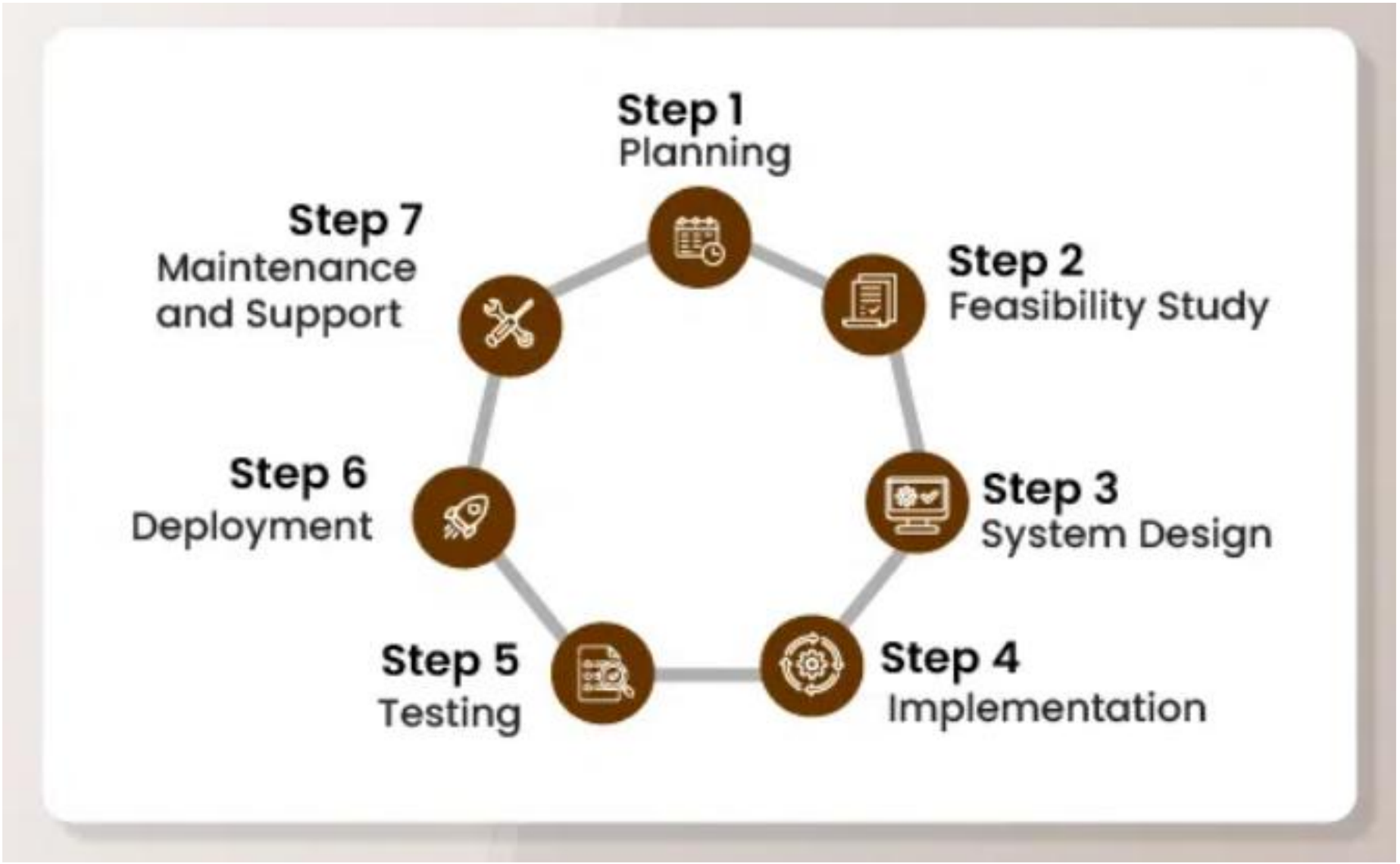
Introduction to System Design

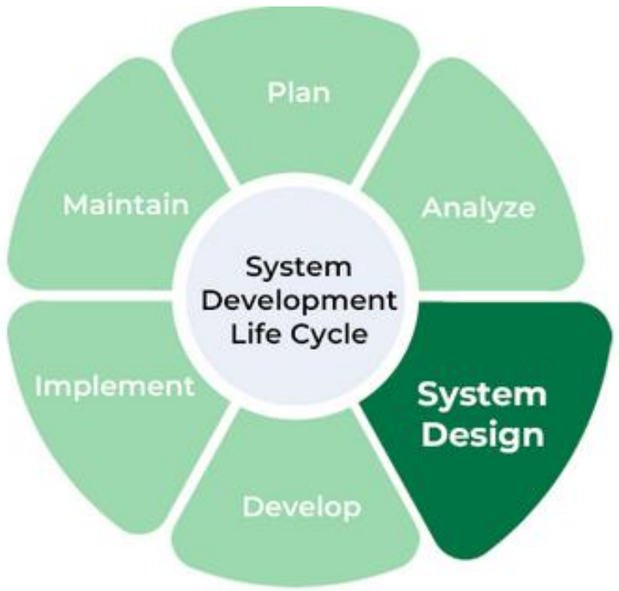
Introduction

- Process of defining the **architecture**, **components**, **modules**, **interfaces**, and **data** for a system to satisfy specified requirements.
- Way of translating user requirements into a detailed blueprint that guides the implementation phase
- AIM: is to create a **well-organized** and **efficient structure** that meets the intended purpose while considering factors like **scalability**, **maintainability**, and **performance**.

