**Vertical scaling –**

When you do this, though, you’re throttling yourself while the machine is taken offline for the upgrade. And, what happens down the road when your traffic is on the rise again and you have to repeat the upgrades? There are only a finite number of times you can go about solving your problem by “scaling up” in this manner.

Vertical scaling is means adding more resources to the server. This can be adding more RAM, disk space/speed, processing power etc..

Usually applications scale well with vertical scaling - depending on the application. If the scaling is done in an area where the application is using up all system resources it’ll be running smoother/faster after scaling vertically.

Vertical scaling is typically costly . It does not make the system fault tolerant , i.e if you are scaling application running with single server , if that server goes down , your system will go down . Also the amount of threads remains the same in vertical scaling . Vertical scaling may require your system to go down for a moment when process takes place . Increasing resources on a server requires a restart and put your system down .

**Horizontal scaling –**

**is almost always more desirable than vertical scaling because you don’t get caught in a resource deficit.** Instead of taking your server offline while you’re scaling up to a better one, horizontal scalinglets you keep your existing pool of computing resources online while adding more to what you already have. **When your app is scaled horizontally, you have the benefit of elasticity.**

You can do exactly this when your infrastructure is [hosted in a Managed Cloud environment](https://www.g2techgroup.com/managed-aws-cloud/).

Other benefits of scaling out in a cloud environment include:

* Instant and continuous availability
* No limit to hardware capacity
* Cost can be tied to use
* You’re not stuck always paying for peak demand
* Built-in redundancy
* Easy to size and resize properly to your needs

**How To Achieve Effective Horizontal Scaling**

There are important best practices to keep in mind to make your service offering super compatible with horizontal scaling.

The first is to **make your application stateless on the server side as much as possible**. Any time your application has to rely on server-side tracking of what it’s doing at a given moment, that user session is tied inextricably to that particular server. If, on the other hand, all session-related specifics are stored browser-side, that session can be passed seamlessly across literally hundreds of servers. The ability to hand a single session (or thousands or millions of single sessions) across servers interchangeably is the very epitome of horizontal scaling.

The second goal to keep square in your sights is to develop your app with a service-oriented architecture. **The more your app is comprised of self-contained but interacting logical blocks, the more you’ll be able to scale each of those blocks independently as your use load demands**. Be sure to develop your app with independent web, application, caching and database tiers. This is critical for realizing cost savings – because without this microservice architecture, you’re going to have to scale up each component of your app to the demand levels of thservices tier getting hit the hardest.

Horizontal Scaling - also referred to as "scale-out" is basically the addition of more machines or setting up a cluster or a distributed environment for your software system. This usually requires a load-balancer program which is a middle-ware component in the standard 3 tier client-server architectural model.

Load-Balancer is responsible to distribute user requests (load) among the various back-end systems/machines/nodes in the cluster. Each of these back-end machines run a copy of your software and hence capable of servicing requests. This is just one of the various functions that load balancer may be performing. Another very common responsibility is "health-check" where the load balancer uses the "ping-echo" protocol or exchanges heartbeat messages with all the servers to ensure they are up and running fine.

Load-Balancer distributes load by maintaining state of each machine -- how many requests are being served by each machine, which machine is idle, which machine is over-loaded with queued requests etc. So the load balancing algorithm considers such things before redirecting the request to an appropriate server machine. It also takes into account the network overhead and might choose the server in the nearest data-center provided it is available to service the requests.

The request-response can also be done in 2 different ways:

1. Load Balancer always acts as an intermediary program for every response - In this case, once the request has been handed over to the server by the load balancer, any response from the server to the user will go through the load balancer. So the server machines that are actually servicing the request will never directly interface with the user machine running the client application. The machine hosting the load balancer program will be handling all the requests/responses to and from the user.
2. Load Balancer does not act as an intermediary for the responses coming from the server machine - In this case, once the server has received the request from load-balancer, it bypasses the load balancer and communicates it responses directly to the client.

Setting up a cluster and load-balancer as a front-end interface to the client application does not really complete our scale-out architecture and design. There are still lots of critical questions to be answered and a number of key design decisions to be made which will affect the overall properties of our system.

We first need to identify our business goals and areas where we would like to add value. These goals will give rise to various requirements. We should then ask ourselves various questions with respect to different systemic properties.

1. Will such a design address our performance requirements ?
2. What performance characteristics do we care about ? Is it the overall system throughput where we are interested in serving maximum number of requests in any given time ? Or is it the system's response time where we design for sending back the response to client in as less time as possible ? Both these and many other types of performance characteristics are related to each other.
3. Will such a design address our availability requirements ? Is the system fault-tolerant ? If so, what is the degree of it ?
4. Is such a design reliable ? Does it impact the correctness ? We should not forget that 100% correctness is an implicit goal of any system.
5. Are we really meeting our scalability goals ? Might be achieving the short-term or immediate ones, but what is going to happen in the long run ?

All these kinds of requirements should have quantifiable measures associated with them.

We should then make important design decisions by questioning ourselves, developing prototypes, and refining the design.

1. Firstly, is using load-balancer the only approach to distribute load and horizontally scale the system ?
2. Do the various back-end servers or nodes communicate with each other ? If yes, then how does the system address the situation where one or more nodes go down -- permanently or temporarily ? If yes, then how does the system address the situation where the network connecting the nodes is down, but all the nodes are up and running ? Most importantly, do we have to differentiate between these two situations ? How ?
3. Whether or not the back-end nodes communicate with each other, does our system need to maintain consistent data across all the nodes ? What level of consistency do we care about ? Is it that At any point in time, data across all the nodes should be consistent. Or later some point in time, data across all the nodes will be consistent. If so, then what is this "later" ? When and how do all the nodes converge to a consistent state ? How will we achieve "total-order" of operations across all the nodes ? Do we have a global clock ? If we are relying on every node's local clock, then how do we synchronize the clocks of all machines. They can easily seem to regress or a machine with an out of order clock might join the cluster. As a consequence, we may ignore the latest data, and consider old/stale data as the latest one.
4. What cluster setup do we have to design for ? Is it a "replica" cluster, where data on each node is replicated to some or every other node. In case of former, what is the replication factor, and how do we decide it ? Or is it a sharded cluster where cluster is divided into various shards or units. A shard is a designated group of nodes. Each shard takes care of a particular partition of data. Data across shards is not replicated, but each shard can adopt replication strategy within itself. Whatever distributed system we design, it should ideally be able to answer the above and many other similar questions.