Applied

Performance Theory

@kavya719

kavya

applying
performance theory
to practice

performance

- What's the additional load the system can support, without degrading **response time**?
- What're the system **utilization bottlenecks**?
- What's the impact of a change on **response time, maximum throughput**?

capacity

- How many **additional servers** to support 10x load?
- Is the system **over-provisioned**?

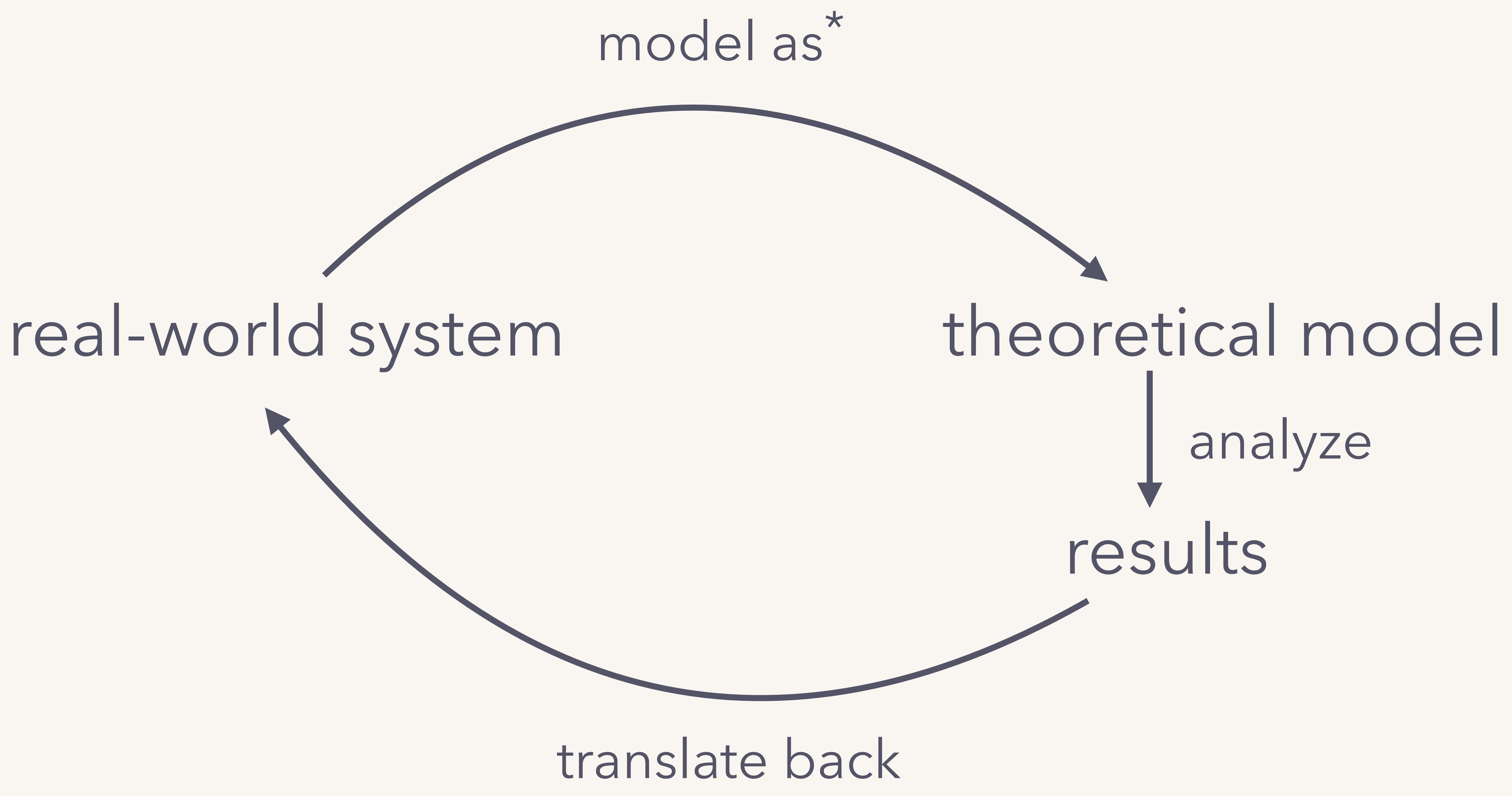
- #YOLO method

```
commit 2184ef07219d2dd5273a0e4ee

go deploy/services: double apiserver CPU

Because it has been CPU-limited several times.
This will hopefully make it perform adequately for now.
```

- load simulation
  Stressing the system to **empirically** determine **actual** performance characteristics, bottlenecks.
  Can be incredibly powerful.

- performance modeling

performance modeling



* **makes assumptions** about the system:
request arrival rate, service order, times.
cannot apply the results if your system does not satisfy them!

a single server

open, closed queueing systems

utilization law, Little's law, the P-K formula

CoDel, adaptive LIFO

a cluster of many servers

the USL

scaling bottlenecks

stepping back

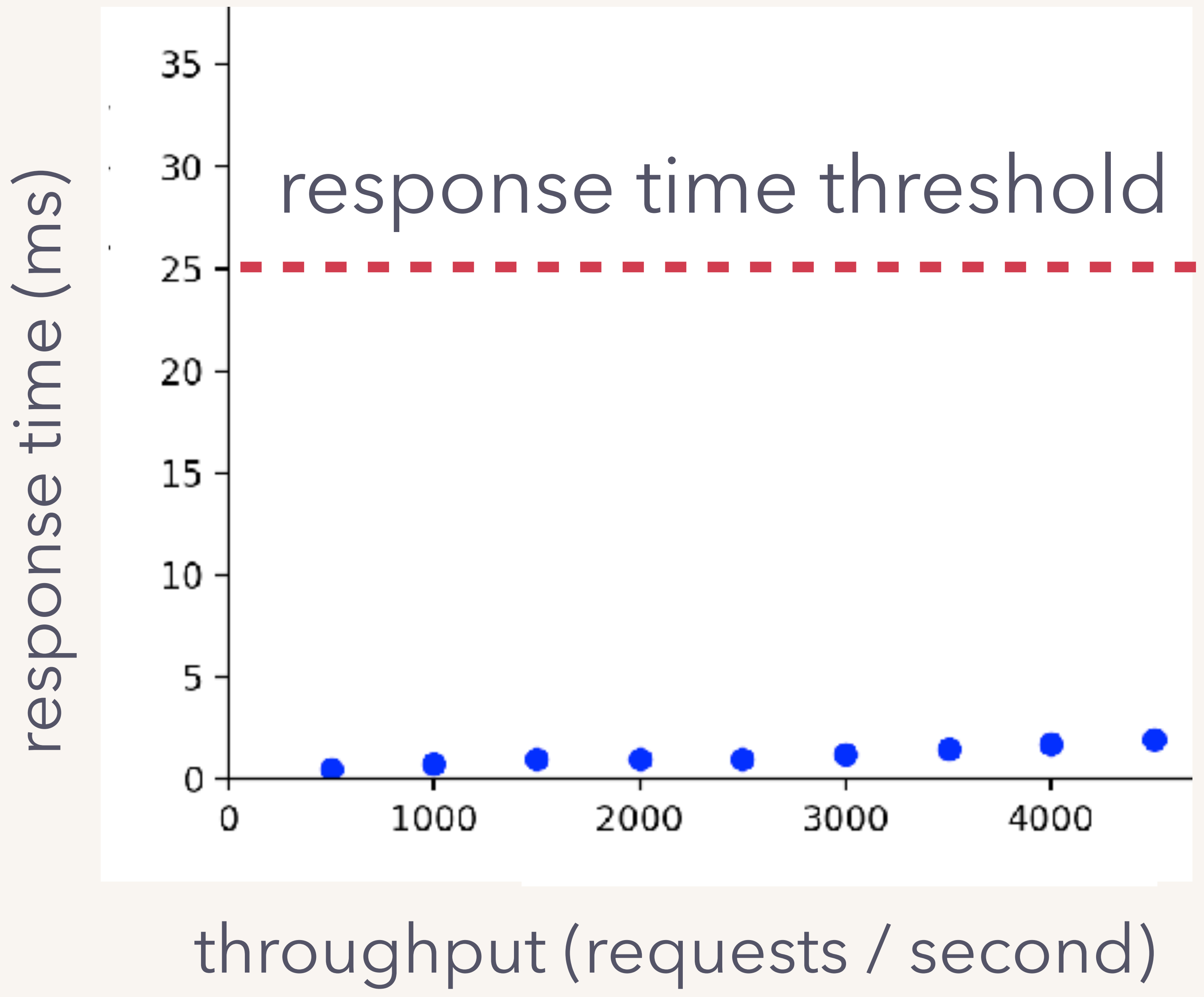
the role of performance modeling

a single server

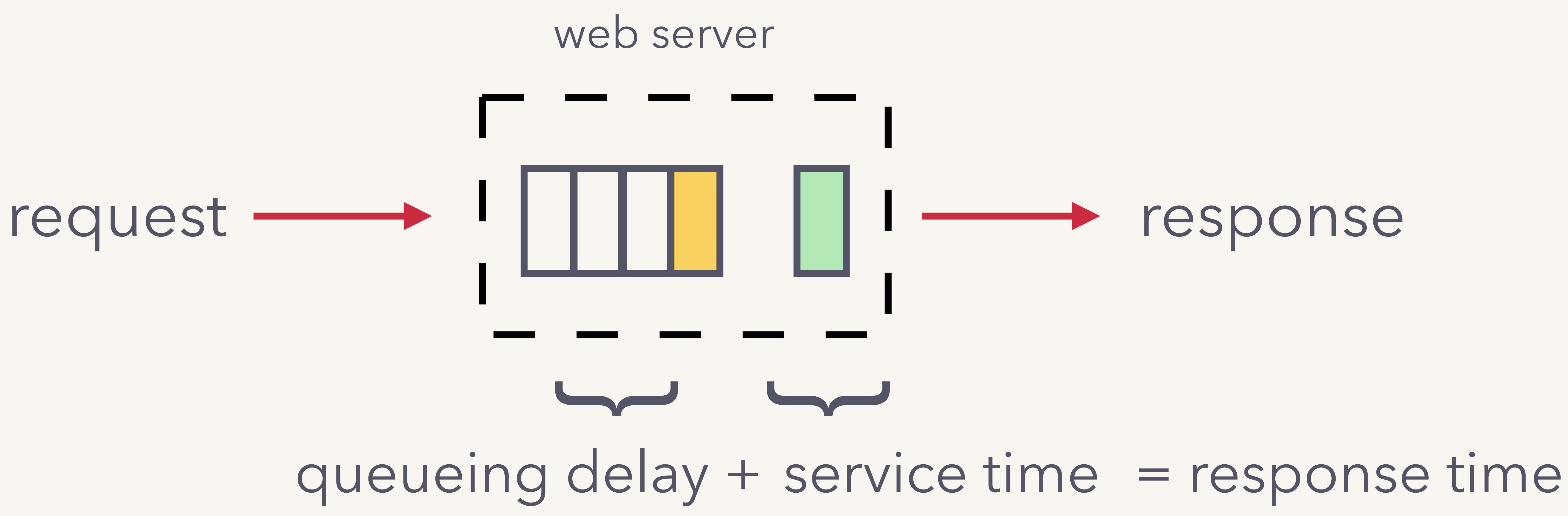
model I

"what's the **maximum throughput** of this server, given a response time target?"

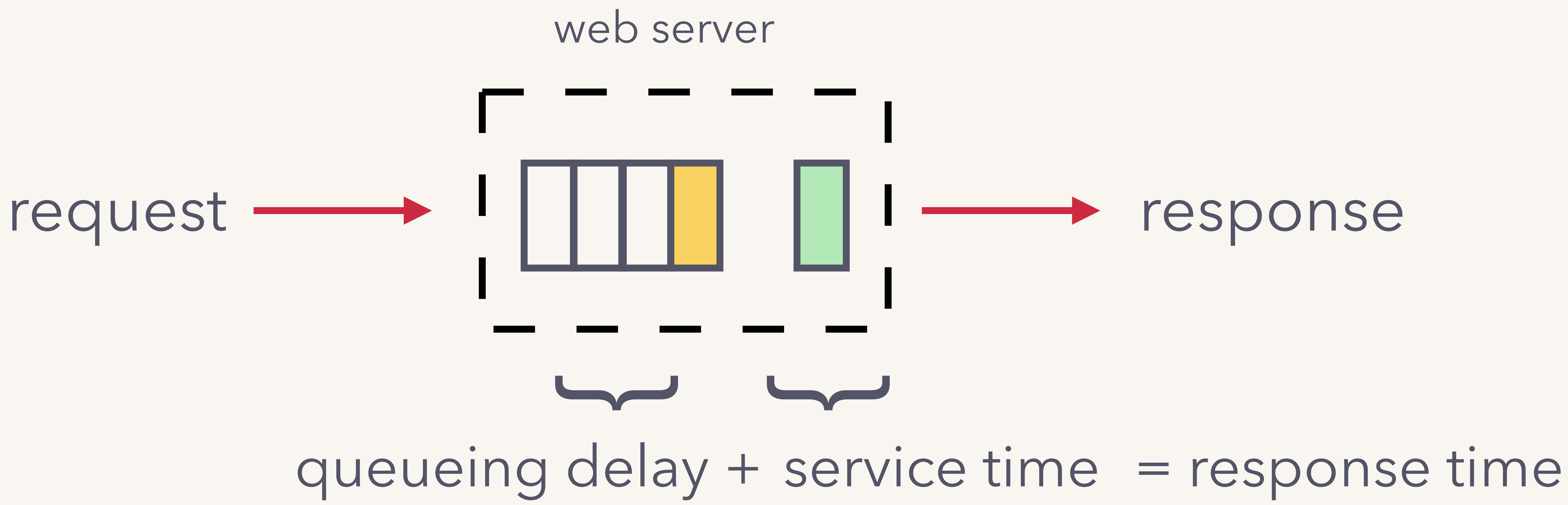
"how can we improve the **mean response time**?"



model the web server as a **queueing system**.



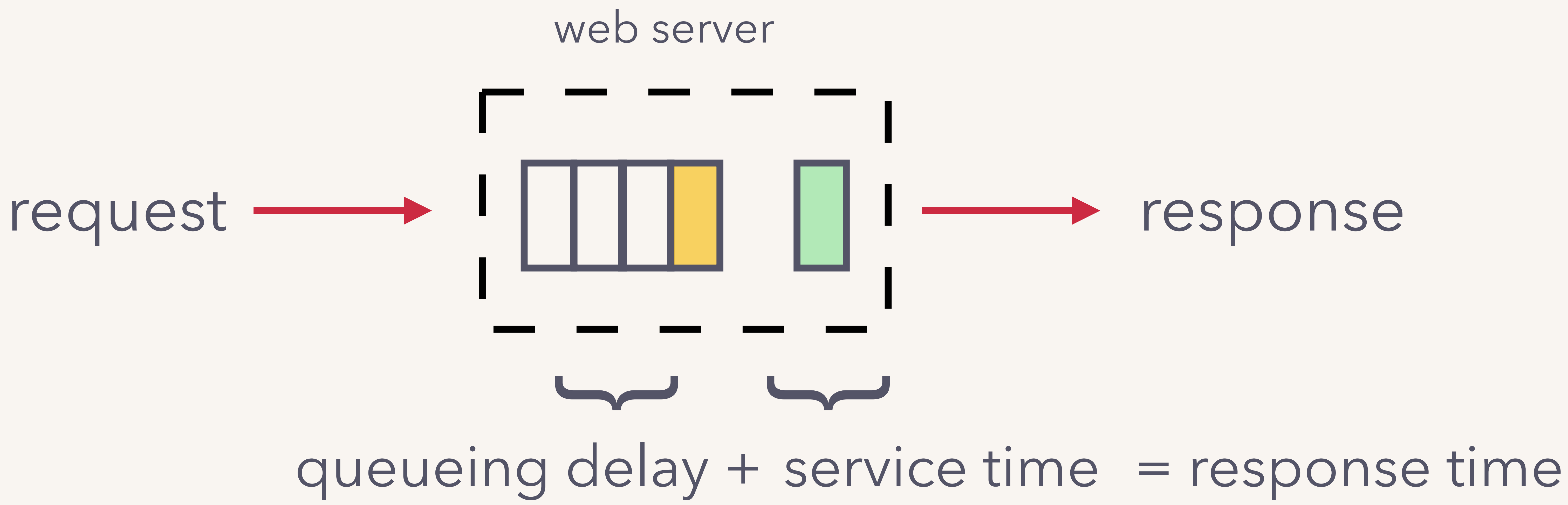
model the web server as a **queueing system**.



assumptions

1. requests are **independent and random**, arrive at some “arrival rate”.

