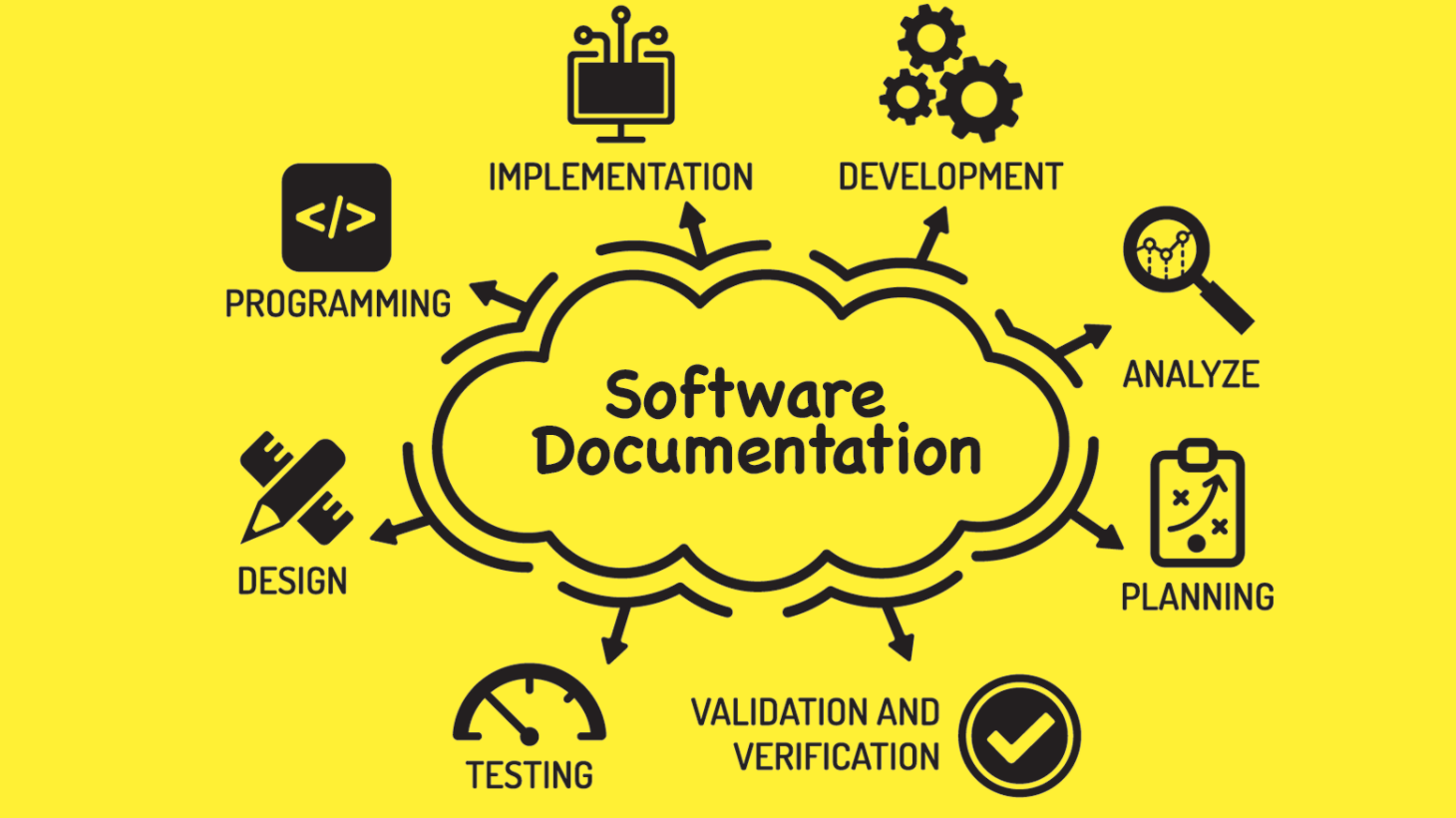
# System Design — TRD Cheatsheet

## The best practice creat technical requirement documentation



<https://www.cubix.co/blog/documentation-software-architecture>

Here is the best practice that I always use when creating TRD or Review TRD for my Team.

