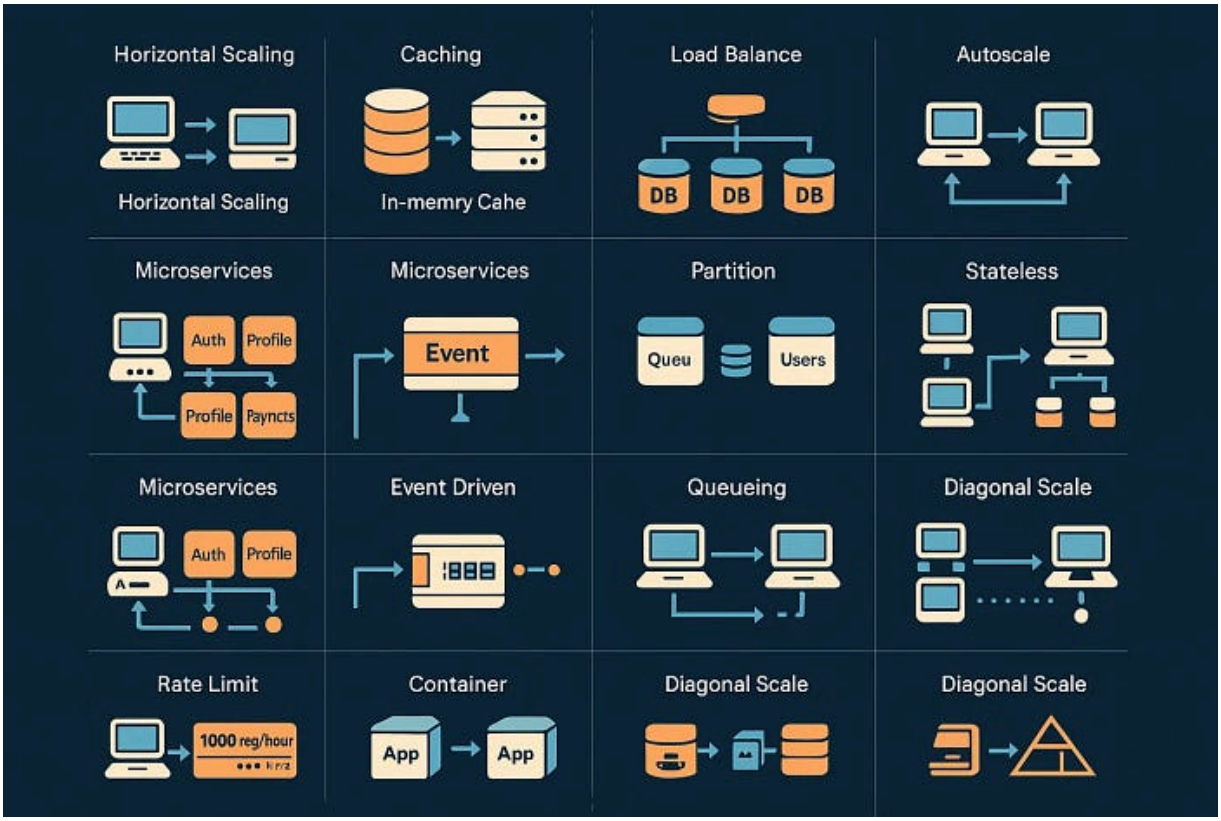
< Go to the original



I Learned System Design the Hard Way. Here Are the 40 Scaling Patterns Nobody Explained

A year ago, our startup's API crashed during a product launch. We had 10,000 concurrent users, and our single server couldn't handle it…

The Latency Gambler

Follow

a11y-light · December 16, 2025 (Updated: December 16, 2025) · Free: No

A year ago, our startup's API crashed during a product launch. We had 10,000 concurrent users, and our single server couldn't handle

Here are all 40 scaling patterns I wish someone had explained to me back then.

The Foundation: Scaling Basics

1. Horizontal Scaling Add more servers instead of upgrading one. Two servers handling 5,000 requests each beats one struggling with 10,000.

