



Metrics

radityo_pw@is.its.ac.id

Data Intensive Apps

- Data – intensive apps vs Compute – intensive apps
 - Many IS □ data-intensive apps
- Standard building block :
 - Store data □ usually database
 - Remember the result of an expensive operation, to speed up reads (caches)
 - Allow users to search data by keyword (search indexes)
 - Send a message to another process to be handled asynchronously (Stream processing)
 - Periodically crunch a large amount of accumulated data (batch processing)

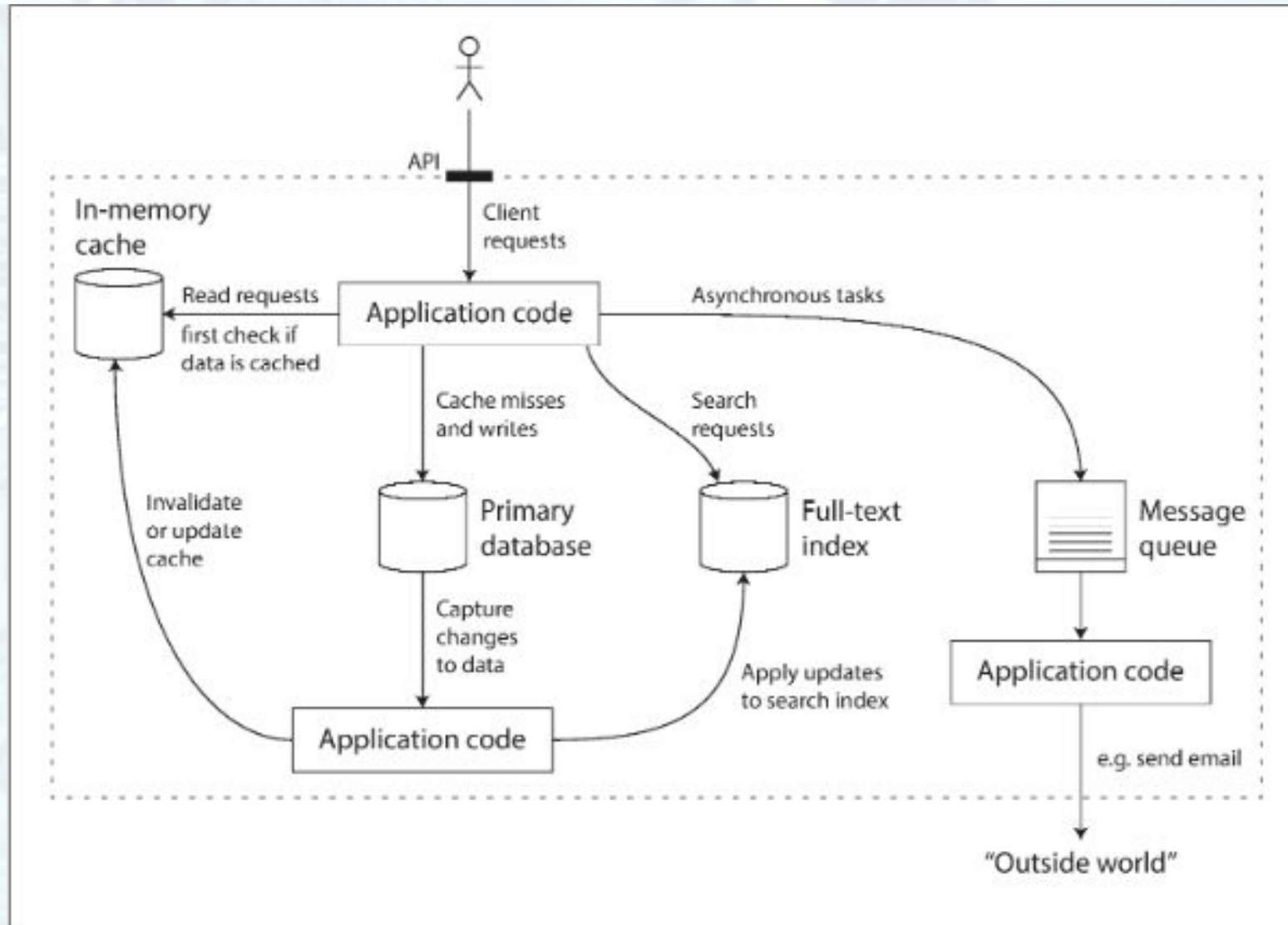
Issue on Data Intensive Apps

- Databases are perfectly good tool for the job
 - Many databases systems with different characteristics
 - Different apps have diff requirements
- Various Approaches for caching
- Several Ways of building search indexes
- Etc..

