Chapter 1 Reliable, Scalable, and Maintainable Applications

Many applications are data-intensive, as opposed to compute-intensive. Major problems are usually the amount of data, the complexity of data, and the speed it is changing.
A data-intensive application is built from standard building blocks that provide common functionality. E.g, applications need databases, caches, search indexes, stream processing, batch processing.

Thinking About Data Systems

- Tools for data storage and processing are optimized for different use cases.
  Tools are bundled together using application code to meet all needs.
  E.g, if you have an application-managed caching layer (Memcached), or a full-text search server (Elasticsearch or Solr) separate from the main database, it is the application code's responsibility to keep those caches and indexes in sync with the main database.

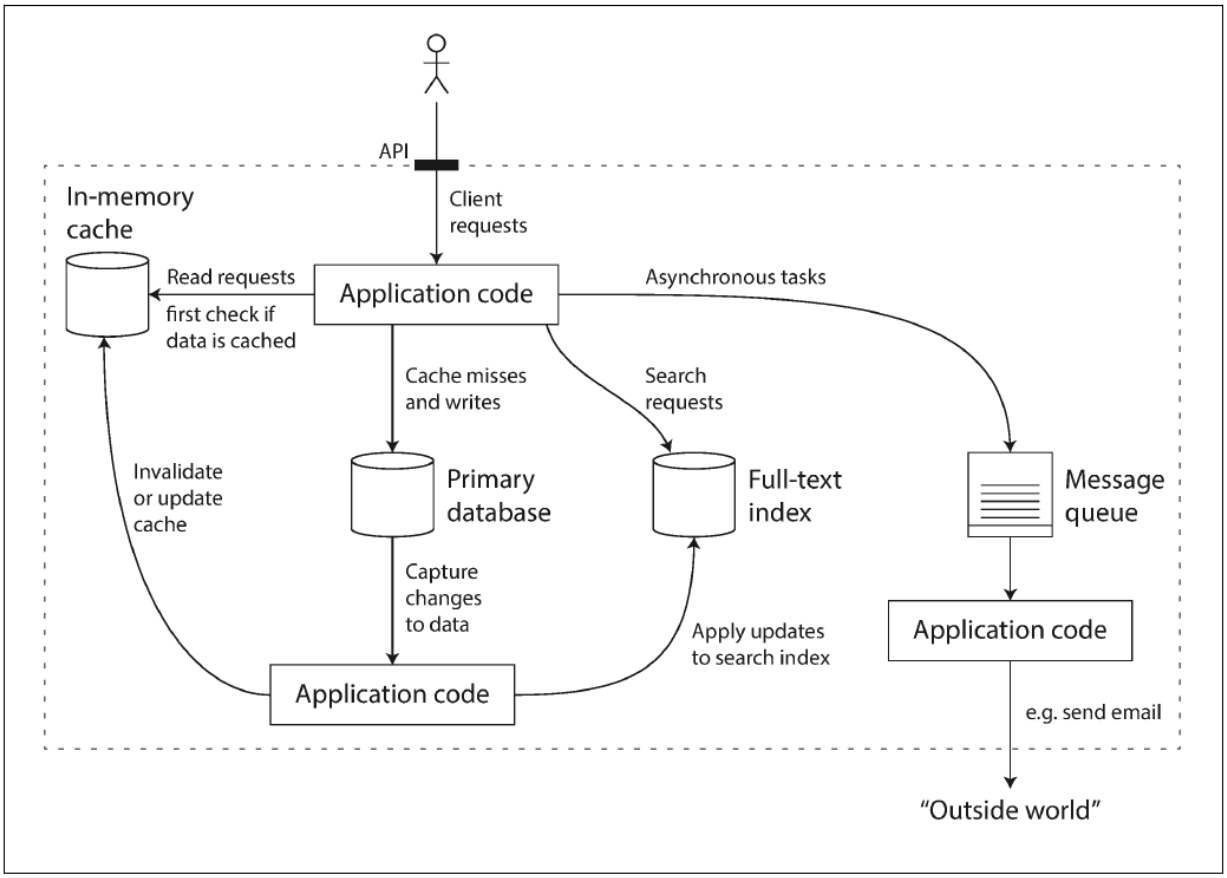

Figure 1-1. One possible architecture for a data system that combines several components.

