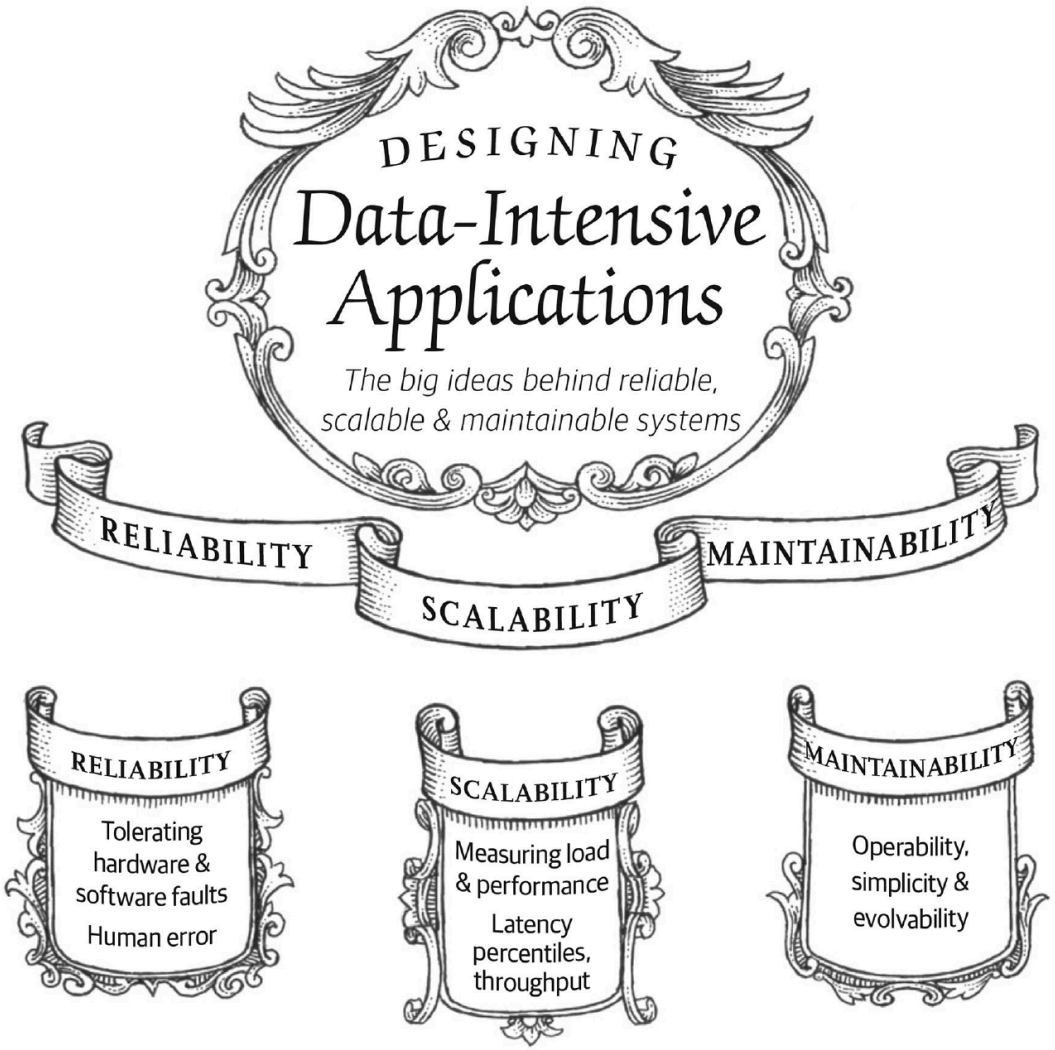
Reliable Scalable Maintainable Applications

Hrachya Tandilyan

2020

Reliable, Scalable, and Maintainable Apps



Data-Intensive vs Compute-Intensive Apps

Many applications today are **data-intensive**, as opposed to **compute-intensive**.

Raw CPU power is rarely a limiting factor for these applications - bigger problems are following:

- *The amount of data*
- *The complexity of data*
- *The speed at which data is changing*

Data-Intensive Apps Building Blocks

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

- **Databases** - Store data so that they, or another application, can find it again later.
- **Caches** - Remember the result of an expensive operation, to speed up reads.
- **Search Indexes** - Allow users to search data by keyword or filter it in various ways.
- **Stream Processing** - Send a message to another process, to be handled asynchronously.
- **Batch Processing** - Periodically crunch a large amount of accumulated data.

