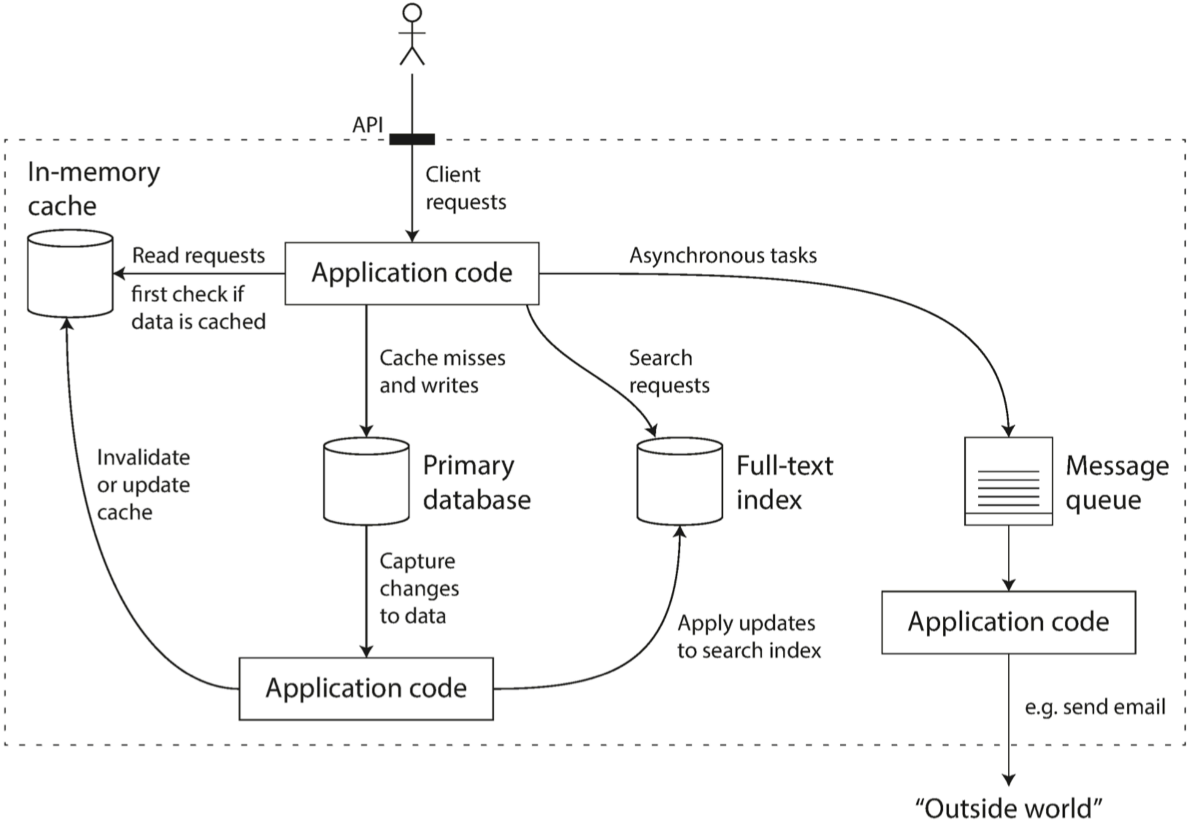
Designing Data-Intensive Applications

Enterprise Architectures for Big Data

Parts of a Modern Application



Parts of a Modern Application

- **Database**
  - Store data so that this or another application can find it again later
- **Cache**
  - Remember the result of an expensive operation, to speed up reads
- **Search index**
  - Allow users to search data by keyword or filter it in various ways
- **Stream processing**
  - Message Queue
  - Send a message to another process, to be handled asynchronously
- **Batch processing**
  - Periodically crunch a large amount of accumulated data

Source: Kleppmann (2017), p. 15

Requirements

- **Functional requirements**

  What the system should do, such as allowing data to be stored, retrieved, searched, and processed in various ways.