Single Tool NEVER enough

- Database vs Queues vs Caches
 - Diff categories ?
 - Some have superficial similarity
 - Datastore as Queue (Redis) and MessageQueue with database-like durability (Kafka)
 - Diff in access pattern
 - Diff Performance characteristics
 - Diff implementations
 - Optimized in variety of use cases
- Complex Apps Req
 - Single Tool can no longer meet All data processing and storage needs.
 - Req broken down into task □ specialized tools
 - Stitched using app code

Typical Complex Apps Architecture



Question about data systems

- How do you ensure data remain correct and complete?
- How do you provide consistently good performance ?
- How do you scale ?
- What does good API ?

Metrik

- Reliability
 - Tolerating Hardware & Software Fault
 - Human Error
- Scalability
 - Measuring load & Performance
 - Latency percentiles, throughput
- Maintainability
 - Operability,
 - Simplicity
 - Evolvability

Reliability

- Performs the function that user expected
- Tolerate the user making mistakes
- Tolerate the use using software in unexpected ways
- Performance good enough for the required use case, under expected load and data volume
- Prevent authorized access and abuse

Reliability

- Continuing to work correctly, even when things go wrong
- Things that go wrong \square faults
 - Systems anticipate faults \square faults tolerance
 - Every possible kind of fault ?
 - Not feasible
 - Certain types only

Fault vs Failure

- A Fault is usually defined as one component of the system deviating from its spec
- A Failure is when a system as a whole stops providing required service to the user.
- Impossible to reduce probability of a fault to zero
 - Design fault-tolerance mechanism that prevent fault from causing failure.
 - Generally prefer tolerating faults over preventing faults.

Hardware Faults

- Hard Disk Crash
 - Mean Time to Failure (MTTF) 10 – 50 years
 - Single disk are easy, how about storage cluster with 10.000 disks ?
- RAM faulty
- Power Grid blackout
- Unplugs wrong cable
- Etc ...
- How to Response ?
 - Redudancy

Software Errors

- Hardware fault □ random and independent from each other
- Software Error □ systematic error within system
 - Software bug □ ex leap second june 30 2012 linux kernel □ apps hang
 - Runaway process use shared resources (cpu, memory , disk , network)
 - Service that the systems depends on start slow down, unresponsive, returning corrupted responses
 - Cascading failure, when small fault trigger other fault, which in turn triggers further fault
- No Quick Solution

Human Errors

- Human everywhere
 - Software & Hardware design by.... Human
 - Software built by Human
 - The Operator..... □ Human
 - Even when they have best intentions , humans are known to be unreliable.
- Configuration errors by operator □ leading cause of outages, hardware faults only 10 – 25 %.
- How To make system reliable :
 - Design systems in a way minimizes opportunities for error.
 - Decouple the places where people make most mistakes from the places where they can cause failures.
 - Test in all levels, from unit test to manual tests
 - Allow quick and easy recovery from human errors
 - Set up detailed and clear monitoring
 - Good Management practices and training.

Why Reliability is Important ?

