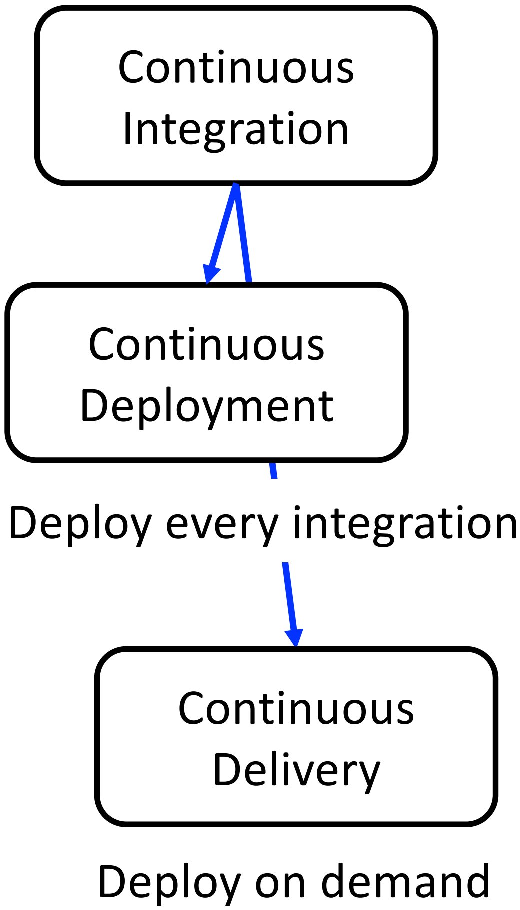
Recall: Deployment is closing the loop

Programs that are never deployed have not fulfilled their purpose. We must deploy!

To do so we must answer:

- Is our application in a working state?
- Do we have the necessary HW/SW resources?
- How do we actually deploy?

Recall: CI, CD and more



CI rigorously tests every integration in production-like environment

- Prevent development-production mismatch
- Test multiple browsers, etc.
- "Stress test" code for performance, fault-tolerance, etc.

Then we deploy!

By deploying frequently, we make what was rare and fraught common and unremarkable!

Recall: DevOps principles

- Involve operations in each phase of a system's design and development,
- Heavy reliance on automation versus human effort,
- The application of engineering practices and tools to operations tasks

*aaS: ___________ as code

Platform-as-a-Service	*Three-tier architecture as code*	1. Deploy (that's it!)
Infrastructure-as-a-Service	*"Infrastructure as code"*	1. Configure (with tools like Ansible, etc.) 2. Deploy
Bare Metal	*Just infrastructure*	1. Rack 2. Configure 3. Deploy

The *aaS division of labor

PaaS handles…	You handle…
“Easy” tiers of horizontal scaling	Minimize load on database
Component-level performance tuning	Application-level performance tuning (e.g., caching)
Infrastructure-level security	Application-level security

What about upgrades? Automation and rigorous processes in action

- Can’t or don’t want to rollout new feature simultaneously to all servers

  Version *n* and *n+1* will co-exist
- Naïve solution: Downtime
- Alternative: Feature flags

  1. Do non-destructive migration
  2. Deploy code protected by feature flag
  3. Flip feature flag on; if disaster, flip it back
  4. Once all records moved, deploy entirely new code
  5. Apply migration to remove old columns

- Other FF uses: A/B testing, …

Kinds of monitoring

“If you haven't ~~tried~~ monitored it, assume it's broken.*”

- At development time (*profiling*)

  Identify possible performance/stability problems *before* they get to production

- In production

  Internal: Instrumentation embedded in application and/or framework

  External: Active probing by other site(s)/tools.

*Google SRE Book

Performance and security metrics

Availability or Uptime

What % of time is site up and accessible?

Responsiveness

How long after a click does user get response?

Scalability

As number users increases, can you maintain responsiveness without increasing cost/user?

Authorization (Privacy)

Is data access limited to the appropriate users?