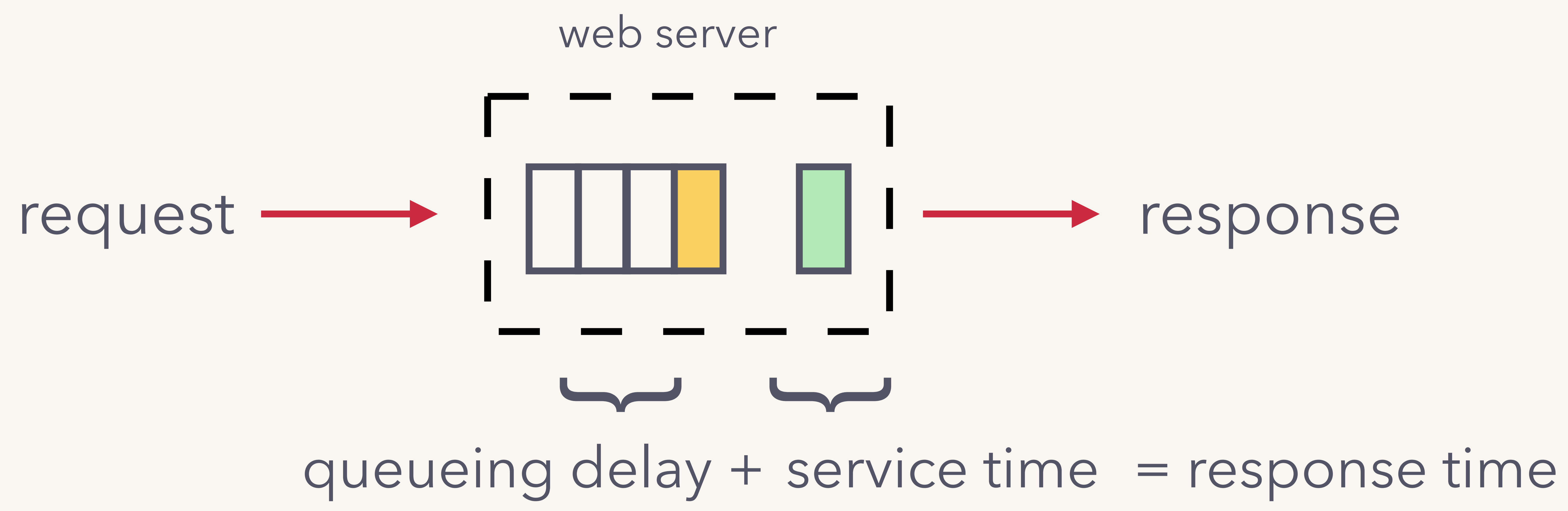
model the web server as a **queueing system**.



assumptions

1. requests are **independent and random**, arrive at some "arrival rate".
2. requests are processed **one at a time, in FIFO order;**
   requests queue if server is busy ("queueing delay").

model the web server as a **queueing system**.



assumptions

1. requests are **independent and random**, arrive at some “arrival rate”.
2. requests are processed **one at a time, in FIFO order;** requests queue if server is busy (“queueing delay”).
3. “service time” of a request is **constant.**

“What’s the maximum throughput of this server?”

i.e. given a response time target

"What's the maximum throughput of this server?"

i.e. given a response time target

arrival rate increases

Utilization law

server utilization increases

utilization = arrival rate * service time

"busyness"

“What’s the maximum throughput of this server?”

i.e. given a response time target

arrival rate increases

↓ **Utilization law**

server utilization increases linearly

"What's the maximum throughput of this server?"

i.e. given a response time target

arrival rate increases

↓ **Utilization law**

server utilization increases **linearly**

↓

P(request has to queue) increases, so
mean queue length increases, so
mean queueing delay increases.

"What's the maximum throughput of this server?"

i.e. given a response time target

arrival rate increases

↓ Utilization law

server utilization increases <u>linearly</u>

↓ P-K formula

P(request has to queue) increases, so
mean queue length increases, so
mean queueing delay increases.

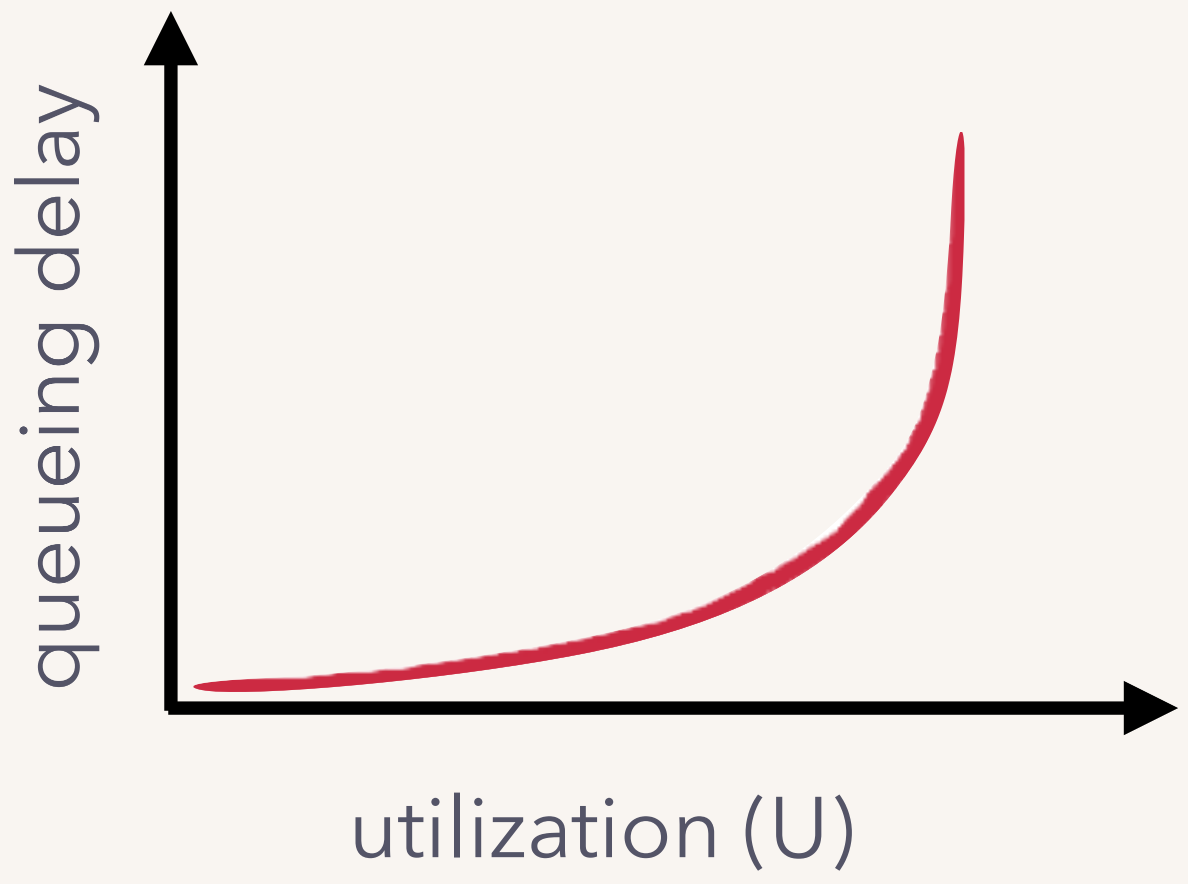
Pollaczek-Khinchine (P-K) formula

$$\text{mean queueing delay} = \frac{U}{(1 - U)} * \text{linear fn (mean service time)} * \text{quadratic fn (service time variability)}$$

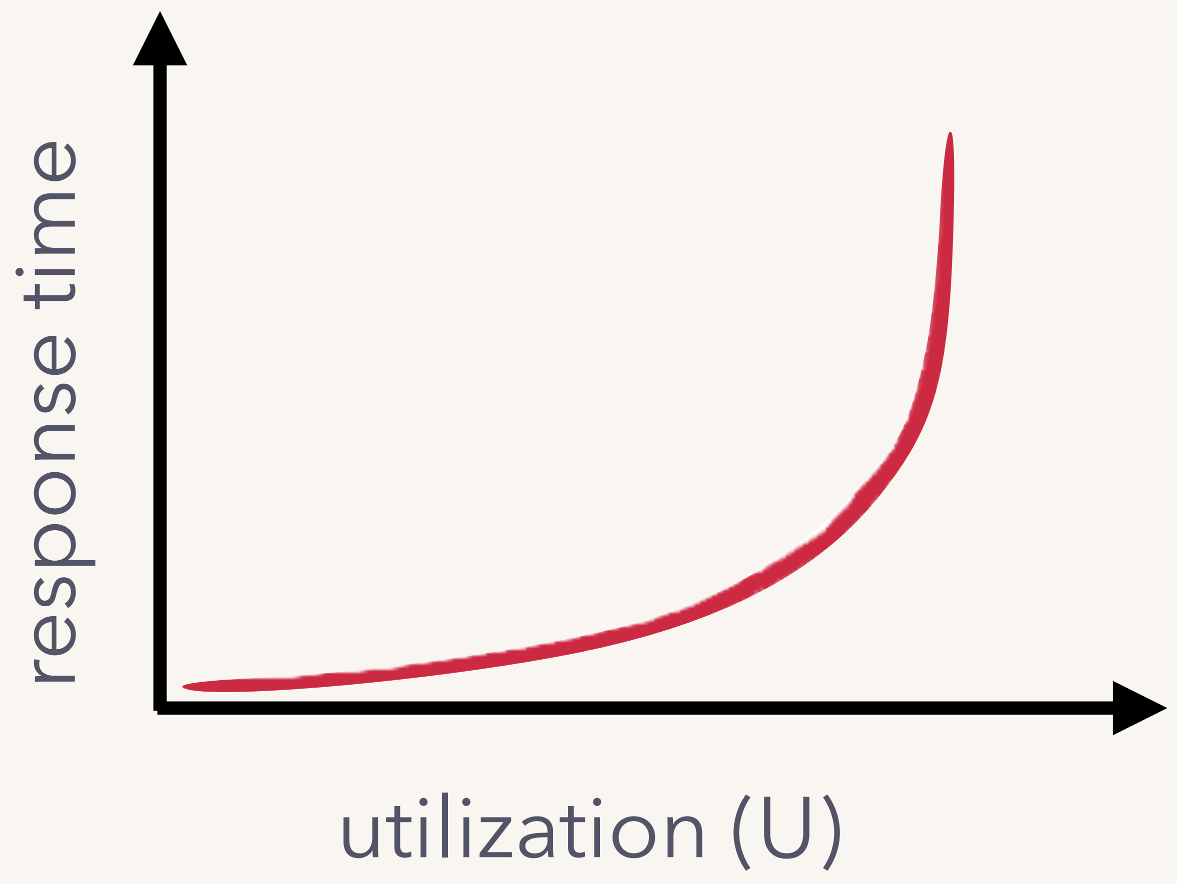
assuming constant service time and so, request sizes:

$$\text{mean queueing delay} \propto \frac{U}{(1 - U)}$$

||



since response time ∝ queueing delay



“What’s the maximum throughput of this server?”

i.e. given a response time target

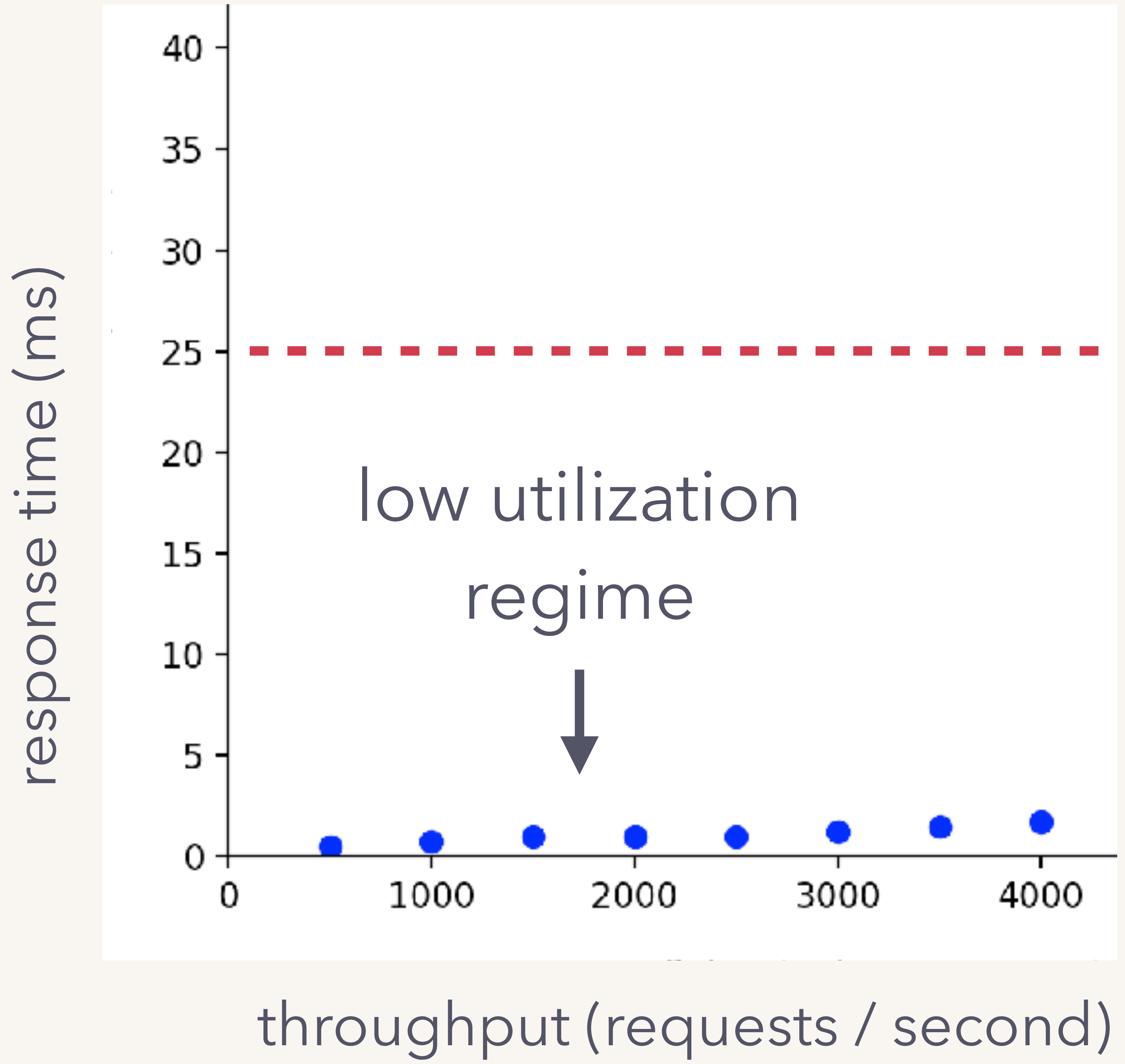
arrival rate increases

↓ **Utilization law**

server utilization increases **linearly**

↓ **P-K formula**

mean queueing delay increases **non-linearly**;
so, **response time too.**



"What's the maximum throughput of this server?"