- **Nonfunctional requirements**

  Are general properties like

  - security
  - reliability
  - compliance
  - scalability
  - compatibility
  - maintainability


DESIGNING
Data-Intensive
Applications
The big ideas behind reliable,
scalable & maintainable systems
RELIABILITY
SCALABILITY
MAINTAINABILITY
RELIABILITY
Tolerating
hardware &
software faults
Human error
SCALABILITY
Measuring load
& performance
Latency
percentiles,
throughput
MAINTAINABILITY
Operability,
simplicity &
evolvability

3 main concerns of most software systems

1. **Reliability**
   - The system should **continue to work correctly** (performing the correct function at the desired level of performance) even in the face of adversity (hardware or software faults, and even human error).
2. **Scalability**
   - As the system grows (in data volume, traffic volume, or complexity), there should be **reasonable ways of dealing with that growth**.
3. **Maintainability**
   - Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be **able to work on it productively**.

Source: Kleppmann (2017)

Reliability

Reliability

- The application executes the functions that the user expected.
- It can tolerate that the users are making mistakes or using the software in unexpected ways.
- Its performance is good enough for the required use case, under the expected load and data volume.
- The system prevents any unauthorized access and abuse.

- How important is reliability?
- How can a system fail?

Source: Kleppmann (2017), p. 6

Fault Types

- Hardware Faults
  - E.g. Hard disks:
    Mean time to failure (MTTF): 10-50 years
    ➔ Cluster of 10,000 disks ➔ 1 disk dies per day
- Software Errors
- Human Errors

- How are hardware and software errors different?
- How to make a system more fault tolerant?

Source: Kleppmann (2017), p. 7

Fault Tolerant Systems

- Strategies of fault tolerant systems
  - Redundancy
  - Testing
  - Well designed abstraction (API)
  - Isolation
  - Decoupling
  - Measuring
  - Monitoring
  - Quick recovery
  - Backup
  - Good management practice and training

Source: Kleppmann (2017), p. 7

Scalability

Scalability

- Definition: the system's ability to cope with increased load

- Not a one-dimensional label

- Meaningless to say "X is scalable" or "Y doesn't scale"

Rather say:

- "If the system grows in a particular way, what are our options for coping with the growth?"
- "How can we add computing resources to handle the additional load?"

Source: Kleppmann (2017), p. 10

Describing Performance

Two different ways to conceptualize performance:

1. When you **increase** a **load parameter** and keep the system **resources** (CPU, memory, network bandwidth, etc.) **unchanged**, how is the **performance** of your system affected?
2. When you **increase** a **load parameter**, how much do you need to **increase** the **resources** if you want to keep **performance unchanged**?

Load Parameters

- Load parameters describe the load
- Depends on your application

- KPIs
  - requests per second to a web server
  - ratio of reads to writes in a database
  - number of simultaneously active users in a chat room

Performance Measurement
Throughput

- Definition:
  - Number of records we can process per second,
  - Alternatively: the total time it takes to run a job on a dataset of a certain size

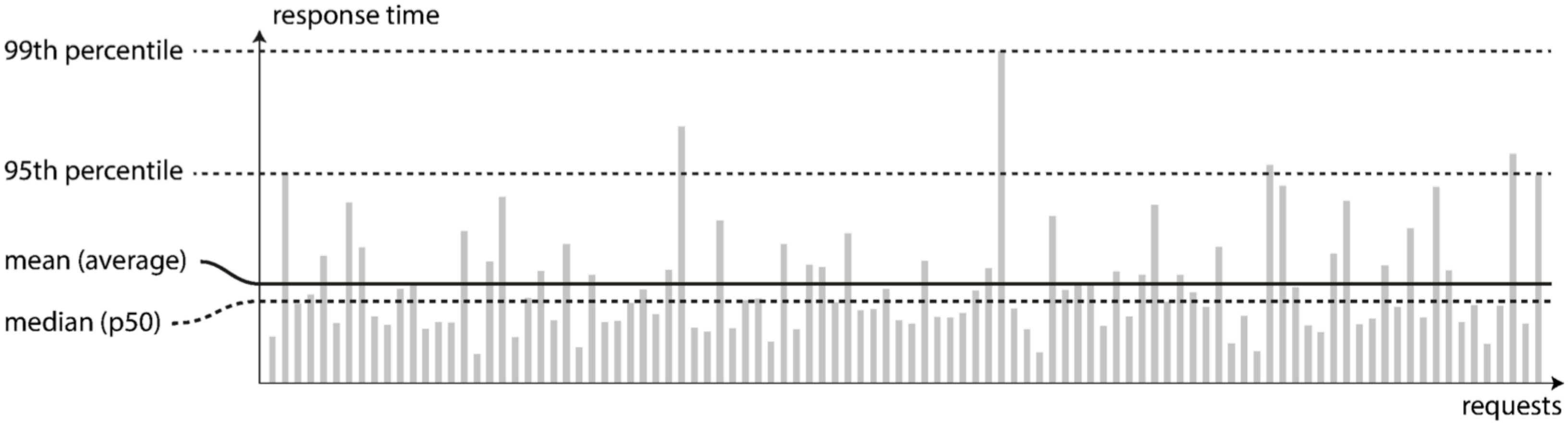
- Important for Batch Systems (like Hadoop)