## Basic Steps

Clarify and agree on the scope of the system. Before writing TRD first we must agree about the scope of the system. All stakeholders must understand and agree. The scope must contain the following.

* **User Cases** (description of sequences of events that, taken together, lead to a system doing something useful).Who is going to use it? How are they going to use it?
* **Constraints**. Mainly identify traffic and data handling constraints at scale. The scale of the system such as requests per second, requests types, data written per second, data read per second). Special system requirements such as multi-threading, read or write oriented.

## High-Level Architecture

Create a high-level architecture design (abstract design). Sketch the important components and connections between them, but don’t go into some details.

* Application service layer (serves the requests).
* List the different services required.
* Data Storage layer.
* eg. Usually, a scalable system includes webserver (load balancer), service (service partition), database (master/slave database cluster), and caching system.

## Component Design

Create a component design that contains the following.

* Component + specific APIs required for each of them.
* Object-oriented design for functionalities.
* The map features to modules: One scenario for one module.
* Consider the relationships among modules.
* Certain functions must have a unique instance (Singletons).
* A core object can be made up of many other objects (composition).
* One object is another object (inheritance).
* Database schema design.

## Bottlenecks of The System

You must understand bottlenecks will come from.

* Perhaps your system needs a load balancer and many machines behind it to handle the user requests. \* Or maybe the data is so huge that you need to distribute your database on multiple machines. What are some of the downsides that occur from doing that?
* Is the database too slow and does it need some in-memory caching?

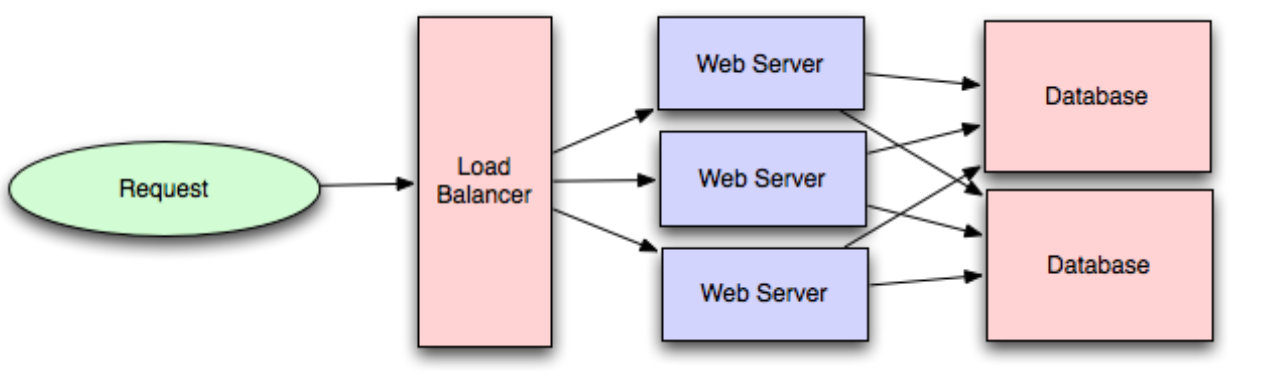
## Scaling Abstract Design

Our design must be able to scale with the following strategy.