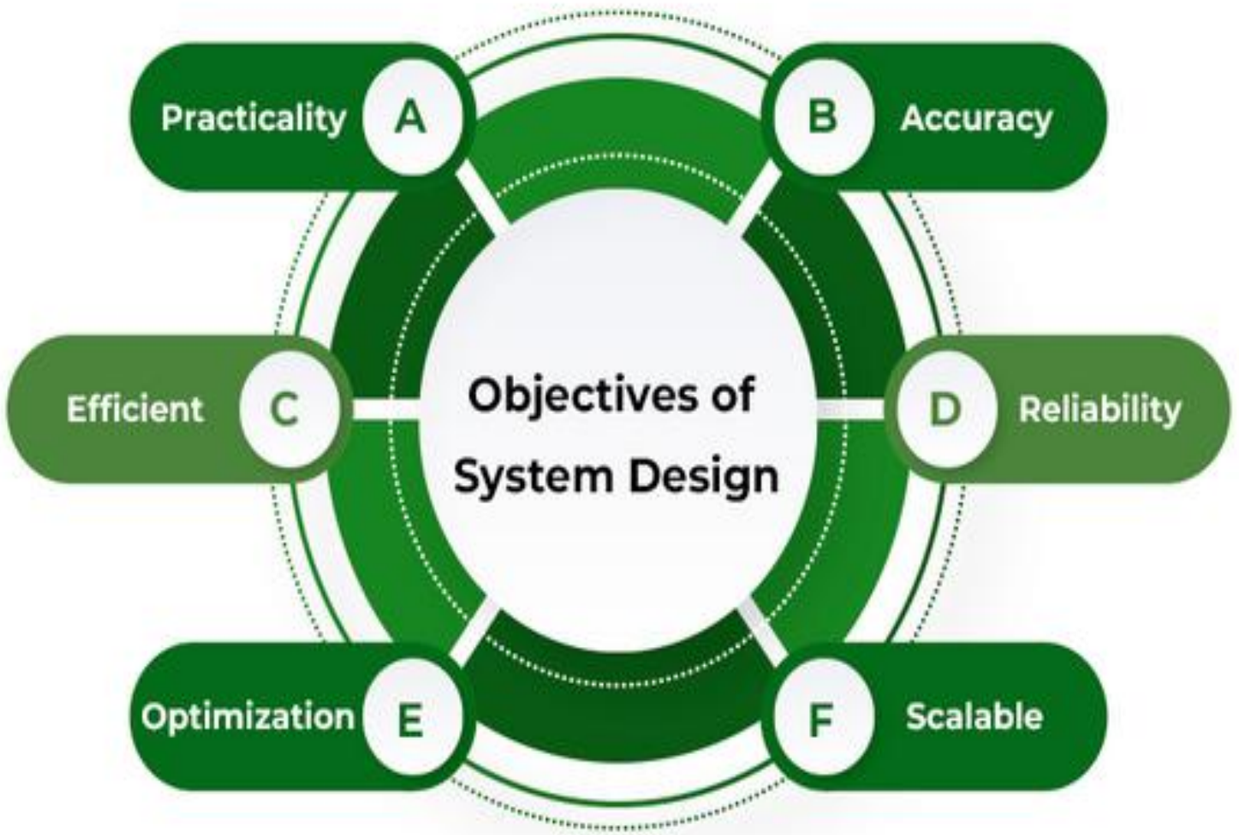

Step 1
Planning
Step 2
Feasibility Study
Step 3
System Design
Step 4
Implementation
Step 5
Testing
Step 6
Deployment
Step 7
Maintenance
and Support

Why do we study system design?

- Without the **designing phase**, you cannot jump to the implementation or the testing part.
- System Design is a vital step in the development of a system but also provides the backbone to handle exceptional scenarios because it represents the **business logic** of the software.



Objectives of System Design



Advantages of System Design

The major advantages of System Design include:

- **Reduces the Design Cost of a Product**: By using established design patterns and reusable components, teams can lower the effort and expense associated with creating new software designs.
