Architecting for Scale

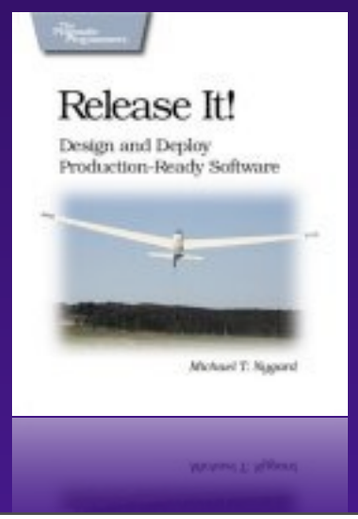
About the Author

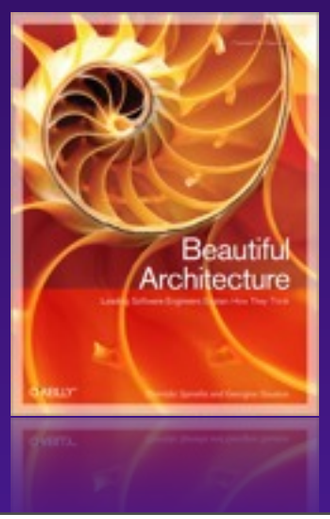
Michael Nygard

Application Developer/Architect – 20 years

Web Developer – 16 years

IT Operations – 8 Years





Agenda

Domain of Applicability

Agenda

Domain of Applicability

Technical Foundations

Amdahl's Law

The Universal Scalability Law

Agenda

Domain of Applicability

Technical Foundations

- Amdahl's Law
- The Universal Scalability Law

Reducing Contention

Reducing Coherence

Agenda

- Domain of Applicability
- Technical Foundations
  - Amdahl's Law
  - The Universal Scalability Law
- Reducing Contention
- Reducing Coherence
- Some Specific Techniques

Questions Wide of the Mark

Bad questions about scalability abound:

"Is it scalable?"

Questions Wide of the Mark

Bad questions about scalability abound:

"Will *technology X* scale?"

Questions Wide of the Mark

Questions Wide of the Mark

Bad questions about scalability abound:

My personal favorite,

Questions Wide of the Mark

Bad questions about scalability abound:

My personal favorite,

“Does Ruby on Rails scale better than XML?”

Questions Wide of the Mark

Think of scalability like a function:

It's a float, not a boolean

It depends on architecture, workload, *and* technology.

Functions exist in specific technical domains.

Comparisons between domains have no meaning.

Nodes

10000

1000

100

10

Medium Scale

App server centric
Master Relational DB
Point to point integration
Some messaging
Some synchronous calls
Manual deployment
Low to moderate use of CDN

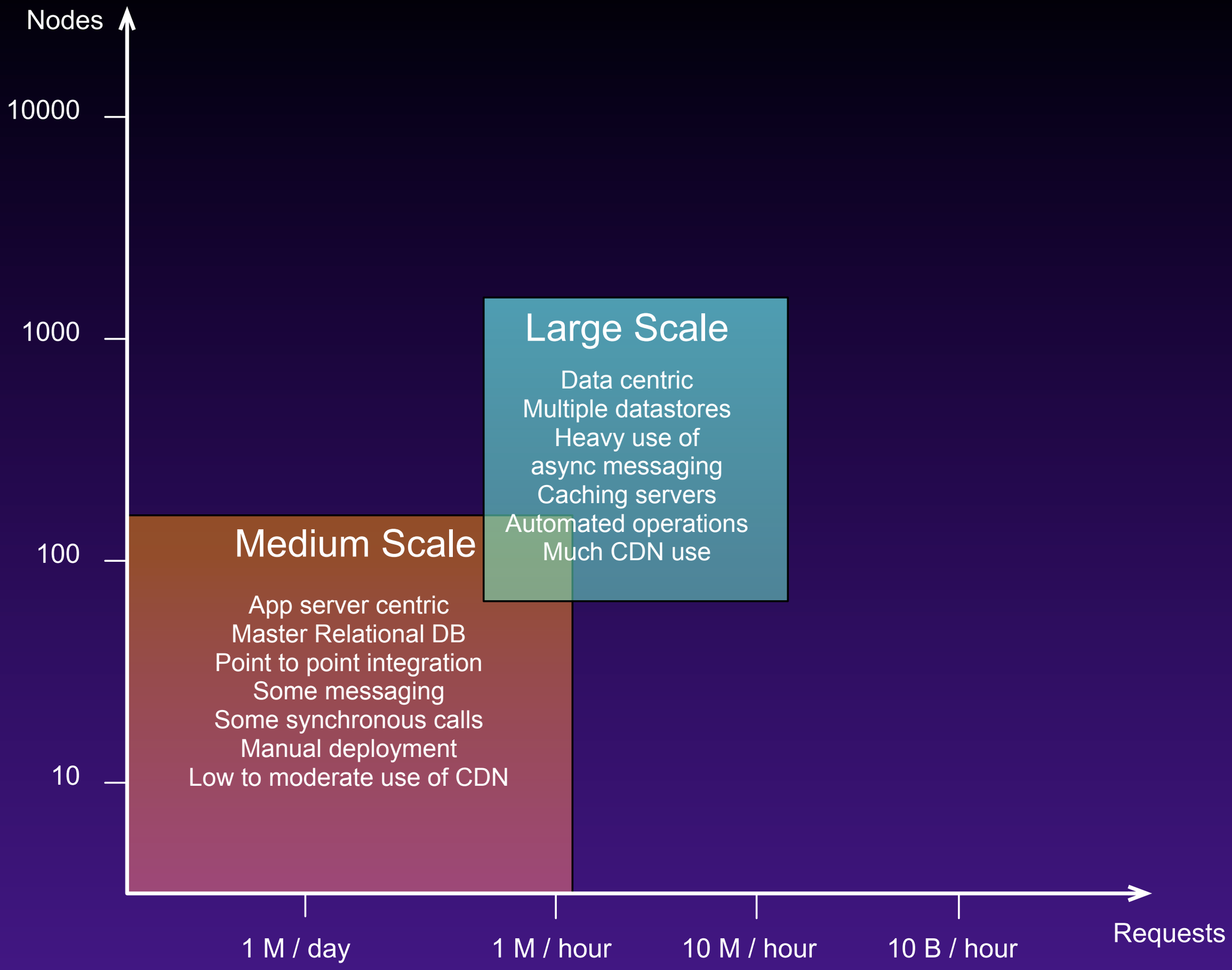
1 M / day

1 M / hour

10 M / hour

10 B / hour

Requests


Nodes
10000
1000
100
10
Large Scale
Data centric
Multiple datastores
Heavy use of
async messaging
Caching servers
Automated operations
Much CDN use
Medium Scale
App server centric
Master Relational DB
Point to point integration
Some messaging
Some synchronous calls
Manual deployment
Low to moderate use of CDN
1 M / day
1 M / hour
10 M / hour
10 B / hour
Requests

Nodes

10000

1000

100

10

Extreme Scale

Operations centric
Distributed & non-relational data storage
Ubiquitous caching
Ubiquitous partitioning
Sharding
Self-managing infrastructure
Build own CDN

Large Scale

Data centric
Multiple datastores
Heavy use of async messaging
Caching servers
Automated operations
Much CDN use

Medium Scale

App server centric
Master Relational DB
Point to point integration
Some messaging
Some synchronous calls
Manual deployment
Low to moderate use of CDN