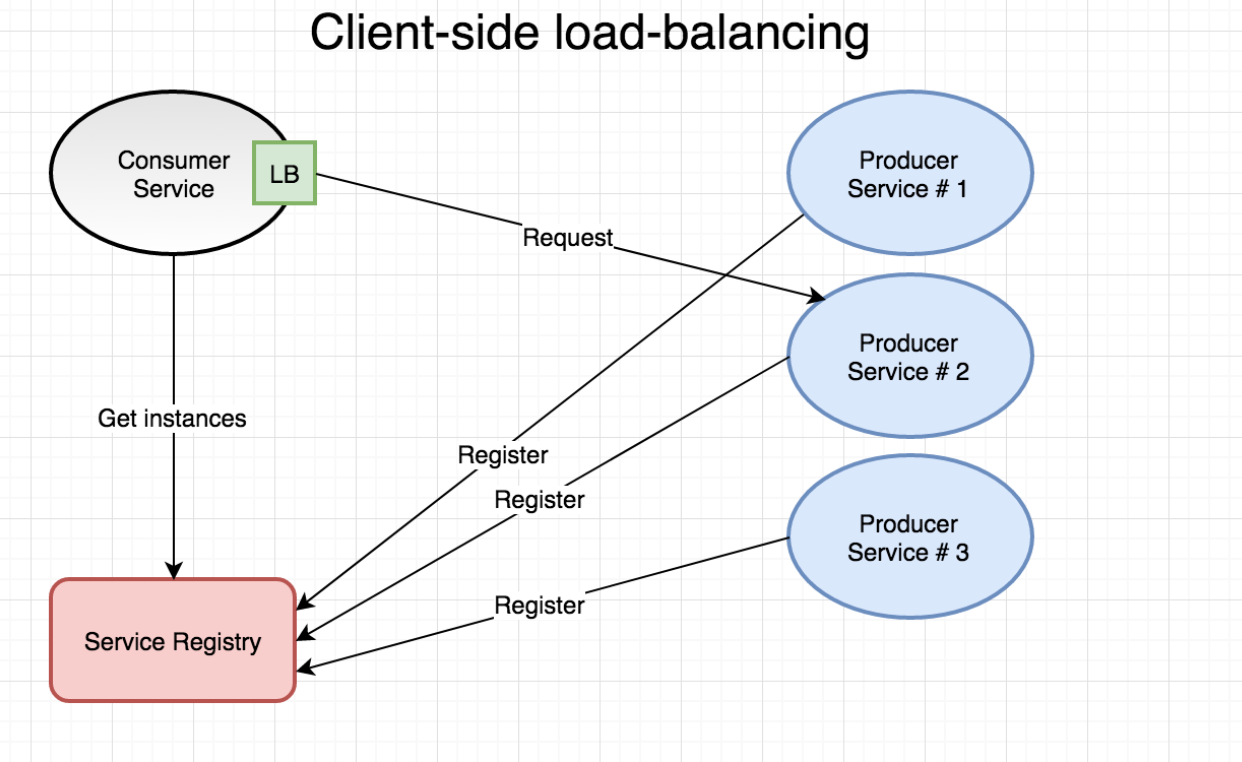
* **Vertical Scaling**. You scale by adding more power (CPU, RAM) to your existing machine.
* **Horizontal Scaling**. You scale by adding more machines into your pool of resources.
* **Caching.**Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.
* Application caching requires explicit integration in the application code itself. Usually, it will check if a value is in the cache; if not, retrieve the value from the database.
* Database caching tends to be “free”. When you flip your database on, you’re going to get some level of default configuration which will provide some degree of caching and performance. Those initial settings will be optimized for a generic use case, and by tweaking them to your system’s access patterns you can generally squeeze a great deal of performance improvement.
* In-memory caches are most potent in terms of raw performance. This is because they store their entire set of data in memory and accesses to RAM are orders of magnitude faster than those to disk. eg. Memcached or Redis.
* eg. Precalculating results (e.g. the number of visits from each referring domain for the previous day),
* eg. Pre-generating expensive indexes (e.g. suggested stories based on a user’s click history)
* eg. Storing copies of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL.

## Load Balancing

* Public servers of a scalable web service are hidden behind a load balancer. This load balancer evenly distributes load (requests from your users) onto your group/cluster of application servers.
* Types: Smart client (hard to get it perfect), Hardware load balancers ($$$ but reliable), Software load balancers (hybrid — works for most systems).



