from: https://www.google.com/about/datacenters/gallery/, in particular: https://www.google.com/about/datacenters/gallery/images/_2000/PRY_20.jpg]

Definitions for this talk

Server: A binary that handles requests.

Example: web server, database server, memcache server

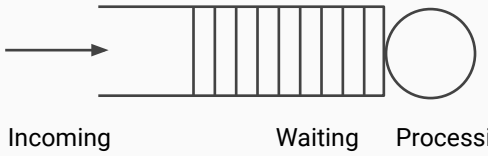
Utilization: Fraction of time that the server is busy in the bottleneck dimension, usually CPU.

Example: Over a 60s interval, the CPU is busy 30s. The utilization is 50%.

Tail latency: 99%ile latency of a set of requests.

Example: Out of 100 requests, the second slowest exhibits the tail latency.

The simplest queue



Queue time

Time spent waiting in the queue

+

Service time

Time spent servicing the request (no waiting)

=

Request latency

User-perceived time: how long it takes from sending a request to receiving the reply

In the simplest case, we have a single processor with a single queue. In other words only one request can be processed at a time, and there are no different request priorities. It's usually assumed that the queue is processed in FIFO order, and that's what we'll assume in the remainder of the talk.

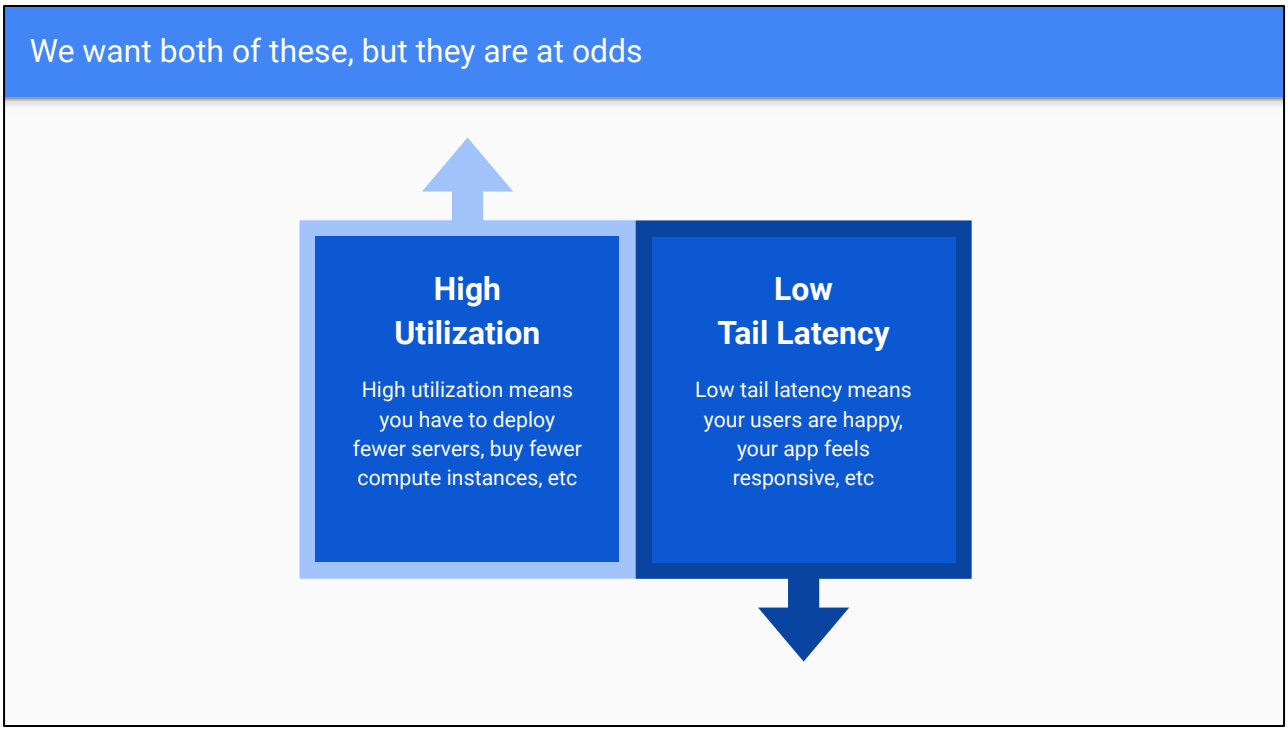
[Read the "equation": Queue time plus Service time equals Request latency]

Queue time is the time spent waiting in the queue. It generally increases when the rate of requests per server goes up.

Service time is the time it actually takes to process the request (e.g. parse the request, retrieve data from memory, assemble result, serialize response). For simplicity we can assume this doesn't change when the rate of requests to the server is increased.

The sum of these makes up the total request latency.

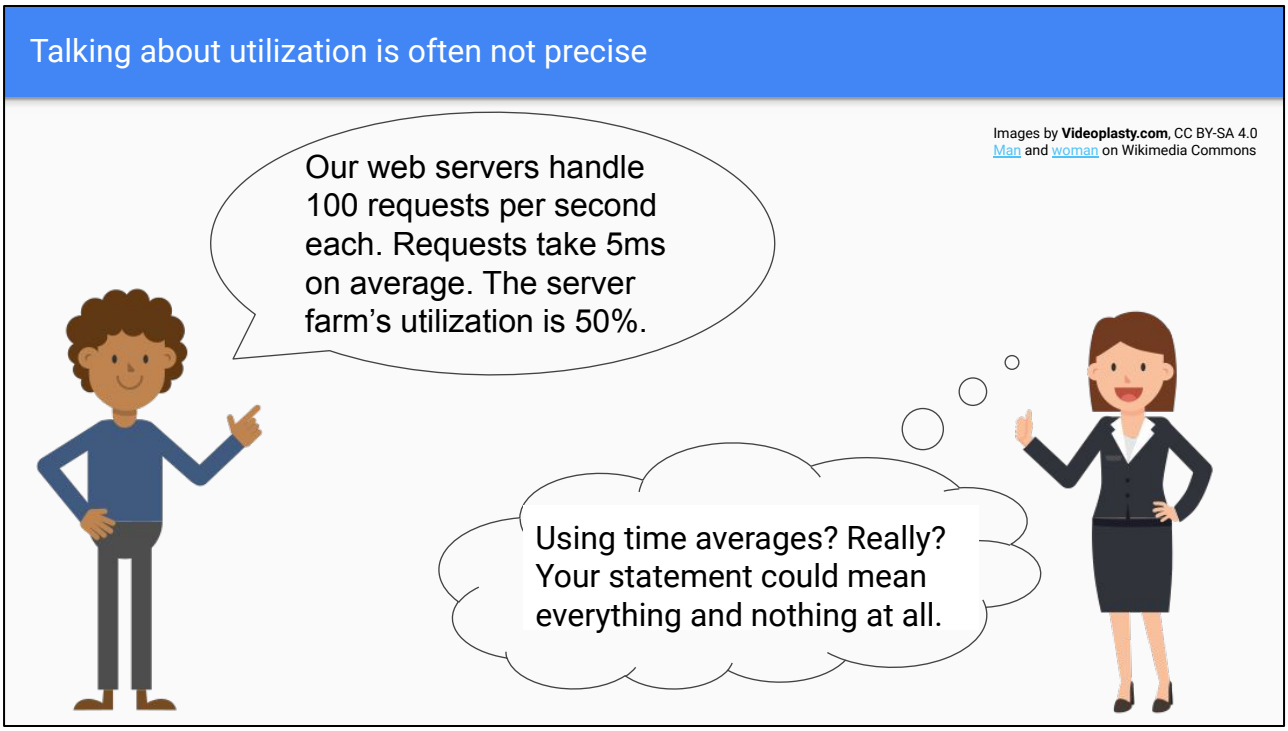
We can control the queue time by controlling how many items are in the queue at any time; we can control the service time by e.g. improving the performance of the binary or using faster CPUs.



