WORCS

Ramkumar Manavalan

# SYSTEM DESIGN

Cheat Sheet

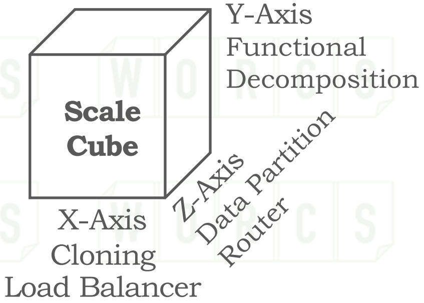
## Scalable Architecture

**Scalability** - Not a metric by itself

- Ability of a system to ensure that all other –ilities are either enhanced or maintained, and, not adversely impacted, when load / volume increases!
- System & Development Scalability

**Software Architecture Basics**

- It's all about decomposition
- 4+1 Architecture View: Logical, Implementation, Process & Deployment Views.
- Styles: Layered (Prez, Biz, Data) – Hexagonal (Dependency Inversion)



**Monolith**

- Can be perceived at any level: Function, Class, Module, App, Suite of Apps
- Scaling: X & Z axis of Scale Cube
- Pros: Low Latency, Low Maintenance (Build, Deployment, Config), Simple to Scale
- Cons: High Load Time, Hard to maintain/ replatform or to scale/treat differentially

**Path Towards MSA** Decompose into independent, differentially treatable and smaller units!

**Scaling** First Decompose Functionally (Y-axis), then scale each service by x or z axis

**Modularity Basics**: Coupling & Cohesion, SRP, Composition vs Aggregation; Avoid Shotgun Surgery & Divergent Change

**Y-Axis Scaling** – Questions to ask: Actors, Usecases, Entities, Volume, Resource Usage, Security, Criticality, Team Structure

**MSA Challenges** High Latency – Multiple Points of Failures – Data Consistency Issues – Operational Challenges – Decomposition is a challenge that requires great skills

**Popular MSA Patterns** Synch RPC (REST, gRPC), Asynch (Messaging Q), Circuit Breaker, Service Discovery Patterns, API Gateway, Saga (compensating transactions), Deployment Patterns (Mesh, VM, Container, Serverless)

| 4+1 Arch View | Monolith | Microservices |
|---|---|---|
| Implementation View | Single Component | Set of multiple components |
| Logical View | No specification | No specification, (mostly hexagonal) |

| | SOA | MSA |
|---|---|---|
| Scope | Enterprise | Local |
| Granularity | Coarse-grained | Fine-grained |
| Communi-cation | Mostly Synchronous | Mostly Asynchronous |

## Scalable Database

| | Replication | Sharding |
|---|---|---|
| Why? | Availability - Scale out – Low Latency | Scale Out – Fault Isolation – Performance |
| Challenges | Synchronization, Failure Handling | Fair Distribution |

**Distributed Consistency** – Replication Methods – Network Faults are inevitable - Leads to CAP Theorem – When distributed, P is not a choice, so it is between CP & AP - ACID → CP, BASE → AP

Safety Guarantees given by a transaction are described by ACID

**Atomicity**: Either All or None – Commit or Rollback/Abort – doesn't deal with Concurrency

**Isolation**: Concurrency - Serializability – 2 Phase Locking – Optimistic Concurrency Control

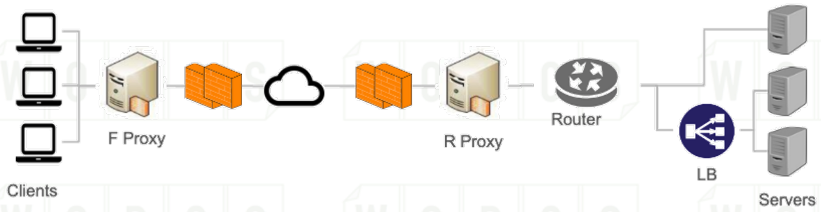
**Consistency**: Validation Rules

**Durability**: Guaranteed replication and write to Storage from Volatile Memory

**BASE** Basically Available – Soft State – Eventually Consistent

| | | |
|---|---|---|
| **RDBMS** | ACID | CA (CP is a challenge) |
| **NoSQL** | ACID or BASE | CP or AP |

This document is for internal use only and not to be distributed



**Firewall**: Intercept & Filter
**Forward Proxy**: Client Anonymity
**Reverse Proxy**: Server Anonymity
**App Router**: Decide & Redirect
**Load Balancer**: Load Distribution

**LB Algos**: Round Robin, Weighted Round Robin, Least Connections, Least Response Time

**LB Types**: Layer4 (TCP) & Layer7 (HTTP – App Awareness)

**Reverse Proxy**

*Security*: Authentication, DDoS Prevention

*Performance*: Caching, SSL Termination, Compression

### Session Management

*Session Identifier Options:* IP Addr, Browser Login, URL Rewriting, Hidden Form Fields, Cookies

*Distributed env*: Sticky Sessions – DB to store – Distributed Data Store (like Redis) – No Server Side Sessions (JWT)

**Performance** responsiveness of a system to execute any action within a given time interval

**Latency** Time spent waiting for other action(s) to be completed

**Availability** Percentage of time the service remains operational under normal circumstances in order to serve its intended purpose. A = (Total time – Sum of downtime)/Total time

### Performance Considerations

- Offload View logic to Client
- No server side sessions
- Separate core from less critical use cases
- Choose the right communication protocol
- Leverage CDN & Caching
- Separate Real-time, near real-time and batch work
- Prefer Async Messaging Queue, wherever applicable – increases reliability too.