2. Vertical Scaling Upgrade CPU, RAM, disk. Works until you hit hardware limits.

Copy

```
// Round-robin load balancing
public class LoadBalancer {
    private List<Server> servers;
    private AtomicInteger counter = new AtomicInteger(0);

    public Server getNext() {
        return servers.get(counter.getAndIncrement() % servers.siz
    }
}
```

3. Caching Store frequent data in memory. Redis reduces database load by 80%.

4. Load Balance Distribute requests across servers. Nginx, HAProxy, or cloud load balancers keep traffic flowing evenly.

5. Sharding Split database horizontally. Users 1–1000 on Shard A, 1001–2000 on Shard B.

7. Partition Divide data by segments. Orders in one database, users in another.

8. Autoscale Dynamically adjust resources based on traffic. AWS Auto Scaling groups spin up instances during spikes.

Architecture Patterns

9. Microservices Break monoliths into independent services. Authentication scales separately from payments.

10. Event Driven Process tasks asynchronously using message queues.

Copy

```
// Event-driven processing
public class EventProcessor {
    private BlockingQueue<Event> queue;

    public void publish(Event event) {
        queue.offer(event);
    }

    public void startProcessing() {
        executor.submit(() -> {
            while (true) {
                Event event = queue.take();
                handleEvent(event);
            }
        });
    }
}
```

12. Stateless Store no session data locally. Every request contains needed information, enabling easy horizontal scaling.

13. Indexing Speed up queries dramatically. Proper indexes turn 10-second queries into 10-millisecond ones.

14. Timeouts Set maximum wait times. Better to fail fast than hang indefinitely.

15. Retries Handle transient failures with exponential backoff. Wait 1 second, then 2, then 4.

16. Rate Limit Control request traffic per user or IP. Protect APIs from abuse and ensure fair usage.

Copy

```
[Client] → [Rate Limiter: 1000 req/hour] → [API Gateway] → [Servic
```

Reliability Patterns

17. Circuit Breaker Stop calling failing services temporarily. Let them recover instead of cascading failures.

Copy

```
public class CircuitBreaker {
    private State state = State.CLOSED;
    private int failures = 0;

    public Response call(Supplier<Response> action) {
        if (state == State.OPEN) {
            throw new CircuitOpenException();
```

```
            onSuccess();
            return res;
        } catch (Exception e) {
            onFailure();
            throw e;
        }
    }
}
```

18. Backpressure Slow down producers when consumers can't keep up. Prevents memory overflow.

19. GeoDNS Route users to nearest data center. DNS returns location-appropriate IPs automatically.

20. Multi Region Deploy across continents. US users hit US servers, reducing latency from 200ms to 20ms.

21. Container Package apps with Docker. Run identically everywhere.

22. Orchestration Manage containers with Kubernetes. Automatic scheduling, scaling, and recovery.

23. Service Mesh Handle service communication with Istio or Linkerd. Manages retries, routing, and observability.

24. Diagonal Scale Combine horizontal and vertical scaling for optimal cost-performance balance.

Performance Optimization

26. Monitoring Track metrics with Prometheus and Grafana. Know problems before users complain.

27. Tracing Follow requests through systems with Jaeger. Find bottlenecks in distributed architectures.

28. Failover Automatically switch to backup when primary fails. Database replicas take over instantly.

29. High Availability Deploy across multiple availability zones. One zone fails, others continue serving.

30. Graceful Degradation Maintain core features when parts fail. Show cached data instead of errors.

31. Consistent Hashing Distribute data evenly. When nodes change, only minimal keys need remapping.

```
Copy
Hash(key) → Node Position on Ring → Clockwise to Next Node
```

32. CAP Tradeoff Choose consistency or availability during partitions. Most systems pick availability with eventual consistency.

33. Modularity Separate cohesive components. Each module changes independently without affecting others.

34. Bulkhead Isolate failures with separate thread pools. Critical operations stay unaffected by non-critical failures.

36. Lazy Load Defer resource allocation until needed. Create database connections only when required.

37. Capacity Planning Forecast future needs. Analyze growth trends and provision infrastructure ahead.

38. Hot Standby Keep backup systems running and synchronized. Zero-downtime failover capability.

39. Read Replica Scale read traffic with multiple database replicas. One primary for writes, five replicas for reads.

40. Write Batching Group multiple operations together.

Copy

```
public class BatchWriter {
    private List<Record> batch = new ArrayList<>();
    private final int BATCH_SIZE = 100;

    public void write(Record record) {
        batch.add(record);
        if (batch.size() >= BATCH_SIZE) {
            database.batchInsert(batch);
            batch.clear();
        }
    }
}
```

The Real Lesson

That night 1 year ago taught me something crucial: knowing these 40 patterns isn't enough. You need to understand when to apply them.

Monitor everything, measure twice, scale once.

System design is about trade-offs. Caching improves performance but adds complexity. Microservices enable scaling but increase operational overhead. Choose patterns that solve your actual problems, not theoretical ones.

Now you know all 40 patterns. The hard part? Picking the right ones for your specific challenge.

What scaling challenges have you faced? Which patterns saved your system? Share your stories below.

#system-design-interview #system-design-concepts #software-engineering #software-development #programming