We want both of these: High Utilization and Low Tail Latency.

- For high utilization, the queue needs to **have at least one request** at all times, so that once a request finishes the server can directly process the next request in the queue (no waiting time for the processor).
- For low tail latency, the queue needs to be **empty**, so that when a request arrives, it'll find the queue empty, and the server can start processing right away (no waiting time for the request).

Obviously, these two are fundamentally at odds.

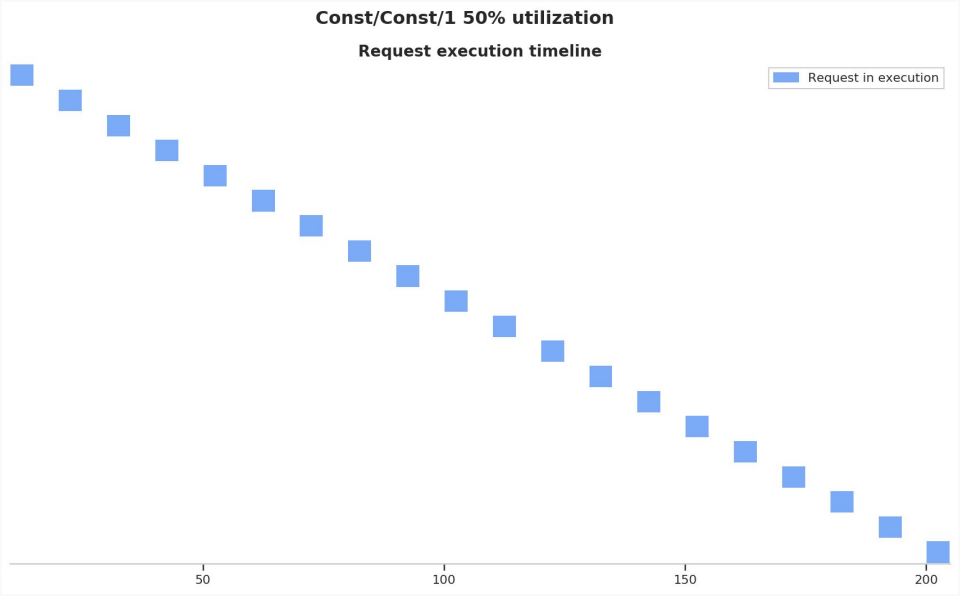


Here's a common problem: When we talk about utilization and request patterns, we're often not very precise, and instead summarize a system using time averages.

Why is the statement not precise? Because it only gives the **average** time between requests (10ms), and only the **average** serving time (5ms); the result is an **average** utilization of 50%.

Averages tell us almost nothing about the **tail** of requests.

This picture of “50% utilization” is very likely misleading



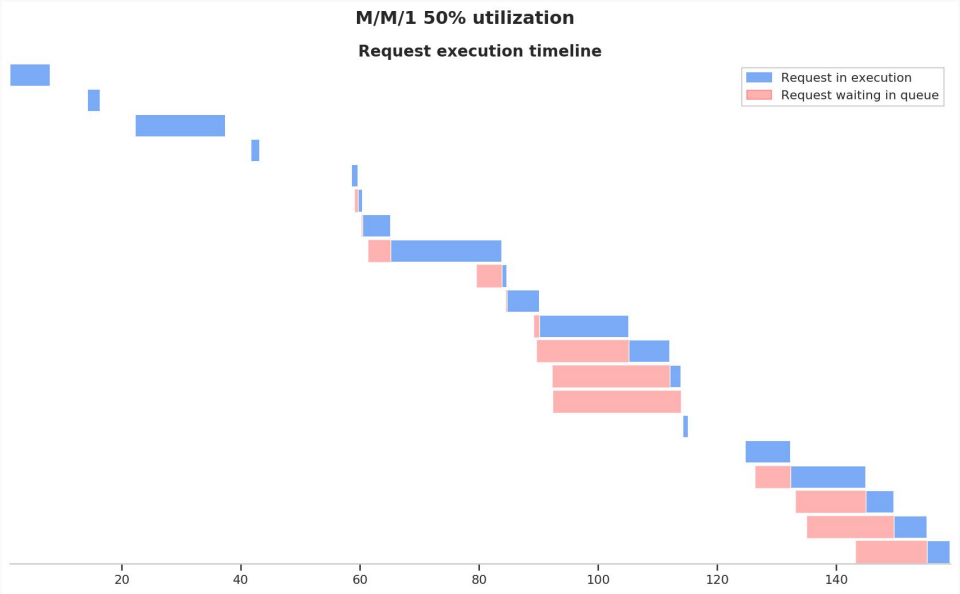
Each row represents a request, and the blue bar represents its processing time. X axis is time in milliseconds. The length of each bar is 5ms, i.e. we assume **constant** service times.

Additionally, the start time of each request is exactly 10ms apart, i.e. we assume **constant** interarrival times.