Source: Kleppmann (2017), p. 11

Performance Measurement
Response time

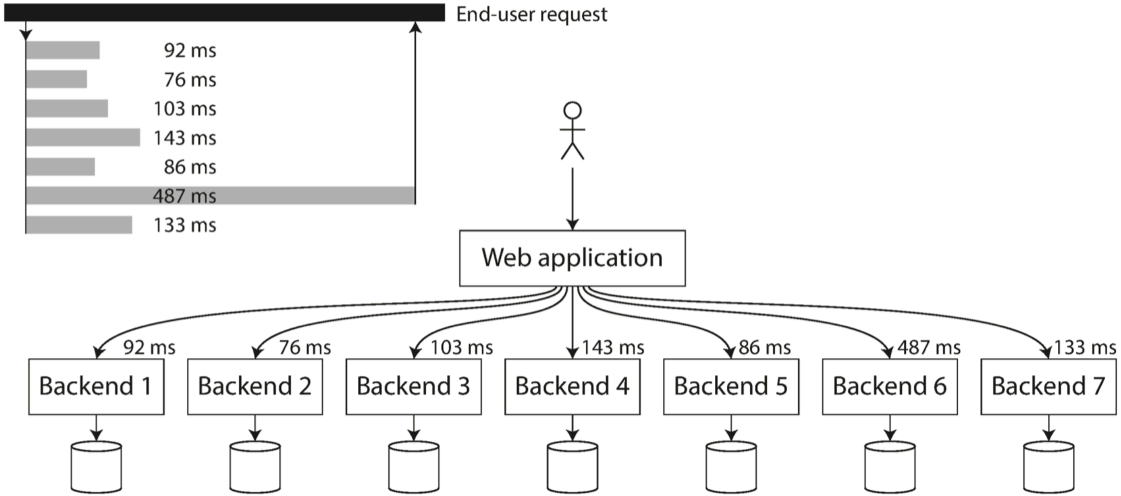
- Definition:
  - Time between a client sending a request and receiving a response
- Important for Online Systems (like an E-Shop)
- Latency vs. response time:
  - Response time: what the client sees
  - Besides the actual time to process the request (the service time), it includes network delays and queueing delays.
- Effect of Response time:
  - Amazon: 100 ms increase in response time reduces sales by 1%
  - 1 s slowdown reduces a customer satisfaction metric by 16%

Response time



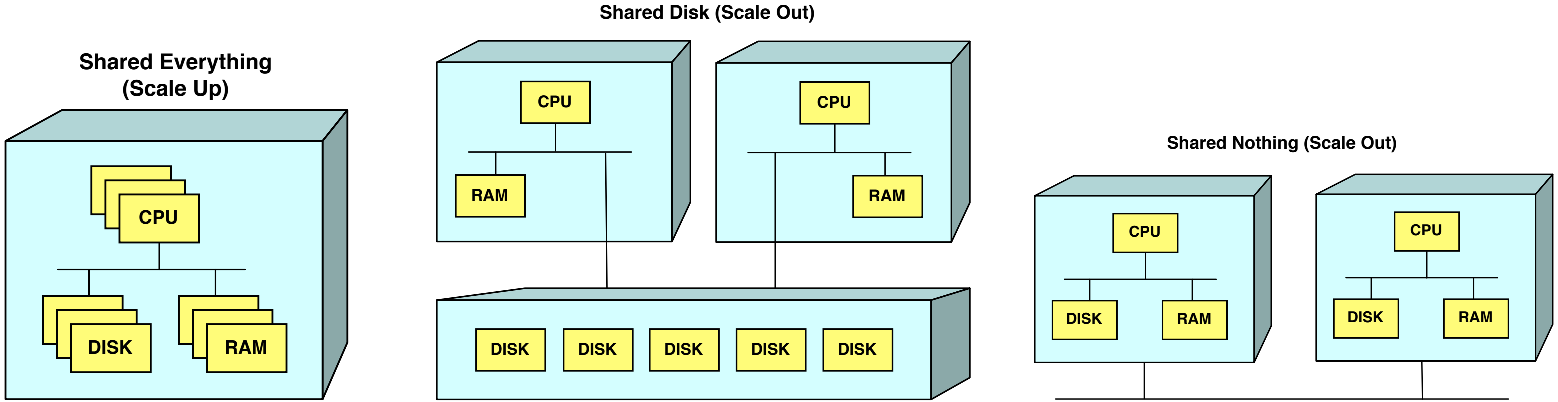
- Aggregations:
  - Not just the average
  - Percentile
- **Tail latencies**, e.g. 99.9th percentile
- Customers with the slowest requests ➔ most data on their accounts ➔ made many purchases ➔ most valuable customers

