­Reliability

# Core Reliability Concepts

* **Reliability**: Continuing to work correctly even when things go wrong
* **Working Correctly** means:
	+ Performs expected functions
	+ Tolerates user mistakes
	+ Maintains adequate performance under load
	+ Prevents unauthorized access
	+ Handles expected data volume

# Fault vs Failure

* **Fault**: Single component deviating from its specification
* **Failure**: Entire system stops providing required service
* Key principle: Build reliable systems from unreliable parts
* Interesting approach: Deliberately trigger faults (e.g., Netflix Chaos Monkey) to test system resilience

# Types of Faults

## 1. Hardware Faults

* Examples:
	+ Disk crashes
	+ RAM failures
	+ Power outages
	+ Network issues
* Solutions:
	+ Component redundancy (RAID, dual power supplies)
	+ Hot-swappable components
	+ Multi-machine redundancy
	+ Software fault-tolerance

## 2. Software Faults (Systematic Errors)

* More dangerous because they:
	+ Are correlated across nodes
	+ Cause more system failures than hardware faults
* Common examples:
	+ Bugs that crash servers on specific inputs
	+ Resource exhaustion
	+ Service dependency failures
	+ Cascading failures
* Mitigation strategies:
	+ Careful testing
	+ Process isolation
	+ Monitoring and analysis
	+ Allow crash and restart
	+ Regular assumption validation

# Key Takeaways

* Prevention isn't always possible (especially for security issues)
* Modern systems favor software fault-tolerance over hardware redundancy
* Regular testing of fault-tolerance mechanisms is crucial
* System assumptions should be continuously verified
* Monitoring and quick recovery mechanisms are essential

Scalability

1. **Describing Load**

**What Is a Load Parameter?**

A **load parameter** is a measurable factor that defines how much "work" a system must handle. It represents the system's activity level or workload and is critical for assessing scalability and performance under different conditions.

**How It Relates to Scalability**

1. **Workload Description**:
	* It quantifies the current demand on a system (e.g., number of requests per second, database writes/reads, active users).
	* Helps describe typical, peak, or extreme cases of system usage.
2. **Growth Projection**:
	* Scalability is about answering: *What happens if this load doubles or grows exponentially?*
	* Understanding load parameters allows you to predict bottlenecks and adjust the architecture to handle increased traffic effectively.
3. **System-Specific**:
	* The choice of load parameter depends on the application. For example:
		+ In **Twitter**, load is tied to fan-out size and read/write operations.





* + - In a **video streaming service**, it might involve concurrent streams or data bandwidth.
		- In an **e-commerce platform**, it might include transaction rates or search queries per second.

**Load Parameters and Scalability Challenges**

Scalability challenges arise when a particular load parameter becomes a bottleneck. For example:

* **Twitter**: Fan-out size (the number of followers receiving tweets) is a primary scalability challenge.
* **E-commerce**: Database writes during peak sale events.
* **Streaming services**: High concurrent users stressing bandwidth and content delivery networks (CDNs).

**Why Defining Load Parameters Is Key**

Without clearly identifying load parameters:

1. You cannot evaluate the current state of the system.
2. Planning for growth or scaling becomes guesswork.
3. Optimizations may miss the real bottleneck (e.g., focusing on average load rather than peak loads).
4. **Describing Performance**

**Performance is about balancing load and resources to ensure the system behaves as expected. Batch systems aim to maximize throughput, while online systems focus on minimizing response times. Both metrics are crucial for identifying scalability bottlenecks and enhancing system performance as workloads increase.**

**1. Ideal case: "The running time of a batch job is the size of the dataset divided by the throughput."**

* **Batch Job**: A batch job processes large volumes of data in chunks or batches, usually running asynchronously.
* **Dataset Size**: This refers to the total volume of data you want to process (e.g., millions of records).
* **Throughput**: Throughput refers to how much data a system can process per unit of time, such as the number of records processed per second.

In an **ideal world**, if we have a batch job that divides the work evenly across multiple worker processes (e.g., CPU cores or machines), the job's **total running time** would be determined by how much data needs to be processed divided by the throughput of each worker.

For example:

* If the dataset has 10 million records.
* And each worker can process 1 million records per minute.
* If you have 10 workers, the total time would be around 1 minute (10 million records ÷ 10 million records per minute).

This is an **idealized scenario** where everything works perfectly.

**2. Real-World Case: "In practice, the running time is often longer, due to skew."**

* **Skew**: Skew occurs when the data isn't evenly distributed across workers. This means some workers may end up with more data to process than others, causing imbalance.