- When combining several tools to provide a service, the service's interface or **application programming interface (API)** hides implementation details from clients. Creating special-purpose data systems from smaller, general-purpose components.
- Your composite data system provides certain guarantees: e.g, cache is invalidated or updated on writes so clients see consistent results.
  You are not only an application developer, but also a data system designer.

Designing a data system or service:

- Ensure that the data remains correct and complete, even when things go wrong internally?
- Provide consistently good performance to clients, even when parts of your system are degraded?
- Scale to handle an increase in load?
- What does a good API for the service look like?

Many factors influence the design of a data system:

- Skills and experience of the people involved.
- Legacy system dependencies, the timescale for delivery.
- Your organization's tolerance of different kinds of risk, regulatory constraints.

We focus on three important concerns in most software systems:

- **Reliability**

The system continues to work correctly (perform **correct function** at desired level of **performance**) in adversity (hardware or software faults, human error).

- **Scalability**
  As the system grows (in **data volume, traffic volume, or complexity**), there are reasonable ways of dealing with the growth.
- **Maintainability**
  Different people who work on the system should be able to work on it productively.

Reliability

Typical expectations include:

- The application performs the function the user expected.
- It tolerates users making mistakes or using it unexpectedly.
- Performance is good for the required use case, under expected load and data volume.
- The system prevents any unauthorized access and abuse.

Reliability means continuing to work correctly even when things go wrong.
Things go wrong called **faults**, systems that anticipate faults and cope with them are **fault-tolerant** or **resilient**.
A fault is not the same as a failure. A fault is one component of the system deviated, a failure is when the system as a whole stops working.
Design fault-tolerance mechanisms to prevent faults from causing failures. We will discuss building reliable systems from unreliable parts.

Hardware Faults

- Hard disks crash, RAM becomes faulty, the power grid has a blackout, someone unplugs the wrong network cable. Things often happen in large datacenters.
- Hard disks have a **mean time to failure (MTTF)** of about 10 to 50 years. On a storage cluster with 10,000 disks, we should expect on average one disk to die per day.
- Add redundancy to individual hardware components to reduce failure rate of the system.
- As data volumes and applications' computing demands increased, applications began to use larger numbers of machines, proportionally increasing the rate of hardware faults.
- In AWS, it is common for virtual machine instances to be unavailable without warning, as the platforms are designed to prioritize flexibility and elasticity over single-machine reliability.
- **Rolling upgrade**: A system that tolerates machine failure can be patched one node at a time, without downtime of the entire system.

Software Errors

- Hardware faults are random and independent from each other, it is unlikely a large number of hardware components fail at the same time.

- Another class of fault is a **systematic error within the system**, they are correlated across nodes, and tend to cause more system failures than hardware faults.
- Things can help to prevent systematic faults in software: 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.
- If a system provides some guarantee (e.g., in message queue, number of incoming messages equals number of outgoing messages), it can constantly check itself and raise an alert if a discrepancy is found.

Human Errors

How to make systems reliable, in spite of unreliable humans? The best systems combine several approaches:

- Design systems minimize opportunities for error. E.g, well-designed abstractions, APIs, and admin interfaces, make it easy to do “the right thing”.
- Decoupling the places people make the most mistakes from the places can cause failures. Provide fully featured non-production sandbox environments for people to experiment safely, using real data, not affecting real users.
- Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests. Automated testing is widely used, well understood, and covers corner cases that rarely arise in normal operation.
- Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure. Fast roll back configuration changes, roll out new code gradually, provide tools to recompute data.
- Set up detailed and clear monitoring, such as performance metrics and error rates. Also referred to as telemetry.
- Implement good management practices and training.

Scalability

Scalability is the term to describe a system’s ability to cope with increased load.
scalability means considering questions like “If the system grows in a particular way, what are our options for coping with the growth?” and “How to add computing resources to handle the additional load?”

Describing Load

Load can be described by load parameters.
Twitter as example, two main operations:

- **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).

Twitter's scaling challenge is not due to tweet volume, but due to **fan-out**: each user follows many people, and each user is followed by many people.