1 M / day

1 M / hour

10 M / hour

10 B / hour

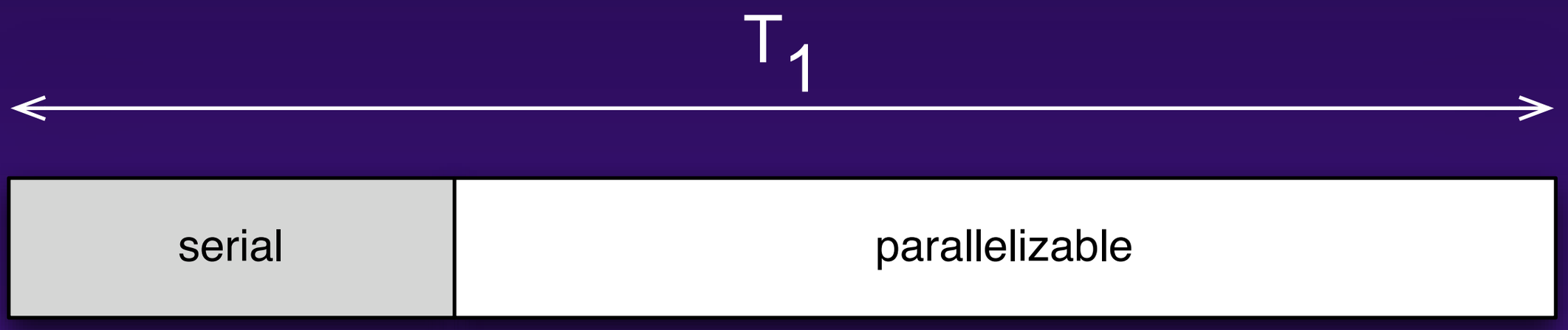
Requests

Technical Foundation

Defining Scalability

Purely technical definition:

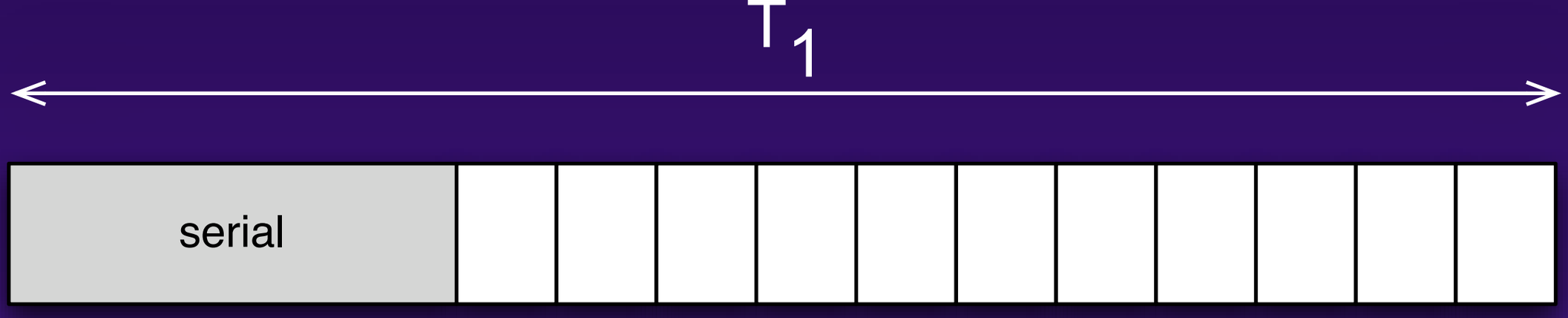
Reduction in elapsed processor time due to parallelization of workload



Defining Scalability

Purely technical definition:

Reduction in elapsed processor time due to parallelization of workload



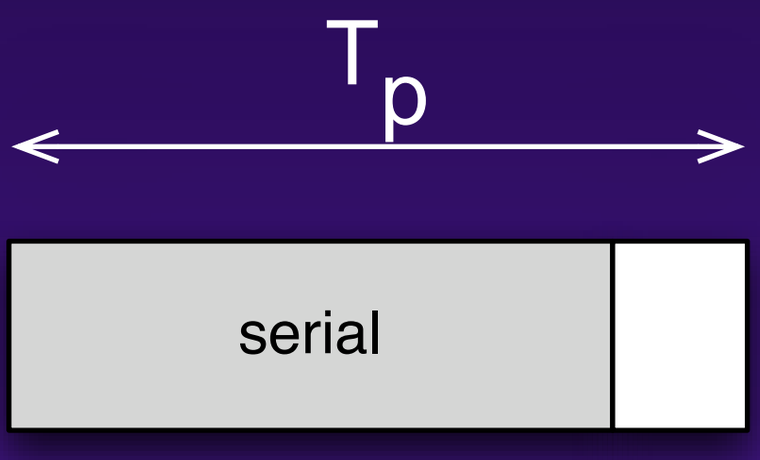
“Serial Fraction” = σ

“Parallel Fraction” = (1 - σ)
Divide into *p* subtasks

Defining Scalability

Purely technical definition:

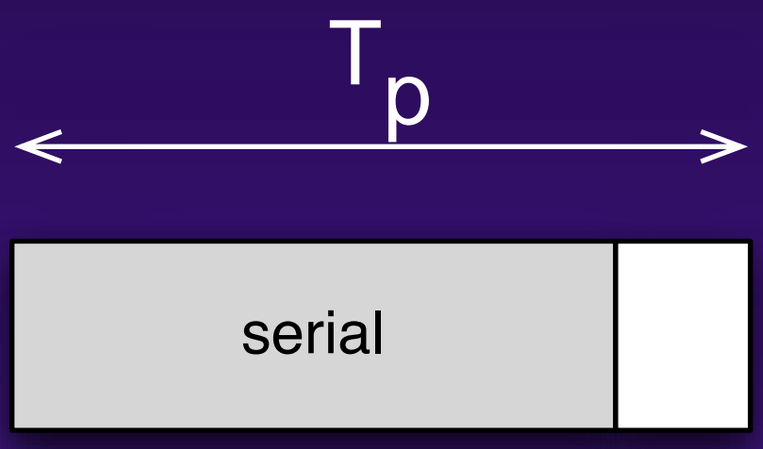
Reduction in elapsed processor time due to parallelization of workload



Defining Scalability

Purely technical definition:

Reduction in elapsed processor time due to parallelization of workload



$$T_p = \sigma T_1 + \frac{(1-\sigma)T_1}{p}$$

Defining Scalability

Speedup “S” is ratio of serial processing time to parallel time.