Both of these are likely not true in practice. The **queue time** in this example is always zero.

[If you wonder about the title of the graph: This is the Kendall notation, it specifies the distribution of interarrival time, distribution of serving time, and number of processors.]

This image is much closer to reality: Variable interarrival and service times



In this graph, requests are made up of a red part (queue wait time) and a blue part (processing time).

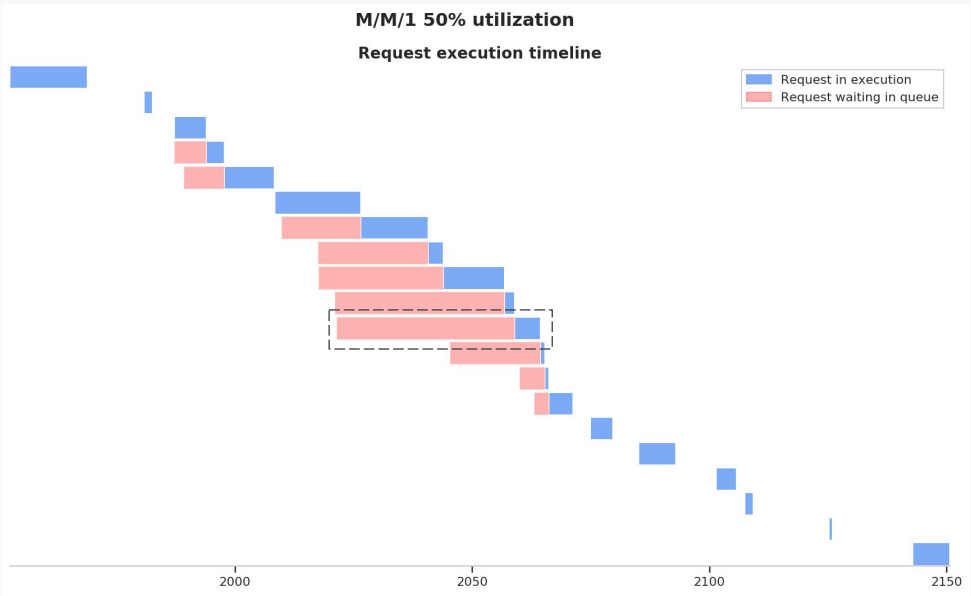
This is a more likely scenario: Some requests take longer to serve, others can be finished quickly. There is variance in the “difficulty” of requests. (E.g. think about built-in caching layers, and how response sizes can vary a lot, e.g. sending 1KiB is far cheaper than sending 1MiB.)

It’s also clear that the requests arrive in an uncoordinated (= random) manner: sometimes two arrive roughly at the same time, and then one has to wait for the (remaining) service time of the prior request. [NB: If you’ve ever heard the term “Poisson arrival process”, this is it.]

We now see where we get “queue time” from. We have nonzero queue time whenever a request arrives and finds the server already busy with another request, and/or other requests already waiting in the queue.

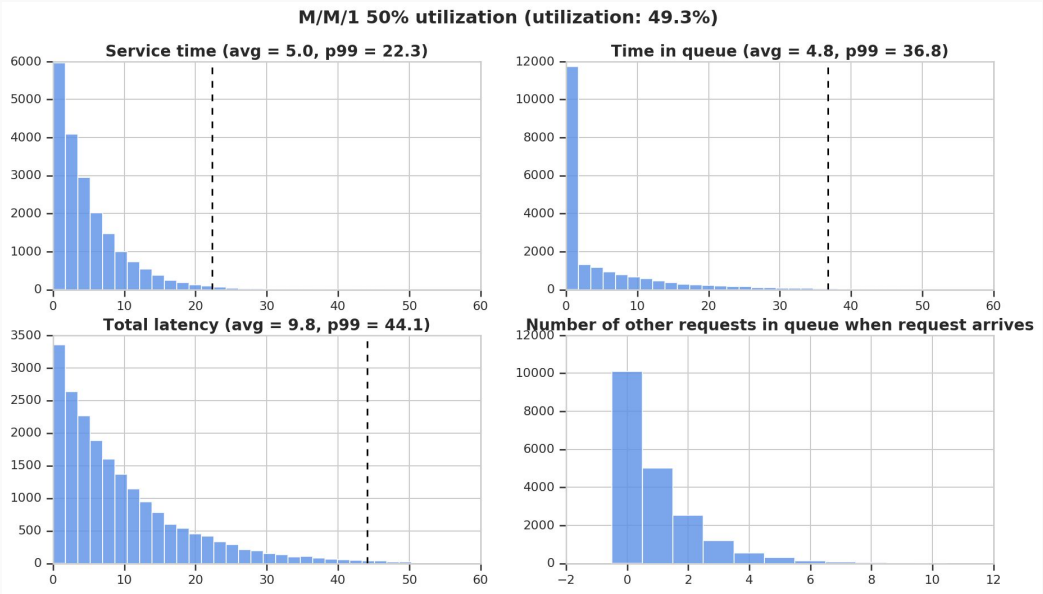
[The M/M/1 graph title refers to the the Kendall notation of this queue: Interarrival time and Service time are both **M**emoryless, and the only continuous memoryless distribution is the Exponential distribution. I.e. we see here samples drawn from an exponential distribution with mean $\mu = 5$ms for the service time, and the interarrival time is sampled from an exponential distribution with mean $\lambda = 10$ms.]

This is a snapshot of a point in time when a request exhibited the 99%ile latency



This is a snapshot from later in the sample process where a request exhibits the 99%ile request latency (middle). Note how at this point in time the requests are coming in faster than they can be processed: this leads to a backlog which causes the request latency to be **dominated** by the queue time.

