DDIA Chapter1:

A data-intensive application is typically built from standard building blocks that provide commonly needed functionality.

Many applications need to:

• Store data so that they, or another application, can find it again later (databases)

• Remember the result of an expensive operation, to speed up reads (caches)

• Allow users to search data by keyword or filter it in various ways (search indexes)

• Send a message to another process, to be handled asynchronously (stream processing)

• Periodically crunch a large amount of accumulated data (batch processing)

The three most important concerns the book focuses on are:

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).

Scalability As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.

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.

Reliability:

Typical expectations include:

• 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.

The things that can go wrong are called faults, and systems that anticipate faults and can cope with them are called fault-tolerant or resilient.

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

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.

Adding redundant hardware components (like RAID for disks, dual power supplies, and backup power) is a common way to reduce system failure rates. When a component fails, the redundant one takes over, allowing for uninterrupted operation for potentially years.

As long as you can restore a backup onto a new machine fairly quickly, the downtime in case of failure is not catastrophic in most applications.

As data and computing needs grow, applications use more machines, increasing the likelihood of hardware failures. Cloud platforms sometimes prioritize flexibility over individual machine reliability, leading to unexpected outages. This trend is driving a shift towards software-based fault tolerance, allowing systems to withstand the loss of entire machines and offering advantages like rolling upgrades without system downtime.

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.

Ex:

• A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth.

• A service that the system depends on 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.

The bugs that cause these kinds of software faults often lie dormant for a long time until they are triggered by an unusual set of circumstances.

Scalability

Even if a system is working reliably today, that doesn't mean it will necessarily work reliably in the future.

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

Describing Load

Load can be described with a few numbers which we call load parameters. The best choice of parameters depends on the architecture of your system: it may be 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, or something else.

Describing Performance

Once you have described the load on your system, you can investigate what happens when the load increases.

• 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?

• When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?

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.

Latency and response time:

Latency and response time are often used synonymously, but they are not the same. 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.

While most requests are fast, occasional slow ones occur due to factors like processing more data or random latency from context switches, network issues, garbage collection, etc. Reporting the average response time isn't ideal for understanding typical user experience. Instead, percentiles are better. The median (p50) shows the time below which half of the requests fall, representing the typical wait time. Higher percentiles (p95, p99, p999) help identify and understand the performance of outlier requests and how bad the slowest experiences are.

Approaches for Coping with Load

To maintain good performance with increasing load, systems need to scale. This can be done by **scaling up (vertical scaling to more powerful machines**) or **scaling out (horizontal scaling by distributing load across multiple machines**, also known as a shared-nothing architecture). While single-machine systems are simpler, **high-end machines can be expensive**, often necessitating scaling out for heavy workloads. Effective architectures often use a combination of both approaches. Systems can also be elastic, **automatically adding resources with increased load**, or scaled manually. Elastic systems are beneficial for unpredictable loads, but manual scaling is simpler.

A well-scaling architecture is designed based on assumptions about common and rare operations. If these assumptions are incorrect, scaling efforts can be wasted or even harmful. In early-stage startups or for unproven products, it's often **more crucial to prioritize rapid iteration on product features** over **premature scaling for potential future load.**

Maintainability

We can and should design software in such a way that it will hopefully minimize pain during maintenance and thus avoid creating legacy software ourselves.

To this end, we will pay particular attention to three design principles for software systems:

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, adapting it for unanticipated use cases as requirements change. Also known as extensibility, modifiability, or plasticity.

Operability: Making Life Easy for Operations

Operations teams are essential for the smooth operation of software systems, with responsibilities including monitoring, **troubleshooting**, maintenance, **security**, **capacity planning**, and establishing operational best practices. Good operability simplifies routine tasks for these teams by **providing system visibility through monitoring**, supporting automation, avoiding single-machine dependencies, **offering clear documentation** and operational models, having good defaults with override options, incorporating self-healing with manual control, and exhibiting predictable behavior.

Simplicity: Managing Complexity

Complexity in software systems can be identified by symptoms like state space explosion and tight coupling, making maintenance hard and increasing the risk of bugs. Reducing complexity is crucial for maintainability, and simplicity should be a primary goal. Accidental complexity, which **arises from implementation rather than the problem itself**, can be removed using abstraction. Good **abstractions hide implementation details**, promote reuse, and lead to higher-quality software, as exemplified by high-level programming languages and SQL.

Evolvability: Making Change Easy

The ease with which you can modify a data system, and adapt it to changing requirements, is closely linked to its simplicity and its abstractions: simple and easy-to understand systems are usually easier to modify than complex ones. But since this is such an important idea, we will use a different word to refer to agility on a data system level: evolvability.