# Reliable, Scalable, and Maintainable Applications

Arnav Gupta

January 12, 2025

## Contents



## 1 Introduction

**Data-intensive applications** are built from standard building blocks that provide commonly needed functionality. These building blocks include databases, caches, search indexes, stream processing, and batch processing.

When building an application, its important to figure out which tools and

approaches are most appropriate for the task at hand.

# 2 Thinking About Data Systems

In recent years, boundaries between categories of data systems are becoming blurred.

Work is broken down into tasks that can be performed efficiently on a single tool, with different tools stitched together using application code.

When combining tools to provide a service, the API hides implementation details from clients.

# 3 Reliability

**Reliability**: the system should continue to work *correctly* (perform correct function at desired level of performance) even in the face of *adversity* (hardware or software faults, even human error)

More roughly, reliability means to continue to work correctly, even when things go wrong.

**Fault**: a component of the system deviating from the spec

**Fault-tolerant/resilient**: systems that anticipate faults and can cope with them

**Failure**: when the system as a whole stops providing the required service to the user

Impossible to reduce probability of a fault to zero, so best to prevent faults from causing failures. Generally better to tolerate faults than prevent faults, but must also sometimes prevent faults.

One strategy involves deliberately inducing faults.

## 3.1 Hardware Faults

Hard disks have a mean time to failure (MTTF), so with many disks, some number of disks will die every day.

Can add redundancy to individual hardware components to reduce the failure rate of the system (like using RAID, backup power, etc). This cannot

completely prevent hardware problems from causing failures, but is well understood and has decent success rate.

Redundancy was sufficient in the past since if a backup is restored onto a new machine fairly quickly, downtime in case of failure is not catastrophic in most applications.

As data volumes and computing demands increased, more applications use more machines and so hardware fault rates have increased. Platforms are now designed to prioritize flexibility and elasticity over single-machine reliability.

Systems should tolerate the loss of entire machines using software and hardware techniques.

### 3.2 Software Faults

Hardware faults are generally have no correlation.

Systematic errors within the system are correlated across nodes and cause more system failures. This includes software bugs, runaway processes, slowdowns, and cascading failures.

Software faults often lie dormant and are triggered by an unusual set of circumstances.

Solutions include:

- considering assumptions made
- thorough testing
- process isolation
- monitoring system behaviour

### 3.3 Human Errors

Humans can be unreliable. To mitigate this:

- design system to minimize opportunities for error
- decouple places where people make mistakes from places where they can cause failures
- test thoroughly at all levels
- allow quick and easy recovery from human errors

- setup detailed and clear monitoring, called **telemetry**
- implement good management practices and training

### 3.4 How Important is Reliability?

Bugs can have large cost with respect to lost revenue and reputation damage.

Can sacrifice reliability to reduce development cost or operation cost, but must be conscious of cost cutting.

## 4 Scalability

**Scalability**: as the system *grows* (data volume, traffic volume, complexity), there should be reasonable ways of dealing with growth (cope with increased load)

### 4.1 Describing Load

Load can be described with **load parameters**.

Choice of load parameters depends on the architecture.

Choice of average vs worst case depends on specific system.

### 4.2 Describing Performance

Can be helpful to examine:

- how system performance is affected by increase on load parameter and keep system resources unchanged
- how much resources must be increased with load to keep performance unchanged

This requires a way to describe performance.

**Throughput**: number of records that can be processed per time unit

**Response time**: time between client sending request and receiving a response

**Latency**: duration that a request is waiting to be handled

Response time can be considered as a distribution. Within this distribution, good metrics can be average response time or percentiles like median, 95th percentile, 99th percentile, etc.

**Tail latencies**: high percentiles of response time, directly affect user experience

Reducing response times at very high percentiles is difficult since they are affected by random events outside control and benefits are diminishing.

Percentiles are used in service level objectives and agreements to define expected performance and availability.

**Head-of-line blocking**: small number of slow requests holding up subsequent requests

To simulate queues accurately, generate load independently of response time.

Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple backend calls, so a higher proportion of end-user requests end up being slow (**tail latency amplification**).

### 4.3 Approaches for Coping with Load

Must rethink architecture on every order of magnitude load increase.

**Shared-nothing architecture**: distributing load across multiple machines

**Elastic system**: can automatically add computing resources when they detect a load increase (good for unpredictable load)

Stateless services are easy to horizontally scale, stateful services are difficult to.

Distributed data systems are default. The way to scale an architecture is specific to the application.

## 5 Maintainability

**Maintainability**: over time, everyone that works on the system (maintaining current behaviour and adapting system to new use cases) should be able to work on it productively

Main cost in software is ongoing maintenance.

## 5.1 Operability: Making Life Easy for Operations

**Operability**: make it easy for operations teams to keep system running smoothly

Operations teams:

- monitor system health
- track down problem causes
- keep software up to date
- keep tabs on system interactions
- anticipate future problems
- establish good practices and tools for deployment
- performance complex maintenance tasks
- define processes that make operations predictable
- preserve organization knowledge about the system

Good operability means making routine tasks easy, allowing operations team to focus efforts on high-value activities.

## 5.2 Simplicity: Managing Complexity

**Simplicity**: make it easy for new engineers to understand the system by removing as much complexity as possible from the system

As projects get larger, they can become complex and difficult to understand.

Reducing complexity greatly improves the maintainability of software.

Making a system simpler does not necessarily mean reducing functionality, it can also mean removing accidental complexity (arises only from implementation).

**Abstraction** can hide implementation detail behind a clean, simple-to-understand facade.

## 5.3 Evolvability: Making Change Easy

**Evolvability**: make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change

Agile working patterns provide a framework for adapting to change. These include test-driven development and refactoring.

The ease with which a system can be adapted to changing requirements is linked to its simplicity and abstractions.