- **Speedy Software Development Process**: Using frameworks and libraries accelerates development by providing pre-built functionalities, allowing developers to focus on unique features.
- **Saves Overall Time in SDLC**: Streamlined processes and automation in the Software Development Life Cycle (SDLC) lead to quicker iterations and faster time-to-market.
- **Increases Efficiency and Consistency of a Programmer**: Familiar tools and methodologies enable programmers to work more effectively and produce uniform code, reducing the likelihood of errors.
- **Saves Resources**: Optimized workflows and shared resources minimize the need for redundant efforts, thereby conserving both human and material resources.

requirements

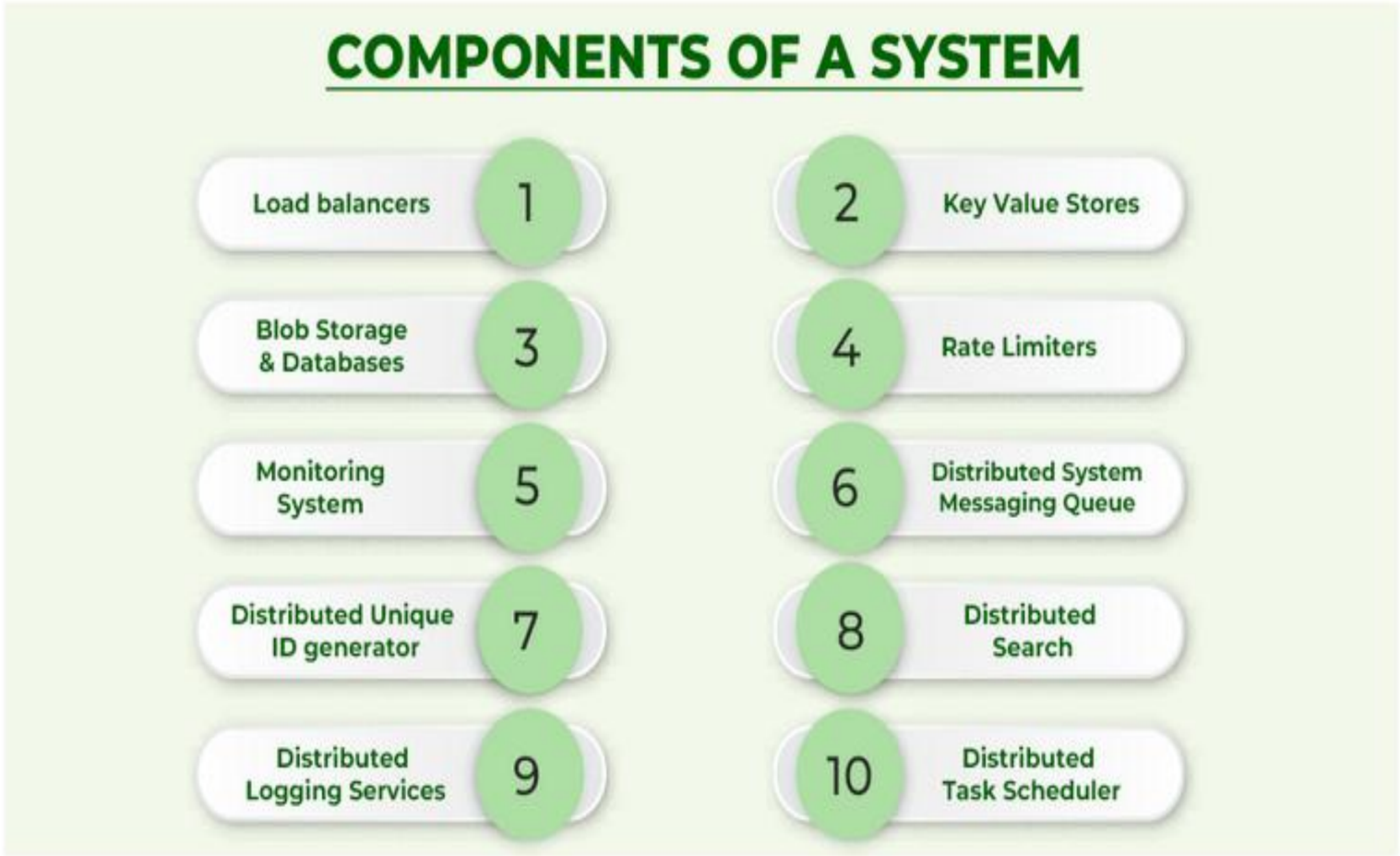
- developed first
- agreement between client and analyst
- describes what the system will provide and any constraints
- natural language and conceptual diagrams