$$S(p) = \frac{T_1}{1 + \sigma(p - 1)}$$

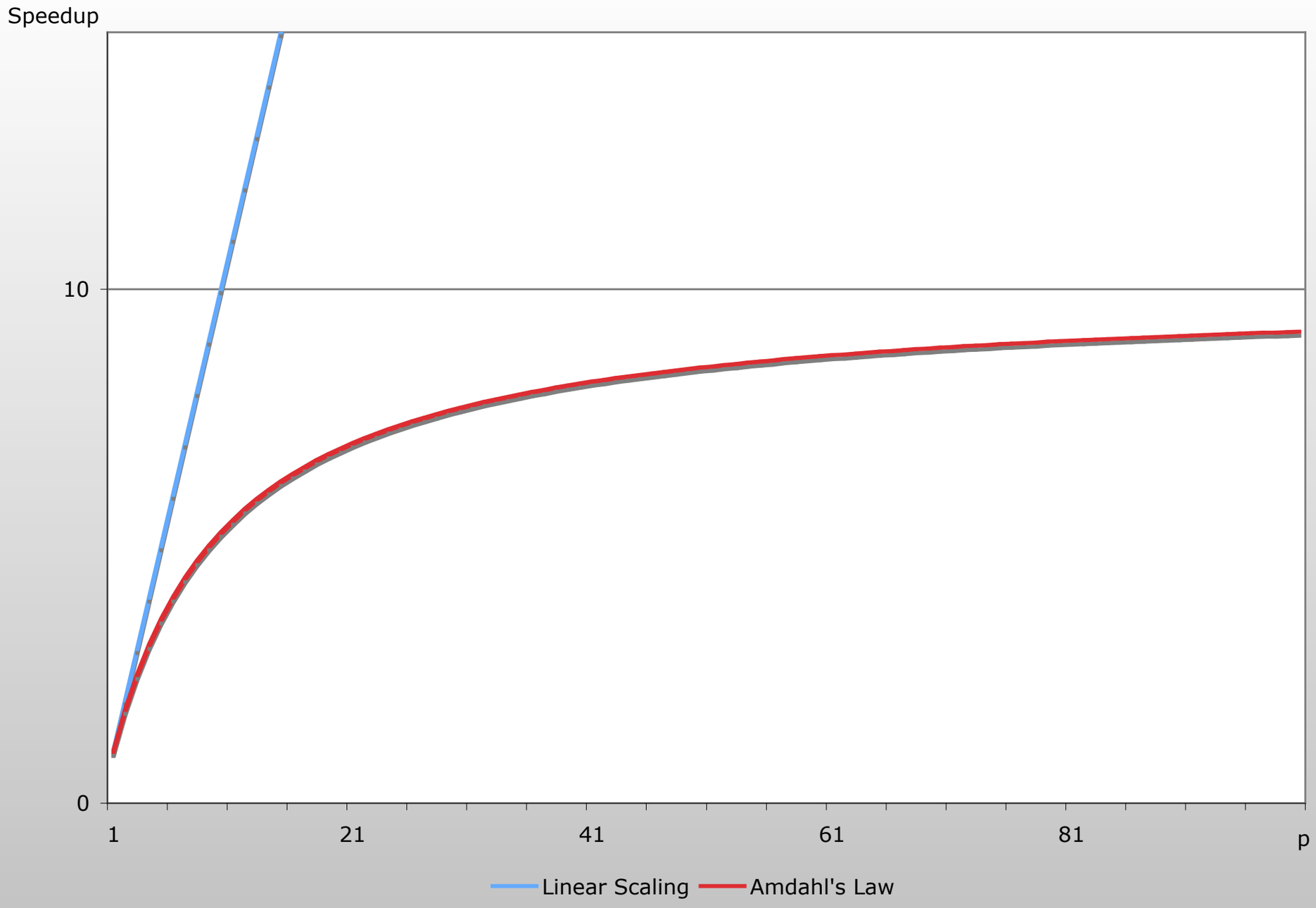
Defining Scalability

Speedup “S” is ratio of serial processing time to parallel time.

$$S(p) = \frac{T_1}{1 + \sigma(p - 1)}$$

Amdahl’s Law

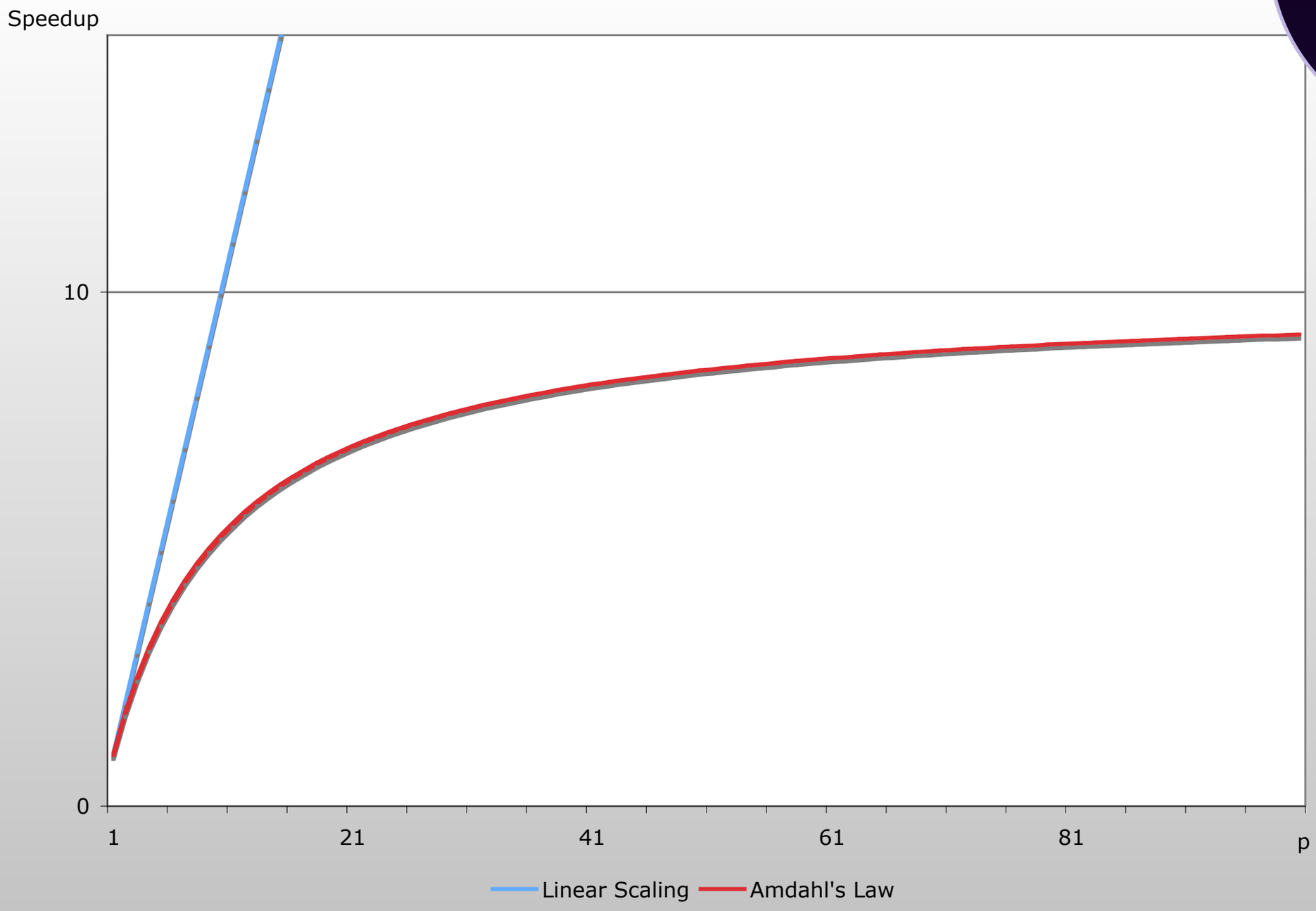
Amdahl's Law versus Linear Scaling



$\sigma = 10\%$

Amdahl's Law versus Linear Scaling

Diminishing Returns



$\sigma = 10\%$

That's pretty bad.

Unfortunately, it's also optimistic.

Contention and Coherency

- Amdahl's Law accounts for contention on serial resources.
- We also need to account for the effect of coherency, time needed to agree on state across multiple processes