i.e. given a response time target

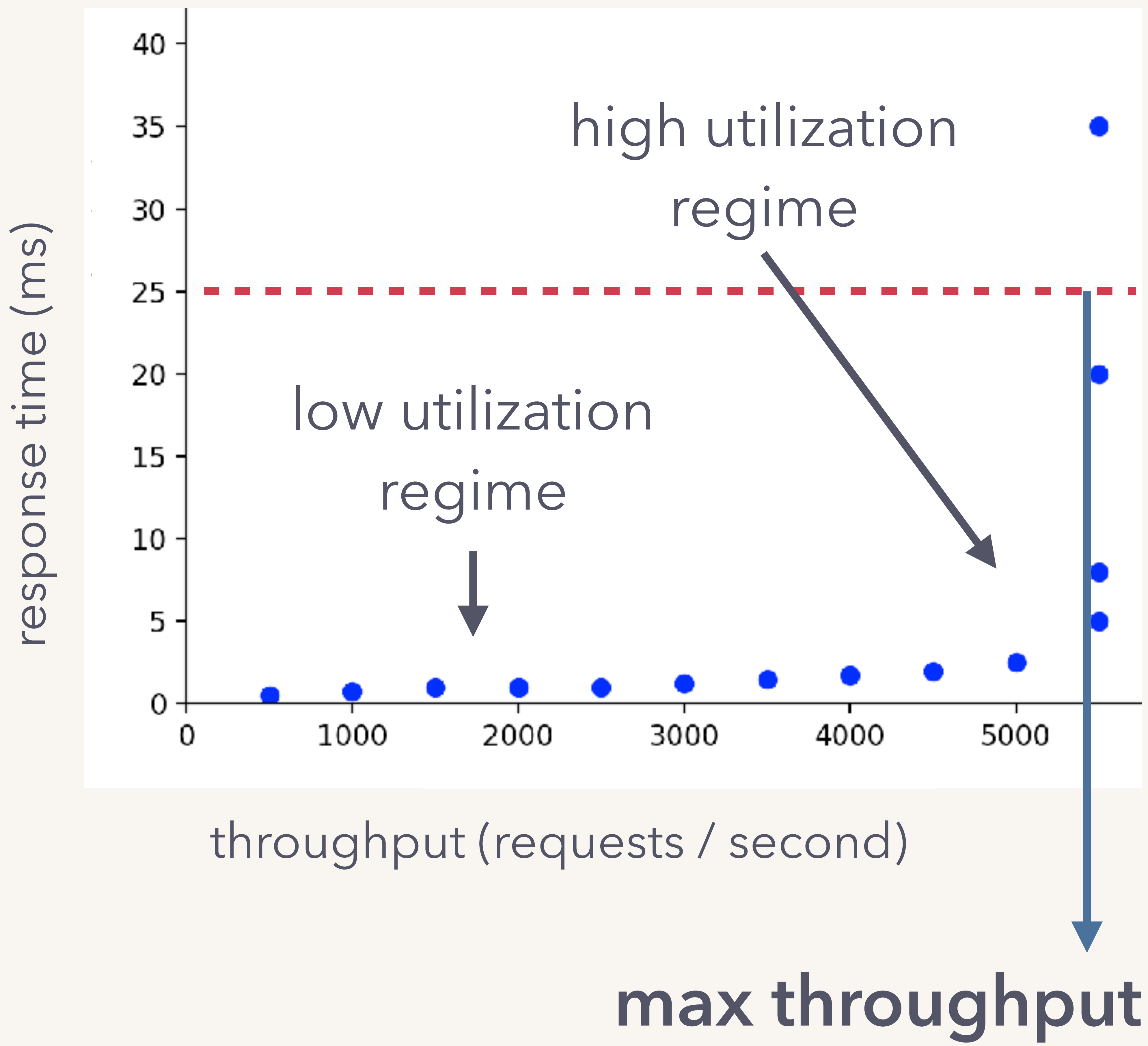
arrival rate increases

↓ **Utilization law**

server utilization increases **linearly**

↓ **P-K formula**

mean queueing delay increases **non-linearly**;
so, **response time too.**



“How can we improve the mean response time?”

"How can we improve the mean response time?"

1. response time ∝ queueing delay

prevent requests from queuing too long

- **set a max queue length**
- **client-side concurrency control**

"How can we improve the mean response time?"

1. response time ∝ queueing delay

prevent requests from queuing too long

- **Controlled Delay (CoDel)**
  in Facebook's Thrift framework

key insight: queues are typically empty

allows short bursts, prevents standing queues

```
onNewRequest(req, queue):
  if (queue.lastEmptyTime() < (now - N ms)) {
    // Queue was last empty more than N ms ago;
    // set timeout to M << N ms.
    timeout = M ms
  } else {
    // Else, set timeout to N ms.
    timeout = N ms
  }
  queue.enqueue(req, timeout)
```

- **set a max queue length**
- **client-side concurrency control**

"How can we improve the mean response time?"

1. response time ∝ queueing delay

prevent requests from queuing too long

- **Controlled Delay (CoDel)**
  in Facebook's Thrift framework

  → key insight: queues are typically empty
  allows short bursts, prevents standing queues

- **adaptive or always LIFO**
  in Facebook's PHP runtime,
  Dropbox's Bandaid reverse proxy.

  → helps when system is overloaded,
  makes no difference when it's not.

  newest requests first, not old requests
  that are likely to expire.

- **set a max queue length**

- **client-side concurrency control**

"How can we improve the mean response time?"

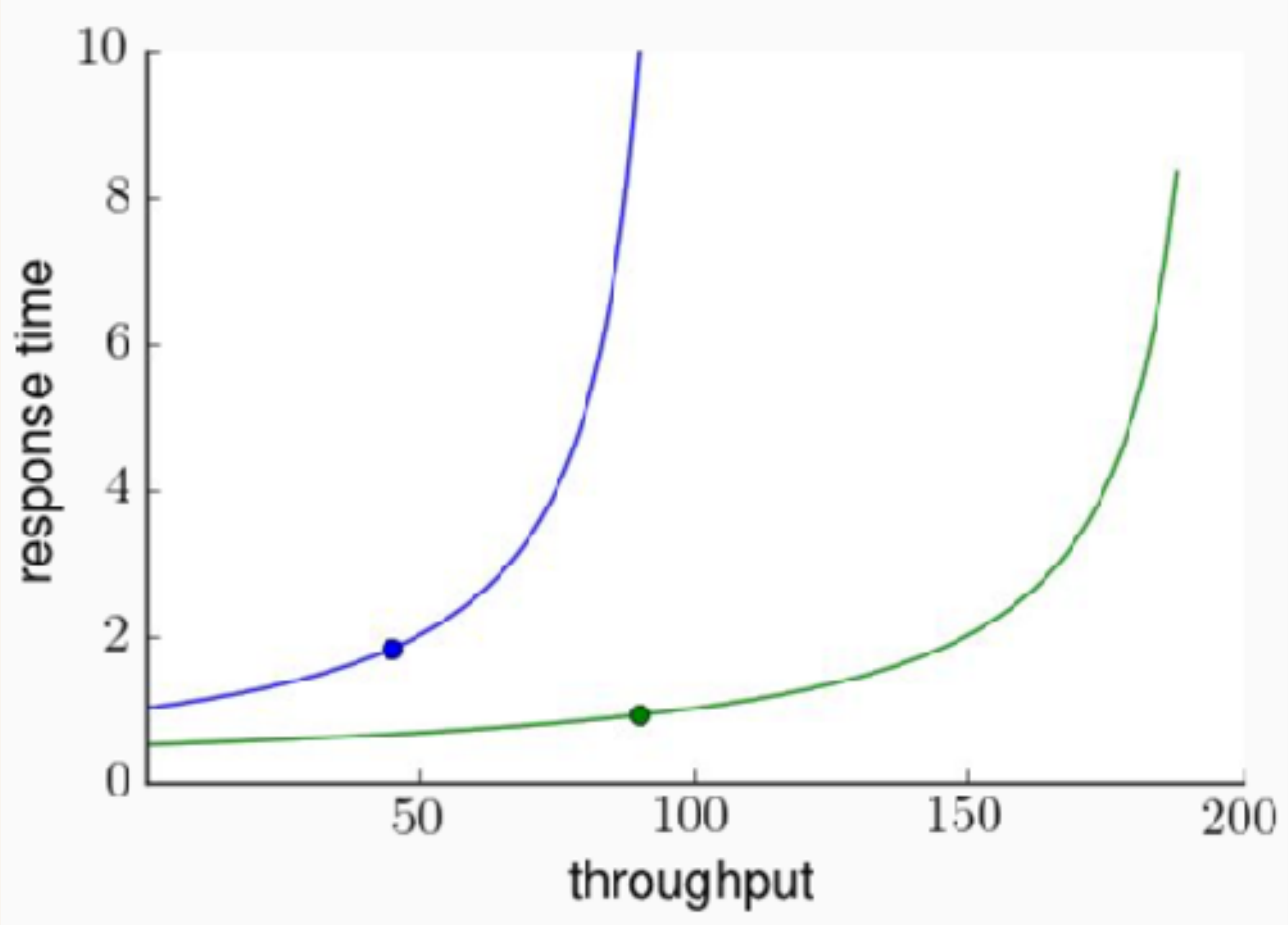
2. response time ∝ queueing delay

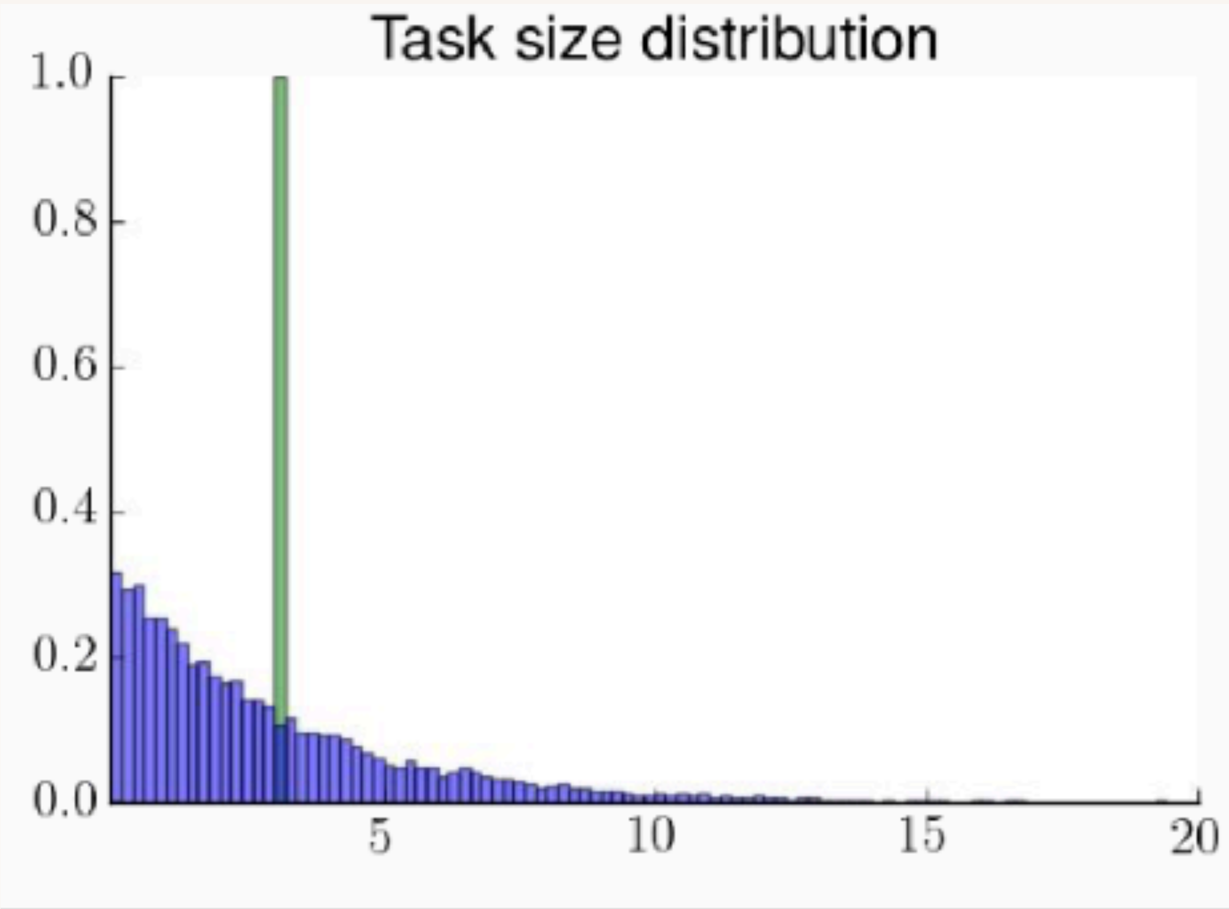
P-K formula

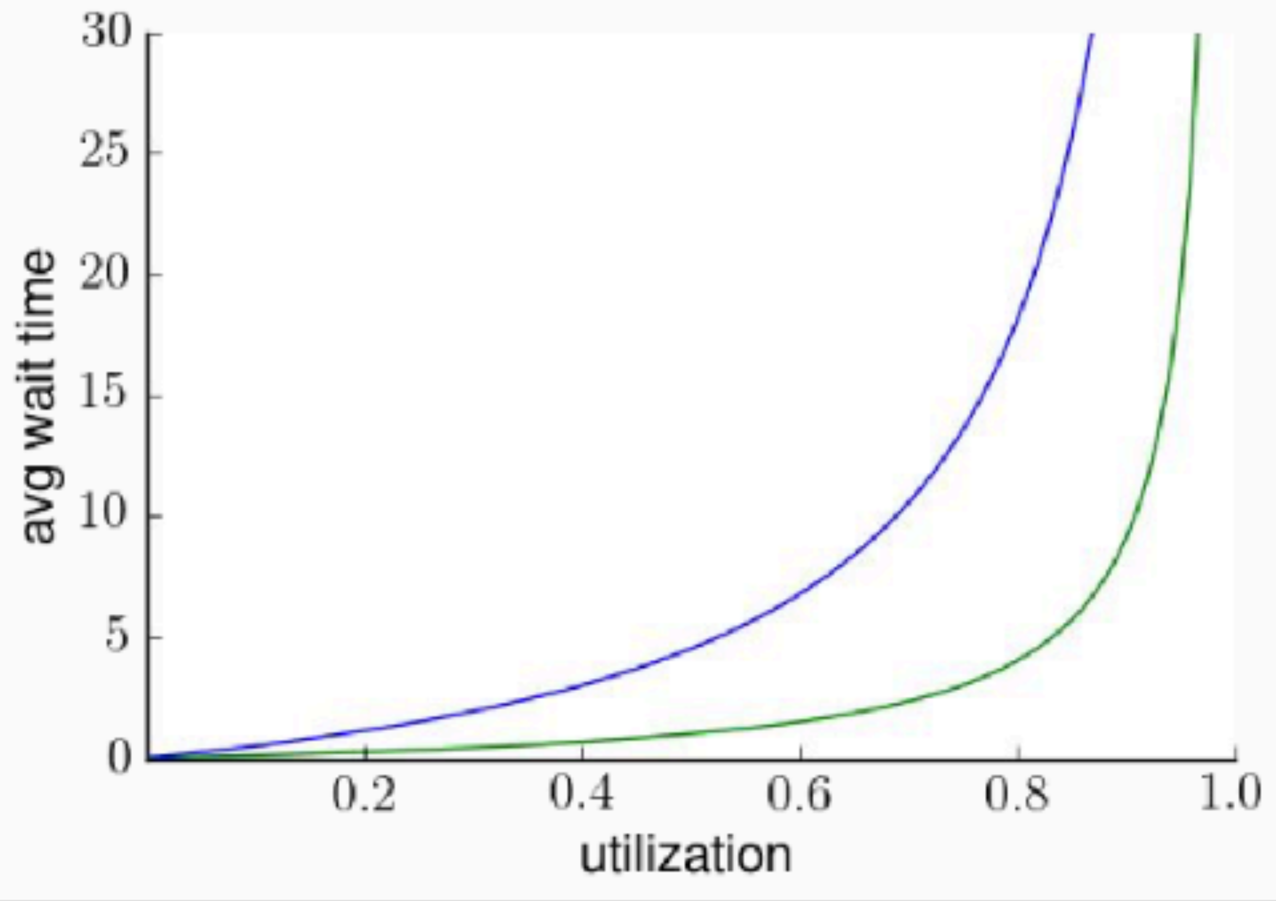
$\frac{U}{(1 - U)}$ * **linear** fn (**mean** service time) * quadratic fn (**service** time **variability**)