specifications

- precise/specific statement of requirements
- agreement between client and developers
- describes the “how” in non-technical terms
- formal notation and more detailed diagrams

→ design

Components of a system



- **Load Balancers**: Distribute incoming traffic across multiple servers to optimize performance and ensure reliability.
- **Key-Value Stores**: Storage systems that manage data as pairs of keys and values, often implemented using distributed hash tables.
- **Blob Storage**: A service for storing large amounts of unstructured data, such as media files (e.g., YouTube, Netflix).
- **Databases**: Organized collections of data that facilitate easy access, management, and modification.
- **Rate Limiters**: Control the maximum number of requests a service can handle in a given timeframe to prevent overload.
- **Monitoring Systems**: Tools that enable administrators to track and analyze infrastructure performance, including bandwidth and CPU usage.
- **Distributed Messaging Queues**: Mediums that facilitate communication between producers and consumers, ensuring reliable message delivery.
- **Distributed Unique ID Generators**: Systems that generate unique identifiers for events or tasks in a distributed environment.
- **Distributed Search**: Mechanisms that allow users to search across multiple data sources or websites for relevant information.
- **Distributed Logging Services**: Systems that collect and trace logs across services to monitor and troubleshoot applications.
- **Distributed Task Schedulers**: Tools that manage and allocate computational resources for executing tasks across a distributed system.

SDLC: System Design Life cycle

Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7
Planning	**Feasibility Study**	**System Design**	**Implement -ation**	**Testing**	**Deploy -ment**	**Maintena -nce and Support**
Define the project scope, goals, and resources.	Asses the practical- ity of the proposed system.	Develop a blueprint of the system architec- ture and compo- nents.	Transform the design into an opera- tional system.	Verify that the system meets the specified require- ments.	Introduce the system to its intend- ed envi- ronment.	Ensure the ongoing function- ality and address any issues that arise.

SYSTEM ARCHITECTURE

- **System Architecture** is a way in which we define how the components of a design are **depicted design** and **deployment of software**. It is basically the skeleton design of a software system depicting components, abstraction levels, and other aspects of a software system. In order to understand it in a layman's language, it is the aim or logic of a business should be crystal clear and laid out on a single sheet of paper.