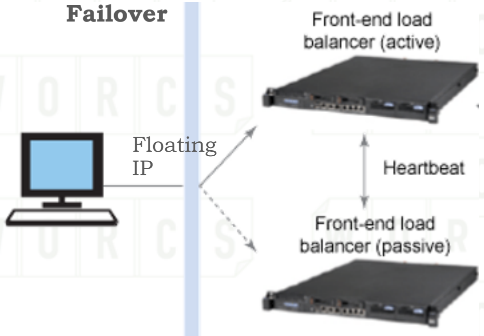
**Availability Patterns:** Failover & Replication – Solution to Single Point of Failures

Replication: Master-Slave, Master-Master (either CP or AP)

### Caching

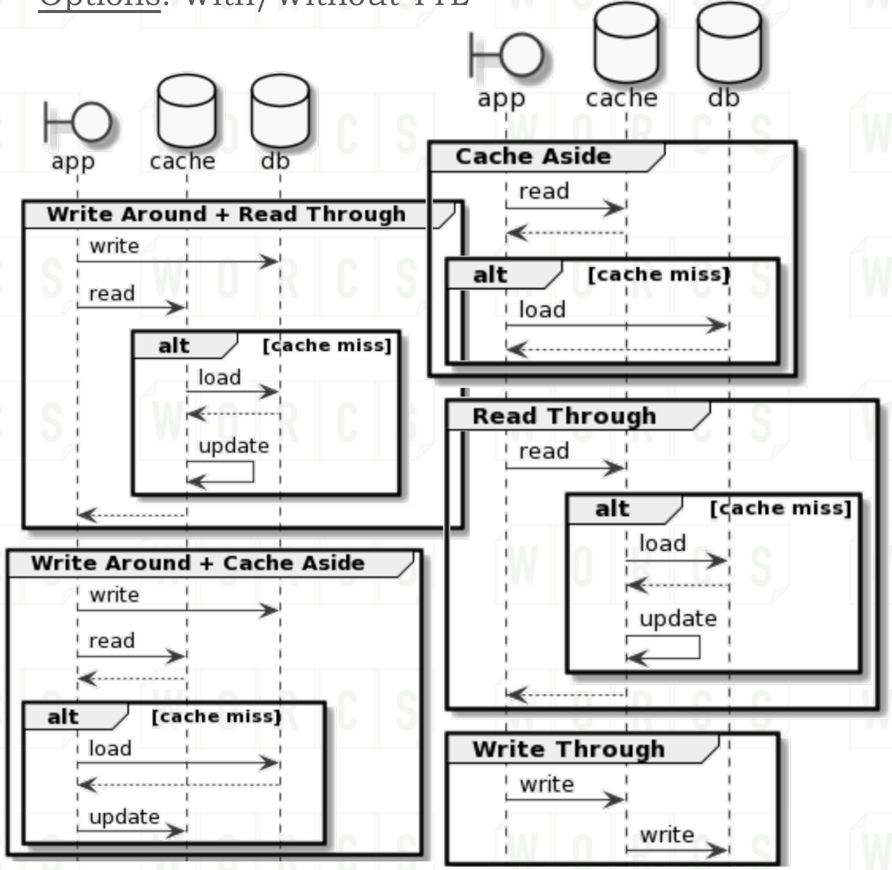
- Client Caching (End User)
- Edge Caching (closer to end users)
- Web Server Caching (Reverse Proxies, etc.)
- Application Caching
- Database Caching (DB Buffers, Indexing, etc.)

**Failover**


Front-end load balancer (active)
Floating IP
Heartbeat
Front-end load balancer (passive)

**DB Caching Strategies**: Cache Aside, Read Through, Write Through, Write Around, Write Back/Behind, Refresh Ahead

Options: With/Without TTL



**Capacity Planning:** Determining the production capacity required to meet changing demands – Periodic Task – Key Resources: CPU, Network, Memory, Disk – Measurement is better than intuition

### Capacity Planning Steps

1. Determine Requirements: Prod Owner / Biz Analyst - User Types, Common/Frequent Use Cases, Short/Long Term Growth – Average/Peak Traffic, Expected Response Time

2. Run Baseline Tests: Test env ~ prod – mock 3rd party calls – stress test common cases for long duration

3. Extrapolate/Estimate: Use the metrics & TPS to extrapolate

This document is for internal use only and not to be distributed