decrease service time
by optimizing application code

decrease request / service size variability
for example, by batching requests

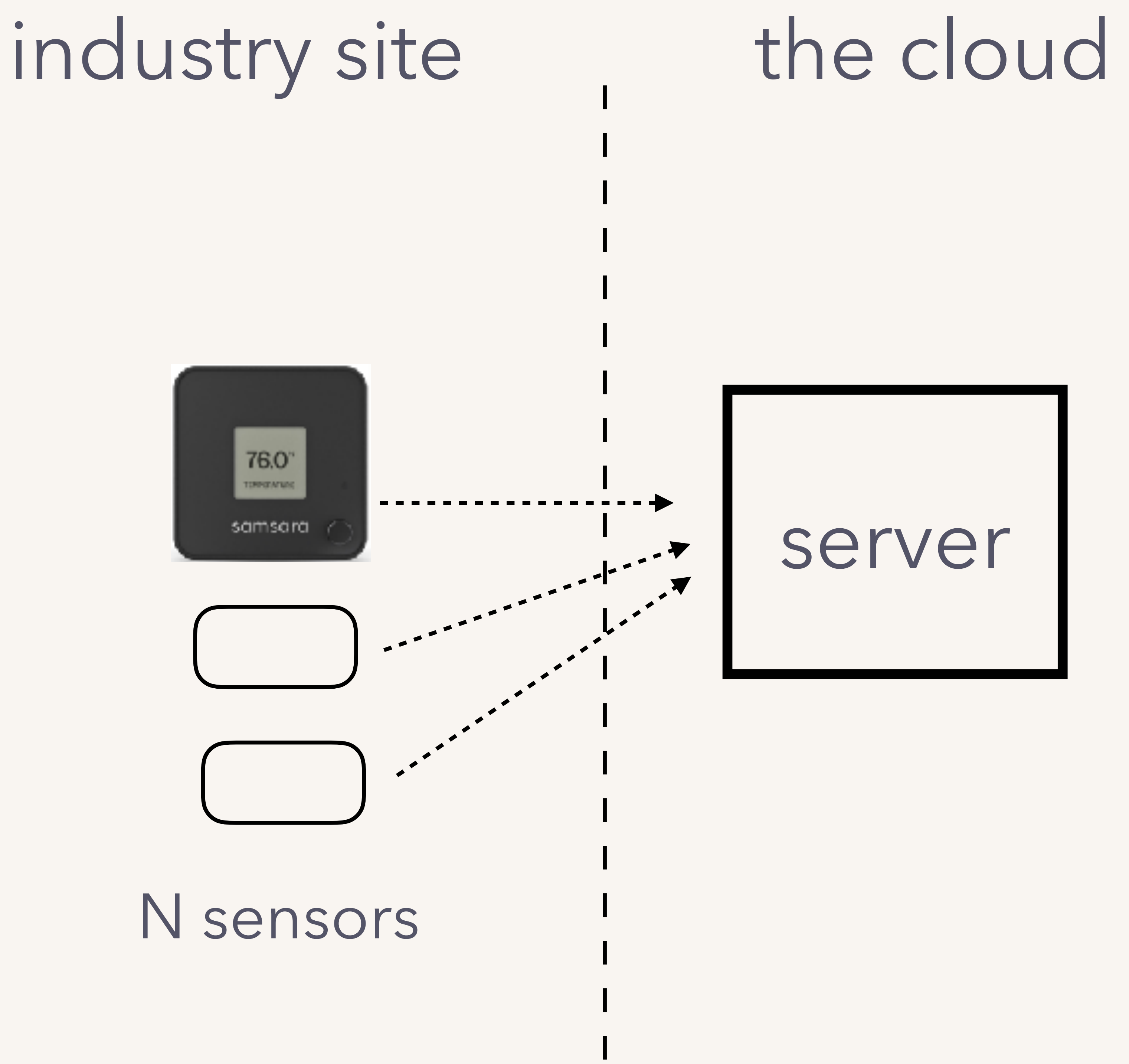




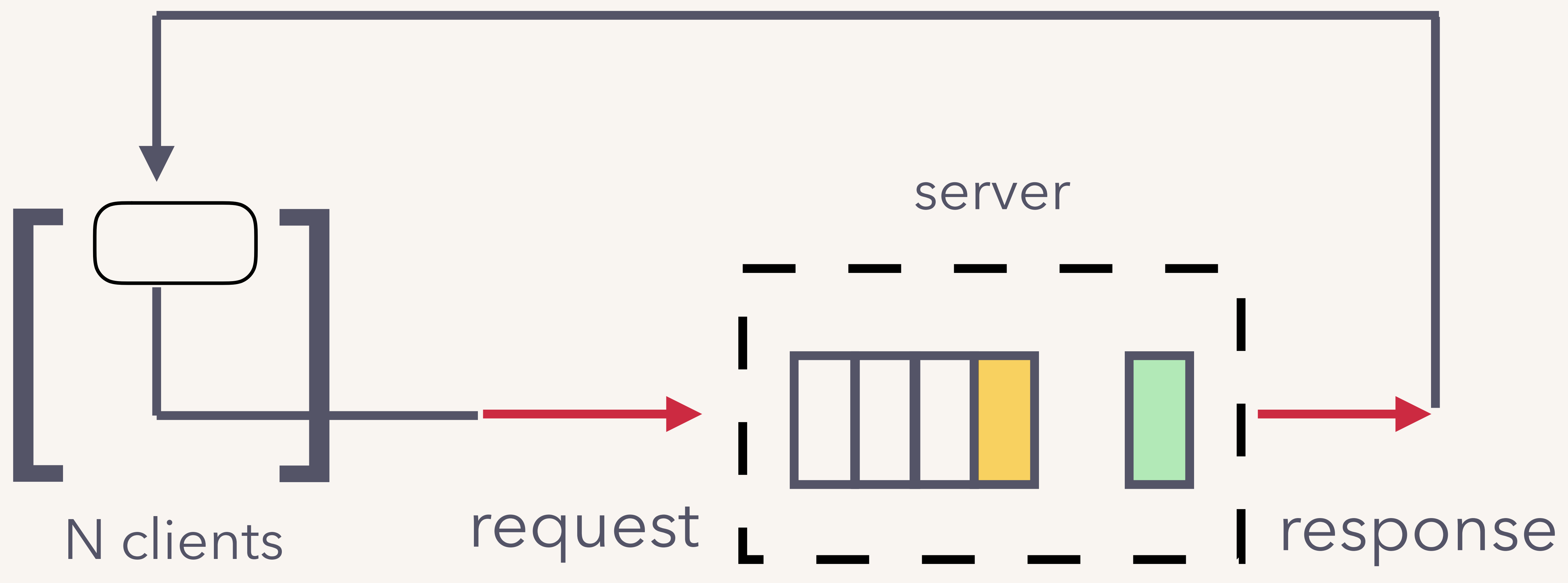


model II

```
while true:
  // upload synchronously.
  ack = upload(data)
  // update state,
  // sleep for Z seconds.
  deleteUploaded(ack)
  sleep(Z seconds)
```



processes data from N sensors



This is called a **closed system.**

super different that the previous web server model (**open system**).

- requests are synchronized.
- fixed number of clients.

throughput depends on response time!

queue length is bounded (<= N),
so **response time bounded**!

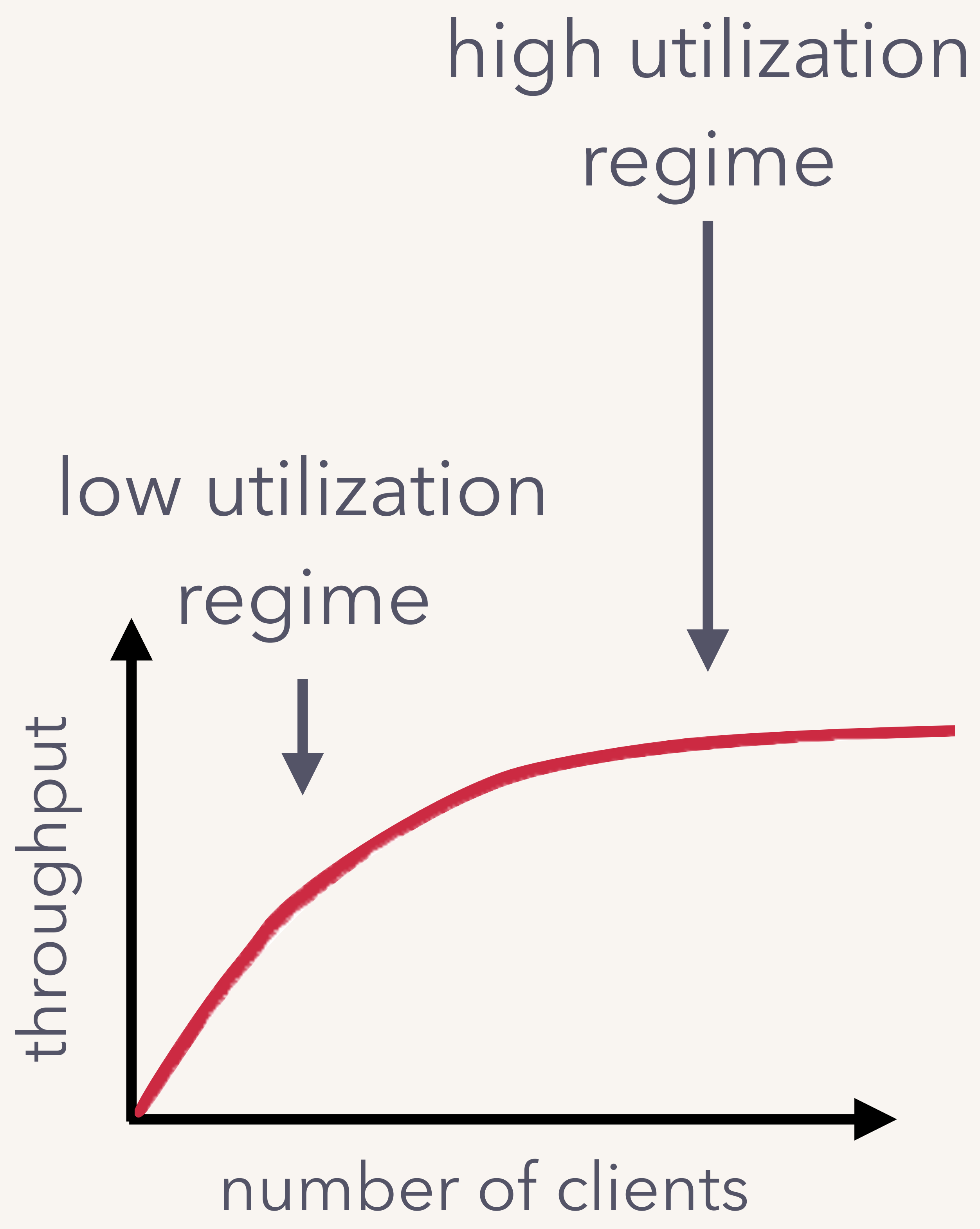
response time vs. load for closed systems

assumptions

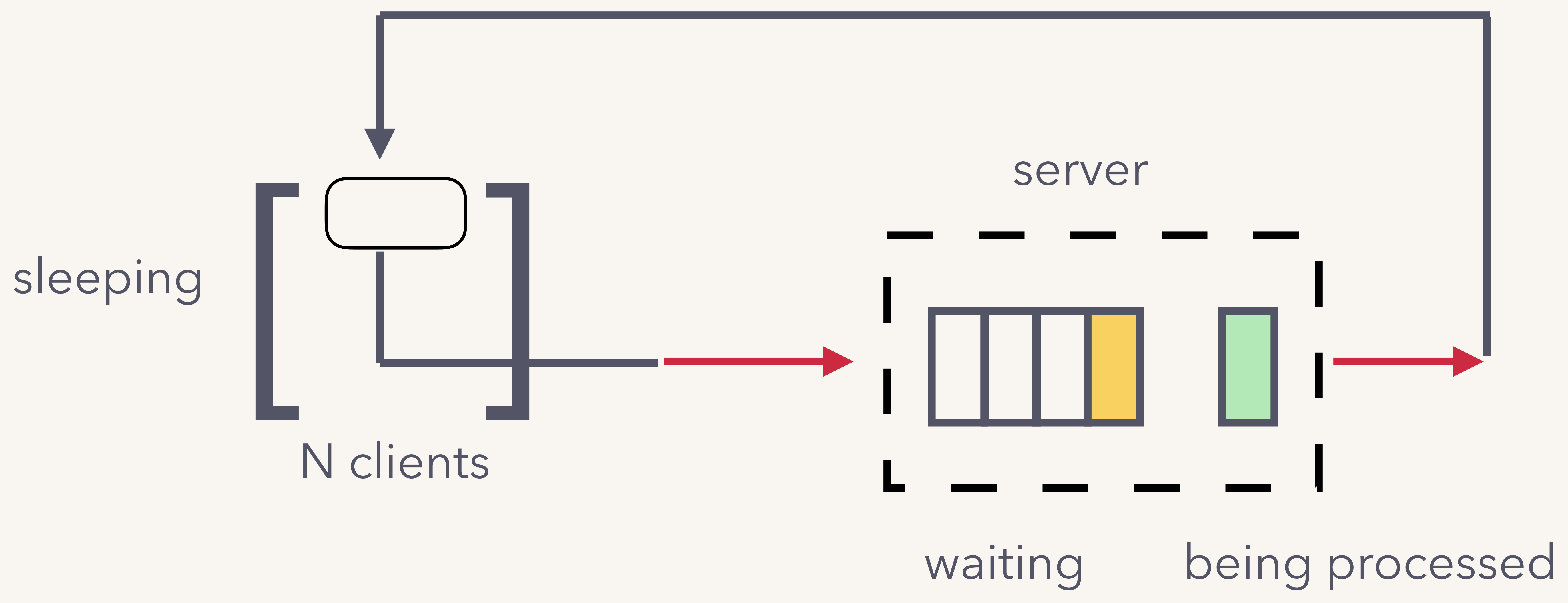
1. sleep time ("think time") is **constant**.
2. requests are processed **one at a time, in FIFO order.**
3. service time is **constant.**

Like earlier, as the **number of clients (N)** increases:
throughput increases to a point i.e. until utilization is high.
after that, increasing N only increases **queuing**.

What happens to **response time** in **this** regime?



Little's Law for closed systems

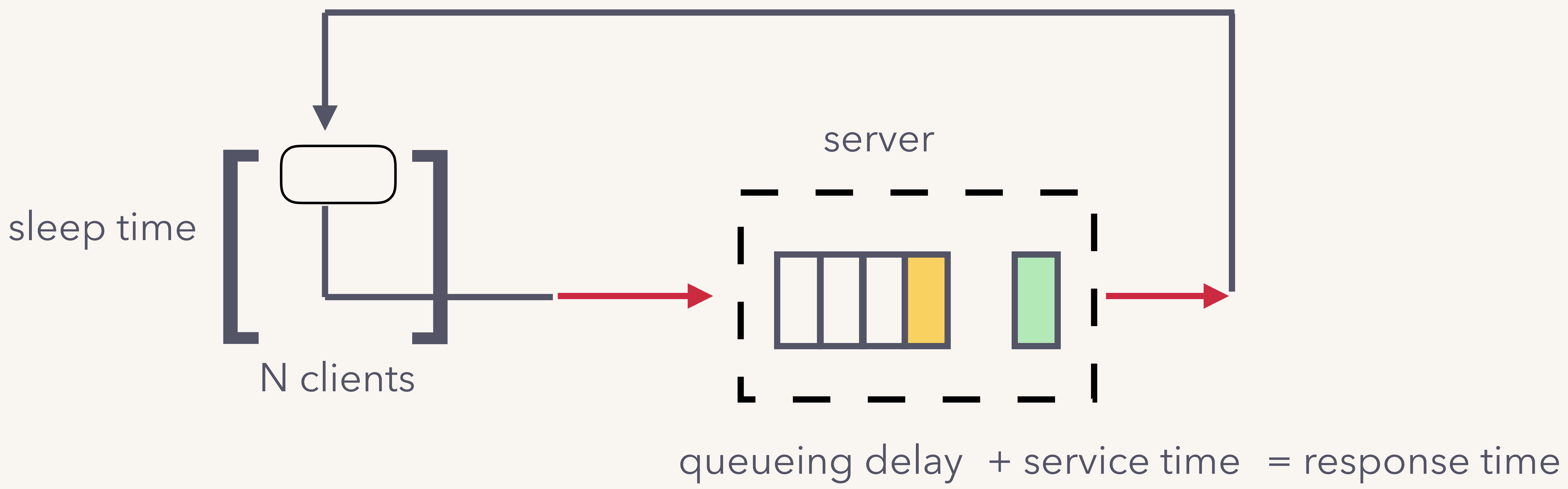


the system in this case is the **entire** loop i.e.

a request can be in one of **three states** in the system:
sleeping (on the device), waiting (in the server queue), being processed (in the server).

the total number of requests in the system includes requests **across the states**.