Architecture Example With Several Components

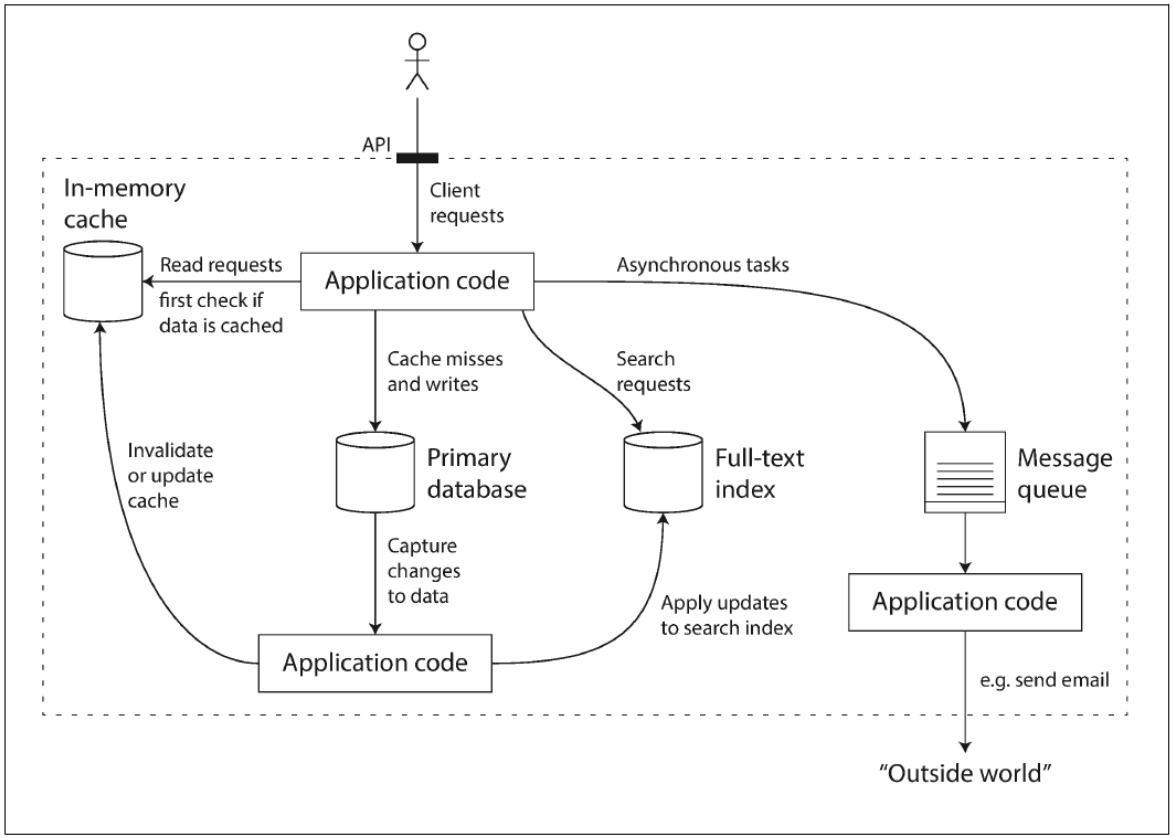


Figure 1-1. One possible architecture for a data system that combines several components.

Thinking About Data Systems

Now you have a new, special-purpose **Data System** from smaller, general-purpose components.

When you combine several components in order to provide a service, the service's interface or *Application Programming Interface* ***(API)*** usually hides implementation details from clients.

Your composite data system may provide **certain guarantees**:
e.g., that the cache will be correctly invalidated or updated on writes so that outside clients see consistent results.

If you are designing a data system or service, a lot of **tricky questions** arise:

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

Reliability, Scalability and Maintainability

In this course, we focus on three concerns that are important in most software systems:

- ***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

For software, typical expectations for **being reliable** 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.

If all those things together mean ***"working correctly"***, then we can understand reliability as meaning ***"continuing to work correctly, even when things go wrong"***.

Faults

The things that can go wrong are called **faults**.

Systems that anticipate faults and can cope with them are called **fault-tolerant** or **resilient**.

The former term is slightly *misleading*: it suggests that we could make a system tolerant of every possible kind of fault, which in reality is not feasible.

Note that a ***fault*** is not the same as a ***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.

It is impossible to reduce the probability of a fault to zero; therefore it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures.

Hardware Faults

Examples of hardware faults:

Hard disks crash, **RAM** becomes faulty, the **power grid** has a blackout, someone unplugs the wrong **network cable**. These things happen all the time when you have a lot of machines.

First response is usually to add redundancy to the individual hardware components: Disks may be set up in a **RAID configuration**, servers may have **dual power supplies** and **hot-swappable CPUs**, and datacenters may have **batteries and diesel generators** for backup power.

When one component dies, the redundant component can take its place while the broken component is replaced. This approach cannot completely prevent hardware problems from causing failures, but it is well understood and can often keep a machine running uninterrupted for years.

Until recently, redundancy of hardware components was sufficient for most applications, since it makes *total failure of a single machine fairly rare*.

Software Errors

Human Errors

How Important Is Reliability?

Scalability

Describing Load

Describing Performance

Approaches for Coping with Load

Maintainability

Operability: Making Life Easy for Operations

Simplicity: Managing Complexity

Evolvability: Making Change Easy

Summary