Types of System Architecture Patterns

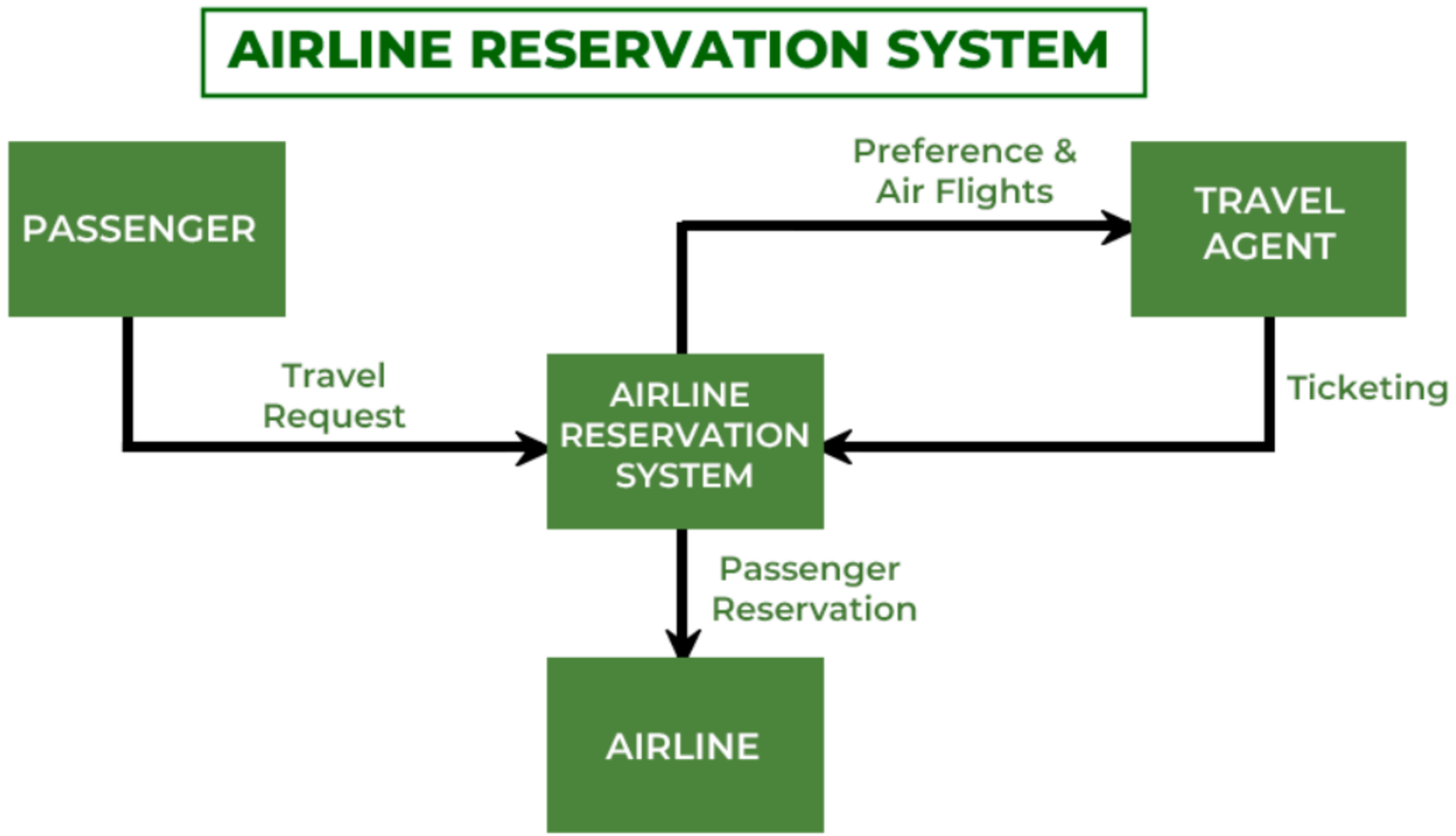
1. Client-Server Architecture Pattern: Separates the system into two main components: clients that request services and servers that provide them.

2. Event-Driven Architecture Pattern: Uses events to trigger and communicate between decoupled components, enhancing responsiveness and scalability.

3. Microkernel Architecture Pattern: Centers around a core system (microkernel) with additional features and functionalities added as plugins or extensions.

4. Microservices Architecture Pattern: Breaks down applications into small, independent services that can be developed, deployed, and scaled independently.