Little's Law for closed systems



requests in system = throughput * round-trip time of a request across the whole system

= sleep time + response time

applying it in the high utilization regime (constant throughput) and assuming constant sleep:

N = constant * response time

So, response time **only grows linearly** with N!

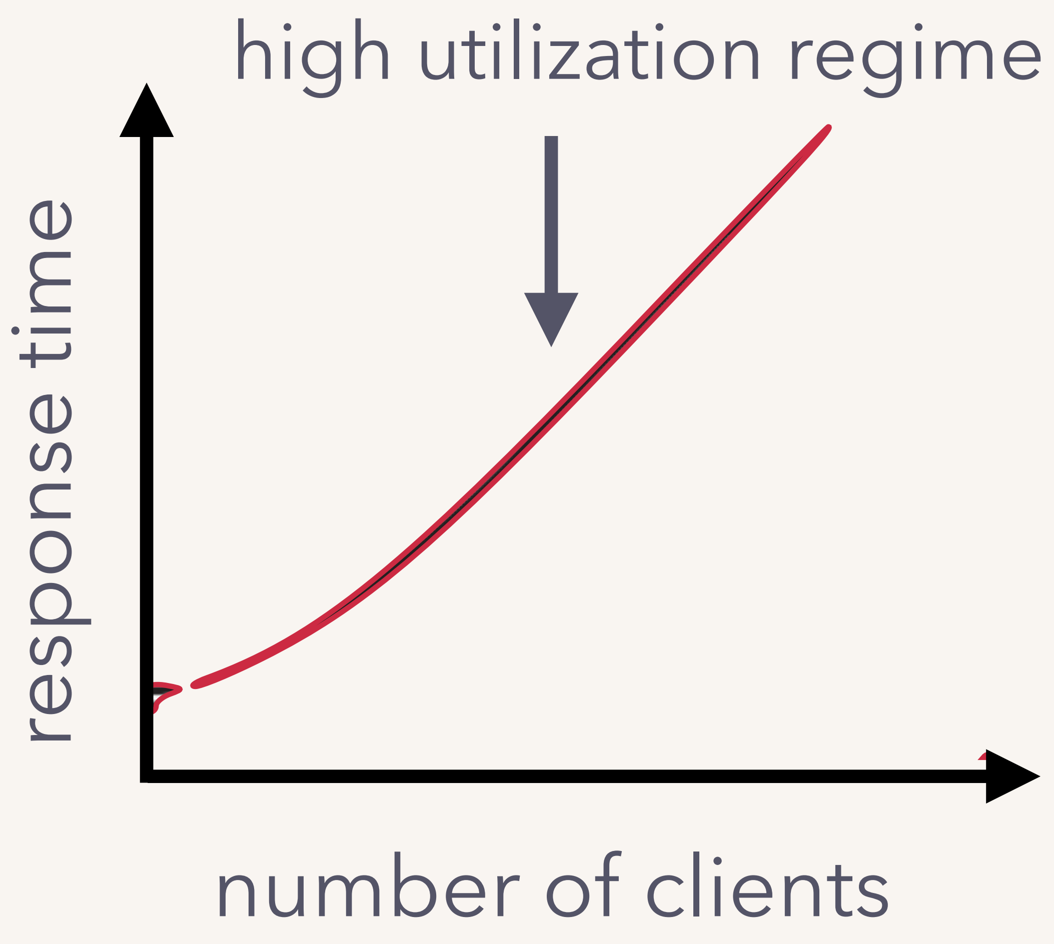
response time vs. load for closed systems

Like earlier, as the **number of clients (N)** increases:
throughput increases to a point i.e. until utilization is high.
after that, increasing N only increases **queuing**.

→ low utilization regime:
response time stays ~same

→ high utilization regime:
grows **linearly** with N.

So, response time for a closed system:



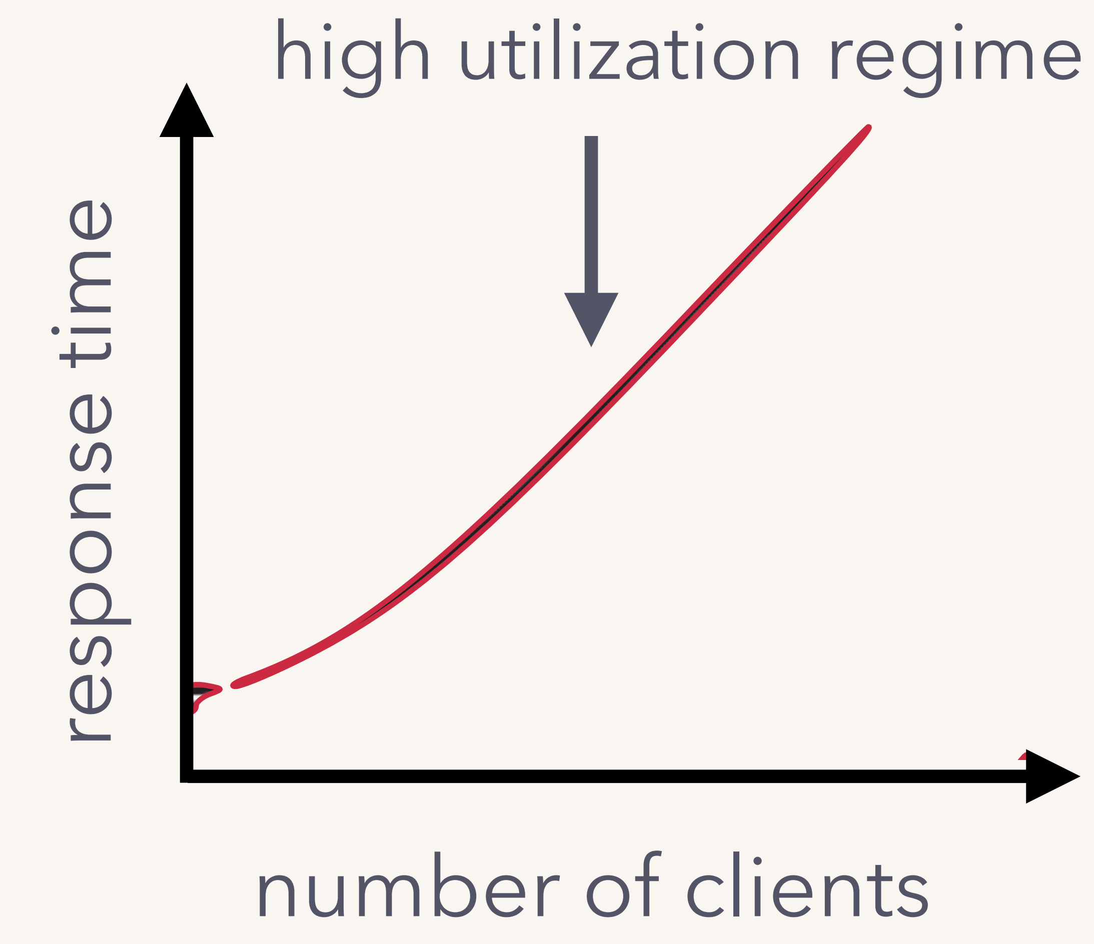
response time vs. load for closed systems

Like earlier, as the **number of clients (N)** increases:
throughput increases to a point i.e. until utilization is high.
after that, increasing N only increases **queuing**.

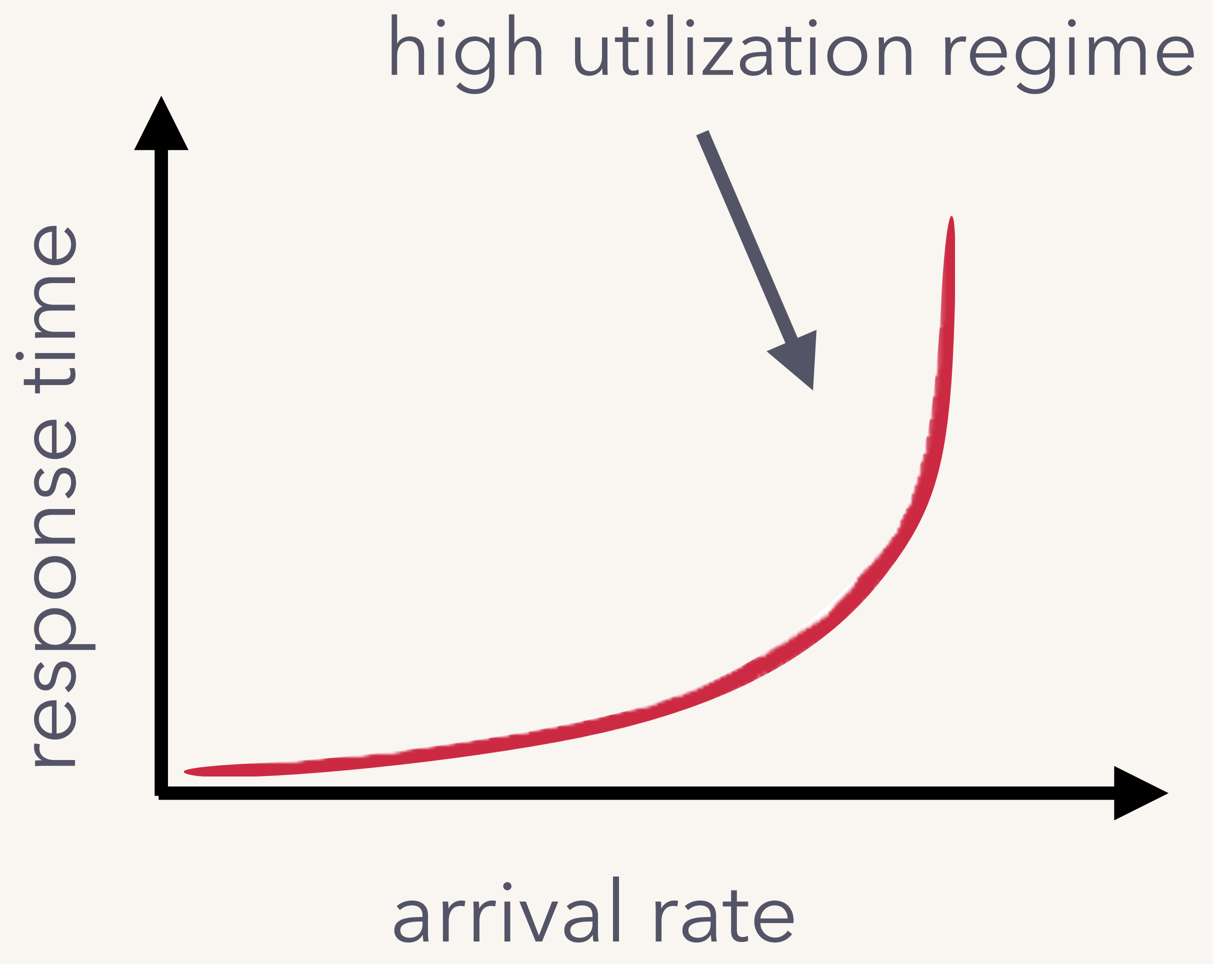
→ low utilization regime:
response time stays ~same

→ high utilization regime:
grows **linearly** with N.

So, response time for a closed system:



way different than for an open system:



open v/s closed systems

closed systems are **very different** from open systems:

- how throughput relates to response time.
- response time versus load, especially in the high load regime.

uh oh…

open v/s closed systems

standard load simulators typically mimic **closed systems**

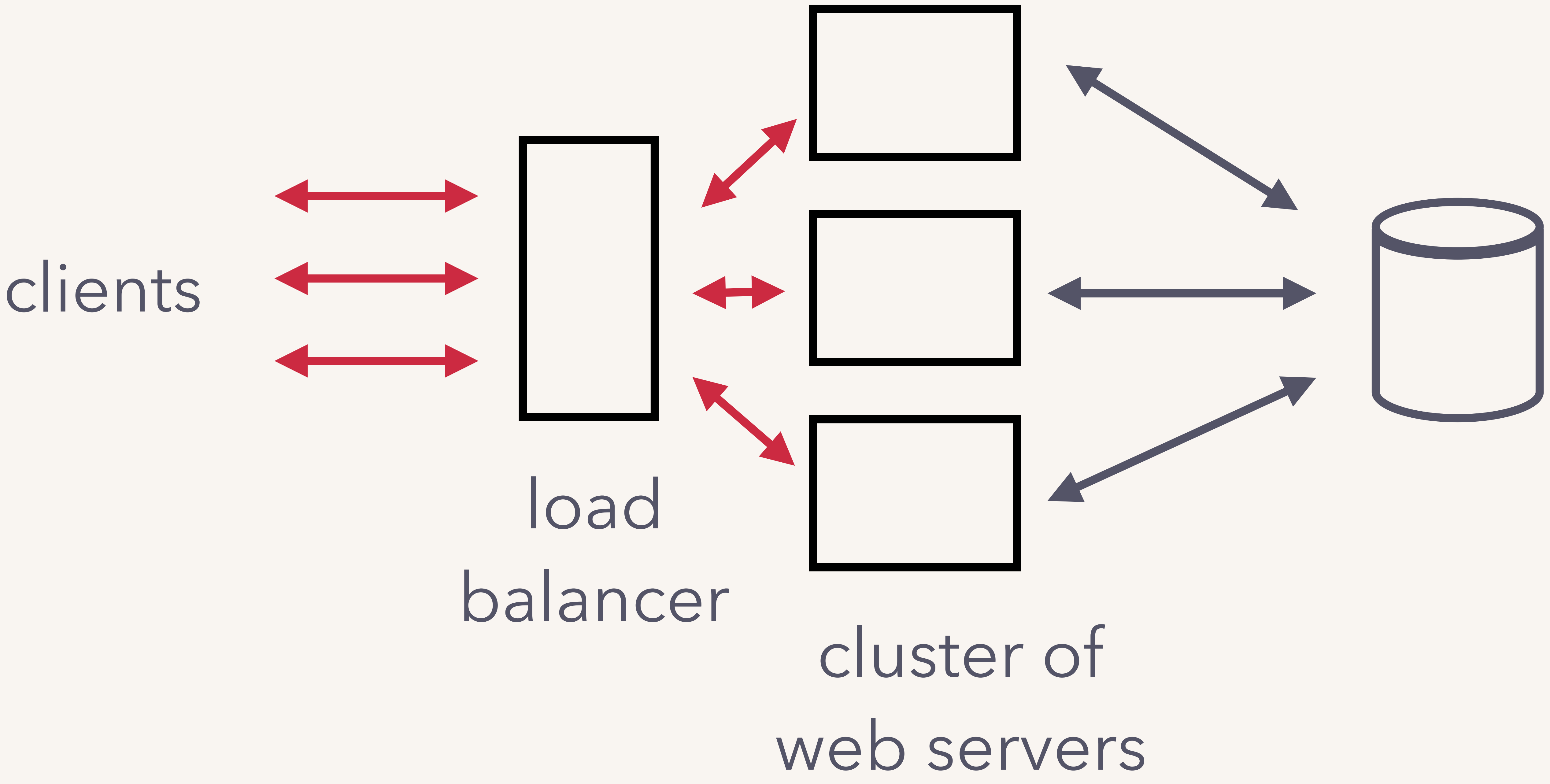
…but the system with real users **may not be one**!

So, load simulation might predict:

- lower response times than the actual system yields,
- better tolerance to request size variability,
- other differences you probably don't want to find out in production…

A couple neat papers on the topic, workarounds:
Open Versus Closed: A Cautionary Tale
How to Emulate Web Traffic Using Standard Load Testing Tools

a cluster of servers



"How many servers do we need to support a **target throughput**?" ← **capacity planning!**
while keeping response time the same

"How can we improve how the system **scales**?" ← **scalability**

"How many servers do we need to support a target throughput?"

while keeping response time the same

max throughput of a cluster of N servers = max single server throughput * N ?

no, systems don't scale linearly.