Response time



- **Tail latency amplification**
- It only takes a small number of slow requests to hold up the processing of subsequent requests
- Queueing delays
- Head-of-line blocking

Scaling Architectures: Scale up vs. Scale out



E.g.
SAN (Storage Area Network) or
NAS (Network Attached Storage)

Distributed Systems

- Stateless vs. stateful data service
  - Stateless services: Distributing across multiple machines fairly straightforward
  - Stateful services: Distributed setup can introduce a lot of complexity

- Tradeoff: Scalability vs. Simplicity (Maintainability)

- Therefore, common wisdom until recently was to **keep your database on a single node** (scale up) until scaling cost or high-availability requirements forced you to make it distributed.
- New tools and abstractions ➔ Change of the tradeoffs

Source: Kleppmann (2017), p. 17

No generic, one-size-fits-all scalable architecture

- The scaling problem may be
  - the volume of reads,
  - the volume of writes,
  - the volume of data to store,
  - the complexity of the data,
  - the response time requirements,
  - the access patterns,
  - or some mixture of all of these plus many more issues.

- An architecture that scales well for a particular application is built around **assumptions**
  - which operations will be frequent, and which will be rare ➔ **load parameters**

- Architectures of systems that operate at large scale are usually **highly specific to the assumptions**
  - E.g. in an early-stage **startup** it’s usually more important to be able to iterate quickly on product features than it is to scale to some hypothetical future load.

- **No generic, one-size-fits-all scalable architecture**
  - No Magic Scaling Sauce
  - However, scalable architecture are based on general-purpose building blocks and arranged in familiar patterns

Source: Kleppmann (2017), p. 17

Maintainability

Maintainability

- **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.
  - Not the same as simplicity of the user interface
- **Evolvability**
  - Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change.

Source: Kleppmann (2017), p. 19

Operability

Keeping a software system running smoothly

- **Monitoring**: Providing visibility into the runtime behavior and internals of the system
- **Automation** (Continuous Integration (CI), Continuous Delivery (CD))
- **Integration** with standard tools
- **Avoiding dependency** on individual machines
- Good **documentation**
- Easy-to-understand **operational model**
- Providing **good default behavior**, optionally override defaults when needed
- **Self-healing** where appropriate, but also giving administrators manual control over the system state when needed
- **Predictable behavior**, minimizing surprises

Source: Kleppmann (2017), p. 20

Simplicity

- Accidental complexity:
  - Not inherent in the problem but arises only from the implementation
- Best tools for removing accidental complexity: **Abstraction**
- Good Abstraction:
    - can hide a great deal of implementation detail behind a clean, simple-to-understand façade.
    - can also be used for a wide range of different applications.
      - Reuse is more efficient than reimplementing a similar thing multiple times
      - Leads to higher-quality software, as quality improvements in the abstracted component benefit all applications that use it.
    - Finding good abstractions is hard

Evolvability

Making Change Easy

- Changing requirements
- Demand of Agility
- Influenced by:
  - Analyzability
  - Extensibility
  - Integrity
  - Portability
  - Testability
  - Changeability
- Linked to Simplicity and Abstraction
- Simple and easy-to-understand systems are easier to modify than complex ones

Source: Kleppmann (2017), p. 21; Breivold, Crnkovic, Eriksson (2008)