* For microservice architecture, client-side load balancing is better than server-side load balancing.



* **Database Replication**. Database replication is the frequent electronic copying data from a database in one computer or server to a database in another so that all users share the same level of information. The result is a distributed database in which users can access data relevant to their tasks without interfering with the work of others. The implementation of database replication for the purpose of eliminating data ambiguity or inconsistency among users is known as normalization.
* **Database Partitioning.**Partitioning of relational data usually refers to decomposing your tables either row-wise (horizontally) or column-wise (vertically).
* **Map-Reduce.**For sufficiently small systems you can often get away with ad-hoc queries on a SQL database, but that approach may not scale up trivially once the quantity of data stored or write-load requires sharding your database, and will usually require dedicated slaves for the purpose of performing these queries (at which point, maybe you’d rather use a system designed for analyzing large quantities of data, rather than fighting your database). Adding a map-reduce layer makes it possible to perform data and/or processing-intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports. eg. Hadoop, and maybe Hive or HBase.
* **Platform Layer (Services).**Separating the platform and web application allows you to scale the pieces independently. If you add a new API, you can add platform servers without adding unnecessary capacity for your web application tier. Adding a platform layer can be a way to reuse your infrastructure for multiple products or interfaces (a web application, an API, an iPhone app, etc) without writing too much redundant boilerplate code for dealing with caches, databases, etc.

## Key Topics for Designing a System

Before write system design or TRD, we must understand the following topics.

**Concurrency.**Do you understand threads, deadlock, and starvation? Do you know how to parallelize algorithms? Do you understand consistency and coherence?

**Networking.**Do you roughly understand IPC and TCP/IP? Do you know the difference between throughput and latency, and when each is the relevant factor?

**Abstraction.**You should understand the systems you’re building upon. Do you know roughly how an OS, file system, and database work? Do you know about the various levels of caching in a modern OS?

**Real-World Performance.**You should be familiar with the speed of everything your computer can do, including the relative performance of RAM, disk, SSD, and your network.

**Estimation.**Do estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few prototypes or micro-benchmarks to write.

**Availability and Reliability.**Are you thinking about how things can fail, especially in a distributed environment? Do you know how to design a system to cope with network failures? Do you understand durability?

**Cost.**When we build something and grow, the first problem is cost. You must calculate the cost of what should we spend on what are you use and compare it with any, and make sure you compare it on the same metric.

## Web App System Design Considerations

When creating system design for a web application we must consider the following things.

**Security** ( XSS, CORS, Clicjacking etc)

**Using CDN.**A content delivery network (CDN) is a system of distributed servers (network) that deliver webpages and other Web content to a user based on the geographic locations of the user, the origin of the webpage, and a content delivery server. This service is effective in speeding the delivery of content of websites with high traffic and websites that have a global reach. The closer the CDN server is to the user geographically, the faster the content will be delivered to the user. CDNs also provide protection from large surges in traffic.

**Full-Text Search.**Using Sphinx/Lucene/Solr, Elasticsearch — which achieves fast search responses because, instead of searching the text directly, it searches an index instead.

Offline support/Progressive enhancement

Service Workers

Web Workers

Server Side rendering

Asynchronous loading of assets (Lazy load items)

Minimizing network requests (Http2 + bundling/sprites etc)

Developer productivity/Tooling

Accessibility

Internationalization

Responsive design

Browser compatibility

## Database Transactional

The most important when creat an application is data consistency. Do the following rule.