We can relate the distributions of queue and service time, and total request latency



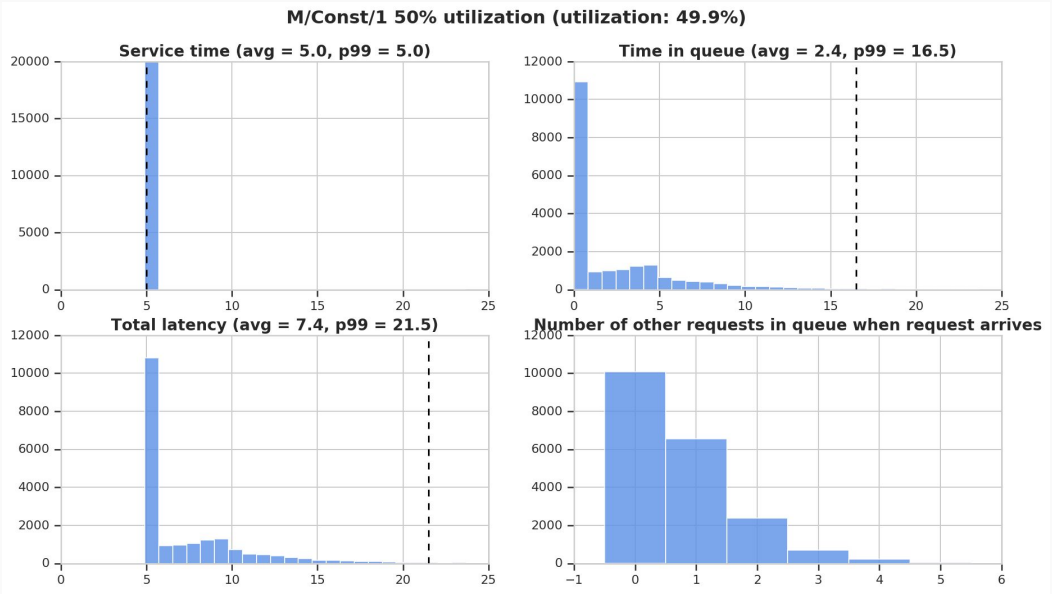
Instead of looking at individual request timelines, we can summarize the relationship of service time and queue time by drawing histograms of the values.

Note how the average service time is 5ms, but has a p99 of 22.3ms (the dotted line). Half of the time we find the queue completely empty (lower right graph; because the server is running at 50% utilization), so the time in queue is negligible more than half of the time. However, at the tail we see that a request can wait 36ms before processing begins!

By comparing the left two graphs, we can see what effect the queue time has on the total request latency: it draws out the distribution to the right. Note though that the total latency p99 is strictly smaller than the sum of service time p99 and queue time p99: It's very unlikely that a slow-to-process request is *also* the request that sees a particularly long queue time.

[Number of samples for the simulation: 20k. NB, Little's Law states that the average number of requests in the system is 1/50% = 2, that's why the average total latency is 2*average service time = 10ms.]

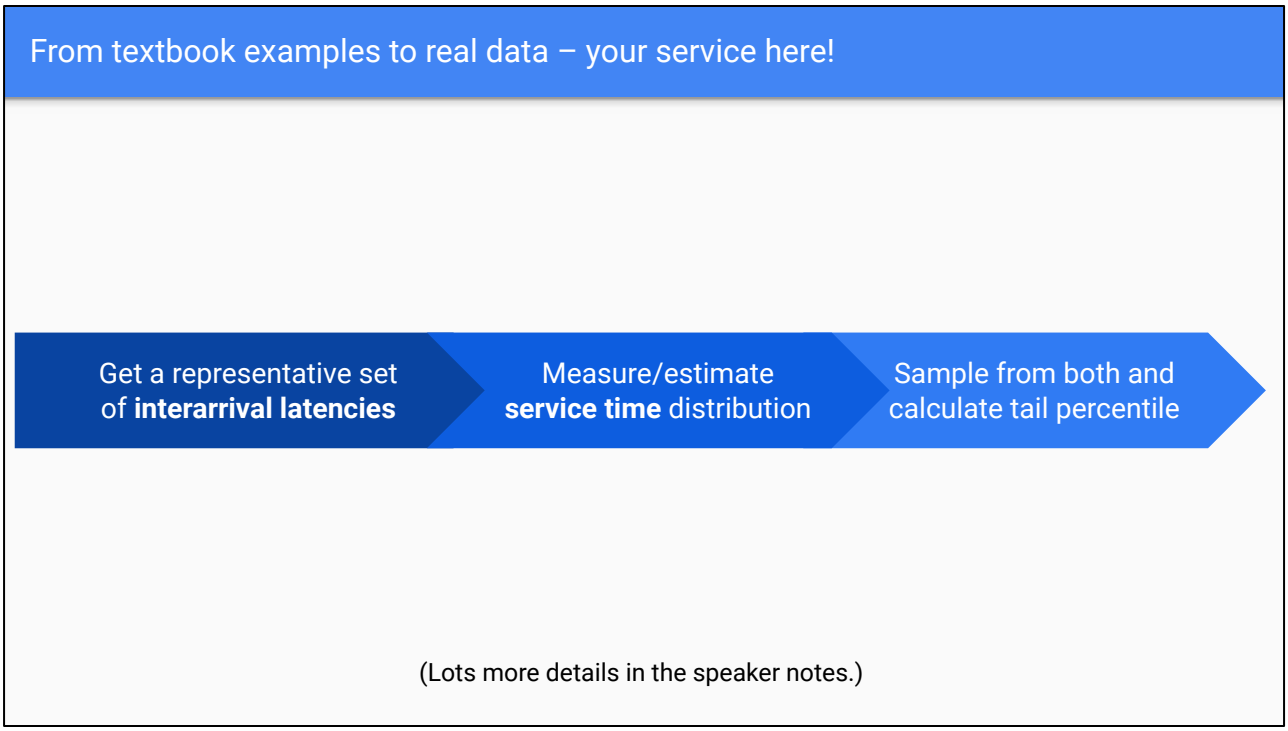
Similar example: With constant service time, queueing effects actually dominate



… you might say that it's unreasonable that the service time varies so wildly, and you might be right. It depends on the server. In the best case, you'd have **constant** service time, i.e. each requests takes as long as any other.

If we assume that the service time is constant 5ms, and just assume independent arrival for a 50% utilization, we see that the tail latency is now **dominated** by queueing effects: The tail latency is about four times as long as a single request takes. That's because there is a 1-in-a-hundred chance that a request will find 3 or more requests in the queue when it arrives, and it has to wait its turn.

Takeaway #1:
A significant amount of tail latency is due to queueing effects



Get a representative set of interarrival latencies:

- Log the timestamp of when a request arrives. If this is expensive, do for a sample of requests only.
  - NB: Logging the arrival timestamp is done after de-queueing a request. If you have a sophisticated tracing framework, log the timestamp of the *client* that is talking to the server. If logging the timestamp is expensive, do this only for a subset of requests.
- Order the requests by arrival time, and compute the delta to the previous request.