In real-world systems, when you split a large dataset into smaller chunks and assign these chunks to different worker processes, you may not get a perfectly even distribution. Some workers might end up with very small chunks of data, while others have much larger chunks. This **uneven distribution** leads to **skew**, which can make the overall running time longer.

For example:

* If you have 10 workers, but due to skew, one worker gets 8 million records while the other 9 workers share the remaining 2 million records.
* The worker with 8 million records will take much longer to process its chunk compared to the other workers. Therefore, the job can't be considered finished until the slowest worker (the one with the most data) finishes processing its task.

**3. Waiting for the Slowest Task: "Needing to wait for the slowest task to complete."**

* In a parallel system, all workers (or tasks) work in **parallel**, but the job doesn’t finish until the **last task finishes**. This means that the overall running time is constrained by the **slowest worker** (also called the **bottleneck**).

If one worker takes significantly longer than the others (due to having more data to process or being slower for any reason), the system has to wait for that worker to finish, causing delays. This is a common issue in distributed systems and can reduce scalability.

**4. Putting It Together:**

* In the **ideal world**, the batch job finishes in a time proportional to the size of the dataset divided by the throughput.
* However, in the **real world**, **skew** and the need to wait for the **slowest worker** increase the time it takes to complete the job. The time spent by the slowest worker becomes the limiting factor, and the overall system performance is constrained by this worker.

**Scalability Implications:**

* **Underutilization of resources**: If you have many workers but one worker takes much longer than the others, many workers sit idle, leading to inefficient use of resources.
* **Performance bottleneck**: The performance of the entire system is limited by the slowest task. If you cannot eliminate this skew or imbalance, the system’s scalability may be compromised as the dataset grows.
* **Solutions**:
	+ **Load balancing**: Distribute the data more evenly to prevent skew.
	+ **Dynamic task allocation**: Allow tasks to be reassigned or adjust the workload during execution to balance the processing time across workers.
	+ **Worker optimization**: Ensure that each worker can handle its task as efficiently as possible, reducing the chances of one task becoming a bottleneck.

In summary, while you might theoretically expect a batch job's running time to be the dataset size divided by throughput, real-world factors like data skew and waiting for the slowest worker often make the job take longer, impacting scalability.



# Response Time Metrics Overview

## Basic Metrics

* **Mean (Average)**: Sum of all values divided by number of values
	+ Less useful for understanding typical user experience
	+ Can be skewed by outliers
* **Median (p50)**: Middle value when response times are sorted
	+ 50% of requests are faster
	+ 50% of requests are slower
	+ Better representation of "typical" user experience

## Important Percentiles

* **p95** (95th percentile): 95% of requests are faster than this threshold
* **p99** (99th percentile): 99% of requests are faster than this threshold
* **p999** (99.9th percentile): 99.9% of requests are faster than this threshold
* Collectively known as "tail latencies"

# Business Impact

* Amazon example:
	+ Uses p999 for internal service requirements
	+ 100ms increase in response time = 1% decrease in sales
	+ 1-second slowdown = 16% decrease in customer satisfaction
	+ p9999 (99.99th percentile) optimization deemed not cost-effective

# Practical Applications

## SLAs and SLOs

* Define expected service performance
* Typical metrics include:
	+ Median response time (e.g., < 200ms)
	+ p99 threshold (e.g., < 1s)
	+ Uptime requirements (e.g., 99.9%)

## Performance Testing Considerations

* Measure response times from client side
* Watch for queueing delays at high percentiles
* Load testing should send requests independently of response times
* "Head-of-line blocking": Slow requests can delay processing of subsequent ones

# Why p999 for Internal Services?

The key insight about p999 (99.9th percentile) is:

* Most valuable customers often have the slowest requests because:
	+ They have larger accounts
	+ They've made more purchases
	+ They have more data to process
* These 1-in-1000 slower requests often correspond to your highest-value customers
* Therefore, optimizing for p999 means protecting your most valuable customer experiences

# Practical Applications Detailed

## Service Level Agreements (SLA)

* **Definition**: Contract between service provider and client defining expected service performance
* **Components**:
	+ Uptime guarantees
	+ Performance metrics
	+ Penalty/refund terms if requirements aren't met

## Service Level Objectives (SLO)

* **Definition**: Specific, measurable targets for service performance
* **Common Metrics**:
	+ Median response time < 200ms (typical user experience)
	+ p99 < 1 second (worst acceptable experience)
	+ Service availability of 99.9% (allowable downtime)

