



Design Data Intensive Application

The Big Ideas Behind Reliable, Scalable,
and Maintainable Systems


O'REILLY®

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE,
AND MAINTAINABLE SYSTEMS



Martin Kleppmann



Chapter1: Reliable, Scalable, and Maintainable Applications

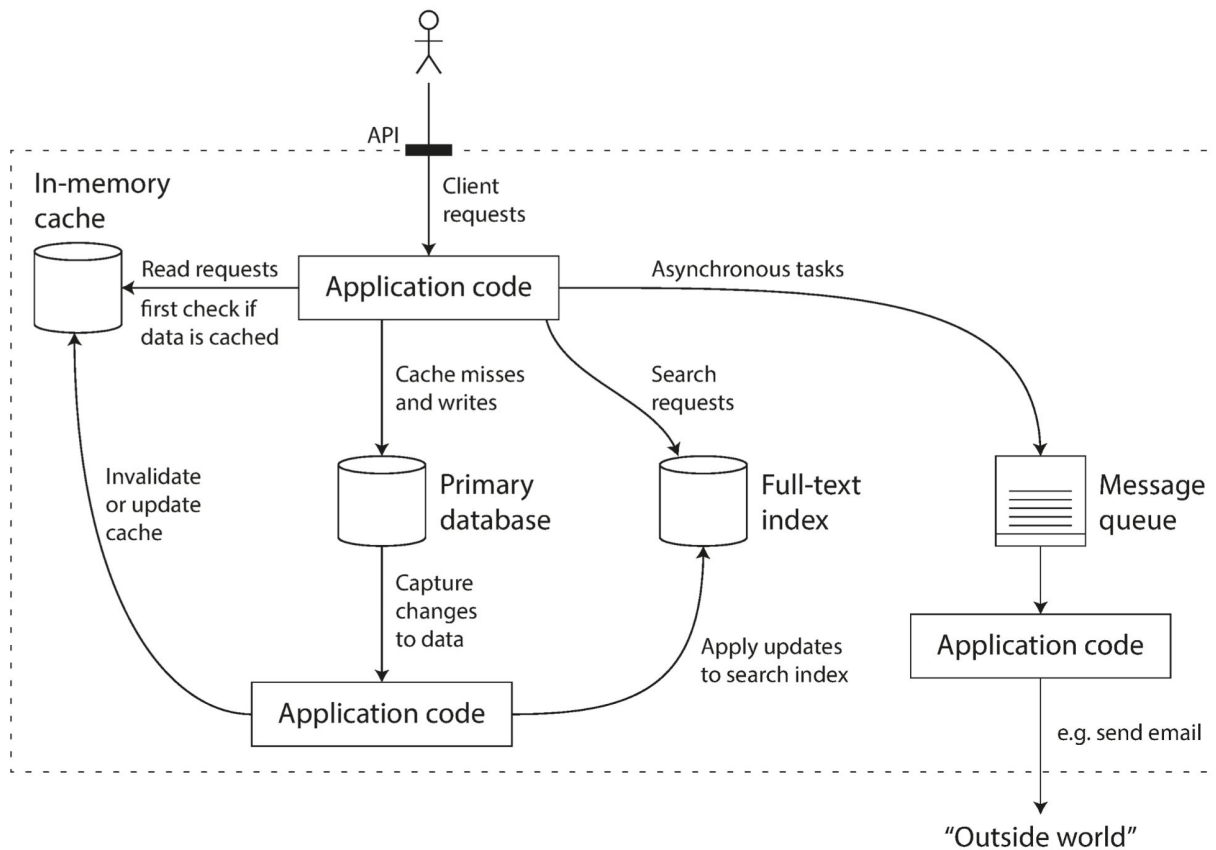


Thinking About Data Systems

We typically think of databases, queues, caches, etc. as being very different categories of tools.

So why should we lump them all together under an umbrella term like data systems?

1. Many new tools are optimized for a variety of different use cases, and they no longer neatly fit into traditional categories
 - Datastores that are also used as message queues (Redis)
 - Message queues with database-like durability guarantees (Apache Kafka)
2. Increasingly many applications now have such demanding or wide-ranging requirements that a single tool can no longer meet all of its data processing and storage needs.