Two ways to implement these two operations:

1. Posting a tweet 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 the users, and merge them (sorted by time). In a relational database, write a query as:

```
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
```

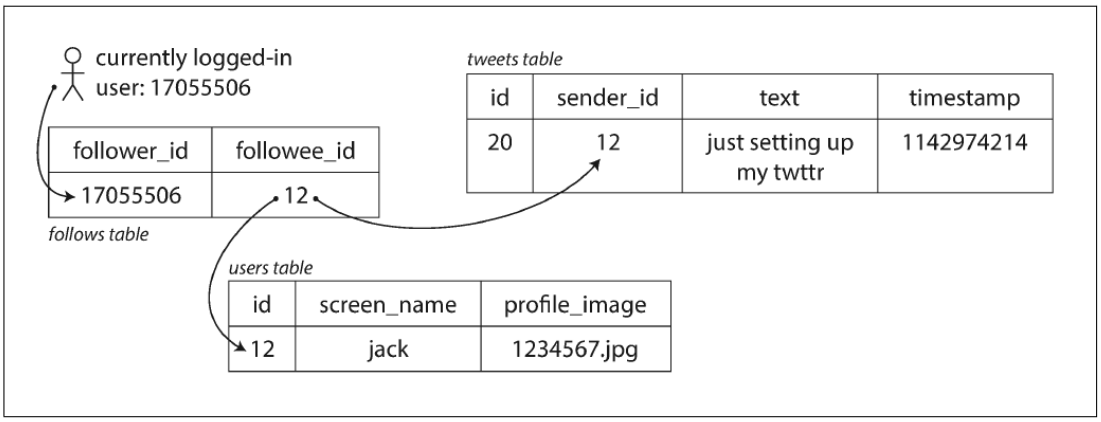

Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

2. 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. Reading the home timeline is cheap, because its result is computed ahead of time.

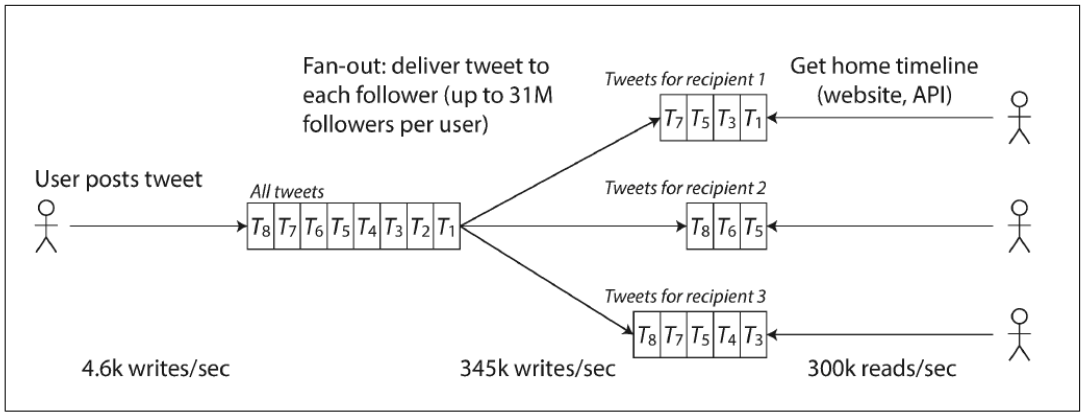

Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- The first version of Twitter used 1, but systems struggled with the load of home timeline queries. 2 works better since the average rate of published tweets is two orders of magnitude lower than the rate of home timeline reads, better to do more work at write time, less at read time.
- Downside of 2: posting a tweet requires extra work. On average, a tweet is delivered to 75 followers, 4.6k tweets/sec become 345k writes/sec to home timeline caches.
- But the number of followers per user varies wildly, some users have 30 million followers. This means a tweet may result in over 30 million writes to home timelines!
  Twitter tries to deliver tweets to followers within five seconds.
- The distribution of followers per user is a key **load parameter** for scalability, it determines the fan-out load.
- The final twist of the Twitter anecdote: **a hybrid of both approaches**.
  Most users' tweets fanned out to home timelines when posted.
  Celebrities are excluded from this fan-out.
  Tweets from celebrities that a user follows are fetched separately and merged with the user's home timeline when read, like 1.
  This hybrid approach is able to deliver consistently good performance.

Describing Performance

- In a batch processing system like Hadoop, we care about throughput: number of records processed per second, or total time to run a job on a certain size of dataset.
  In online systems, service's response time: the time between a client sending a request and receiving a response.
- Latency and response time are not the same:
  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 a request is waiting to be handled.
- Response time is not as a single number, but a distribution of values to measure.
  Service reported average response time. The mean is not a good metric to know your response time, because it doesn't tell how many users actually get delayed.
- It is better to use **percentiles**. Sort response times from fastest to slowest, the median is the halfway point. The median is the **50th percentile or p50**.
  If the median response time is 200 ms, it means half requests return in less than 200 ms, and half requests take longer.