- How about nuclear power stations ?
- Air traffic control software ?
- Bank and other finances apps ?
- Even for noncritical apps

Scalability

- Reliability <> Scalability
 - Even if system is working reliably today, doesn't mean it will necessarily work reliably in the future.
- Common reason for **performance** degradation is increased **load**.
 - Ex from 10.000 to 100.000 concurrent users
 - Ex processing much larger data volumes

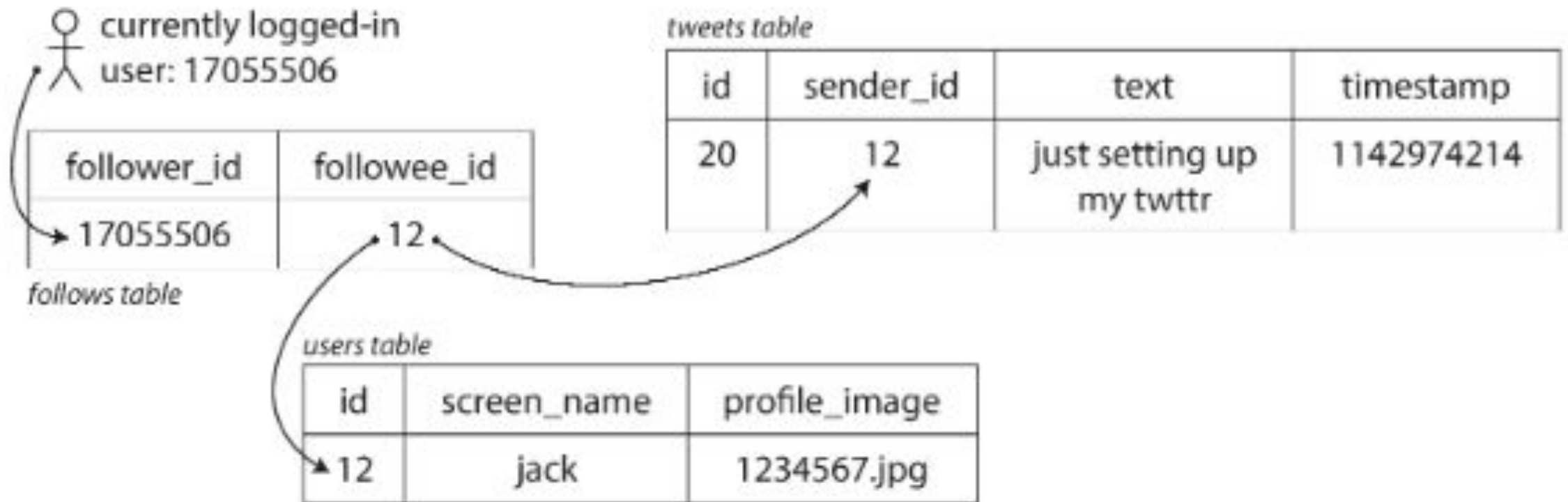
Load

- Load can be described with a few number □ load parameters
 - Depend on architecture
 - May be request per second for a web servers
 - May be ratio of reads to writes for database
 - May be the number of simultaneously active users in a chat room
 - May be hit rate on a cache

