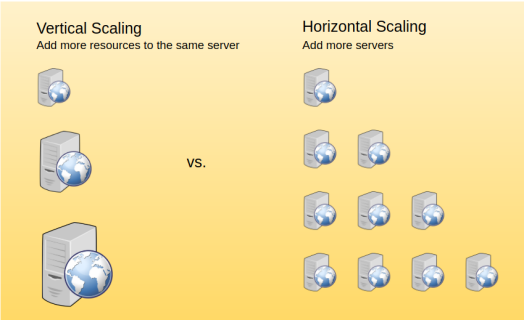
Scalability

- Definition
  - the capability of a system, process, or network to grow and manage increased demand.
  - a system is scalable if it is designed so that it can handle additional load and will still operate efficiently
  - Any distributed system that can continuously evolve in order to support the growing amount of work is considered to be scalable.
  - A scalable architecture avoids this situation and attempts to balance the load on all the participating nodes evenly.
- Reasons
  - an increased data volume
  - an increased amount of work; e.g., the number of transactions
  - a scalable system would like to achieve this scaling without performance loss.
- Notes
  - Generally, the performance of a system, although designed (or claimed) to be scalable, declines with the system size due to the management or environmental cost
  - For instance, network speed may become slower because machines tend to be far apart from one another
  - More generally, some tasks may not be distributed, either because of their inherent atomic nature or because of some flaw in the system design.
- Types
  - Horizontal:
    - scale by adding more servers into your pool of resources
    - often easier to scale dynamically by adding more machines into the existing pool
    - Cassandra and MongoDB provide an easy way to scale horizontally by adding more machines to meet growing needs.



    - Vertical:
        - scale by adding more power (CPU, RAM, Storage, etc.) to an existing server.
        - usually limited to the capacity of a single server and scaling beyond that capacity often involves downtime and comes with an upper limit.
        - MySQL allows for an easy way to scale vertically by switching from smaller to bigger machines.
- Keywords

    distributed system, scalable architecture, balance the load, data volume, amount of work, performance loss, declines, a flaw in, pool of resources, downtime, an upper limit

Reliability

- Definition
    - the probability a system will fail in a given period
    - a system is reliable if it can perform the function as expected, it can tolerate user mistakes, is good enough for the required use case, and it also prevents unauthorized access or abuse.
    - a distributed system is considered reliable if it keeps delivering its services even when one or several of its software or hardware components fail.
- Reasons
    - any failing machine can always be replaced by another healthy one, ensuring the completion of the requested task.
- Example
    - If a user has added an item to their shopping cart, the system is expected not to lose it
- Notes
    - A reliable distributed system achieves this through the redundancy of both the software components and data.
    - If the server carrying the user's shopping cart fails, another server that has the exact replica of the shopping cart should replace it.

    - Obviously, redundancy has a cost and a reliable system has to pay that to achieve such resilience for services by eliminating every single point of failure
- Keywords

  delivering its services, components, healthy service, redundancy, replica, resilience, eliminating, single point of failure

Availability

- Definition
    - the time a system remains operational to perform its required function in a specific period.
    - a system is available if it is able to perform its functionality (uptime/total time).
    - Availability takes into account maintainability, repair time, spare availability, and other logistics considerations.
- Availability Vs. Reliability
    - If a system is reliable, it is available
    - if it is available, it is not necessarily reliable
    - Example:
        - an online retail store that has 99.99% availability for the first two years after its launch. However, the system was launched without any information security testing. The customers are happy with the system, but they don't realize that it isn't very reliable as it is vulnerable to likely risks. In the third year, the system experiences a series of information security incidents that suddenly result in extremely low availability for extended periods of time.
        - This results in reputational and financial damage to the customers.