System Design Example: Airline Reservation System



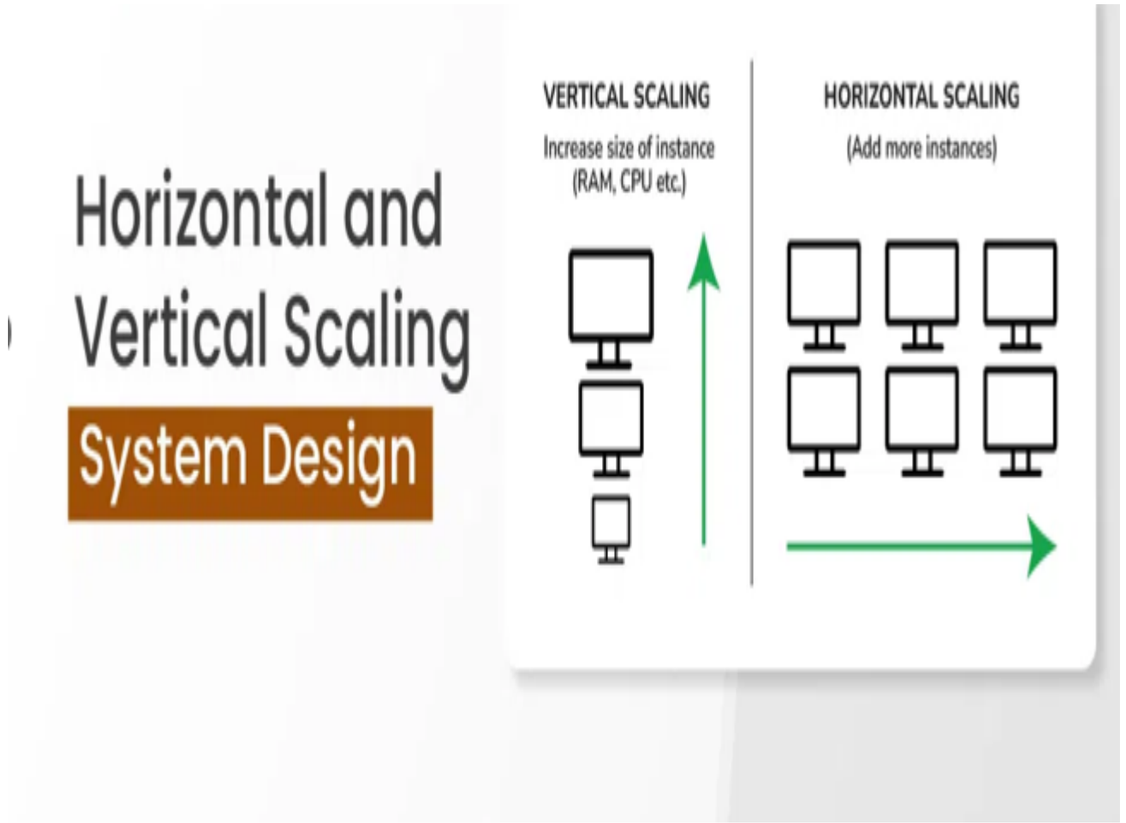
System Design Characteristics:

Availability, Reliability, Scalability: Vertical Scaling (Scaling Up), Horizontal Scaling (Scaling Out), Maintainability, Consistency, Fault Tolerance, latency, Throughput

Horizontal and Vertical Scaling | System Design

In system design, scaling is crucial for managing increased loads.

Horizontal scaling *and* ***vertical scaling*** *are two different approaches to scaling a system, both of which can be used to improve the performance and capacity of the system.*