Measure/estimate service time distribution

- Measuring the service time distribution is hard. E.g. you'd need very precise per-thread counters. Make sure you don't include latency or wait time from the backends.
- An easy way is to make up the service time distribution: assume a (truncated) Normal or Log-normal distribution around the time-average mean.

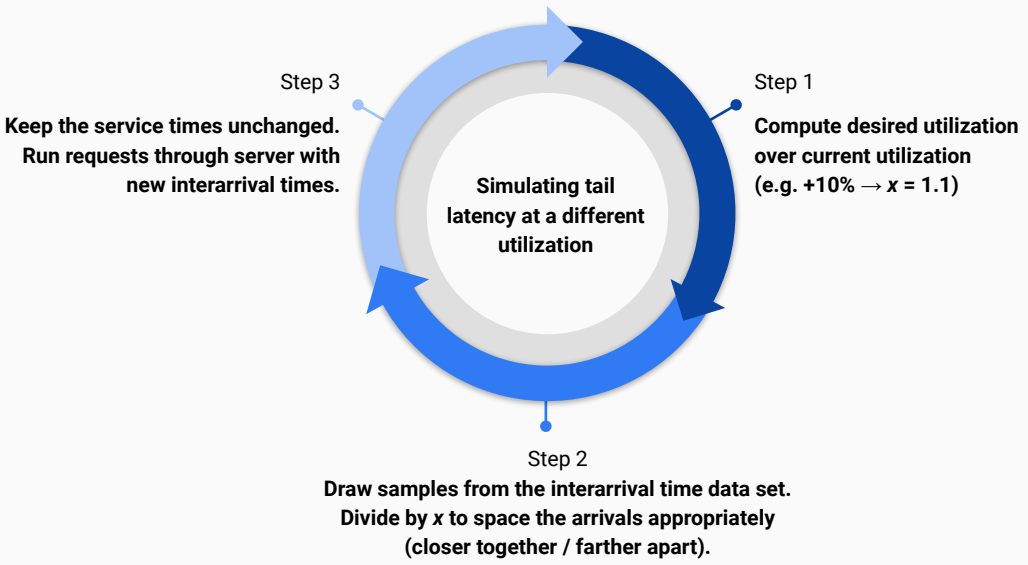
Sample from both and calculate tail percentile

- Draw *N* sample requests of the form *(interarrival, service)* from the data sets.

- E.g. use numpy.random.choice() to sample from an array of interarrival times. Use the scipy.stats module to sample from well-known distributions.
- Run them through a simplified server implementation. Calculate the tail request latency after all requests have been processed. Colab that produced the diagrams for this talk.

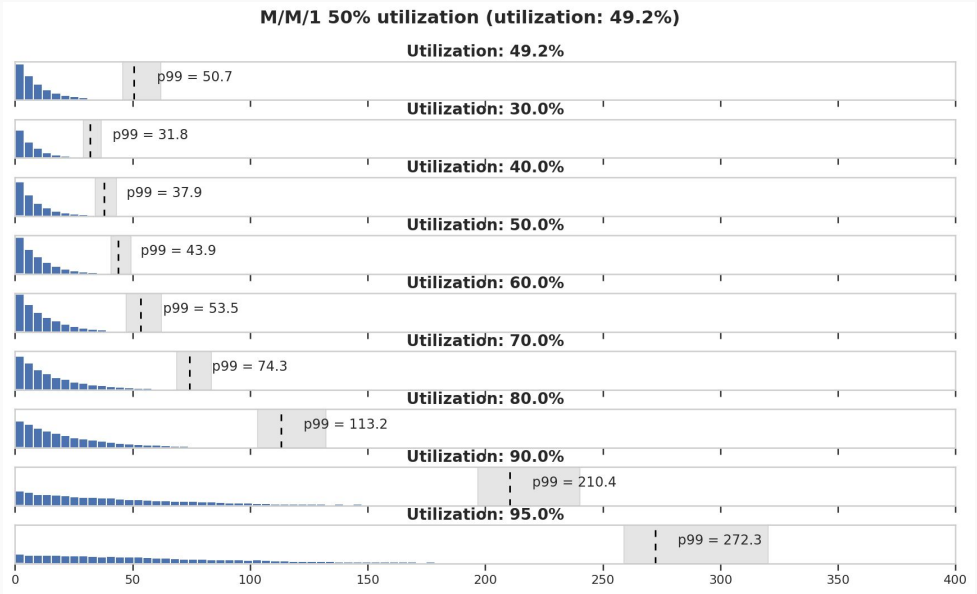
https://colab.research.google.com/drive/14dwS_9gkSfPd835Q2EO-FflitRZgJvCh

Simulating tail latency at different utilizations



Look at the Colab notebook linked at the previous slide for an implementation of this idea.

The dire result



The first graph is the baseline of 50% utilization (or almost, due to sampling errors). The dashed line is the 99%ile, and the grey area surrounding it is the interval from 98.5%ile to 99.5%ile.

Dropping the utilization doesn't yield very much benefit. However, increasing the utilization beyond 60% or so yields terrible tail latency. Remember that in this example the **average** service time is still 5ms.

Also note how the tail latency increases nonlinearly with utilization increase.

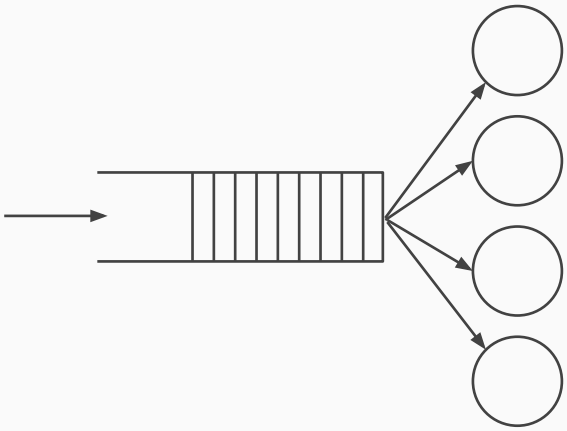
[NB: In this particular case of an M/M/1 queue you can actually calculate the p'th percentile latency via the formula $-1/(\mu-\lambda)*\ln((100-p)/100)$, with μ the service rate and λ the arrival rate, and $p \in (0;100)$; in general, i.e. for a G/G/1 queue, this is not possible.]

Takeaway #2: Queueing effects increase superlinearly with utilization

Increasing utilization by 10 percentage points will cause the tail latency to drag out by a certain factor. Increasing utilization by another 10 percentage points will lead to an increase in tail latency that is larger than the previous increase.