Universal Scalability Law

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)}$$

σ = Contention

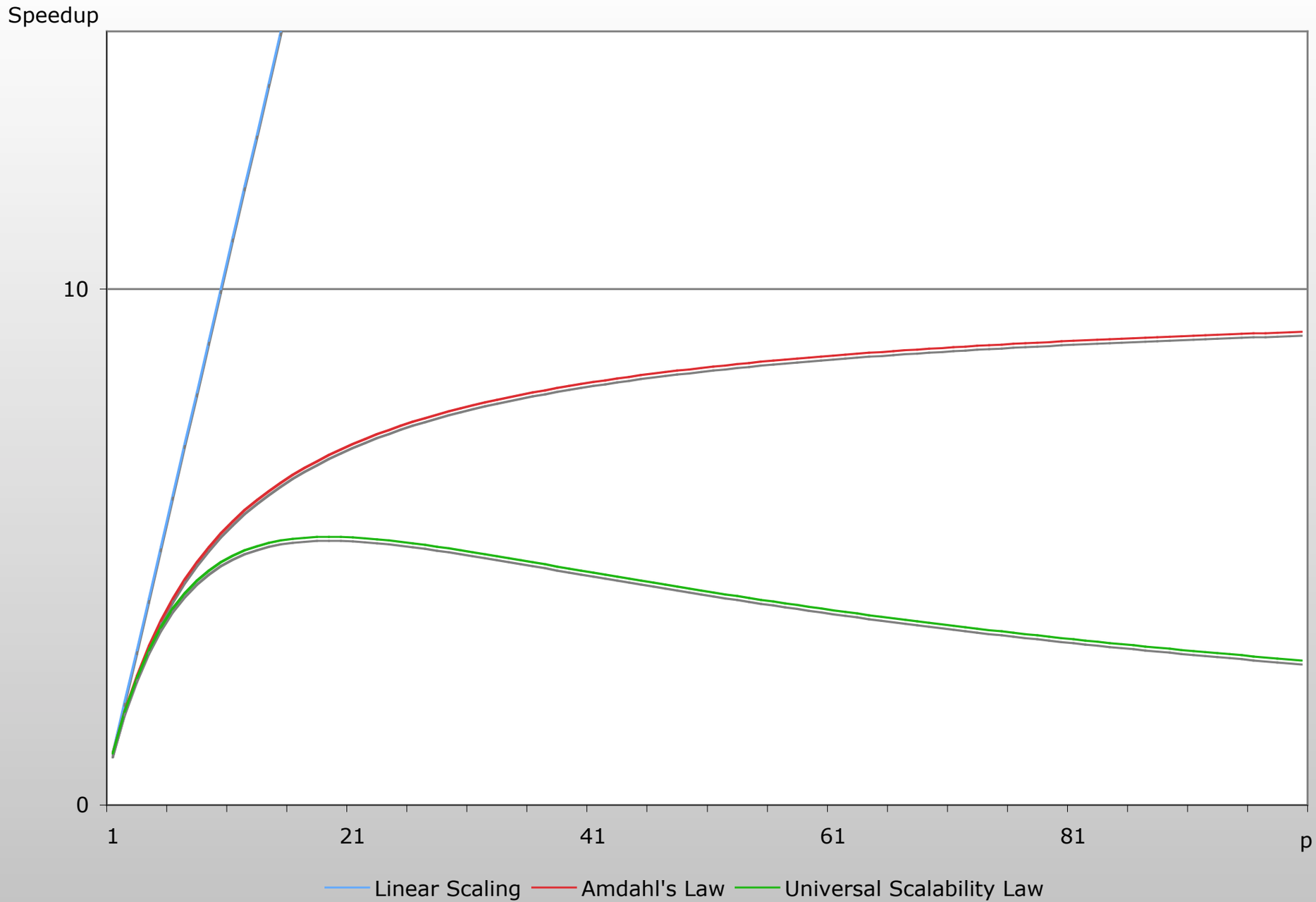
Degree of serialization on shared writable data, contention for resources.

κ = Coherency

Penalty for maintaining consistency of shared writable data.

From “Guerilla Capacity Planning”, by Dr. Neil Gunther.

Amdahl's Law versus Linear Scaling

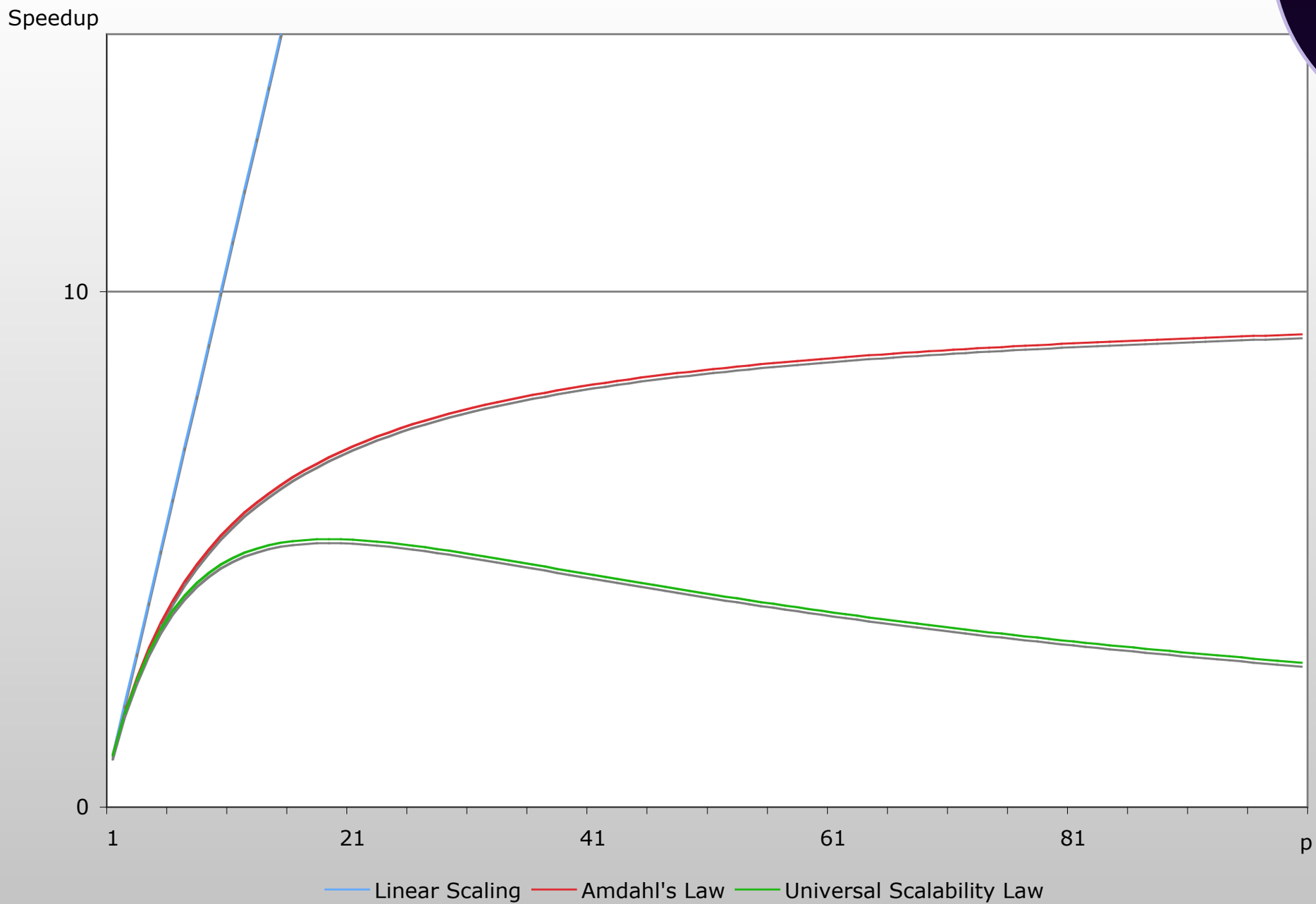


$$\sigma = 10\%$$

$$\kappa = 0.0025$$

Amdahl's Law versus Linear Scaling

Capacity Maximum, Negative Returns



$$\sigma = 10\%$$

$$\kappa = 0.0025$$

How shall we respond to this?

General Scalability Principles

Improving Scalability

There are only three strategies:

1. Reduce p
2. Reduce σ
3. Reduce κ

Why isn't

"improve performance"