Reliability

- The application performs the function that the user expected.
- It can tolerate the user 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.

Continuing to work correctly, even when things go wrong



Reliability: Fault vs Failure

A **fault** is usually defined as one component of the system deviating from its spec, whereas a **failure** is when the system as a whole stops providing the required service to the user.

Impossible to reduce the probability of a fault to zero.

Design fault-tolerance mechanisms that prevent faults from causing failures.



Reliability: Hardware Faults

- Hard disks crash
- RAM becomes faulty
- The power grid has a blackout
- Someone unplugs the wrong network cable.



Reliability: Hardware Faults

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years.

Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.



Reliability: Hardware Faults

Add redundancy to the individual hardware components in order to reduce the failure rate of the system.

- Disks may be set up in a RAID configuration,
- Servers may have dual power supplies and hot-swappable CPUs
- Datacenters may have batteries and diesel generators for backup power.



Reliability: Hardware Faults

As data volumes and applications' computing demands have increased, more applications have begun using larger numbers of machines, which proportionally increases the rate of hardware faults. (VM on AWS become unavailable) using software fault-tolerance techniques in preference or in addition to hardware redundancy.



Reliability: Hardware Faults

A single-server system requires planned downtime if you need to reboot the machine.

A system that can tolerate machine failure can be patched one node at a time, without downtime of the entire system

Rolling upgrade



Reliability: Software Errors

Another class of fault is a systematic error within the system. Such faults are harder to anticipate, and because they are correlated across nodes, they tend to cause many more system failures than uncorrelated hardware faults



Reliability: Software Errors

- A software bug that causes every instance of an application server to crash when given a particular bad input. For example, consider the leap second on June 30, 2012, that caused many applications to hang simultaneously due to a bug in the Linux kernel.
- A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth.
- A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.
- Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults



Reliability: Software Errors

- Carefully thinking about assumptions and interactions in the system
- thorough testing
- process isolation
- allowing processes to crash and restart
- measuring, monitoring, and analyzing system behavior in production.
- It can constantly check itself while it is running and raise an alert if a discrepancy is found.



Reliability: Human Errors

- Design systems in a way that minimizes opportunities for error.
- Decouple the places where people make the most mistakes from the places where they can cause failures.
- Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests.
- Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure.
- Set up detailed and clear monitoring, such as performance metrics and error rates.
- Implement good management practices and training.



Scalability

Scalability is the term we use to describe a system's ability to cope with increased load.

Perhaps the system has grown from 10,000 concurrent users to 100,000 concurrent users, or from 1 million to 10 million.



Scalability: Describing Load

Load can be described with a few numbers which we call **load parameters**. depends on the architecture of your system

- **requests per second** to a web server,
- the **ratio of reads to writes** in a database
- the **number of simultaneously active users** in a chat room
- the **hit rate** on a cache



Scalability: Twitter Example

Post tweet: A user can publish a new message to their followers (4.6k requests/sec on average, over 12k requests/sec at peak).