As the utilization reaches 100%, the tail latency grows unbounded. (There is too much work coming in to ever catch up.)

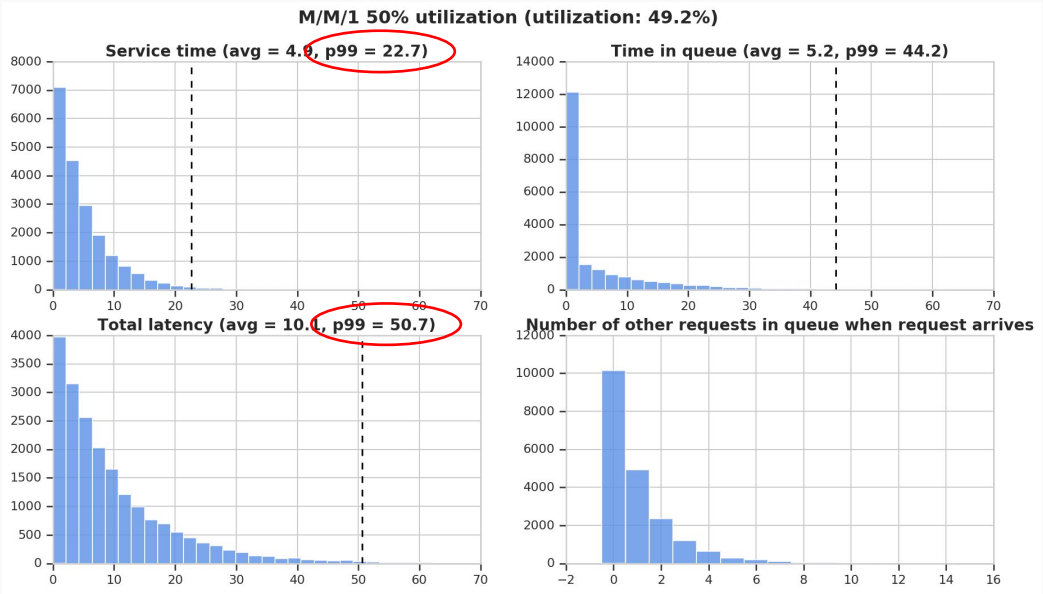
A single queue, but four cores (your system probably looks more like this)



But all is not lost: Your system is probably not a queue with a single processor. Mostly we run thread pools over *K* cores that have a single queues. Typical setups are 2, 4 or 8 cores.

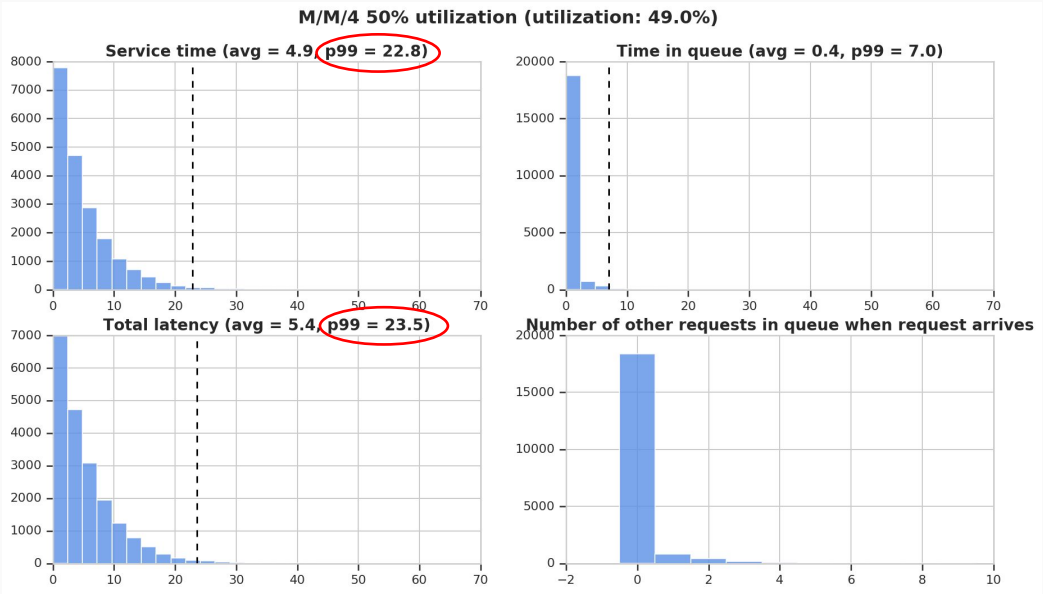
Case *K* = 4: What this means is that when a request arrives at the server, it has four chances of finding an empty processor; and if it has to wait, it only has to wait for the shortest of the currently running 4 requests to finish. This draws in the tail latency significantly.

Previously: 50% utilization with a single core



We saw previously: With a single processor, the tail latency is clearly influenced by the queue latency.

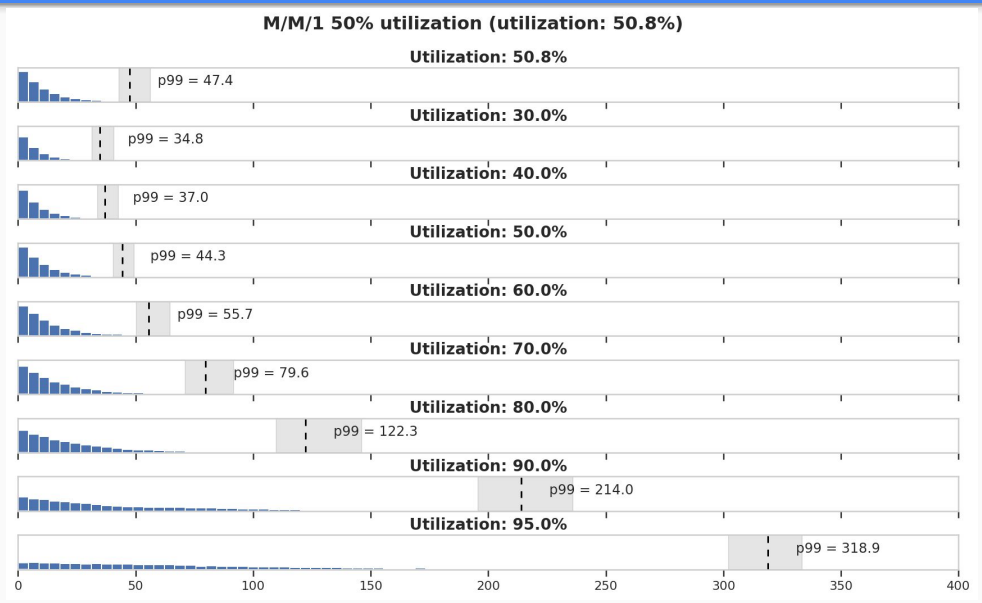
Now: 50% utilization with four cores: Barely any queueing effects



However, if you run e.g. one-quarter as many instances that have four cores each, the tail latency is almost unaffected by the queue latency.