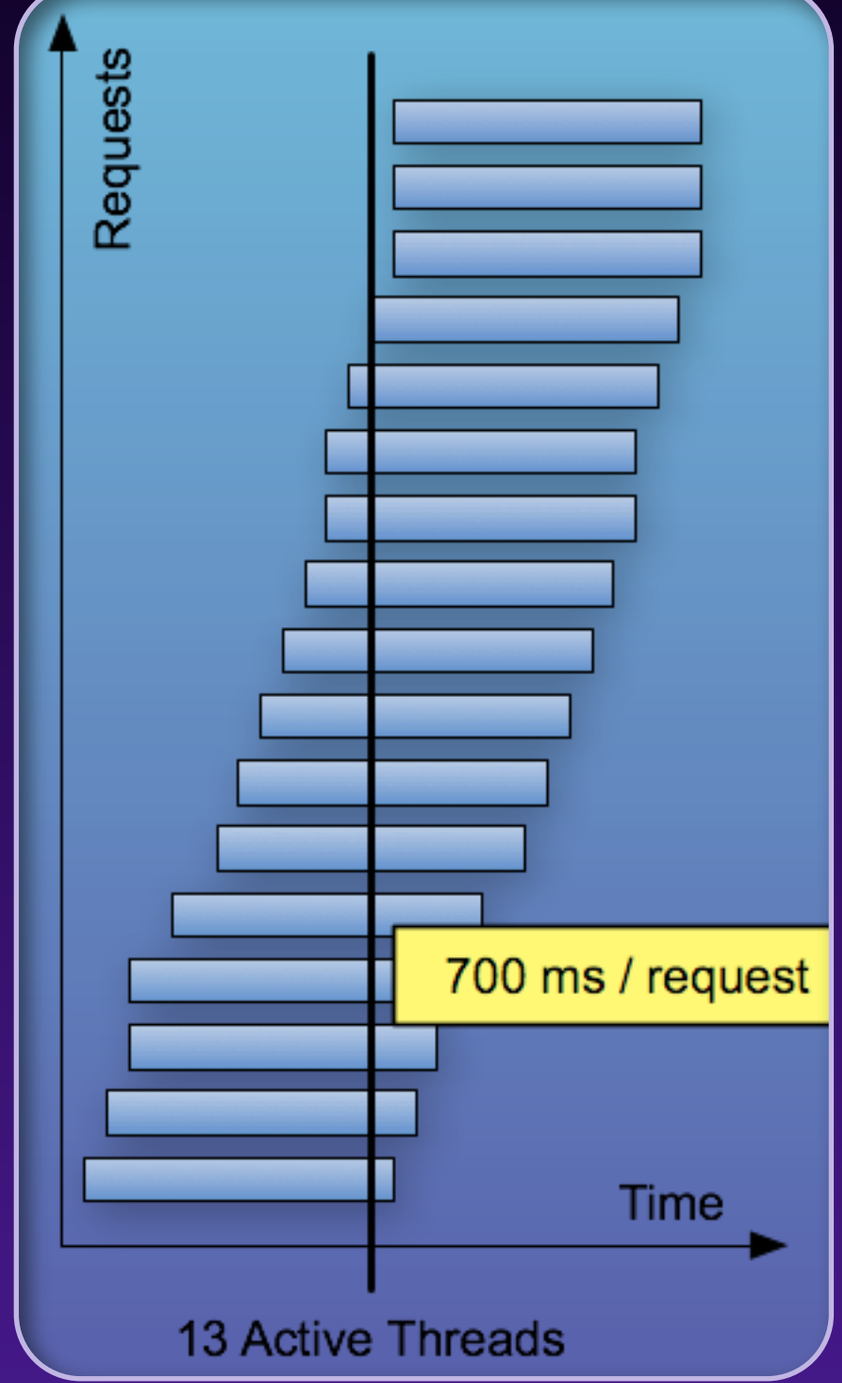
on that list?

A Brief Aside About Performance

- Performance determines capacity for a given set of resources.
- Scalability measures capacity increase for additional resources.
- Increasing performance reduces your *need* for scalability, but by itself, does nothing to benefit scalability.

The Effect of Performance on Capacity

- Each request consumes resources during processing.
- Once the request completes, those resources can be used for new requests.
- The shorter the response time, the greater a system's capacity.

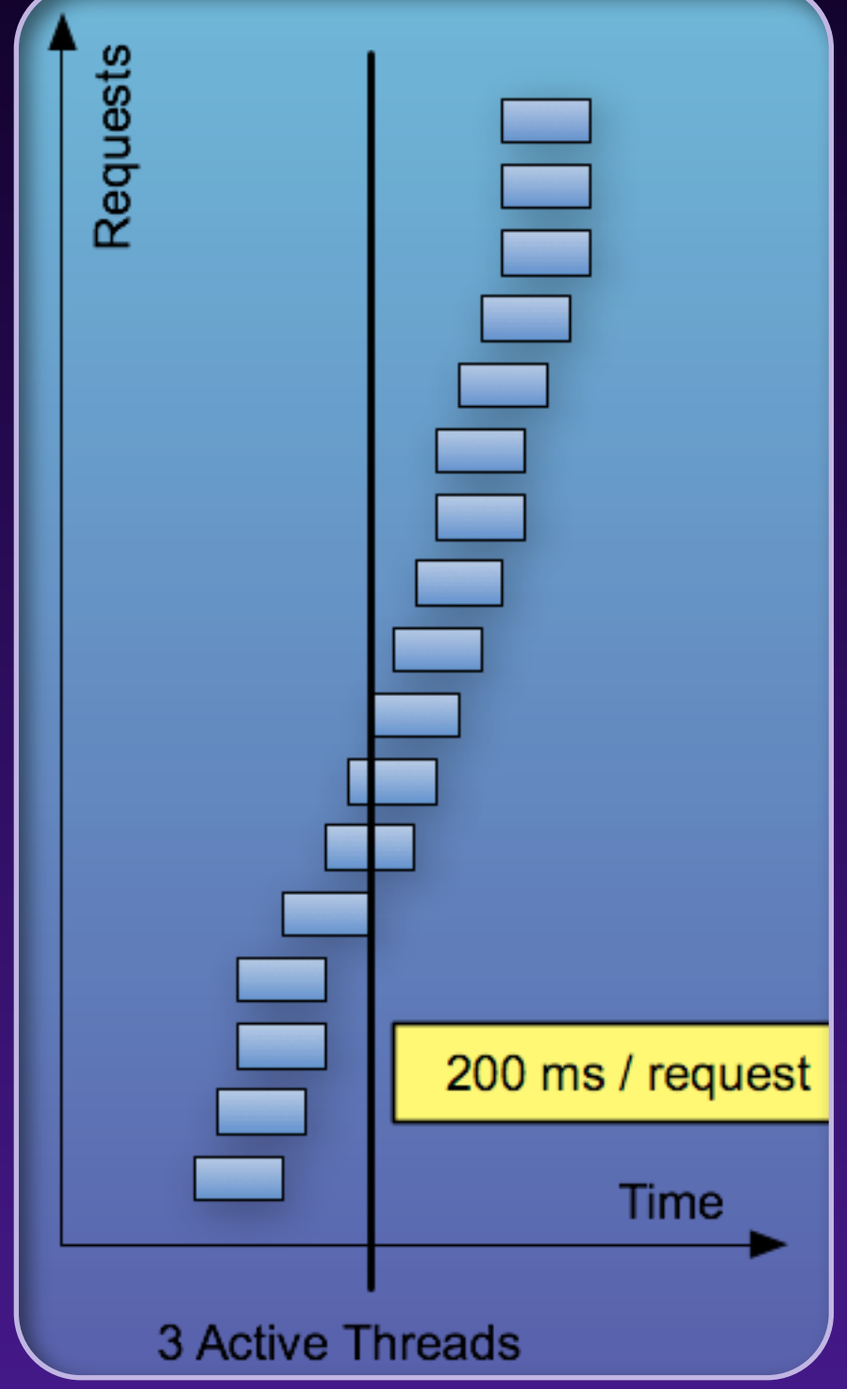


The Effect of Performance on Capacity

Corollary:

Slower response time means you need more hardware to serve the same capacity.

Faster response time means more capacity on the same hardware.



Reducing p

Are you suggesting that we become more scalable by *reducing* the number of computers?

Partitioning

“If you can't split it, you can't scale it.”