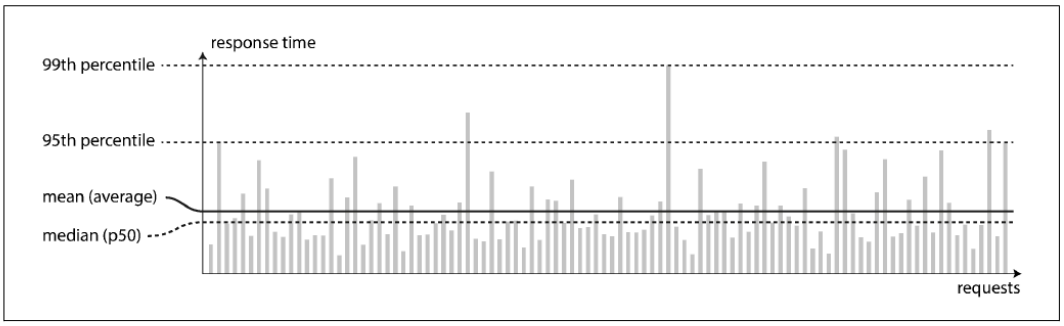

Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

- Look at higher percentiles to see how bad your outliers are: 95th, 99th, and 99.9th percentiles (p95, p99, and p999).
- **Tail latencies**, **high percentiles of response times** are important because they directly affect users' experience of the service.
  E.g, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it affects 1 in 1,000 requests. The customers with the slowest requests are those that have the most data on their accounts because they made many purchases, they're the most valuable customers.
  Amazon also observed that a 100 ms increase in response time reduces sales by 1%, a 1-second slowdown reduces a customer satisfaction metric by 16%.
- On the other hand, optimizing the 99.99th percentile (the slowest 1 in 10,000 requests) is too expensive. Reducing response times at high percentiles is difficult because they are easily affected by random events outside of control, benefits are diminishing.
- **Percentiles** are used in **service level objectives (SLOs)** and **service level agreements (SLAs)**, **contracts that define the expected performance and availability of a service**.
  An SLA may state the service is considered to be up if it has a median response time of less than 200 ms and a 99th percentile under 1s, and the service is required to be up at least 99.9% of the time.
  These metrics set expectations for clients of the service and allow customers to demand a refund if the SLA is not met.
- **Queueing delays account for a large part of the response time at high percentiles.**
- It only takes a few slow requests to hold up the processing of subsequent requests, as a server only processes a few things in parallel. This effect is **head-of-line blocking**.
  A high proportion of end-user requests being slow, this effect is **tail latency amplification**.
- It is important to measure **response times** on the **client** side.
- When generating load artificially to test the scalability of a system, the load-generating client needs to keep sending requests independently of the response time.
- The naïve implementation of response time percentiles monitoring is to keep a list of response times for all requests within the time window and sort the list every minute. Or there are algorithms that calculate approximation of percentiles at minimal CPU and memory cost. Aggregate response time data by adding the histograms.

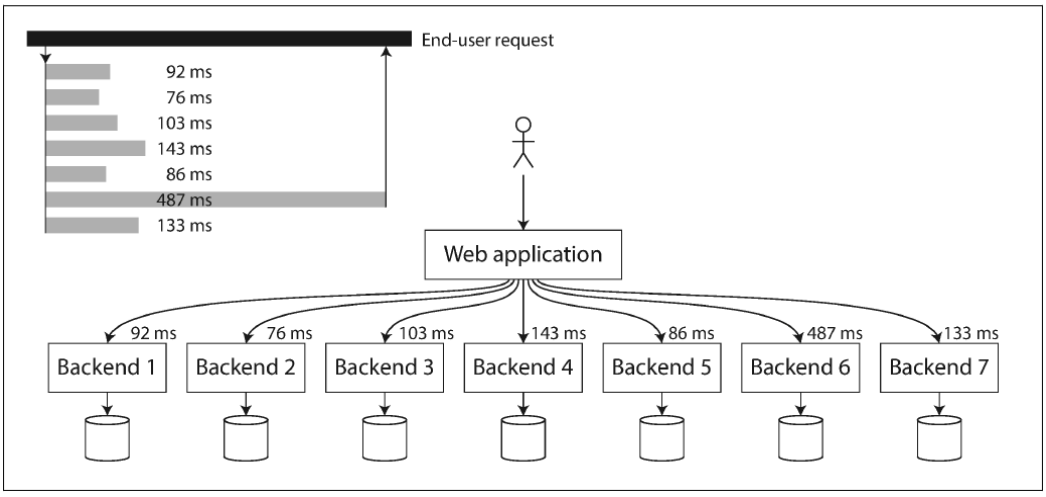

Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Approaches for Coping with Load