Authentication

Can we trust that user is who s/he claims to be?

Data integrity

Is users' sensitive data tamper-evident?

Performance & Stability

Security

Google's 4 "golden" signals

- Latency

  Can be confounded by errors. How?

  Time to service a request

- Traffic

  Application specific metric: requests/s, I/O rate, …

  How much demand is being place on your system

- Errors

  Rate of requests that fail

- Saturation

  How "full" your system is (when will you hit ceiling?)

“Premature optimization is the root of all evil”*

- Users expect speed!

  99 percentile matters, not just “average”
- There are lots of reasons for “too slow”
- Don’t assume, measure!

  Monitoring is your friend: measure twice, cut once!

*Variously attributed to Hoare, Knuth, Dijkstra, ….

Simplified (& false) view of response time

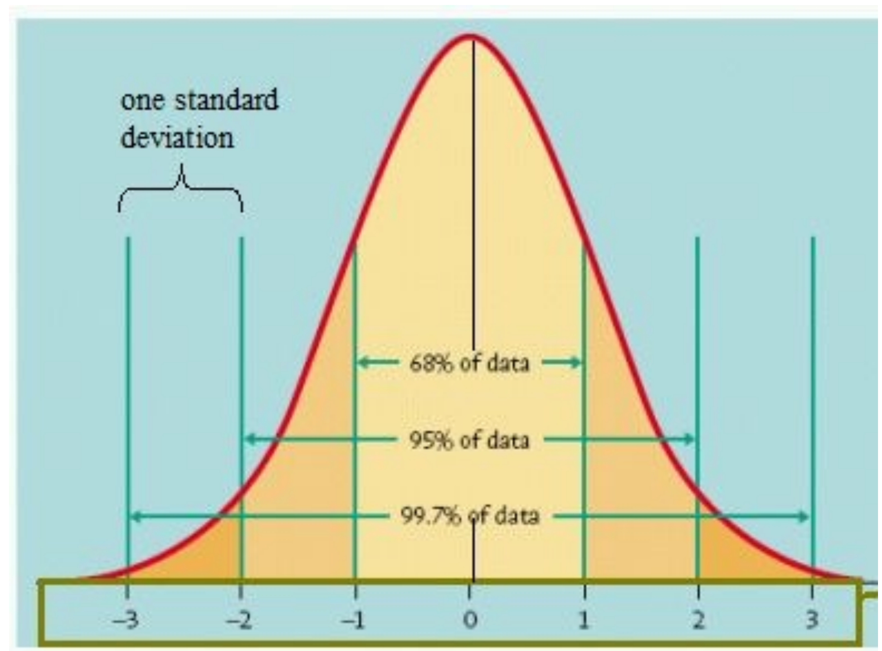
For *normal distribution* of response times:

$\pm$2 standard deviations around mean is 95% CI

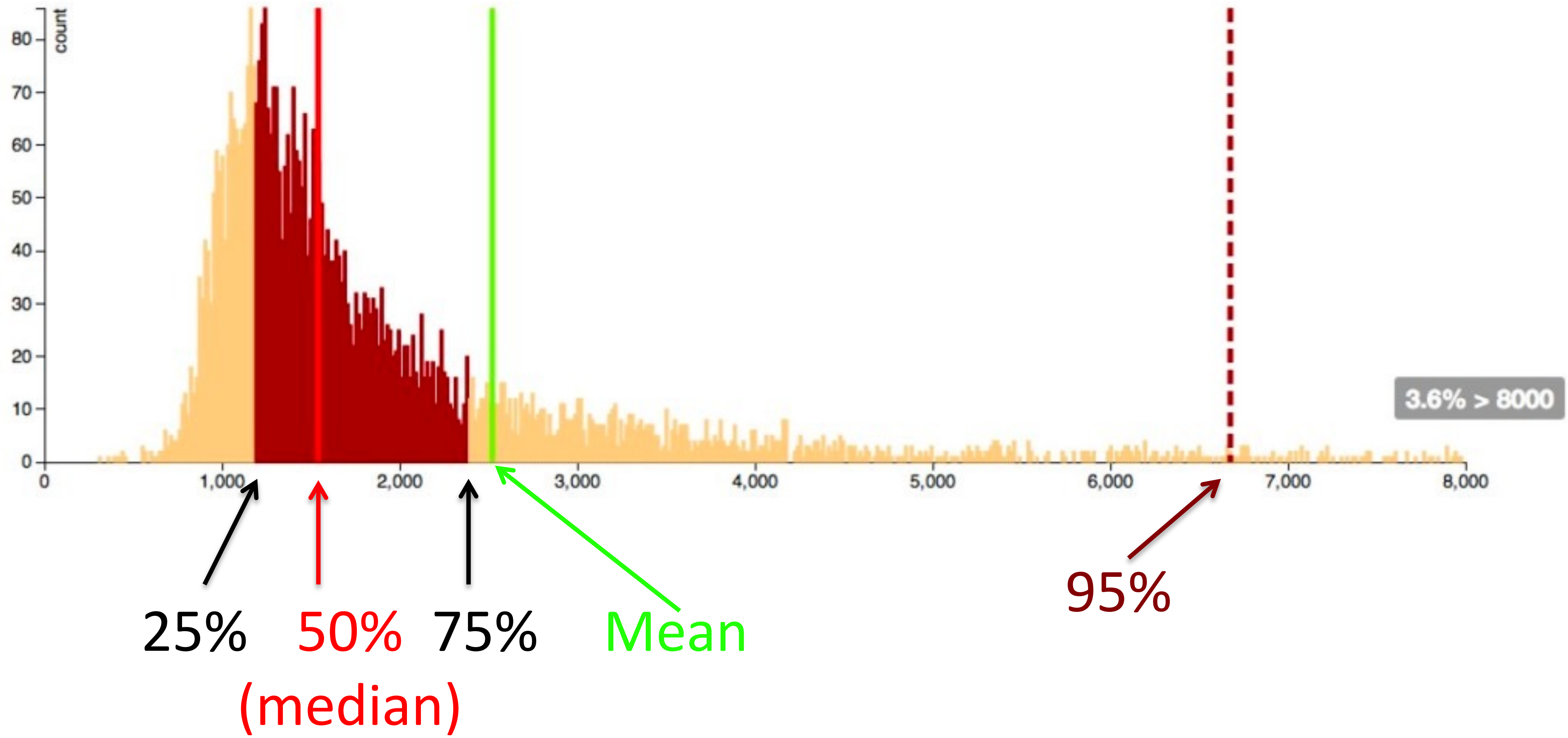
Average response time *T* means:

95%ile users are getting T+2σ

99.7%ile users get T+3σ



A real example: The long tail



https://blog.newrelic.com/2013/09/10/breaking-down-apdex/

Service Level Objective (SLO): Target value for your service

Instead of worst case or average metric, specify a percentile, target and window

- *99% of requests complete in < 1 second, averaged over a 5 min. window*

SLOs set customer expectations

- Make sure you have a safety margin
- Overachieving can be problematic too! How?

Service Level Agreements (SLAs) attach contractual obligations to SLOs

How can you fix "slow"?

- Add more resources, i.e., over-provision
  - Easy to scale presentation and logic tiers for small sites (readily automated in the "cloud")
  - More expensive for larger sites (10% of 10,000 machines is a big number!)
- Make your application more efficient
  - Most effective when there is one bottleneck

The fastest computation is the one you don't do

- Don't forget big-O and CS fundamentals, e.g.

  `Array.include` vs. `Set` for unique

  Smart use of DB indexes

- Caching (and memoization more generally)
- Avoid "toxic" queries, e.g.