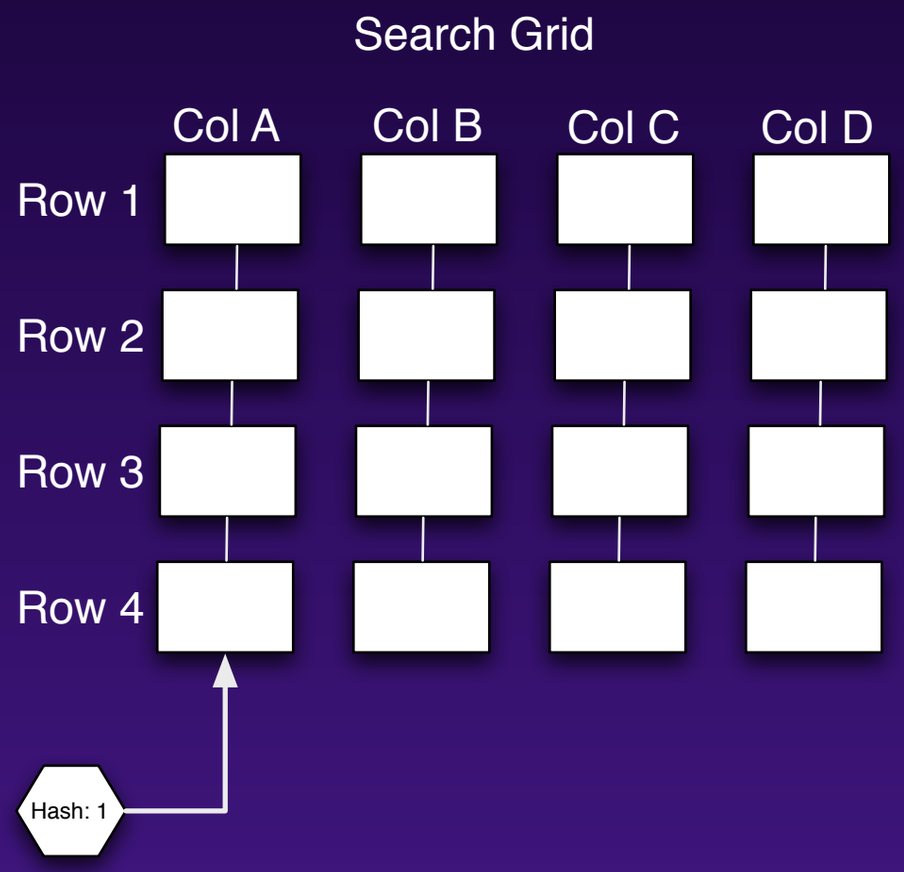
–Randy Shoup, eBay

Horizontal Partitioning

Dispatch workload according to attributes of the task.

Example:

Hash an item ID into 4 bins, each served by a separate cluster.



Best applied by application logic.

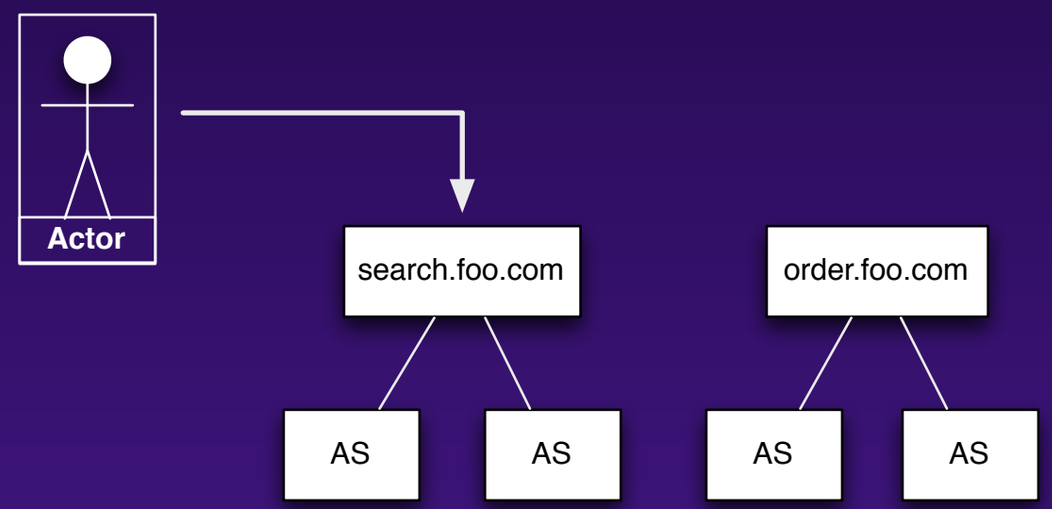
1. At the callout from an app.
2. By a dispatching proxy.

Functional Partitioning

Dispatch transaction types to different clusters.

Example:
- Availability lookups handled separately from reservations.

Best applied as close to the user as possible:
- Client side
- Load balancer/content switch



Geographic Partitioning

Dispatch workload to nearby clusters.

Example:

Akamai DNS responds with nearest point-of-presence.

“Nearby” in network terms means lowest latency.

Shortens transmission delays (inherently serial) due to the effect of latency on bandwidth.

The Key to Partitioning

Partitioning strategies all assume no cross-cluster dependencies on shared data.

Shared writable data requires serialized access.

(Higher σ)

Reducing σ

Network Latency Effects

Slow client connections cause TCP stalls.

TCP stalls keep sockets open on the web server and consume RAM for buffered responses.

In case of poor connectors, stalled web servers will cause app server to stall with full TCP write buffers.

Solutions to Network Latency

Solutions to Network Latency

Reverse proxy with lots of RAM

Solutions to Network Latency

Reverse proxy with lots of RAM

Web accelerator (F5, Cisco, etc.)

Solutions to Network Latency

Reverse proxy with lots of RAM

Web accelerator (F5, Cisco, etc.)

Content Delivery Network (Akamai, Limelight)

Solutions to Network Latency

Reverse proxy with lots of RAM

Web accelerator (F5, Cisco, etc.)

Content Delivery Network (Akamai, Limelight)

Smaller responses.

Caching

Every form of caching is built to reduce serialization time.

- Caching proxies
- App server caching
- Cache servers

A poorly sized or tuned cache can cause *more* contention, though. Monitor accordingly.

Publishing

Publishing static assets reduces both serialization and coherency requirements.

Static content is inherently parallel!

Reducing κ

Brewer's Conjecture

Eric Brewer, UC Berkeley

Choose at most two:

- Consistency
- Availability
- Partition-tolerance

Que pasa?

Consistency:

There exists a total ordering on all operations, and all nodes in the system agree on that ordering at every point in time.

I.e., changes to system state are Atomic, Consistent, Isolated, and Durable.

Que pasa?

Availability:

Every request received by a non-failing node must result in a response. (Every algorithm must terminate.)

Que pasa?

Partition-tolerance:

The network may lose arbitrarily many messages from any subset of nodes to any other subset of nodes.

Formal definitions from Gilbert, Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services” ACM SIGACT News, 2002.

Pick Two

Consistent & Available	Consistent & Partitionable	Available & Partitionable
Partitioning is not allowed. (But tell me again how you propose to prevent it?)	Consistency can only be guaranteed if the service is unavailable during partitions. Otherwise, one subset will see a different history than the other.	We maintain availability in the face of partition by allowing different subsets to report different histories. “Agreement” protocols are therefore forbidden.

Reality Bites

Like Heisenberg's Uncertainty Principle, or Gödel's Theorem, we'd like to pretend that Brewer's Conjecture doesn't exist.

- We cannot choose to eliminate partitions.
- We *must* choose consistency or availability.
- I'll assume that availability is paramount.

Database Transactions Require Agreement

ACID properties demand agreement by all nodes at all times.

Therefore, ACID databases inherently select “Consistency”.

Does this mean we have to abandon transactions?

Data Without Transactions?

Depends on your scale. There may be other ways to reduce κ without giving up transactions.