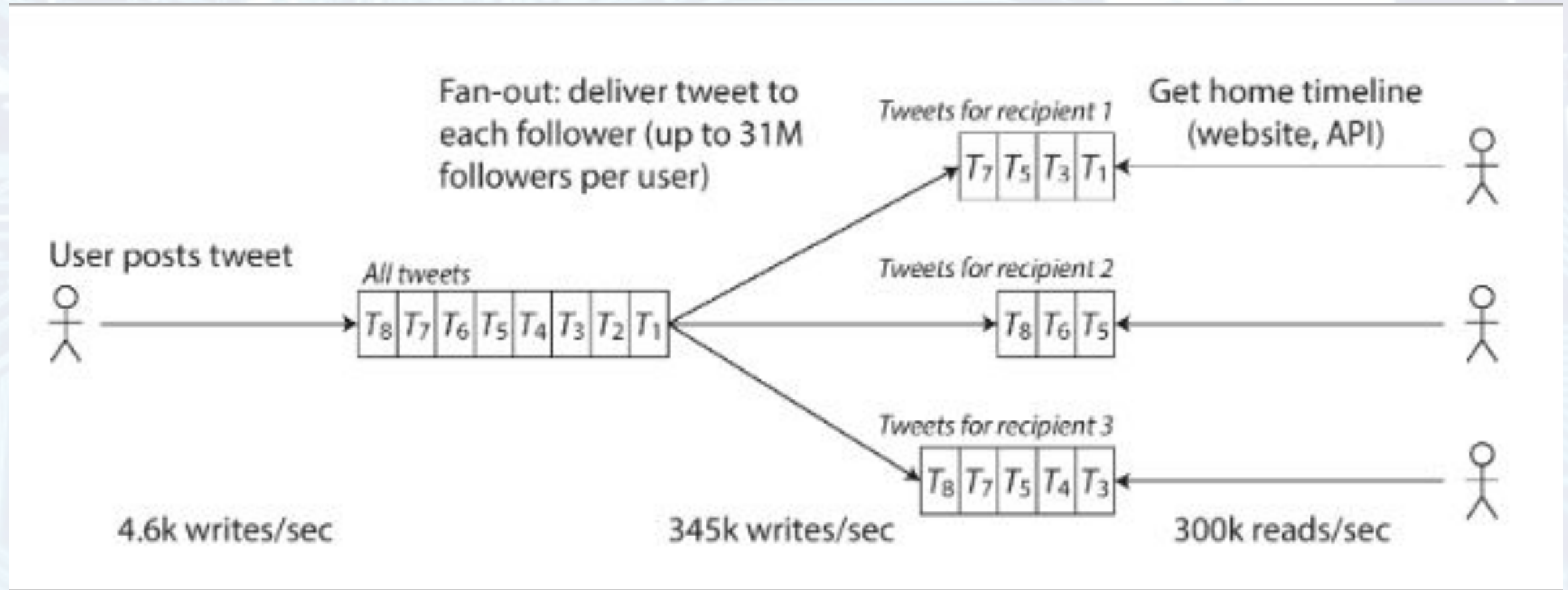
Lets Tweets

- Twitter main operation :
 - Post Tweet
 - A User can publish a new message to their followers (4.6 K requests / sec on average, over 12 K requests / sec at peak)
 - Home Timeline
 - A user can view tweets posted by the people they follow (300 K requests / sec)
- Tweeter scaling challenge not in tweet data volume (handling post tweet fairly easy), the problem is fan-out, each user follows many people, and each user is followed by many people.
- There two broadly ways of implementing two operations :
 - Posting a tweet simply insert the new tweet into a global collection of tweet. When a user requests their home timeline, look up all the poeple they follow, find all tweets for each users, and merge them (sort by time). □ achitecture1
 - Maintain cache for each users timeline (like mailbox of tweets for each recipient user), when user post a tweet look up all people who follow that users, and isert the new tweet into each of their home timeliline. □ architecure2

Tweeter architecture 1



Tweeter architecture 2



Tweeter Arch 1 vs Arch 2

- Arch 1 simple, easy to start
- Arch 1 struggled to keep up with the load of home timeline
- Arch 2 works better because the average rate of published tweets is lower than rate of home timeline read
- Downside of Arch2 that posting a tweet now requires a lot of extra work. On average a tweet is delivered to about 75 followers, so 4.6K tweet per second become 345K writes per second to the home timeline caches.
- The distribution of followers per user is key load parameter.

Performance ?

- Basic Question :
 - When increase a load parameter and keep the system resources (CPU, memory, network bandwidth, etc.) unchanged, how is the performance affected ?
 - When increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged ?
 - Both questions require **performance numbers** ...
- In a batch processing system □ throughput □ the number of record we can process per second, or the total time it takes to run a job on a dataset of a certain size.
- In Online systems □ response time □ the time between a client sending a request and receiving a response.

Response Time vs Latency

- Response Time \square is what the client sees (processing time + network delay + queueing delay)
- Latency \square is the duration that a request is waiting to be handled.

Response Time : Average vs Percentiles

- Its common to see the average response time.
- The mean is not a very good metric if you want to know your typical response time.
 - It doesnt tell you how many users actually experience that delay.
 - Percentile (median) is a good metric if you want to know how long users typically have to wait.
- Median □ 50th percentile / p50
- To identify how bad your outliers are, you can look at higher percentiles : p95, p99 and p99.9 □ tail latencies
 - Ex if p95 response time is 1.5 seconds, that means 95 of 100 requests take less than 1.5 seconds, and 5 of 100 requests take 1.5 seconds or more.