  "n+1" query for associations

DB is one of the hardest components to scale, aim to *be kind to your database.*

Indexes: O(< n) queries

Index is a tree, hash-table or other data structure optimized for efficient queries

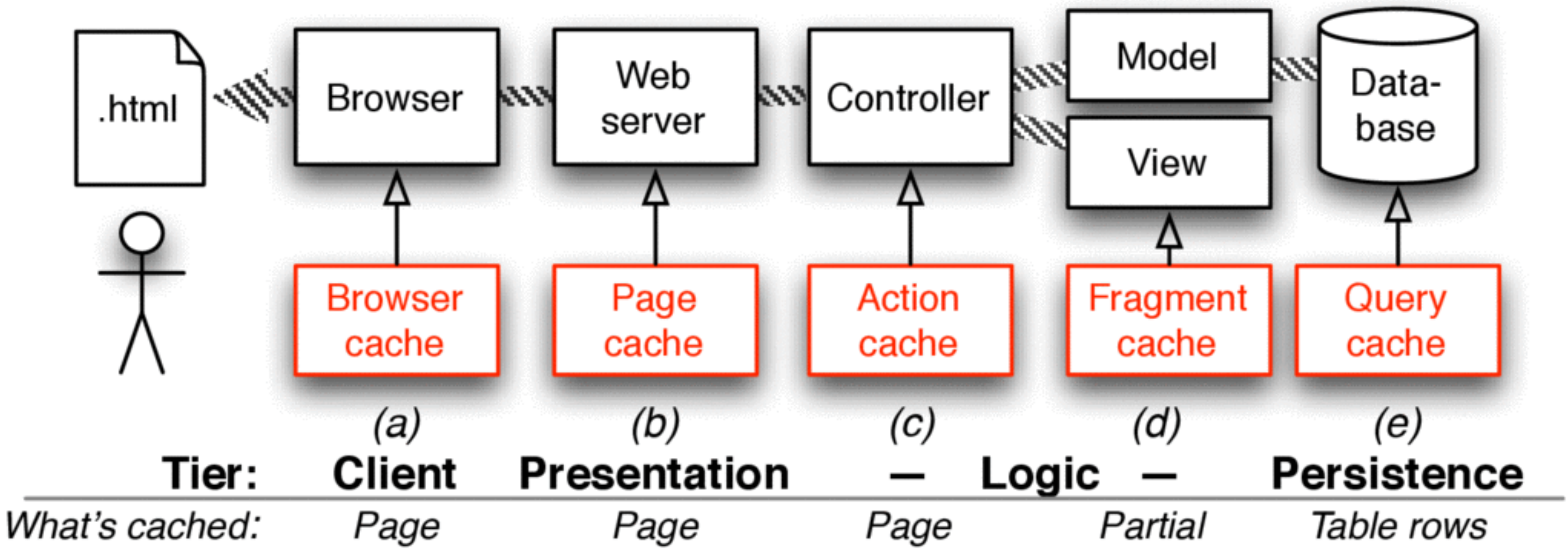
# of reviews:	2000	20,000	200,000
Read 100, no indices	0.94	1.33	5.28
Read 100, FK indices	0.57	0.63	0.65
Performance	166%	212%	808%

Sub-linear scaling!

Why not use an index for every field?

- Requires additional storage space for each index
- Slows down insert/edit (need to update the index)

Cache what hasn't changed



"There are only two hard things in Computer Science: cache invalidation and naming things." –Phil Karlton

n+1 queries (or leaky abstractions)

Recall in the Film Explorer a user ”has many” films “through” ratings

```
User.query().where('zip', '05753').then((fans) => {
  fans.forEach((fan) => {
     fan.$relatedQuery('films')…
  });
});
```

1 query for each user (i.e. *n+1* queries for *n* users)
More subtle for other ORMs, e.g. `fan.films()` is really a query

```
User.query()
  .where('zip', '05753')
  .eager('films')
  .then((fans) => {
    fans.forEach((fan) => {
      fan.films …
    });
});
```

Just 1 or 2 queries, but DB “leaking” through ORM abstraction