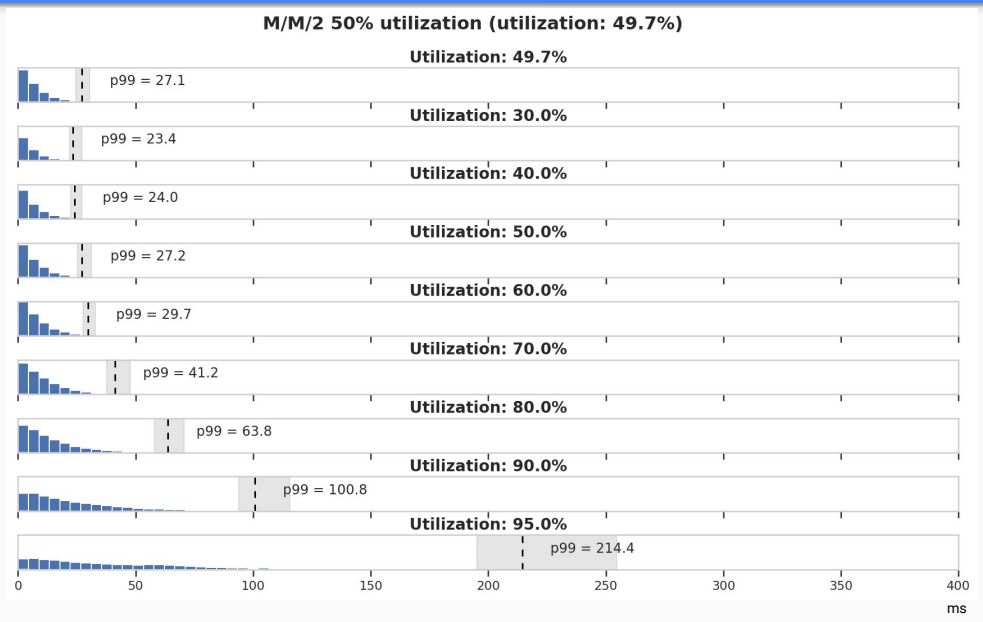
That's a phenomenal discovery! Let's just have fewer, larger instances!

Different utilizations with $K = 1$

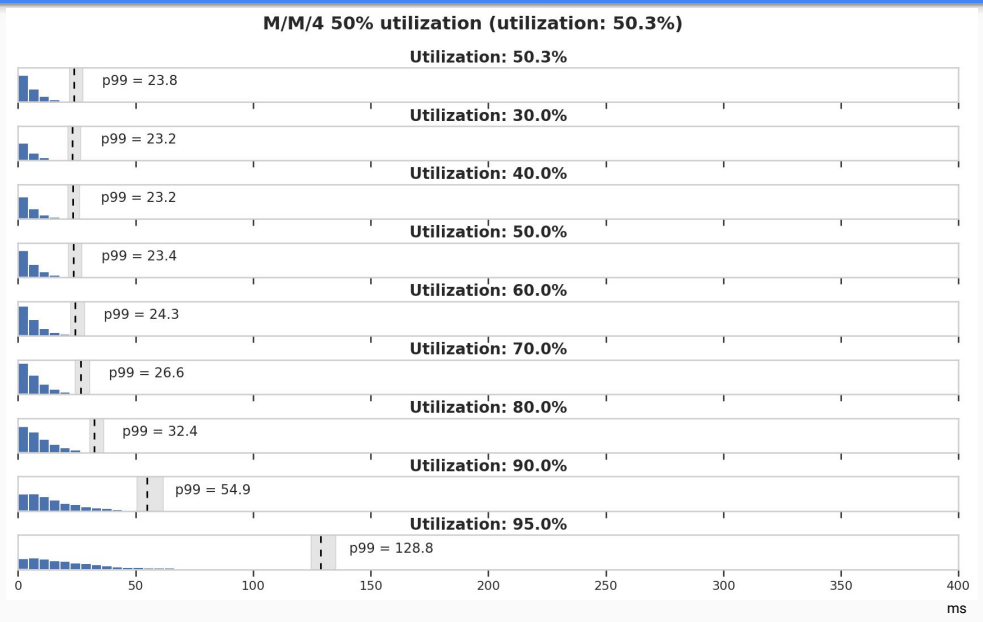


Base line: single processor. Clear nonlinear increase of tail latency.