- contention penalty → αN

  due to serialization for shared resources.

  examples: database contention, lock contention.

"How many servers do we need to support a target throughput?"

while keeping response time the same

max throughput of a cluster of N servers = max single server throughput * N ?

no, systems don't scale linearly.

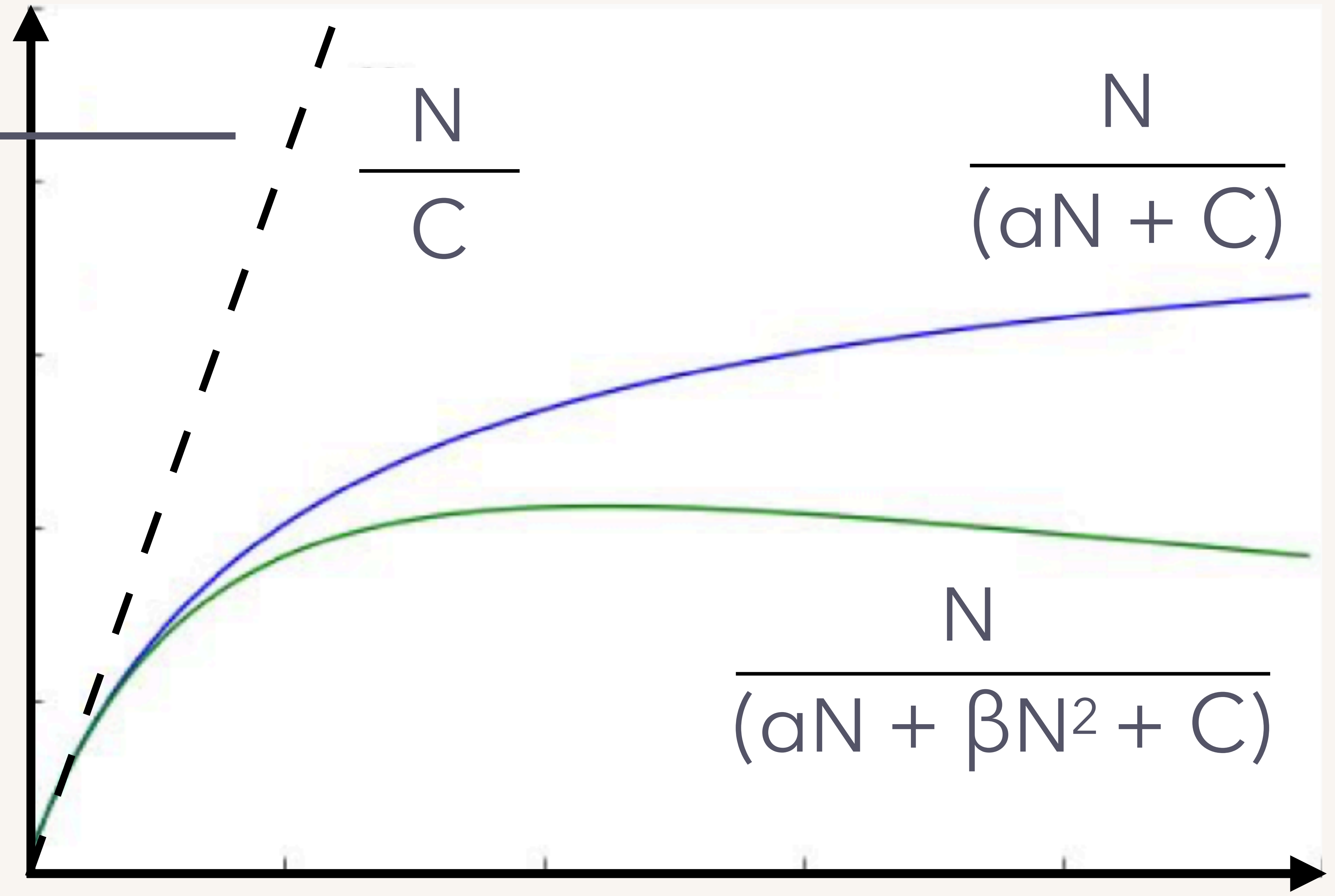
- contention penalty → αN
  due to serialization for shared resources.
  examples: database contention, lock contention.

- crosstalk penalty → βN^2
  due to coordination for coherence.
  examples: servers coordinating to synchronize mutable state.

Universal Scalability Law (USL)

$$\text{throughput of N servers} = \frac{N}{(\alpha N + \beta N^2 + C)}$$

linear scaling



contention

contention and crosstalk

“How can we improve how the system scales?”

Avoid **contention (serialization)** and **crosstalk (synchronization)**.

- smarter data partitioning, smaller partitions
  in Facebook’s TAO cache

- smarter aggregation
  in Facebook’s SCUBA data store

- better load balancing strategies: best of two random choices
- fine-grained locking
- MVCC databases
- etc.

stepping back

the role of performance modeling

most useful **in conjunction with** empirical analysis.

load simulation, experiments

modeling requires assumptions that may be difficult to practically validate.

but, gives us a **rigorous framework** to:

- **determine what experiments to run**
  run experiments needed to get data to fit the USL curve, response time graphs.

the role of performance modeling

most useful **in conjunction with** empirical analysis.

load simulation, experiments

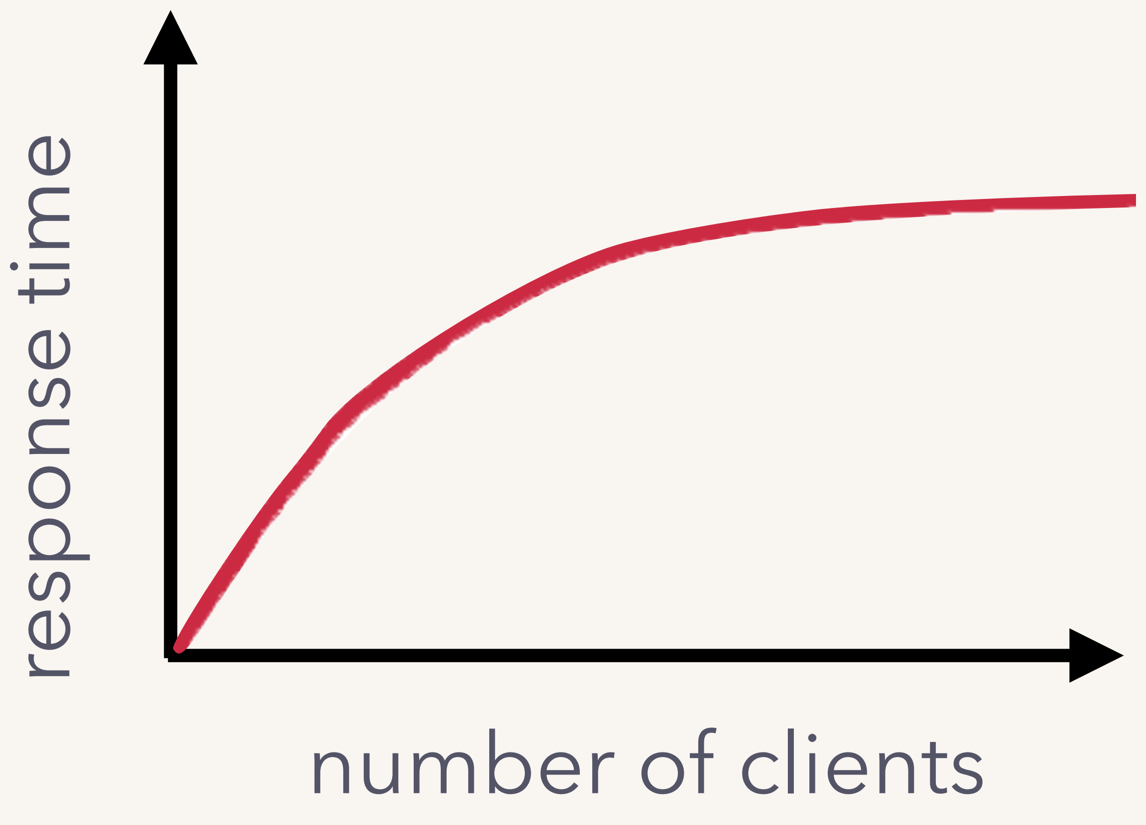
modeling requires assumptions that may be difficult to practically validate.

but, gives us a **rigorous framework** to:

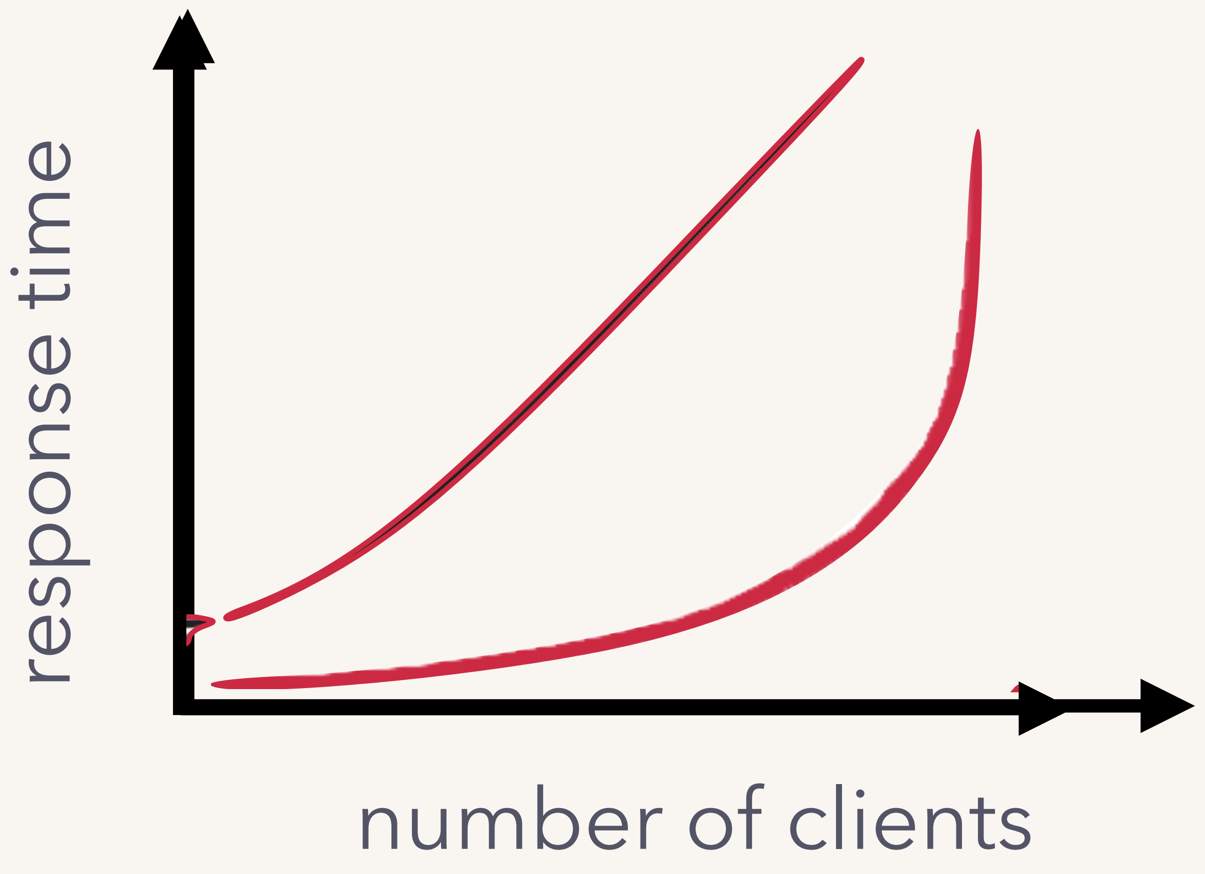
- **determine what experiments to run**
  run experiments needed to get data to fit the USL curve, response time graphs.

- **interpret and evaluate the results**
  load simulations predicted better results than your system shows

load simulation results with increasing number of virtual clients (N) = 1, …, 100



wrong shape
for response time curve!



should be
one of the two curves above

… load simulator hit a bottleneck.

the role of performance modeling

most useful **in conjunction with** empirical analysis.

load simulation, experiments

modeling requires assumptions that may be difficult to practically validate.

but, gives us a **rigorous framework** to:

- **determine what experiments to run**
  run experiments needed to get data to fit the USL curve, response time graphs.

- **interpret and evaluate the results**
  load simulations predicted better results than your system shows

- **decide what improvements give the biggest wins**
  improve mean service time, reduce service time variability, remove crosstalk etc.

References

Performance Modeling and Design of Computer Systems, Mor Harchol-Balter
Practical Scalability Analysis with the Universal Scalability Law, Baron Schwartz
Open Versus Closed: A Cautionary Tale
How to Emulate Web Traffic Using Standard Load Testing Tools
Queuing Theory, In Practice
Fail at Scale
Kraken: Leveraging Live Traffic Tests
SCUBA: Diving into Data at Facebook