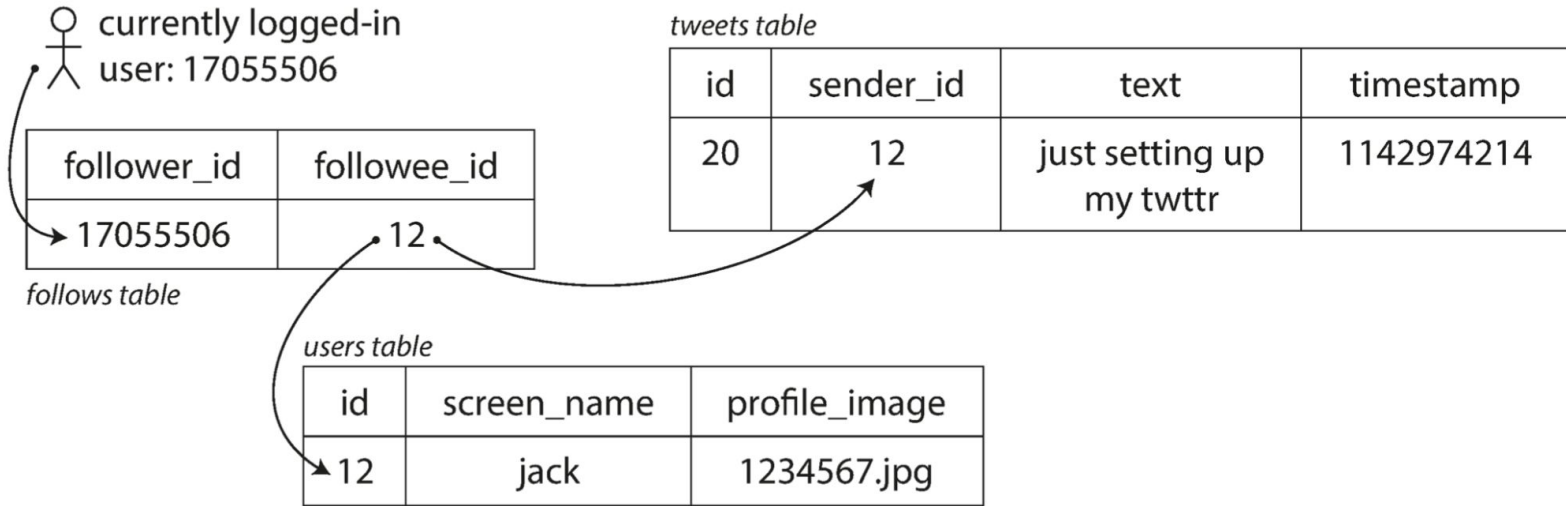
Home timeline: A user can view tweets posted by the people they follow (300k requests/sec).

fan-out: each user follows many people, and each user is followed by many people.



Scalability: Twitter Example

Posting a tweet simply inserts the new tweet into a global collection of tweets. When a user requests their home timeline, look up all the people they follow, find all the tweets for each of those users, and merge them (sorted by time).

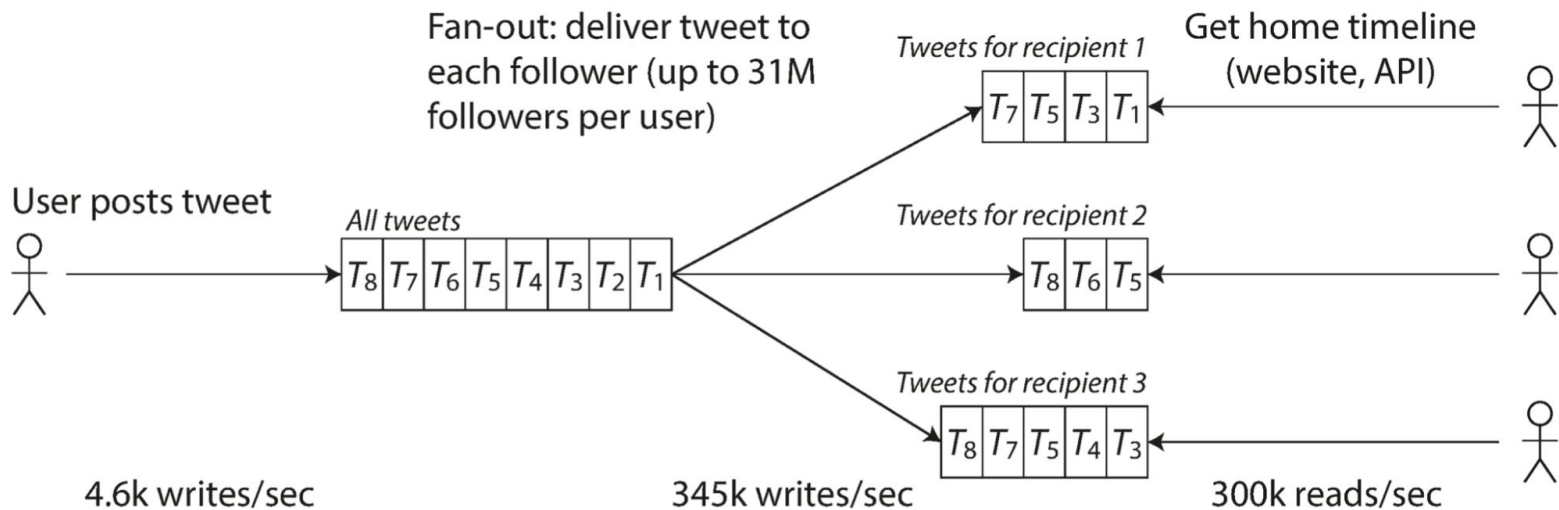


```
SELECT tweets.*, users.* FROM tweets
JOIN users ON tweets.sender_id = users.id
JOIN follows ON follows.followee_id = users.id
WHERE follows.follower_id = current_user
```



