Vertical Scaling

Vertical Scaling (also called "scaling up") is the process of adding more power to an existing machine—increasing its CPU, RAM, storage, or network capacity to handle more load.

What Can Be Scaled Vertically

CPU (Processing Power)

- Upgrade from 4 cores to 8, 16, or 32 cores
- Increase clock speed
- Better processors with improved performance per core

RAM (Memory)

- Increase from 8GB to 16GB, 32GB, 64GB, or more
- Allows handling more concurrent requests and caching more data in memory

Storage

- Add more disk space (1TB to 2TB, etc.)
- Upgrade to faster storage (HDD → SSD → NVMe)
- Improves read/write speeds and capacity

Network

- Upgrade network interface cards for higher bandwidth
- Move from 1 Gbps to 10 Gbps or higher

How it works in practice

- App stays **exactly the same**
- No load balancer needed
- No code changes (usually)
- Just reboot with better hardware
- For cloud environments – you can simply upgrade instances
- For physical servers – it requires physically installing new hardware—more RAM sticks, replacing CPUs, adding storage drives. Often requires downtime

Pros ✅

- Simple to implement; No code changes
- Fast performance (no network hops); Reduced latency
- Great for monoliths and databases
- Easy debugging
- Maintains data consistency

Cons ❌

- Physical limit (you can't scale forever)
- Single point of failure
- Can require downtime
- Often expensive at higher tiers

Horizontal scaling

Horizontal Scaling (also called "scaling out") is the process of adding more machines to your infrastructure to handle increased load, rather than making existing machines more powerful.

Basic flow:

1. User makes a request
2. **Load balancer** receives the request
3. Load balancer routes request to one of many available servers
4. That server processes the request and responds
5. If traffic increases, add more servers behind the load balancer

Pros ✅

- Near-infinite scalability
- High availability (one node dies, others survive)
- No physical limit like vertical scaling
- Perfect for cloud-native & microservices
- Geographic distribution possible
- Rolling updates with zero downtime
- Fault tolerance

Cons ❌

- More complexity
- Needs stateless services
- Requires load balancing
- Data consistency can be tricky
- Network latency
- Debugging difficulties

Load Balancing Strategies

Round Robin Distributes requests sequentially. Server 1 → Server 2 → Server 3 → Server 1... Simple but doesn't account for server load differences.

Least Connections Sends requests to the server with fewest active connections. Better for long-lived connections like websockets.

Least Response Time Routes to the server responding fastest. Good for ensuring consistent user experience.

IP Hash Routes based on client IP address, ensuring the same user always hits the same server. Useful when you need session affinity (though generally you want to avoid this).

Weighted Distribution Assigns different capacities to servers. Powerful servers get more traffic than weaker ones. Useful in heterogeneous environments.

Geographic/Location-Based Routes users to the nearest server geographically. Reduces latency for global applications.

Real-World Scenario: Your web application runs on one server handling 1,000 requests/second. Traffic grows to 5,000 requests/second.

Horizontal scaling solution:

1. Deploy 4 additional identical servers (5 total)
2. Add a load balancer in front
3. Each server now handles ~1,000 requests/second
4. Move session data to Redis (shared cache)
5. All servers connect to the same database

Vertical Scaling	**Horizontal Scaling**
Scale **up**	Scale **out**
Bigger server	More servers
Simple	Complex
Limited	Virtually unlimited
Risky failure	Fault tolerant
Good for: databases, monolithic apps, getting started	Good for: web servers, stateless applications, microservices