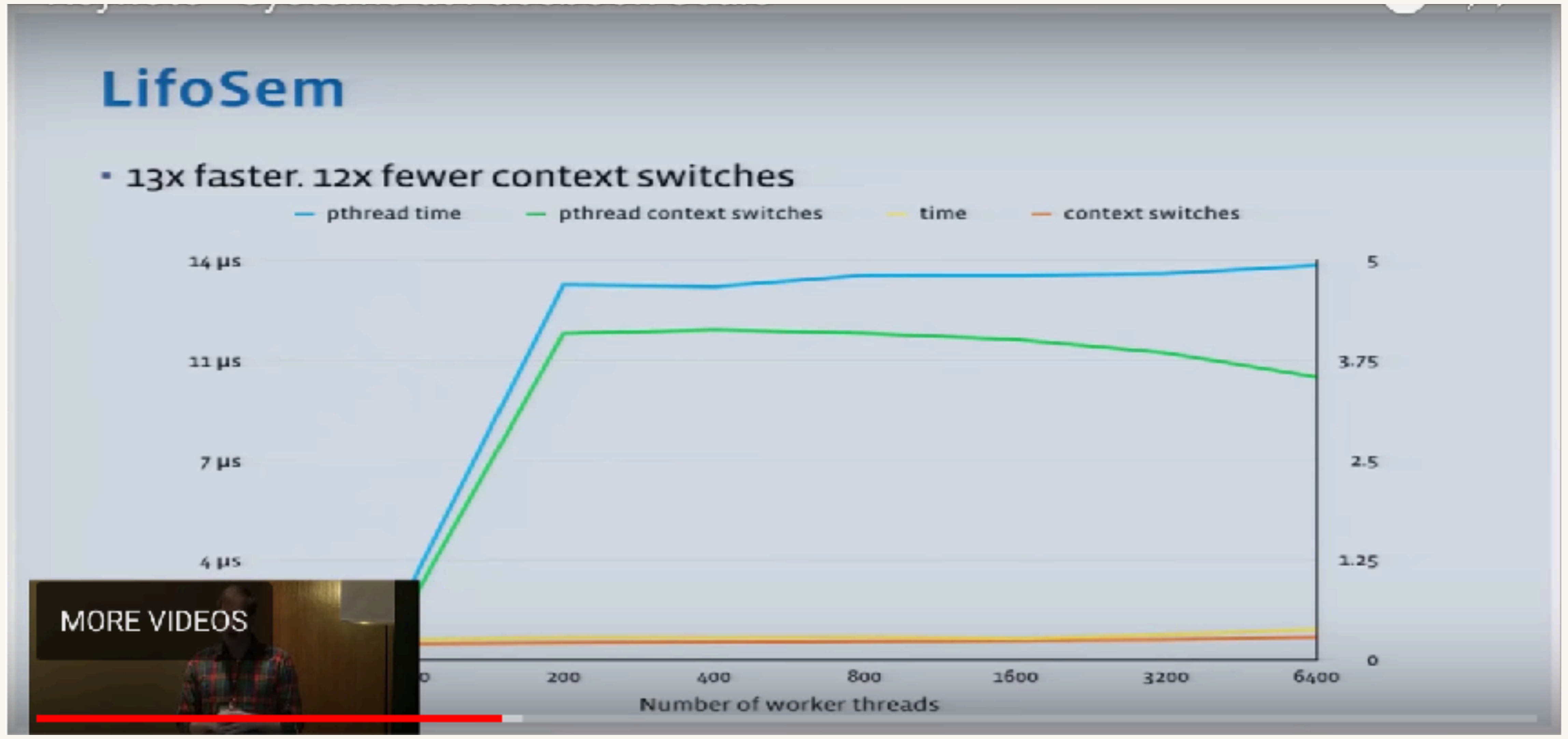
@kavya719

speakerdeck.com/kavya719/applied-performance-theory

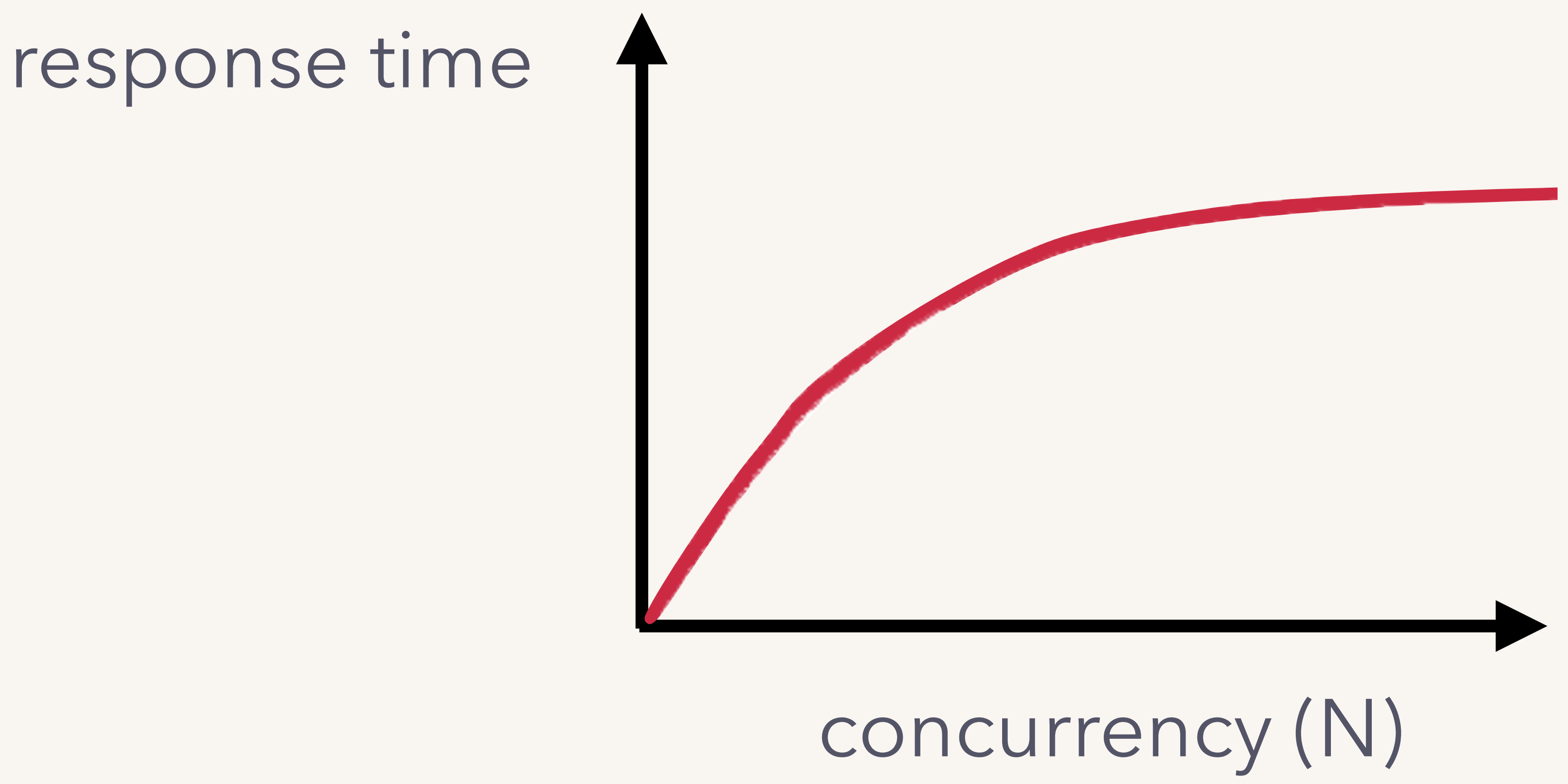
Special thanks to Eben Freeman for reading drafts of this

On CoDel at Facebook:
“An attractive property of this algorithm is that the values of M and N tend not to need tuning.
Other methods of solving the problem of standing queues, such as setting a limit on the number of items in the queue or setting a timeout for the queue, have required tuning on a per-service basis.
We have found that a value of 5 milliseconds for M and 100 ms for N tends to work well across a wide set of use cases. “

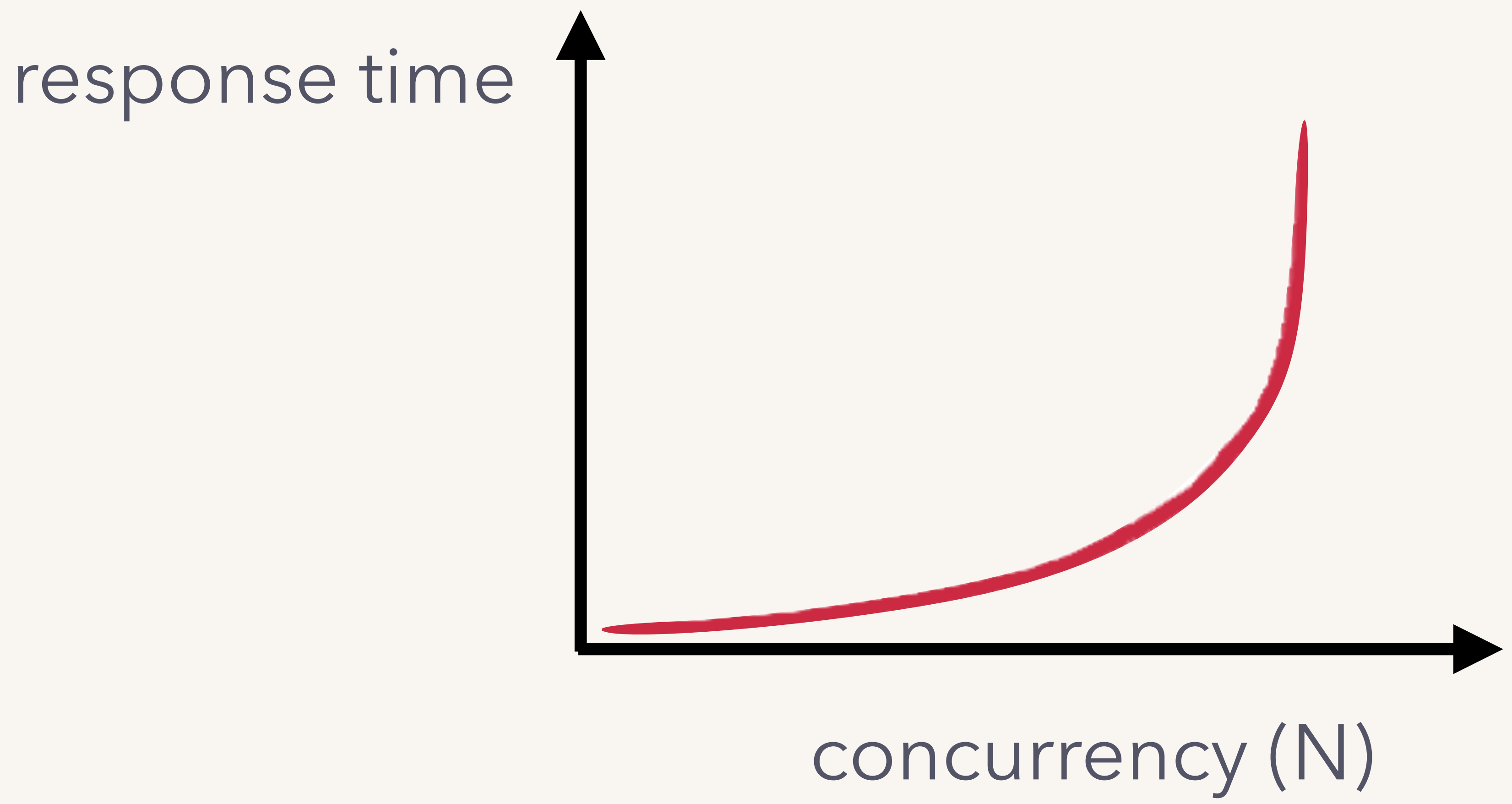
Using LIFO to select thread to run next, to reduce mutex, cache trashing and context switching overhead:



number of virtual clients (N) = 1, …, 100



wrong shape
for response time curve!



should be

… load simulator hit a bottleneck!

utilization = throughput * service time (Utilization Law)

"busyness"

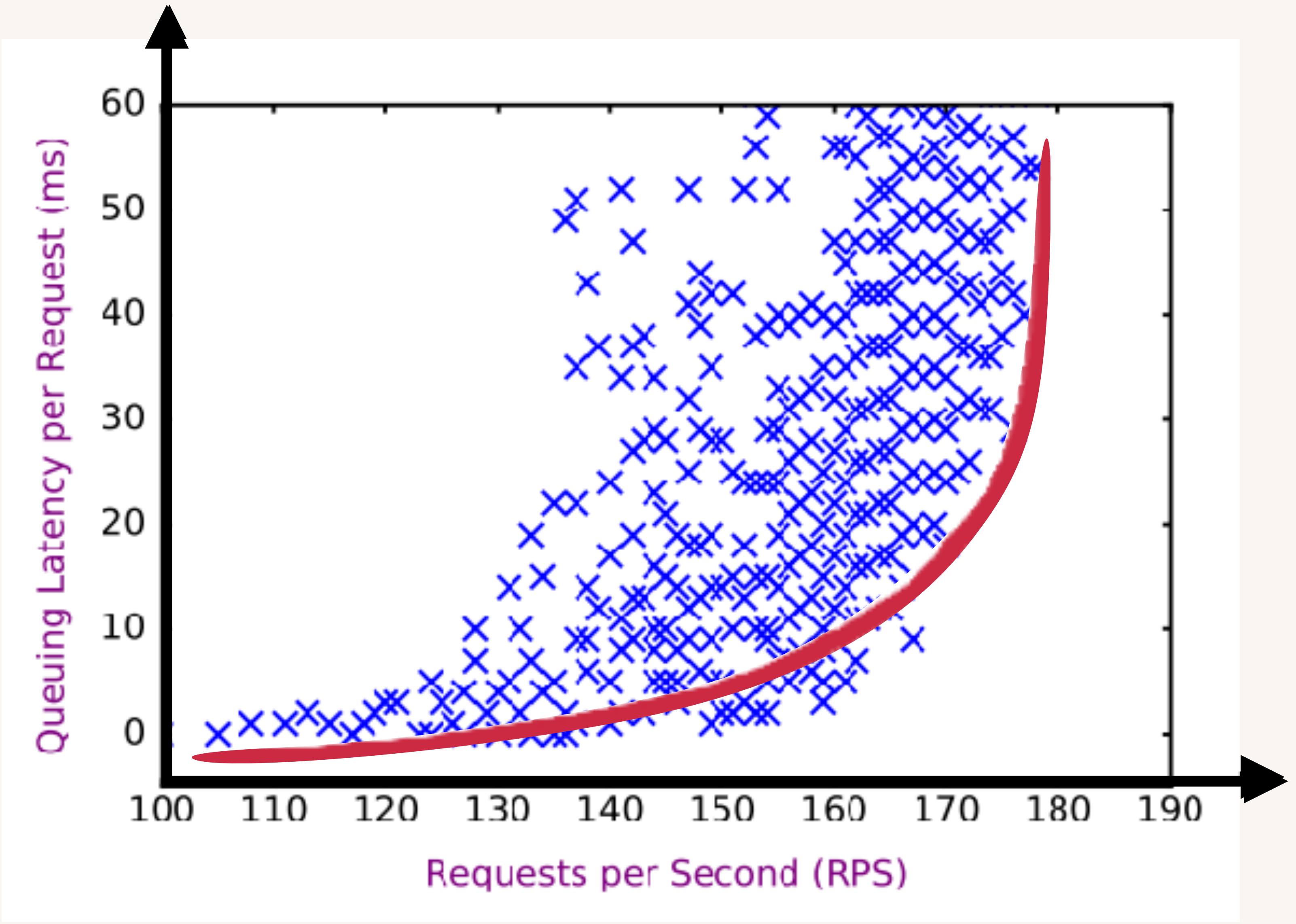
throughput increases

↓

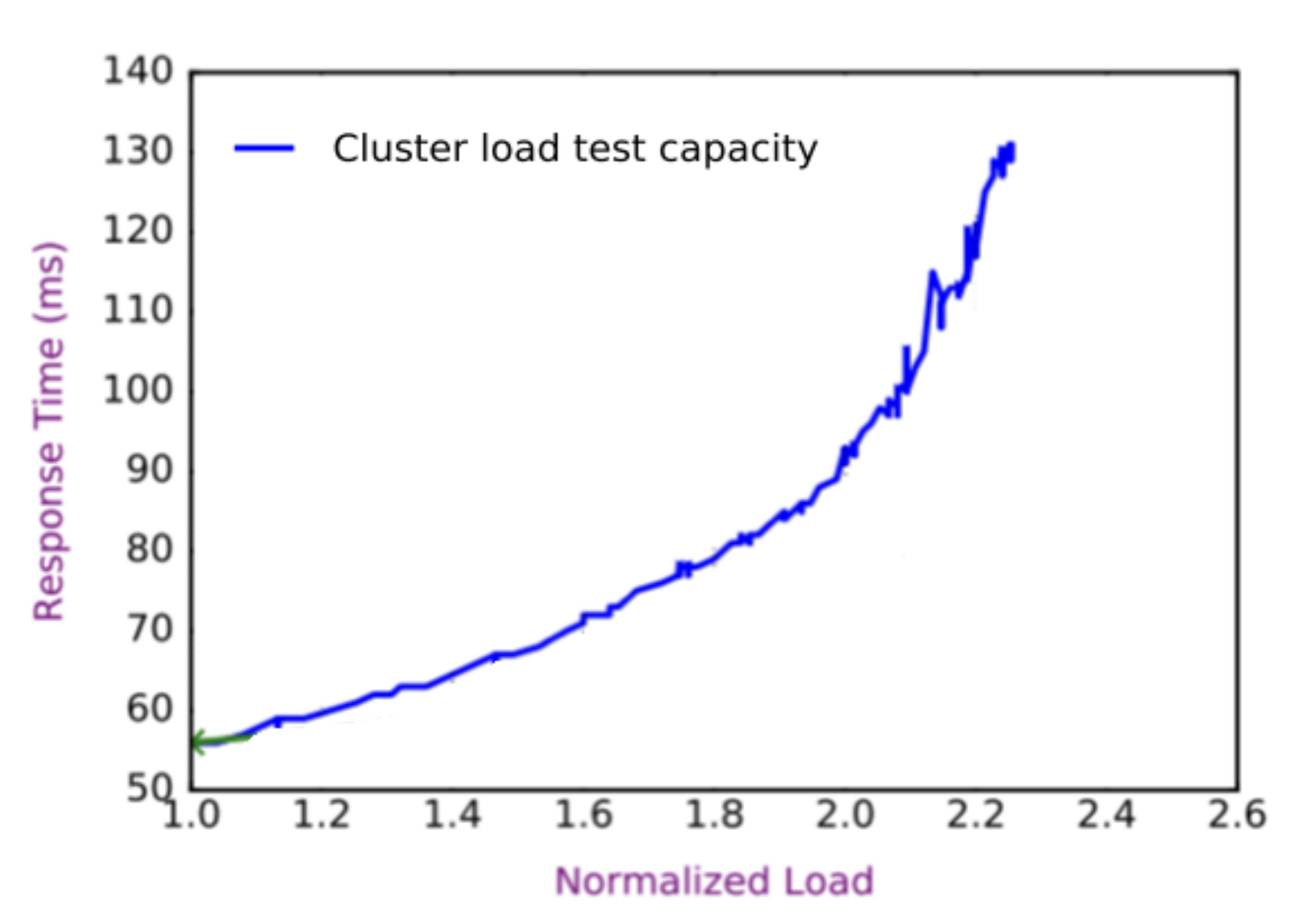
utilization increases

↓

queueing delay increases (non-linearly);
so, **response time**.

