Catalog

# Ultimate System Design Cheatsheet

## Foreword: Thinking Like an Architect

System design is fundamentally an exercise in navigating constraints and making reasoned trade-offs. Unlike algorithmic problems that often have a single optimal solution, system design challenges are open-ended, mirroring the complexities of real-world engineering. There is no universally "correct" answer; instead, there are solutions that are more or less appropriate for a given set of requirements, constraints, and business goals. The mark of an expert architect is not knowing a secret, perfect design, but rather possessing the framework to systematically deconstruct a problem, articulate the available options, and justify the chosen path.

This book is structured to build that framework. It moves from foundational, non-negotiable principles to the practical application of architectural patterns and components. The core philosophy is to first understand *why* certain qualities like scalability and availability matter, then to learn the *what*—the building blocks like load balancers and caches—and finally, to master the *how* by applying this knowledge to complex, real-world scenarios. A recurring theme is the critical importance of asking the right questions at the outset of any design process. Before a single line of code is written or a single server is provisioned, the architect must clarify the functional requirements, anticipate the scale, and define the non-functional goals that will shape every subsequent decision. This cheatsheet is designed to be both a comprehensive guide for deep learning and a quick reference for mastering that decision-making process.

---

## Part 1: The Foundations of Modern System Design

This part establishes the fundamental, non-negotiable principles that underpin any robust, large-scale system. It translates abstract concepts into measurable goals and provides the vocabulary for discussing system quality.

### Chapter 1: The Four Pillars of System Architecture

This chapter introduces the core non-functional requirements that dictate the success or failure of a system. These are the primary metrics by which a system's quality is judged.

## 1.1. Scalability: The Art of Growth

Scalability is a system's ability to handle a growing amount of work by adding resources to the system. It is a crucial measure of a system's capacity to grow gracefully without a degradation in performance.

#### Defining Scalability vs. Performance

A common point of confusion is the distinction between performance and scalability. The two are related but distinct concepts. Performance measures how fast a system can complete a task for a single user. Scalability, on the other hand, measures the system's ability to maintain performance under an increasing load (more users, more data, or more transactions).

A useful analogy is a local delivery service. The performance of the service could be measured by how fast one truck can complete a single delivery. The scalability of the service is its ability to handle a city-wide surge in orders by adding more trucks to its fleet. A system can be highly performant for a single user but fail completely under heavy load if it is not scalable.

#### Vertical Scaling (Scaling Up)

Vertical scaling involves increasing the resources of a single server—adding a more powerful CPU, more RAM, or faster storage. This is the "bigger truck" option.

- **Pros:** This approach is often simpler to implement and manage because it doesn't fundamentally change the application's architecture. All code continues to run on a single machine, avoiding the complexities of distributed computing.
- **Cons:** Vertical scaling has hard limits. There is a physical maximum to how much a single machine can be upgraded, and the cost of high-end hardware can become astronomical. Furthermore, it creates a single point of failure; if that one powerful server goes down, the entire system is offline.

#### Horizontal Scaling (Scaling Out)

Horizontal scaling, the foundation of modern cloud architecture, involves adding more servers to a pool of resources and distributing the load among them. This is the "more trucks" strategy.

- **Pros:** This approach offers nearly limitless potential for growth and enhances resilience. If one server fails, the others can continue to operate and pick up the slack, improving the system's overall availability.
- **Cons:** It introduces the complexities inherent in distributed systems, such as inter-service communication, data consistency across nodes, and service discovery.

| **Feature** | **Vertical Scaling (Scaling Up)** | **Horizontal Scaling (Scaling Out)** |
|---|---|---|
| **Scalability Limit** | Capped by maximum machine size | Virtually limitless |
| **Cost** | Exponentially expensive for high-end hardware | Linear cost increase with commodity hardware |
| **Fault Tolerance** | Single point of failure | High; resilient to individual machine failures |
| **Architectural Complexity** | Low; application logic remains centralized | High; requires handling distributed state, communication, and consistency |
| **Maintenance** | Simpler; one machine to manage | More complex; requires orchestration and management of many machines |

### 1.2. Availability: Designing for Resilience

Availability ensures that a system is operational and accessible to users when they need it. It is typically measured as a percentage of uptime over a given period. High availability means the system experiences minimal downtime.

#### Understanding Availability in Numbers

Availability is often expressed in "nines." For example, 99.9% availability ("three nines") translates to approximately 8.77 hours of downtime per year. In contrast, 99.999% availability ("five nines") allows for only about 5.26 minutes of downtime per year. Understanding these numbers is critical for setting realistic and business-appropriate service level objectives (SLOs).

#### Achieving High Availability

The core principle behind high availability is "designing for failure". It assumes that individual components *will* fail and builds a system that can gracefully handle those failures. Key patterns include:

- **Redundancy:** This involves having backup components for critical parts of the system. If one component fails, a redundant one can take over, eliminating single points of failure.
- **Replication:** This is a specific form of redundancy where data and services are duplicated across multiple machines or data centers. Common database replication patterns include master-slave and master-master replication.
- **Fail-over:** This is the automated process of detecting a component failure and switching traffic to a redundant, standby component. Fail-over can be active-passive (the standby is idle until needed) or active-active (all components are handling traffic simultaneously).