Example: In-memory data grid

1. App writes to cache server: local, fast, no κ
2. Cache server writes through to DB asynchronously: incurs coherence penalty

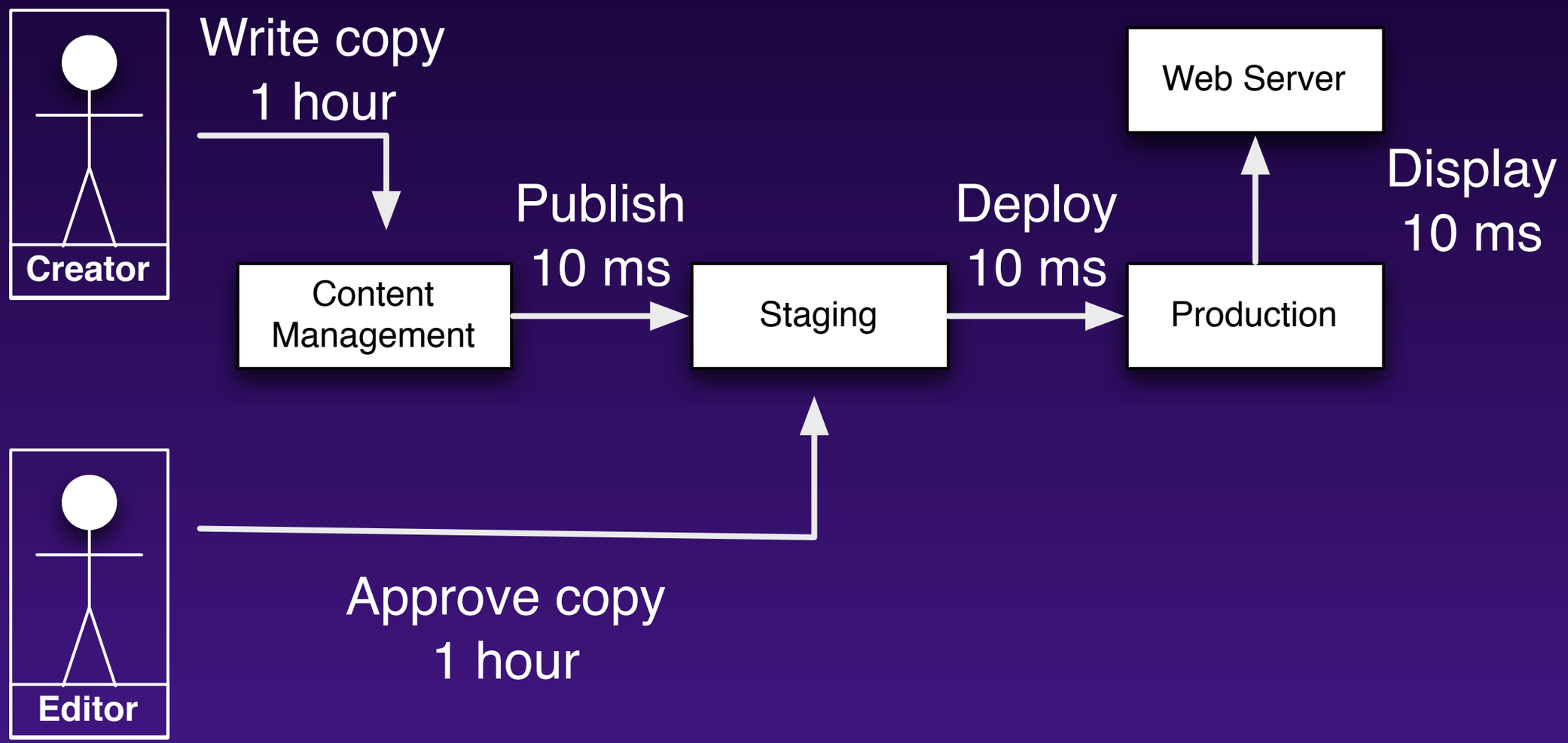
Sufficiently Consistent

"Always consistent" isn't always necessary.

Use latency to your advantage.

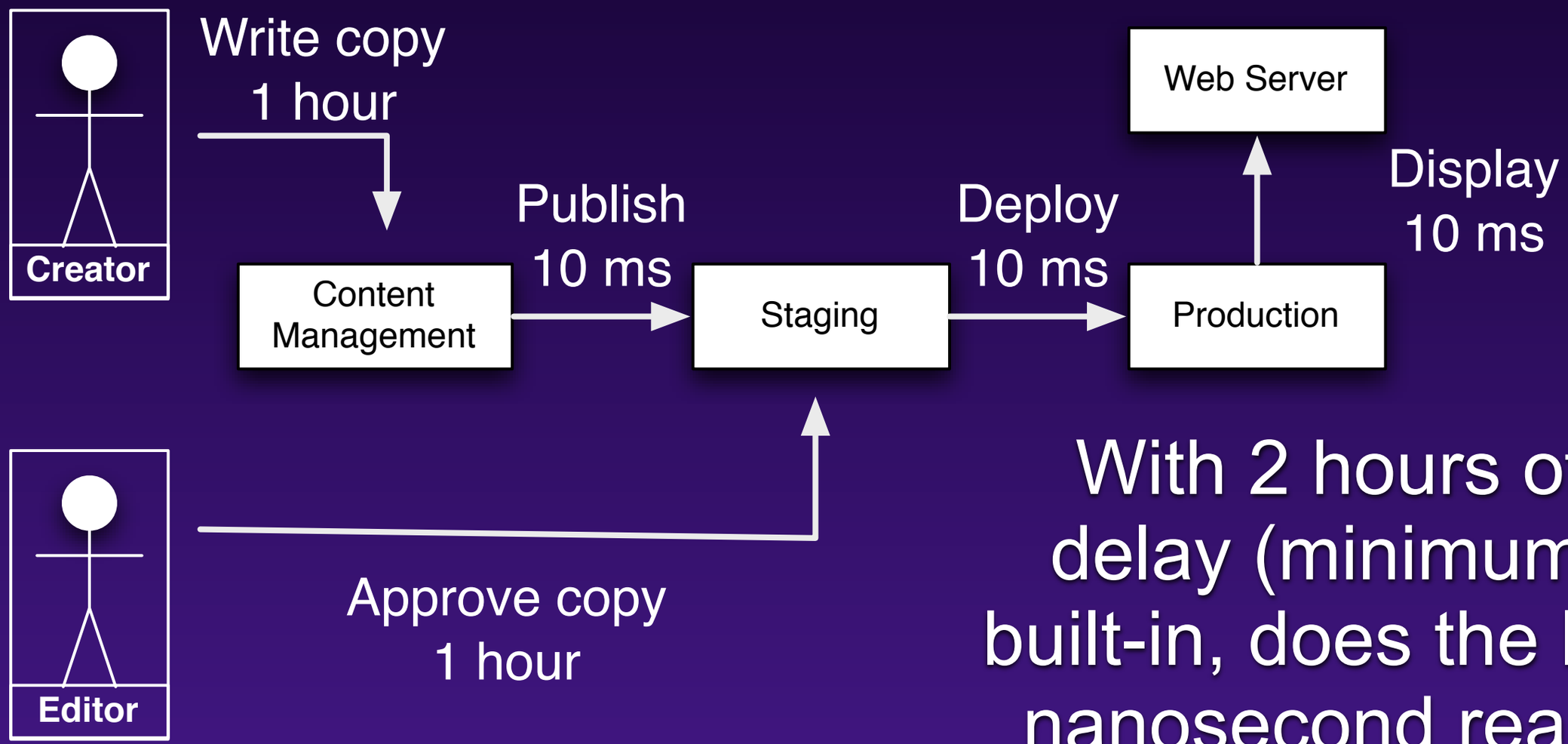
Use Latency To Reduce K

Does the chain of custody start with a human?



Use Latency To Reduce K

Does the chain of custody start with a human?