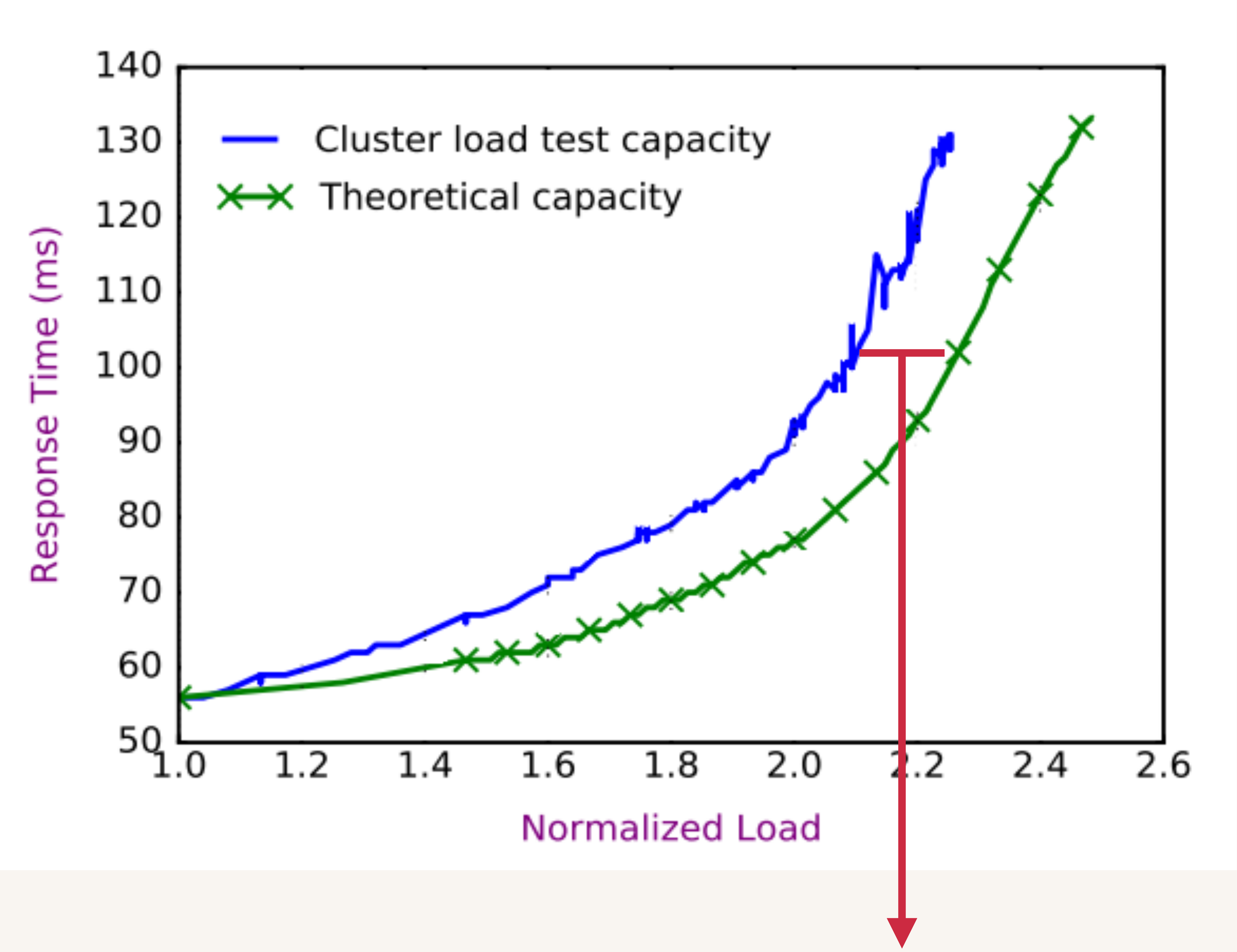


Facebook sets target cluster capacity = 93% of theoretical.



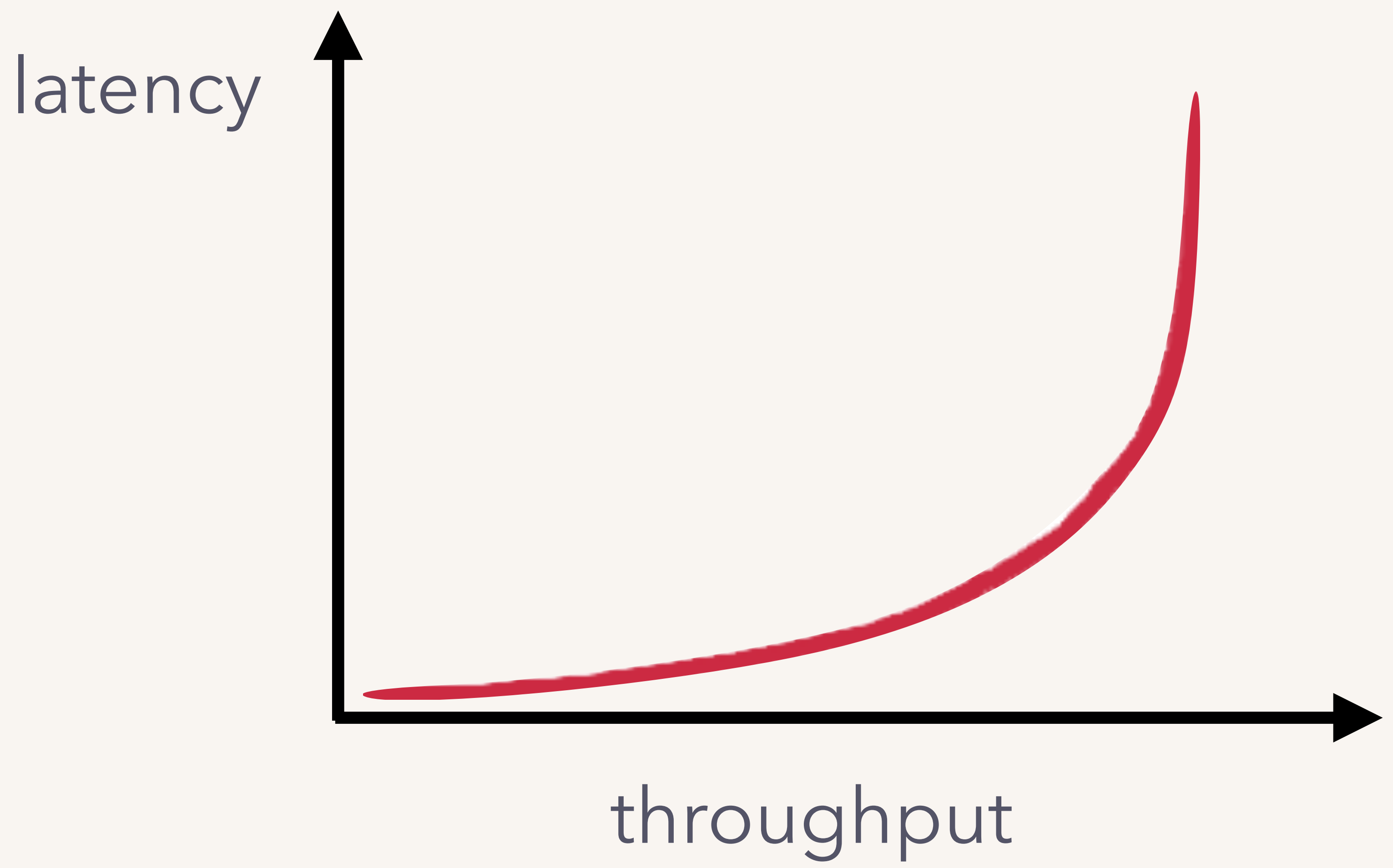
…is this good or is there a bottleneck?

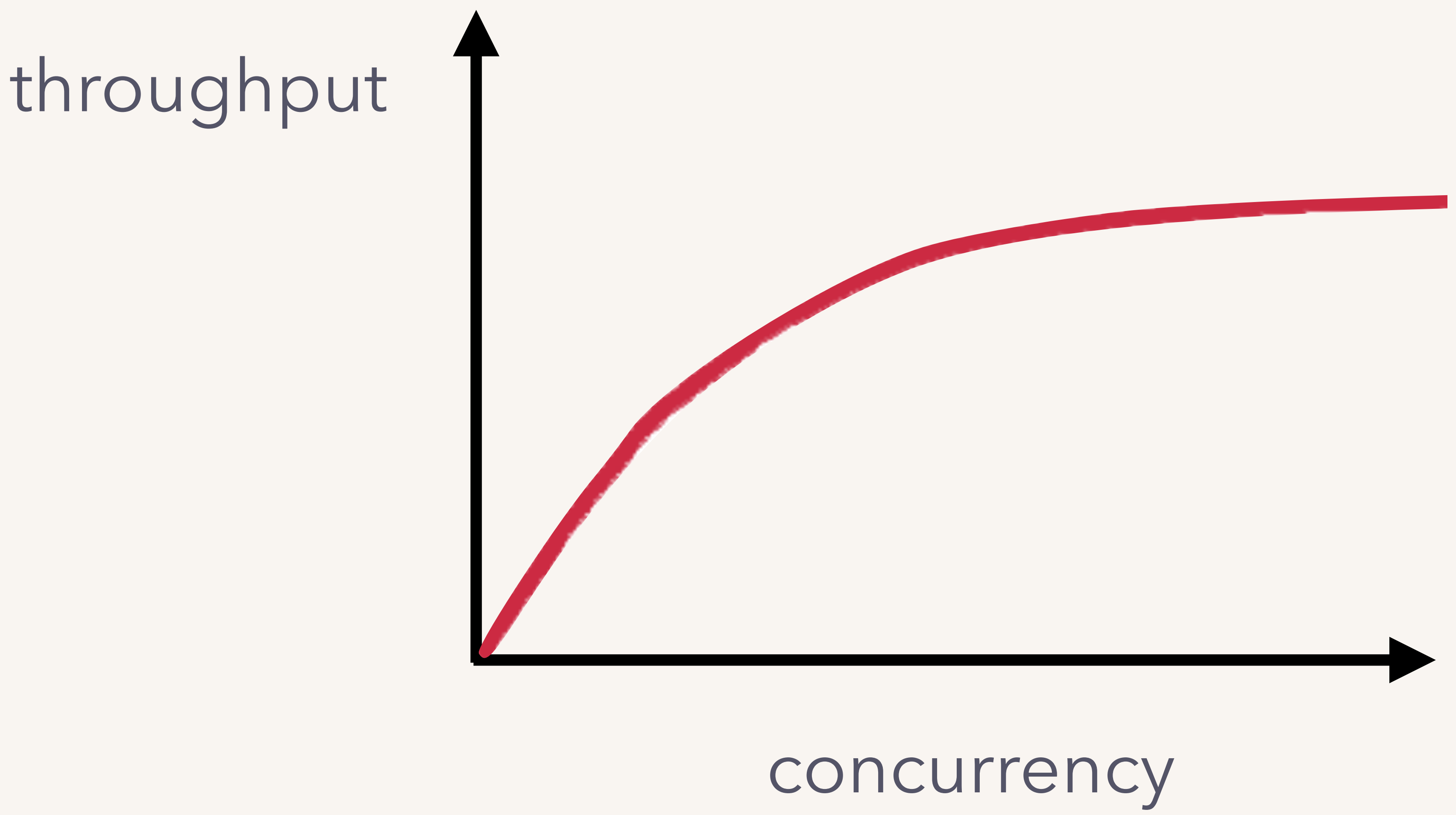
Facebook sets target cluster capacity = 93% of theoretical.



cluster capacity is ~90% of theoretical,

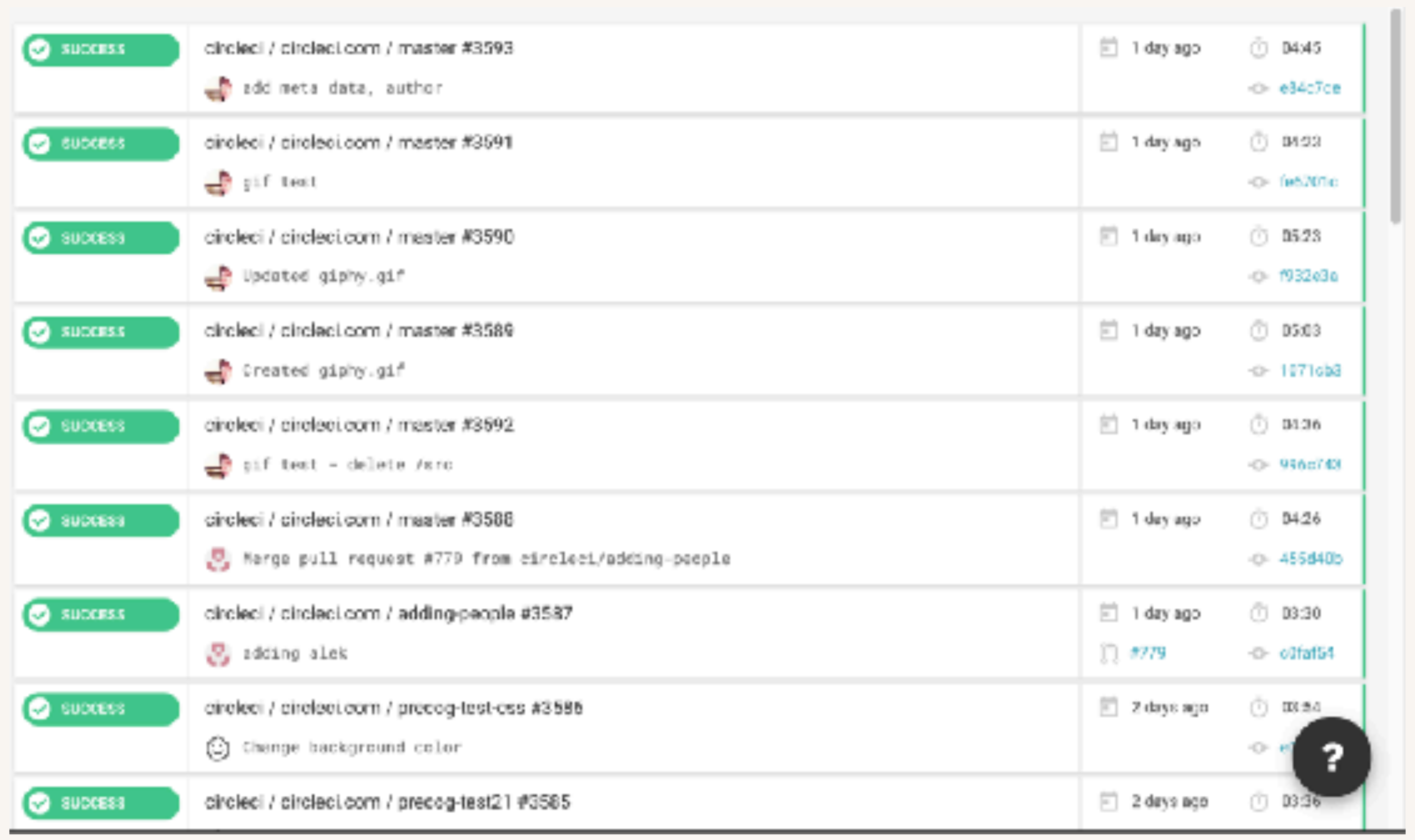
so there's a bottleneck to fix!





non-linear responses to load

non-linear scaling



microservices:
systems are complex

continuous deploys:
systems are in flux

load generation

need a **representative workload**.

profile (read, write requests)
arrival pattern including traffic bursts

…use **live traffic**.

capture and replay
traffic shifting

traffic shifting

adjust weights that control load balancing,
to increase the fraction of traffic to a cluster, region, server.

edge weight || cluster weight || server weight ||