## 1.3. Reliability: Building Trustworthy Systems

Reliability and availability are closely related but distinct. Availability means the system is operational; reliability means the system functions correctly and delivers the right output when it is operational. A system can be available (it responds to requests) but unreliable (it returns incorrect data).

A helpful analogy is a calculator: an available calculator turns on, but a reliable calculator consistently provides the correct answer to every calculation. Reliability builds user trust. It is achieved through robust error handling, fault tolerance mechanisms, and comprehensive testing strategies, including stress tests and simulations of real-world failure scenarios.

## 1.4. Performance: Latency and Throughput

Performance refers to the efficiency and responsiveness of a system. It is primarily measured using two

key metrics: latency and throughput.

#### Key Performance Metrics

- **Latency:** The time it takes to perform a single action or for a request to travel from the client to the server and back. This is what a user perceives as "speed".
- **Response Time:** A closely related metric that includes not just the network travel time (latency) but also the time the system takes to process the request.
- **Throughput:** The number of actions or operations a system can handle per unit of time (e.g., requests per second). This measures the system's capacity.

The fundamental goal in system design is often to maximize throughput while keeping latency within an acceptable range for a good user experience. A delay of just a few hundred milliseconds can be enough to frustrate users and negatively impact business outcomes.

| **Operation** | **Approximate Time** |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 25 ns |
| Main memory reference | 100 ns |
| Compress 1 KB with Zippy | 3,000 ns (3 µs) |
| Send 2 KB over 1 Gbps network | 20,000 ns (20 µs) |
| Read 1 MB sequentially from memory | 250,000 ns (250 µs) |
| Round trip within same datacenter | 500,000 ns (500 µs) |

| | |
|---|---|
| Disk seek | 10,000,000 ns (10 ms) |
| Read 1 MB sequentially from disk | 20,000,000 ns (20 ms) |
| Send packet CA->Netherlands->CA | 150,000,000 ns (150 ms) |

*(Data sourced from common industry knowledge benchmarks referenced in S6, S19)*

This table provides a crucial mental model for performance optimization. It highlights the immense difference in speed between operations. Accessing data from memory is orders of magnitude faster than accessing it from a disk, which in turn is orders ofmagnitude faster than a network round trip across continents. An architect with these numbers internalized can quickly identify potential performance bottlenecks in a design.

# Chapter 2: Core Principles of Distributed Systems

This chapter covers the inescapable laws and models that govern systems built across multiple machines. Understanding these theoretical foundations is crucial for making informed architectural decisions.

## 2.1. The CAP Theorem: The Foundational Trilemma

The CAP theorem, also known as Brewer's theorem, is a fundamental principle for distributed data stores. It states that in a distributed system, it is impossible to simultaneously provide more than two of the following three guarantees: Consistency, Availability, and Partition Tolerance.

#### Consistency, Availability, Partition Tolerance Explained

- **Consistency:** This guarantee means that every read operation receives the most recent write or an error. In a consistent system, all nodes see the same data at the same time. When data is written to one node, it is replicated, and any subsequent read from any other node will return that new data.
- **Availability:** This guarantee means that every request receives a non-error response, though it may not contain the most recent data. The system remains operational and responsive even if some

nodes are down or unable to communicate.

- **Partition Tolerance:** This is the system's ability to continue operating despite a network partition—a communication break between nodes. Messages may be dropped or delayed between nodes, but the system as a whole does not fail.

#### Why Partition Tolerance is Non-Negotiable

In any real-world distributed system that communicates over a network, partitions are inevitable. Network hardware can fail, connections can become congested, and data centers can lose connectivity. Because an architect cannot simply wish away network failures, Partition Tolerance (P) must be a given. This reality forces a direct trade-off: during a network partition, a system must choose between maintaining Consistency (C) or Availability (A).

#### Navigating the Trade-off: CP vs. AP Systems

- **CP (Consistency + Partition Tolerance):** A CP system chooses to sacrifice availability to ensure consistency. When a partition occurs, the system may be unable to process requests that require communication with the partitioned node. To prevent returning stale data, it will return an error or time out. This model is essential for systems where data correctness is paramount, such as banking applications or financial ledgers. Seeing a slightly outdated account balance is often less acceptable than seeing a temporary error message.
- **AP (Availability + Partition Tolerance):** An AP system chooses to sacrifice consistency to ensure availability. During a partition, nodes will respond to requests with the most recent version of the data they have locally, even if it is not the globally latest version. This model is suitable for systems where being online and responsive is more critical than having perfectly up-to-date data. Examples include social media feeds, where seeing a post a few seconds late is acceptable, or e-commerce shopping carts, where it is better to let a user add an item than to show an error.

The choice between CP and AP is not merely a technical one; it is a business decision. The CAP theorem provides a framework for architects to ask stakeholders: "What is the business cost of returning stale data versus the cost of being unavailable?" This reframes the theorem from a theoretical constraint into a practical tool for aligning system behavior with business requirements. While the theorem is often presented as a rigid "pick two" choice, modern interpretations emphasize that the goal is to maximize the desired combination of consistency and availability that makes sense for the specific application, with plans for how to operate during a partition and recover afterward.

## 2.2. The PACELC Theorem: Extending CAP to Normal Operation