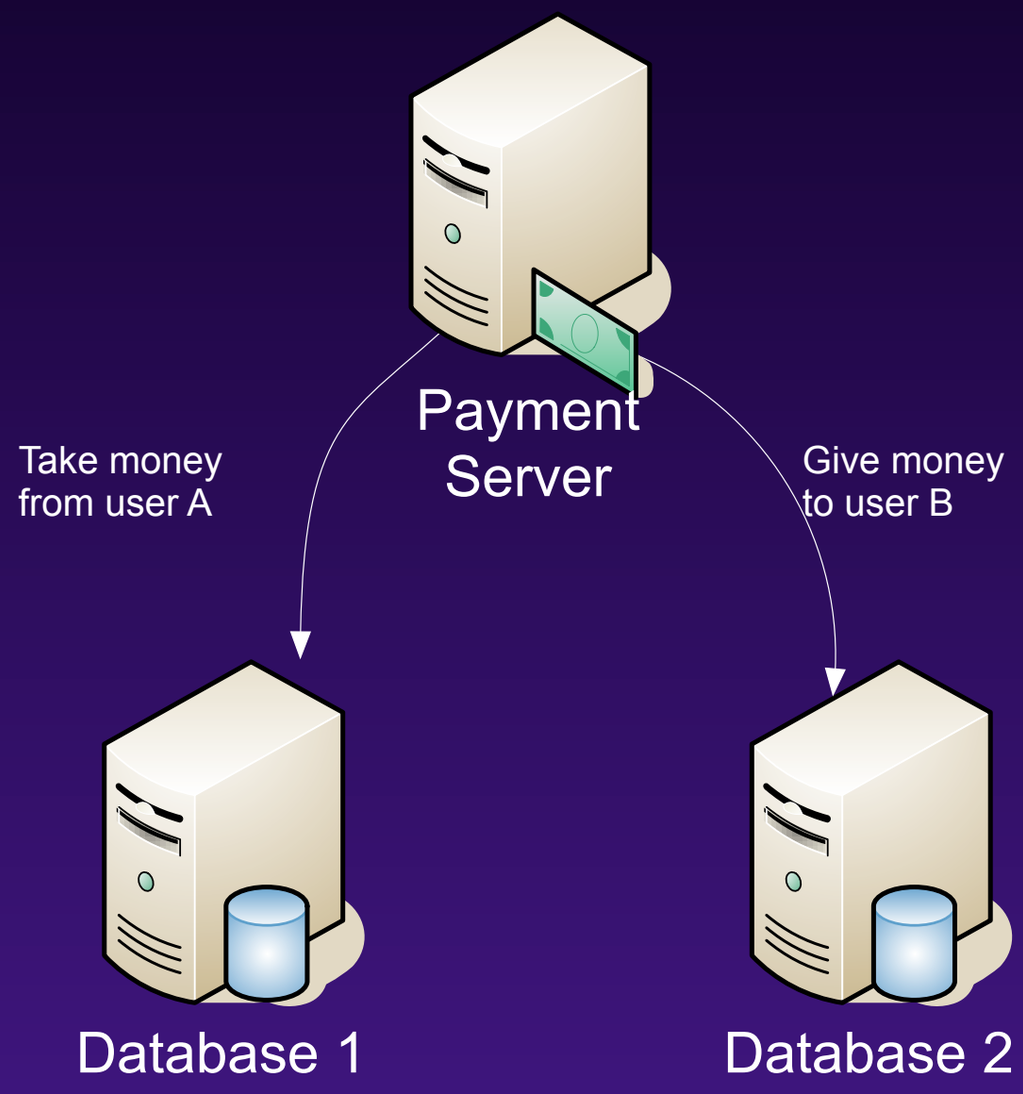
With 2 hours of delay (minimum) built-in, does the last nanosecond really matter?

Always ask yourself:

"Does it matter if this changed in the last millisecond?"

Consistency Without Transactions

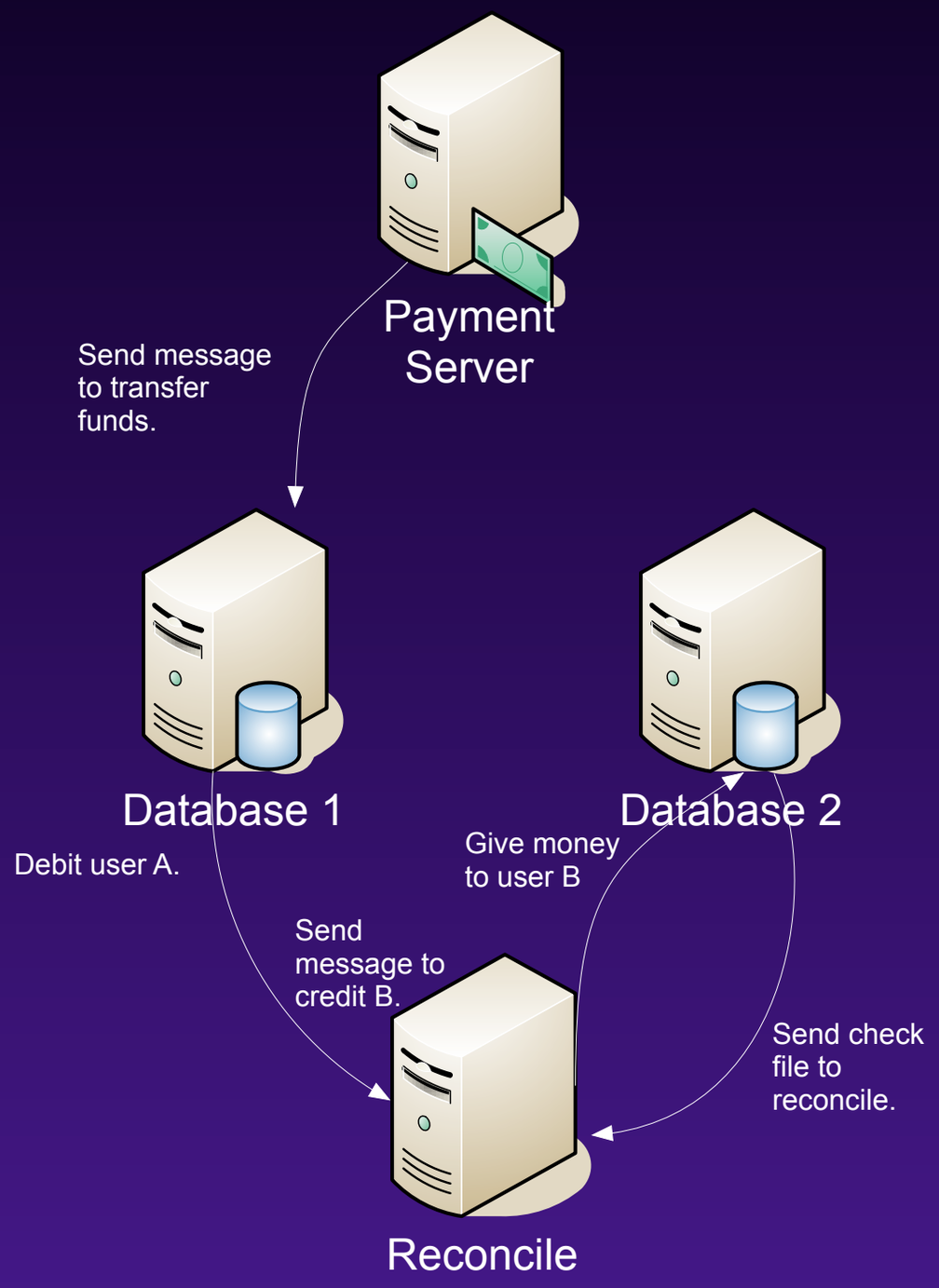
The classic case for transactions. Either both legs of the transfer occur or neither do.



Same Thing, No Transactions

Real banks don't use distributed two-phase commit. They clear transactions asynchronously.

Exception processes are absolutely required.



About Consistency

- Instead of “always consistent,” design for “eventually consistent”.
- RDBMSs do this under the covers. They just hide the convergence time while committing your transaction.
- The time required to achieve consistency is the primary component of κ.

Useful Technology for Eventual Consistency

Post-relational databases

SimpleDB, BigTable, Hypertable

In-Memory Data Grid

GigaSpaces, Coherence, Terracotta

Never Forget Operations

Cost of scaling includes cost of operations.

Operations cost increase is supralinear:

- More boxes require more admins.
- More admins require additional management.

Hallmarks of Scalable Operations

Automatic discovery & provisioning

Pull-mode configuration (cfengine, puppet)

Software package repository

Declarative deployments:

- Execute in waves
- Concurrent versions are allowed (may be necessary)

Questions?

Please fill out a session evaluation.

Michael Nygard
michael.nygard@n6consulting.com
www.michaelnygard.com/blog