Scalability: Twitter Example

Maintain a cache for each user's home timeline—like a mailbox of tweets for each recipient user. When a user posts a tweet, look up all the people who follow that user, and insert the new tweet into each of their home timeline caches. The request to read the home timeline is then cheap, because its result has been computed ahead of time.





Scalability: Twitter Example

Most users' tweets continue to be fanned out to home timelines at the time when they are posted

Tweets from any celebrities that a user may follow are fetched separately and merged with that user's home timeline when it is read.



Scalability: Describing Performance

- When you increase a load parameter and keep the system resources unchanged, how is the performance of your system affected?
- When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?



Scalability: Describing Performance

In a batch processing system such as Hadoop, we usually care about throughput—the number of records 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, what's usually more important is the service's response time—that is, the time between a client sending a request and receiving a response.



Scalability: Describing Performance

Latency vs Response time

The response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queueing delays.

Latency is the duration that a request is waiting to be handled—during which it is latent, awaiting service



Scalability: Describing Performance

It's common to see the average response time of a service reported. However, the mean is not a very good metric if you want to know your “typical” response time, because it doesn't tell you how many users actually experienced that delay.

Usually it is better to use **percentiles**.



Scalability: Describing Performance

In order to figure out how bad your outliers are, you can look at higher percentiles: the 95th, 99th, and 99.9th percentiles are common (abbreviated p95, p99, and p999). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold.



Scalability: Describing Performance

High percentiles of response times, also known as tail latencies, are important because they directly affect users' experience of the service.

Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the **slowest** requests are often those who have the **most data** on their accounts because they have made **many purchases**—that is, they're the most **valuable** customers



Scalability: Describing Performance

It's important to keep those customers happy by ensuring the website is fast for them: Amazon has also observed that a 100 ms increase in response time reduces sales by 1%, and others report that a 1-second slowdown reduces a customer satisfaction metric by 16%

On the other hand, optimizing the 99.99th percentile was deemed too expensive and to not yield enough benefit for Amazon's purposes.



Scalability: Describing Performance

As a server can only process a small number of things in parallel, it only takes a small number of slow requests to hold up the processing of subsequent requests—an effect sometimes known as **head-of-line blocking**.

It is important to measure response times on the client side.



Scalability: Approaches for Coping with Load

scaling up (vertical scaling, moving to a more powerful machine)

scaling out (horizontal scaling, distributing the load across multiple smaller machines).



Maintainability: Operability

Make it easy for operations teams to keep the system running smoothly.

A good operations team typically is responsible for the following.

Data systems can do various things to make routine tasks easy.



Maintainability: Simplicity

Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system.

Reducing complexity greatly improves the maintainability of software, and thus simplicity should be a key goal for the systems we build.



Maintainability: Simplicity

Making a system simpler does not necessarily mean reducing its functionality.

Complexity as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation. One of the best tools we have for removing accidental complexity is **abstraction**.



Maintainability: Evolvability

Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as **extensibility**, **modifiability**, or **plasticity**.

Agile working patterns provide a framework for adapting to change.

Simple and easy-to-understand systems are usually easier to modify than complex ones.



Summary

functional requirements: what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways

nonfunctional requirements: general properties like security, reliability, compliance, scalability, compatibility, and maintainability

Reliability: means making systems work correctly, even when faults occur.

Scalability: means having strategies for keeping performance good, even when load increases.

Maintainability: has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system.