Important of Tail Latencies

- Directly affect users experience of the service.
- Amazon □ response time requirement for internal service p99.9 □ 1 of 1000 request.
 - This is because the customers with the slowest request are often those who have most data on their accounts because the have made many purchases.
 - Most valuable customers
 - Easily affected by random events outside of your control.
- Amazon observed that 100 ms increase in response time reduces sales by 1 % , others report thath a second slowdown reduce a customer satisfaction metric by 16%.
- Often used in Service Level Objectives (SLOs) and Service Level Agreements (SLA).
 - Service may considered up if it has a median response time of less than 200 ms and p99 under 1 second.

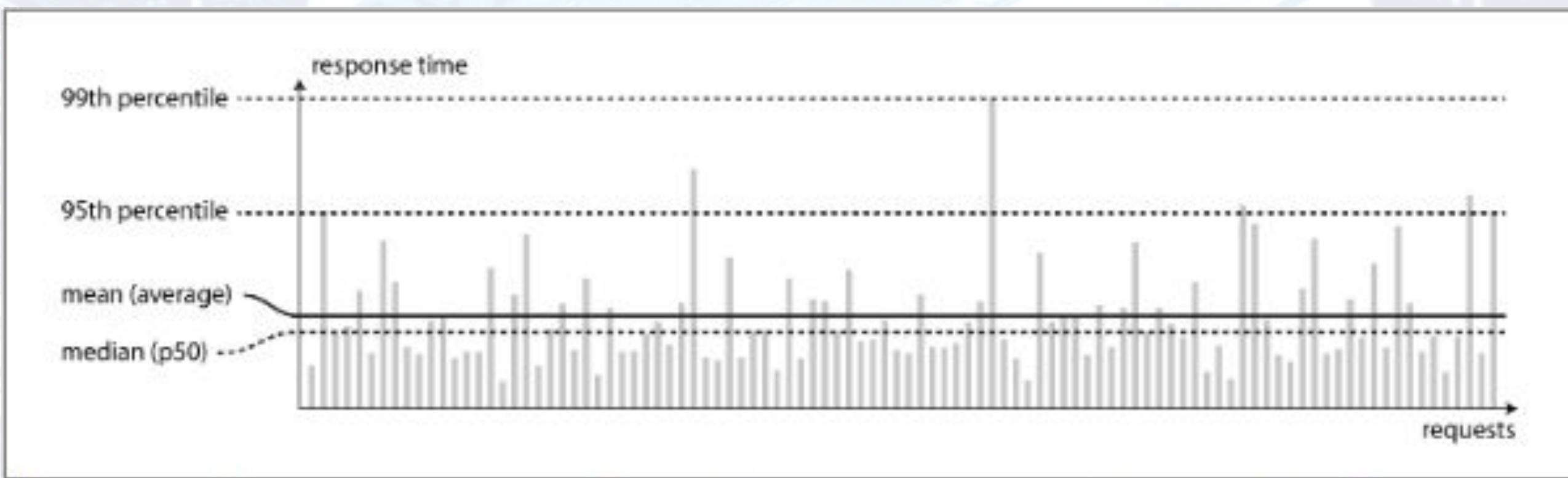


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

How to Handle Load

- Scaling
 - Scaling Up
 - Scaling Out
 - Shared – Nothing
- Good Architecture usually involve a pragmatic mixture of approaches
 - Ex using several fairly powerful machines can still be simpler and cheaper than a large number of small virtual machines.
 - There is no such thing as a generic , one-size-fits-all scalable architecture.
- Distributing stateless service across multiple machines is fairly straightforward.
- Taking stateful data systems from single node to a distributed setup can introduce a lot of additional complexity.
 - Keep Your databases on a single node (Scale Up), until scaling cost or High-availability requirements forced you to make it distributed.

Maintainability

- Majority of cost of software is not in its initial development. But in its ongoing maintenance
 - Fixing bugs
 - Keeping its system operational
 - Investigating failures
 - Adapting it to new platforms
 - Modifying it for new use cases
 - Adding new features.
- Design software in such way minimize pain during maintenance :
 - Operability
 - Make it easy for operations teams to keep the system running smoothly
 - Simplicity
 - Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system
 - Evolvability
 - Make it easy for engineers to make changes to the system in the future.