- How to maintain good performance when load parameters increase by some amount?
  If you are working on a fast-growing service, you need to rethink your architecture on every order of magnitude(10 times) load increase.
- **Scaling up / Vertical scaling**: moving to a more powerful machine.
  Scaling out / Horizontal scaling: distributing load across multiple small machines.
  Distributing load across multiple machines is a shared-nothing architecture.
- Some systems are **elastic**: meaning they can **automatically add computing resources when detecting a load increase**, whereas other systems are scaled manually.
  An elastic system is useful if load is highly unpredictable, but manually scaled systems are simpler and have fewer operational surprises.
- While distributing stateless services across multiple machines is straightforward, taking stateful data systems to a distributed setup can introduce complexity.
  For this reason, keep your database on a single node (scale up) until scaling cost or high availability requirements force you to make it distributed.
- The architecture of systems that operate at large scale is highly specific to the application, there is no generic scalable architecture. The 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 these add more issues.
- An architecture that scales well for a particular application is built around assumptions of which operations to be common and which to be rare, the load parameters.
  If those assumptions are wrong, the engineering effort for scaling may be counterproductive. In an early-stage startup or an unproven product it's more important to iterate quickly on product features than to scale to some hypothetical future load.

- Scalable architectures are built from general-purpose building blocks, arranged in familiar patterns.

Maintainability

Majority of the cost of software is not in development, but in its ongoing maintenance. Fixing bugs, keeping its systems operational, investigating failures, adapting to new platforms, modifying for new use cases, repaying technical debt, and adding new features.

Three design principles for software systems:

1. **Operability**: Make it easy for operations teams to keep the system running smoothly.
2. **Simplicity**: Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system.
3. **Evolvability / Extensibility / Modifiability / Plasticity**: Make it easy for engineers to make changes to the system later, adapting it for future use cases as requirements change.

Operability: Making Life Easy for Operations

Operations are vital to keeping a software system running smoothly. Operations team is responsible for the following:

- Monitoring the health of the system and quickly restore service if it goes into a bad state.
- Tracking down the cause of problems, like 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.(e.g., capacity planning)
- Establishing good practices and tools for deployment, configuration management, etc.
- 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 keep the production environment stable.
- Preserving the organization's knowledge about the system, even as people come and go.

Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities. **Data systems can do various things to make routine tasks easy** including:

- 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.
- Providing good documentation and an easy-to-understand operational model.

- 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

- As projects get larger, they become complex and difficult to understand. This complexity slows down who works on the system, increasing the cost of maintenance.
- In complex software, Greater risk of introducing bugs when making a change. Reducing complexity improves the maintainability of software. **Simplicity should be a key goal for the systems we build.**
- Making a system simpler does not mean reducing functionality, it means removing accidental complexity.
- One of the best tools to remove accidental complexity is **abstraction**.
  A good abstraction hides implementation detail behind a clean, simple-to-understand façade. It can be used by different applications.
  This reuse is efficient than reimplementing, it leads to higher-quality software, as quality improvements in the abstracted component benefit all applications using it.
- E.g, high-level programming languages are abstractions that hide machine code, CPU registers, and syscalls. SQL is an abstraction that hides complex on-disk and in-memory data structures, concurrent requests from other clients, and inconsistencies after crashes.

Evolvability: Making Change Easy

- **Evolvability: agility on a data system level.**
- In terms of organizational processes, Agile working patterns provide a framework for adapting to change. The Agile community developed technical tools and patterns that are helpful when developing software in a frequently changing environment, such as test-driven development (TDD) and refactoring.
- We look for ways to increase agility on the level of a larger data system.
- The ease to modify a data system, and adapt it to changing requirements, is linked to simplicity and abstractions: simple and easy-to-understand systems are easier to modify than complex ones.