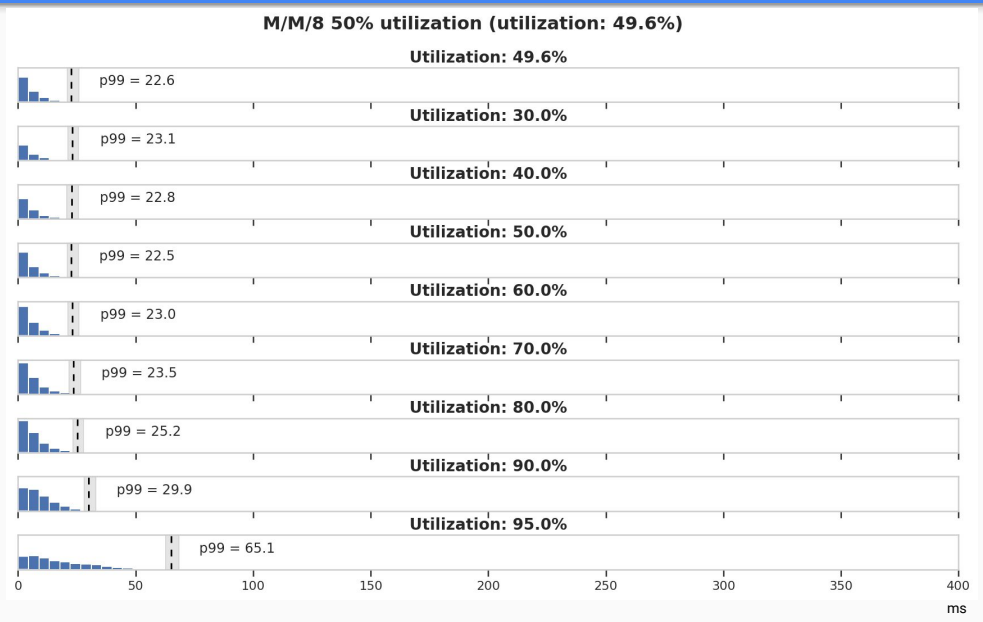
Different utilizations with $K = 2$



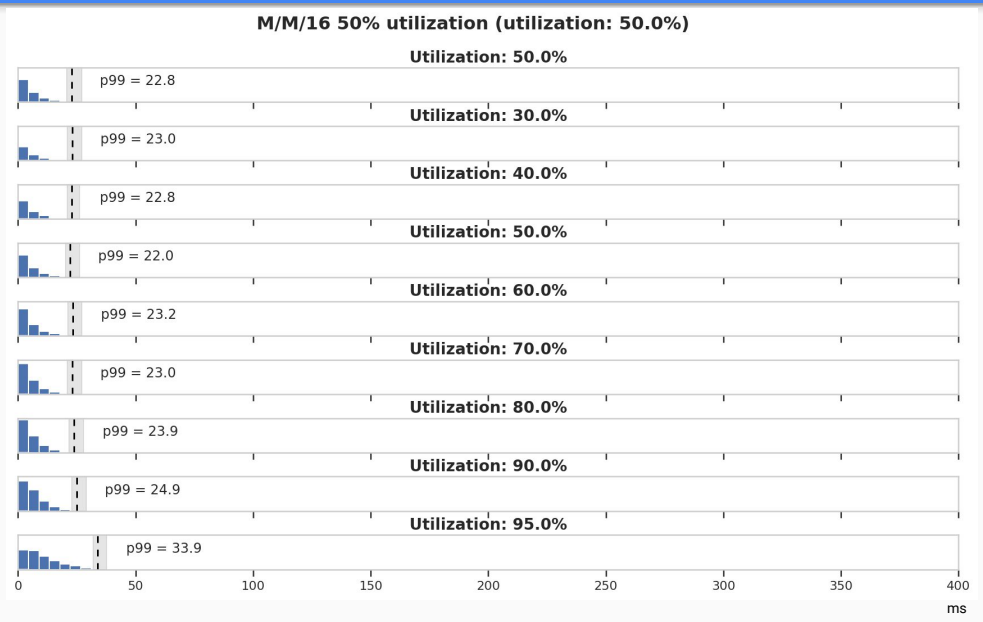
Different utilizations with $K = 4$



Different utilizations with $K = 8$



Different utilizations with $K = 16$



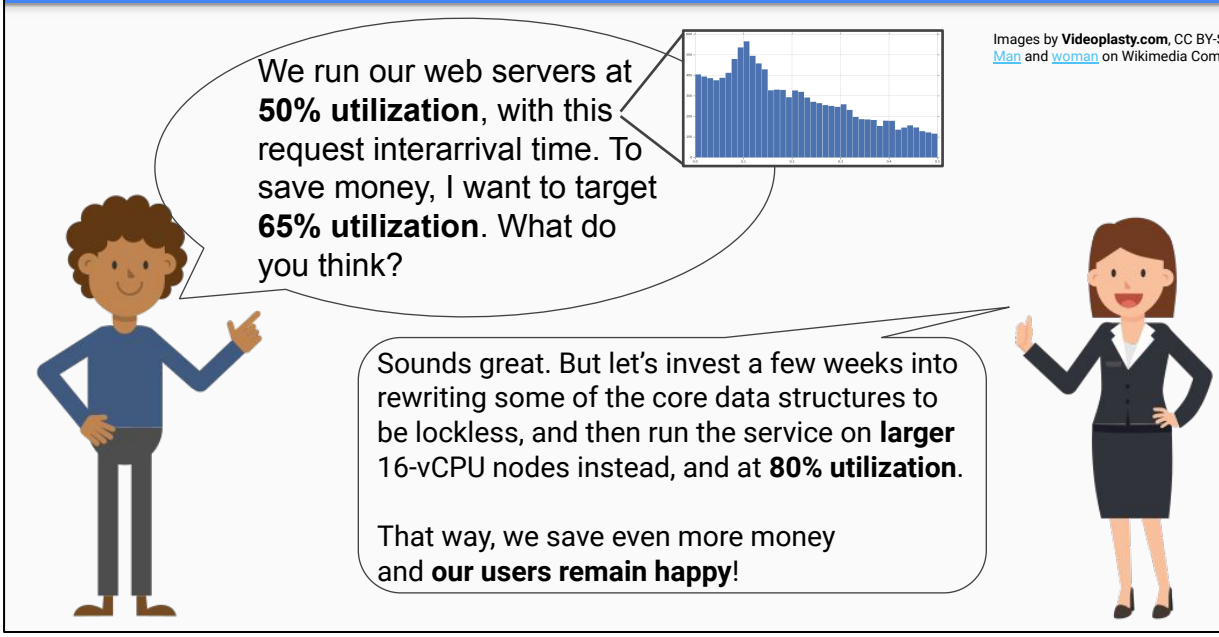
Takeaway #3:
Increasing the parallelism in your server will pull in the tail latency(*)

(*) Of course, if increased parallelism leads to e.g. more contention on shared resouces that are protected by a mutex, you might actually perceive a slowdown. See also: Amdahl's law and the Universal Scalability Law.

NB, fewer, larger jobs also help with load balancing. See my SRECon 2018 talk about randomized load balancing.

See also the Square Staffing Rule, which would come to the same conclusion.

How our conversations about utilization *should* go



When talking about utilization, always think about the fundamental tradeoff with tail latency. Bring plenty of data to the table, not just time averages!

Recap of the three takeaways:

#1 – A significant amount of tail latency can be due to queueing effects.

#2 – Queueing effects increase superlinearly with utilization.

#3 – Increasing the parallelism in your server will pull in the tail latency.

#1 might hold for you, or it might not. The only way to find out is to measure.
#2 is qualitatively true, but again you'd need to measure or simulate to gain confidence on what effect changes will have.
#3 leads to many benefits, and you should do it for other benefits too if at all possible.

Thank you!

Q&A