Operability : Making Life Easy for Operations

- Good operations can often work around the limitations of bad software
- Good software cannot run reliably with bad operations
- A good operations team typically is responsible for the following :
 - Monitoring the health of the system and quickly restoring service if it goes into a bad state
 - Tracking down the cause of problems, such as system failures or degraded performance
 - Keeping software and platforms up to date, including security patches
 - Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage
 - Anticipating future problems and solving them before they occur
 - Establishing good practices and tools for deployment, configuration management, and more
 - Performing complex maintenance tasks such as moving an application from one platform to another
 - Maintaining the security of the system as configuration changes are made
 - Defining processes that make operations predictable and help keep the production environment stable
 - Preserving the organization's knowledge about the system, even as individual people come and go.

Operability : Making Life Easy for Operations

- Good Operability means making routine tasks easy, allowing the operations team to focus on high value activities :
 - Providing visibility into the runtime behavior and internals of the system, with good monitoring
 - Providing good support for automation and integration with standard tools
 - Avoiding dependency on individual machines
 - Allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted
 - Providing good documentation and an easy-to-understand operational model
 - If i do x, y will happen
 - Providing good default behavior, but also giving administrators the freedom to override defaults when needed
 - Self-healing where appropriate, but also giving administrators manual control over the system state when needed
 - Exhibiting predictable behavior, minimizing surprises

Simplicity : Managing Complexity

- Big Ball of Mud : a software project mired in complexity
 - Explosion of the state space
 - Tight coupling of modules
 - Tangled dependencies
 - Inconsistent naming and terminology
 - Hacks aimed at solving performance problems
 - Special-casing to work around issues elsewhere
- When complexity make maintenance hard, budgets and schedules are often overrun.
 - Hidden assumptions
 - Unintended consequences
 - Unexpected interactions

Simplicity : Managing Complexity

- Making simpler not necessarily mean reducing its functionality.
 - Removing accidental complexity.
 - Complexity as accidental if it is not inherent in the problem that the software solves, but arises only from the implementation
- Removing accidental complexity is abstraction
 - Hide a great deal of implementation detail behind a clean, simple-to-understand facade.
 - Ex : High Level Programming Languages
 - SQL

Evolvability : Making Change Easy

- Your systems requirement will remain unchanged forever ?
 - BIG NO !
 - Learn new facts
 - Previously unanticipated use cases emerge
 - Business priorities change
 - Users request new features
 - New platforms replace old platforms
 - Legal or regulatory requirements change
 - Growth of the system forces architectural changes
- Increasing agility on data level.
 - Ex : how to refactor “twitter architecture 1 to architecture 2”



DESIGNING Data-Intensive Applications

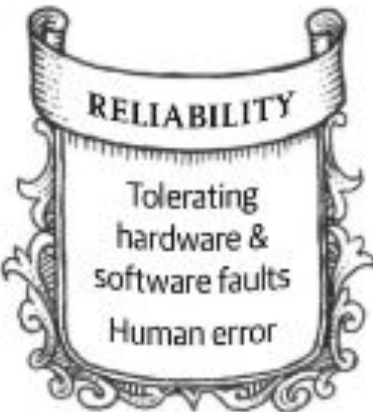
*The big ideas behind reliable,
scalable & maintainable systems*



RELIABILITY

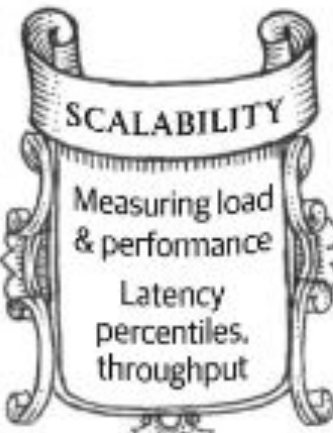
MAINTAINABILITY

SCALABILITY




RELIABILITY

Tolerating
hardware &
software faults
Human error



SCALABILITY

Measuring load
& performance
Latency
percentiles,
throughput



MAINTAINABILITY

Operability,
simplicity &
evolvability