## Performance Testing Best Practices

1. **Client-Side Measurement**
	* **Why**: Captures full user experience
	* **What**: Includes network delays, queuing, processing time
2. **Queue Management**
	* **Head-of-line blocking**: When slow requests delay faster ones behind them
	* **Impact**: Even fast requests can appear slow due to queuing
3. **Load Testing Guidelines**
	* **Do**: Send requests continuously, independent of response times
	* **Don't**: Wait for each response before sending next request
	* **Why**: Waiting creates artificially short queues, giving inaccurate results
* The Wrong Way ❌
	+ Request 1 ➜ Wait for Response 1 ➜ Request 2 ➜ Wait for Response 2 ➜ Request 3
* **Problem**:
	+ Creates unrealistic gaps between requests
	+ System appears less loaded than reality
	+ Queue never builds up properly
	+ Results are overly optimistic
* The Right Way ✅
	+ Request 1 ➜ Request 2 ➜ Request 3 ➜ Request 4 (Continuous)
	+ ↓ ↓ ↓ ↓
	+ Response 1 Response 2 Response 3 Response 4 (Independent)
* **Benefits**:
	+ Simulates real-world traffic patterns
	+ Allows proper queue buildup
	+ Shows true system behavior under load
	+ Reveals actual latency issues

# Key Terms

* **Tail Latency**: Response times at high percentiles (p95, p99, p999)
* **Head-of-line blocking**: Delay of fast requests by slow ones in front
* **Response Time**: Total time from request to response
* **Queue Delay**: Time spent waiting for processing to begin
* **Load Testing**: Simulating real-world usage patterns to test performance





# Approaches for Coping with Load

## Key Concepts

### Scaling Strategies

* **Vertical Scaling (Scale Up)**: Increasing capacity of existing machines
* **Horizontal Scaling (Scale Out)**: Adding more machines to distribute load

### Scalability Challenges

* Architecture suitable for one load level may not work for 10x higher load
* Need to rethink architecture for each order-of-magnitude load increase

### Shared-Nothing Architecture

* Distributing load across multiple machines
* Avoids single powerful machine becoming a bottleneck

### Elasticity

* Automatic scaling up/down based on detected load changes
* Useful for unpredictable load, but manually scaled systems are simpler

### Stateful vs Stateless Services

* Distributing stateless services is straightforward
* Scaling stateful data systems introduces significant complexity

## Scalability Principles

* No "one-size-fits-all" scalable architecture
* Design based on expected load parameters and access patterns
* General-purpose building blocks arranged in familiar patterns

## Key Takeaways

* Scaling approach depends on application's specific requirements
* Vertical and horizontal scaling are complementary, not mutually exclusive
* Elasticity trades complexity for better handling of unpredictable load
* Stateful services are more challenging to scale than stateless ones

# Stateful vs Stateless Services

**Stateless Services**:

* Services that don't maintain any persistent data or session information
* Each request can be handled independently without relying on previous interactions
* Examples: Web servers, API endpoints, simple caching layers

**Stateful Services**:

* Services that maintain persistent state or session information
* Each request depends on the current state of the system
* Examples: Databases, message queues, user session management

## Scaling Differences

**Scaling Stateless Services**:

* Straightforward to distribute across multiple machines
* Can load balance requests across servers without coordination
* Easy to add more capacity as needed

**Scaling Stateful Services**:

* Much more complex to distribute across a cluster
* Need to manage data partitioning, replication, and consistency
* Coordination between nodes introduces overhead and potential failures

The key point is that stateful services, like databases, have inherent complexity around managing data and state across a distributed system. This makes scaling them significantly more challenging compared to scaling stateless services.

Stateless services can be easily scaled out by adding more independent servers, but stateful services require carefully architecting the data storage and management layer to maintain reliability and consistency.

By "*coordination*" in the context of scaling stateful services, I'm referring to the necessary communication and synchronization between the various nodes or servers in the distributed system.

Some examples of the coordination required:

1. **Data Partitioning**: Deciding how to split up the data across multiple servers (e.g. sharding a database)
2. **Replication**: Ensuring data is replicated across servers for redundancy and high availability
3. **Consistency**: Making sure all the replicated copies of data stay in sync and consistent
4. **Failover**: Automatically shifting traffic to backup nodes when primary nodes fail
5. **Load Balancing**: Distributing requests across the available servers in the cluster

This coordination introduces overhead and complexity that is not present in simple stateless services. The nodes in a stateful distributed system need to constantly communicate and negotiate to maintain the overall state of the system.

In contrast, stateless services can scale out more independently, as each server can handle requests without needing to coordinate with the others.