**Monolith**

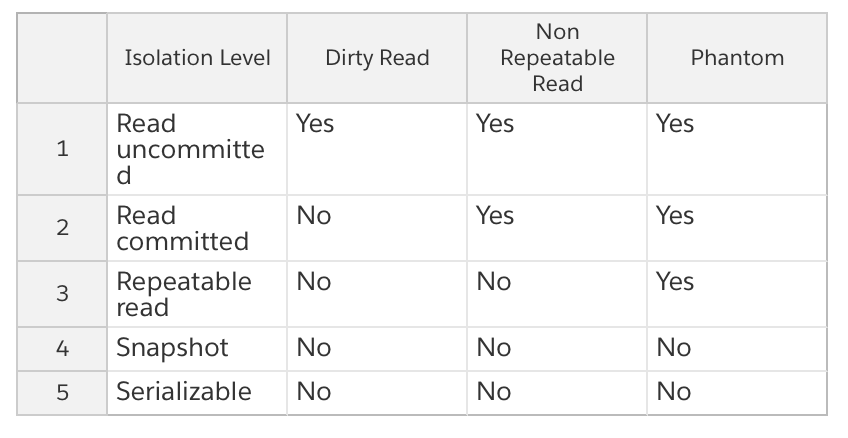
1. Local transaction in one Database
2. Global transaction in a transaction at more than one Database with more than one Kind Database ex using Oracle and MySql (we should use Transaction coordination like JTA in java and application server like Wildfly or WebLogic or [Enduro/X](https://en.wikipedia.org/wiki/Enduro/X) in Go)
3. ACID ( Atomicity, Consistency, Durability, Isolation)

* **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
* **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
* **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
* **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction

Transaction isolation levels control the following:

1. Whether locks are taken when data is read, and what type of locks are requested.
2. How long the read locks are held.
3. Whether a read operation referencing rows modified by another transaction.
4. Block until the exclusive lock on the row is freed.
5. Retrieve the committed version of the row that existed at the time the statement or transaction started.
6. Read the uncommitted data modification.
7. Choosing a transaction isolation level doesn’t affect the locks that are acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies and holds that lock until the transaction completes, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions.
8. A lower isolation level increases the ability of many users to access data at the same time but increases the number of concurrency effects, such as dirty reads or lost updates, that users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users might encounter, but requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level. The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, but it does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, can retrieve data that has been modified but not committed by other transactions. All concurrency side effects can happen in reading uncommitted, but there’s no read locking or versioning, so overhead is minimized.

The following table shows the concurrency side effects allowed by the different isolation levels.



**Microservice**

For microservice architecture to handle data consistency, we can use event base service communication, [Saga Pattern](https://medium.com/@dhfr123/saga-pattern-f46c989dc452), Event Sourcing, and [CQRS](https://medium.com/@dhfr123/cqrs-81b2de39dfd5) pattern.

**Lock Strategy**

1. Lock Pessimistic ( Serialisation method)
2. It’s when you maintain an exclusive lock while editing a record. For the end-user, it looks like they can’t start editing a product until the person who’s editing it currently releases the lock.
3. Most relational databases support pessimistic locking out of the box but you can also implement it yourself in the application code.
4. When it comes to choosing between optimistic and pessimistic locking, I would say go with the former by default. Pessimistic locking is useful when the cost of merging simultaneous changes is high. Which is not the case in the vast majority of domains. Pessimistic locking is too cumbersome to implement and too annoying for the users, so it doesn’t worth it in most cases
5. Lock Optimistic: [Optimistic locking](https://en.wikipedia.org/wiki/Optimistic_concurrency_control) is a way to manage concurrency in multi-user scenarios. You generally want to avoid situations when one user overrides changes made by another user without even looking at them. Locking — optimistic locking in particular — is a way to do that. User Like version or by created time. The optimistic lock must combine with automatic retry.
6. Optimistic locking is when you check if the record was updated by someone else before you commit the transaction.
7. Pessimistic locking is when you take an exclusive lock so that no one else can start modifying the record.
8. You cannot combine optimistic locking with the automatic retry. Optimistic locking assumes a manual intervention in order to decide whether to proceed with the update.
9. If you don’t care about the previous updates applied to the record, don’t implement any locking strategy.

**Last we must keep in mind that API not only runs from server to server but also from people to people.**