- An application has to meet various requirements in order to be useful. Some nonfunctional requirements: security, reliability, compliance, scalability, compatibility, and maintainability.
- Data-intensive applications - the amount of data, the complexity of data, and the speed at which it changes. These days those are the most common applications.
- Compute-intensive applications - CPU power is a limiting factor.

Applications usually need to:
- store data (*databases*),
- speed up reads (*caches*),
- search data (*search indexes*),
- send a message to another process asynchronously (*stream processing*),
- periodically crunch data (*batch processing*).

The border is quite blurred because there are datastores that are also used as message queues (Redis), and there are message queues with database-like durability guarantees (Apache Kafka).

Reliability:
- Reliability means making systems work correctly, even when faults occur.
- A **fault** is not the same as a **failure**, fault is about one component, and a failure is about the whole system. We need to prevent faults from causing failures. Faults in a system can be handled with **fault-tolerant** or resilient systems.
- Examples of 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.
    - faulty RAM,
    - blackouts in the power grid,
    - an accidental disconnection of network cables
- Software errors are systematic faults that can cause many system failures. Unlike hardware faults, these errors are often correlated across nodes.Measures to reduce the impact of software errors include:
    - thorough testing
    - process isolation
    - allowing processes to crash and restart
    - careful consideration of assumptions and interactions in the system.
- Human errors are a leading cause of outages, with configuration errors being the primary culprit. Only a small percentage of outages are due to hardware faults.
- Prevention:
    - Design systems to minimize opportunities for error, including well-designed abstractions, APIs, and admin interfaces.
    - Provide fully-featured non-production sandbox environments for experimentation.
    - Test thoroughly at all levels, from unit tests to whole-system integration tests and manual - tests.

    - Allow quick and easy recovery from errors.
    - Set up detailed and clear monitoring, such as performance metrics and error rates. This is referred to as telemetry.
    - Provide training to operators.

Scalability:

- Scalability is the ability of a system to maintain performance as load increases
- An architect needs to describe the right parameters to measure. Load parameters describe system load and may include requests per second to a web server, read-to-write ratios in a database, active users in a chat room, cache hit rates, or other relevant factors. Once you have described the load on your system, you can investigate what happens when the load increases.
- What happens when the load increases:
    - How is the performance affected?
    - How much do you need to increase your resources?
- In a batch processing system like Hadoop, we usually care about **throughput**, or how many requests an app can handle simultaneously.
- Online systems prioritize **response time,** specifically the time between a client sending a request and receiving a response.
- Response time and latency are not synonyms. Response time includes network and queuing delays in addition to the actual service time, whereas **latency** is how fast a system responds to a single request. Percentiles are commonly used to measure response time, with high percentiles (95th, 99th, and 99.9th) important for understanding **tail latencies** (refers to high latencies that clients see fairly infrequently. Things like: “my service mostly responds in around 10ms, but sometimes takes around 100ms”)
- We want to **maximize throughput** while **minimizing latency**

How to deal with increased load?

- Scaling up or **vertical scaling**: Moving to a more powerful machine
- Scaling out or **horizontal scaling**: Distributing the load across multiple smaller machines.
- **Elastic systems**: Automatically add computing resources when detected load increases. Quite useful if the load is unpredictable.
- It is relatively easy to distribute **stateless services** across multiple machines. However, transitioning **stateful data systems** from a single node to a distributed setup can be quite complex. In the past, it was generally advised to keep databases on a single node until it became necessary to distribute them due to scaling costs or high availability requirements.
    - Stateless means there is no memory of the past. Every transaction is performed as if it were being done for the very first time.
    - Stateful means that there is memory of the past. Previous transactions are remembered and may affect the current transaction.
- **Shared-nothing architecture**: each processor has its own set of disks. Data is horizontally partitioned across nodes, so each node has a subset of the rows from each

table in the database. Each node is then responsible for processing only the rows on its own disks.

To design an architecture that scales effectively for a particular application, it is crucial to make assumptions about the frequency of common and rare operations, referred to as load parameters.

Maintainability:

- Maintainability aims to improve the experience of engineering and operations teams working with the system. Three critical design principles for software systems to enhance maintainability are operability, simplicity, and evolvability.
- Operability ensures that the system is easy for operations teams to manage and maintain to ensure it runs smoothly.
  - Monitoring the system’s health and restoring service promptly during issues
  - Identifying the root cause of problems like system failures or reduced performance
  - Ensuring that software and platforms are updated, including security patches
  - Monitoring the system’s interdependencies with other systems
  - Maintaining the system’s security
  - Good operability is achieved by simplifying routine tasks, freeing up the operations team’s time to focus on higher-value activities.
- Simplicity reduces complexity as much as possible to make it easier for new engineers to understand the system. However, note that simplicity in this context doesn’t refer to the simplicity of the user interface.
  - Symptoms:
    - Explosion of state space
    - Tight coupling of modules
    - Tangled dependencies
    - Inconsistent naming and terminology
    - Performance-related hacks
    - Special cases to work around issues elsewhere
  - Abstraction as a tool for reducing complexity:
    - Hides implementation details behind a simple interface
    - High-level programming languages abstract machine code, CPU registers, and syscalls
    - SQL abstracts complexities of on-disk and in-memory data structures, concurrent requests, and inconsistencies after crashes.
- Evolvability makes it easy for engineers to make changes to the system as the requirements change, adapting it for unanticipated use cases. This principle is also known as extensibility, modifiability, or plasticity.
  - Agile provides a framework for adapting to change, and technical tools such as TDD and refactoring aid software development in a constantly changing environment. A data system’s simplicity and abstractions affect its ability to be modified and adapted, referred to as evolvability.