Efficiency

- Definition
  - a system is efficient if it is able to perform its functionality quickly.
- Standard Measures
  - Response time (or Latency)
    - denotes the delay to obtain the first item
  - Throughput (or bandwidth)
    - denotes the number of items delivered in a given time unit (e.g., a second).
- Notes
  - The two measures correspond to the following unit costs:
    - The number of messages globally sent by the nodes of the system regardless of the message size.
    - The size of messages representing the volume of data exchanges.
  - The complexity of operations supported by distributed data structures (e.g., searching for a specific key in a distributed index) can be characterized as a function of one of these cost units.
  - The analysis of a distributed structure in terms of 'the number of messages' is over-simplistic. It ignores the impact of many aspects, including the network topology, the network load, and its variation, the possible heterogeneity of the software and hardware components involved in data processing and routing, etc.
  - However, it is quite difficult to develop a precise cost model that would accurately take into account all these performance factors; therefore, we have to live with rough but robust estimates of the system behavior.

- KeyWords

  Latency, bandwidth, nodes of the system, volume of data exchanges

Serviceability or Manageability

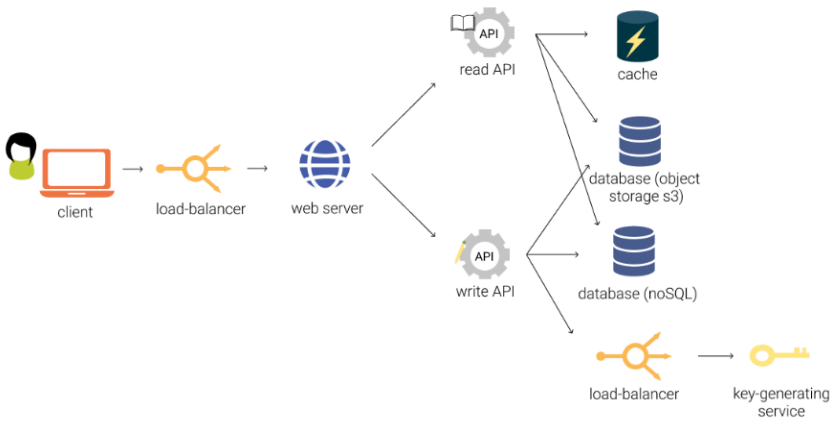
- Definition
  - the simplicity and speed with which a system can be repaired or maintained
  - a system is maintainable if it easy to make operate smoothly, simple for new engineers to understand, and easy to modify for unanticipated use cases.
- Things to Consider
  - the ease of diagnosing and understanding problems when they occur
  - the ease of making updates or modifications
  - how simple the system is to operate (i.e., does it routinely operate without failure or exceptions?).
- Notes
  - if the time to fix a failed system increases, then availability will decrease
  - early detection of faults can decrease or avoid system downtime.
    - For example, some enterprise systems can automatically call a service center (without human intervention) when the system experiences a system fault.

- KeyWords

  Maintenance, diagnosing, routinely operates, detection of faults, intervention

Example

Let us consider a simple write system.
The web service is designed to allow users to write and save information, and get an associated URL with the associated text that they have written (like PasteBin).



- Scalable
  - The use of load-balancers and caches and the choice of databases (object storage S3 and noSQL) will help facilitate additional load on the system from additional users.
- Reliable
  - The system will be able to reliably handle the functionality (i.e. where some number of clients are able to write text and get an associated URL with the text they have written).
  - And by how the APIs are defined, it should be able to handle user mistakes.
- Efficient
  - The introduction of the caching allows for greater efficiency for read requests by the user.
  - The load-balancing also distributes the load more evenly across servers to make the system more efficient.
- Available
  - The load-balancers are distributing the load across multiple servers, meaning there is less likelihood of failure that will cause the system to crash.

    - Additional databases can be used for redundancy in the case of error or failures.
- Maintainability.
    - The APIs are designed in a way that allows for modularity (i.e. separation of read and write calls).
    - This allows for easy maintainability of the system.

Final Notes

- There are so many components, algorithms, and architectures that support optimality in the above dimensions of distributed system design.
- You will quickly find that each element in the system design toolbox may be great with respect to one dimension but at the cost of being low-performing in another
    - i.e. a component might be extremely efficient but not as reliable, or a certain design choice might be really effective at supporting one functionality like providing read access but not as efficient with write access).
- In other words, there are tradeoffs that you must consider as you make informed system design choices. The best way to understand these tradeoffs is to understand how each of these components, algorithms and architectural designs work
-