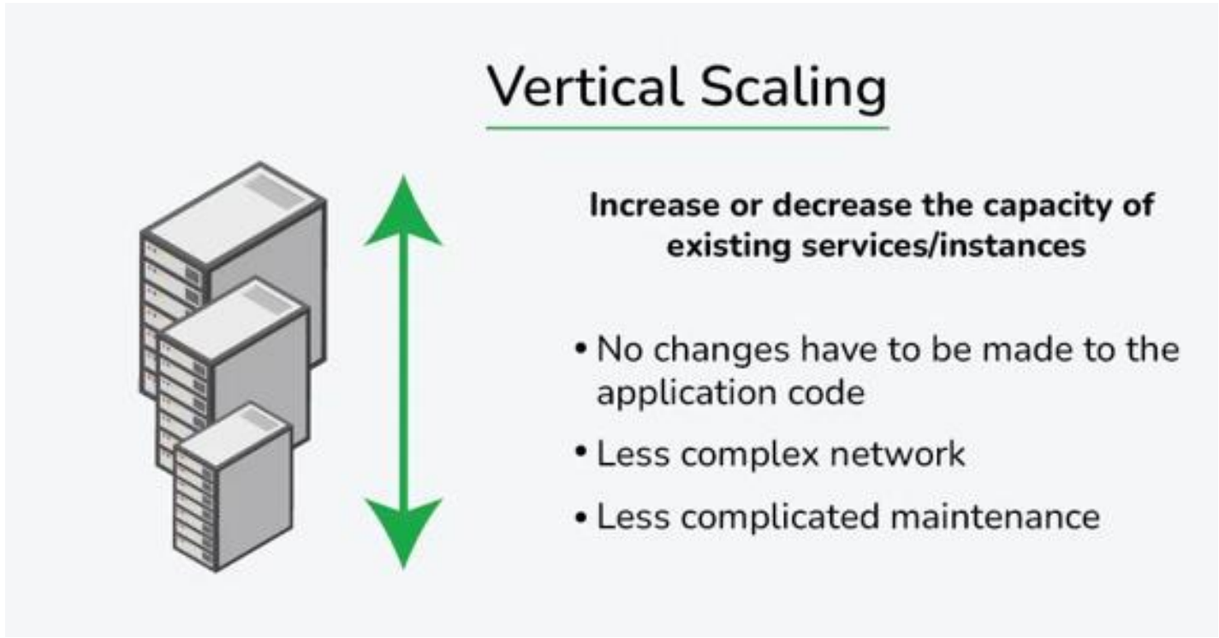


What is Vertical Scaling?

Vertical scaling, also known as scaling up, refers to the process of increasing the capacity or capabilities of an individual hardware or software component within a system.

•You can add more power to your machine by adding better processors, increasing RAM, or other power-increasing adjustments.

•Vertical scaling aims to improve the performance and capacity of the system to handle higher loads or more complex tasks without changing the fundamental architecture or adding additional servers.



Characteristics of the Vertical Scaling

•This approach is also known as the '**scale-up**' approach.

•It doesn't require any partitioning of data and all the traffic resides on a **single node with more resources**.

•Its implementation is easy.

•Less administrative effort as you need to manage just one system.

•Application compatibility is maintained.

•Mostly used in small and mid-sized companies.

•MySQL and Amazon RDS is a good examples of vertical scaling.

Advantages and Disadvantages of Vertical Scaling

•Advantages of vertical scaling

- **Increased capacity:** A server's performance and ability to manage incoming requests can both be enhanced by upgrading its hardware.
- **Easier management:** Upgrading a single node is usually the focus of vertical scaling, which might be simpler than maintaining several nodes.

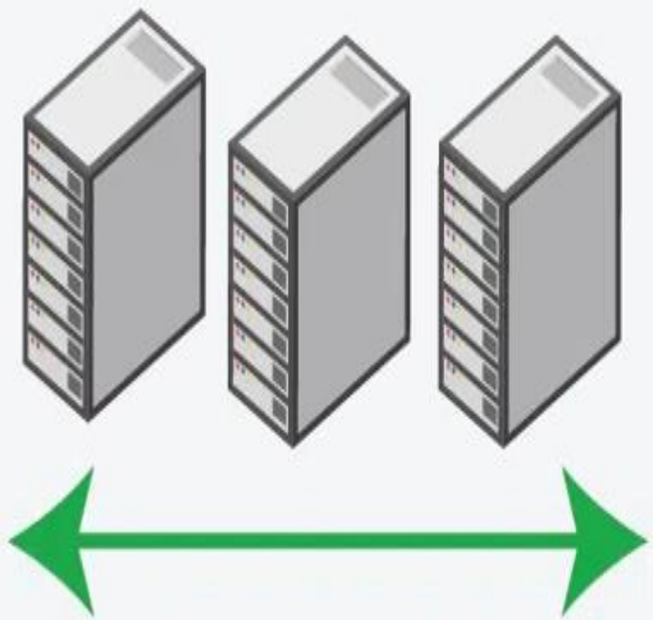
•Disadvantages of vertical scaling:

- **Limited scalability:** While horizontal scaling can be readily extended by adding more nodes and vertical scaling is constrained by the hardware's physical limitations.
- **Increased cost:** It may be more costly to upgrade a server's hardware than to add extra nodes.
- **Single point of failure:** One server still receives all incoming requests thus increasing the possibility of downtime in the event of a server failure.

What is Horizontal Scaling?

- Horizontal scaling, also known as scaling out, refers to the process of increasing the capacity or performance of a system by adding more machines or servers to distribute the workload across a larger number of individual units.
- In this approach, there is no need to change the capacity of the server or replace the server.
- Also, like vertical scaling, there is no downtime while adding more servers to the network.

Add more resources like virtual machines to your system to spread out the workload across them.



- Increases high availability
- Fewer periods of downtime
- Easy to resize according to your needs

Characteristics of the horizontal scaling

•This approach is also known as the '**scale-out**' approach.

•Horizontal scalability can be achieved with the help of a distributed file system, clustering, and load–balancing.

•Traffic can be managed effectively.

•Easier to run fault tolerance.

•Easy to upgrade.

•Easy to size and resize according to your needs.

•Implementation cost is less expensive compared to scaling up.

•Google with its Gmail and YouTube, Yahoo, Facebook, eBay, Amazon, etc. are heavily utilizing horizontal scaling.

•Cassandra and MongoDB are good examples of horizontal scaling.

•**Advantages of horizontal scaling**

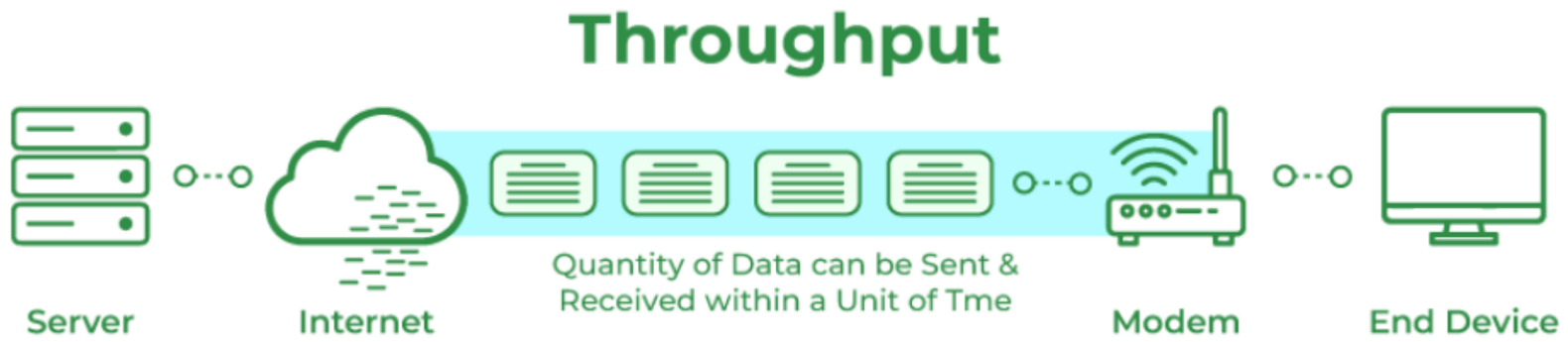
- **Increased capacity:** More nodes or instances can handle a larger number of incoming requests.
- **Improved performance:** By distributing the load over several nodes or instances, it is less likely that any one server will get overloaded.
- **Increased fault tolerance:** Incoming requests can be sent to another node in the event of a node failure, lowering the possibility of downtime.

•**Disadvantages of horizontal scaling**

- **Increased complexity:** Managing multiple nodes or instances can be more complex than managing a single node.
- **Increased cost:** Adding more nodes or instances will typically increase the cost of the system.

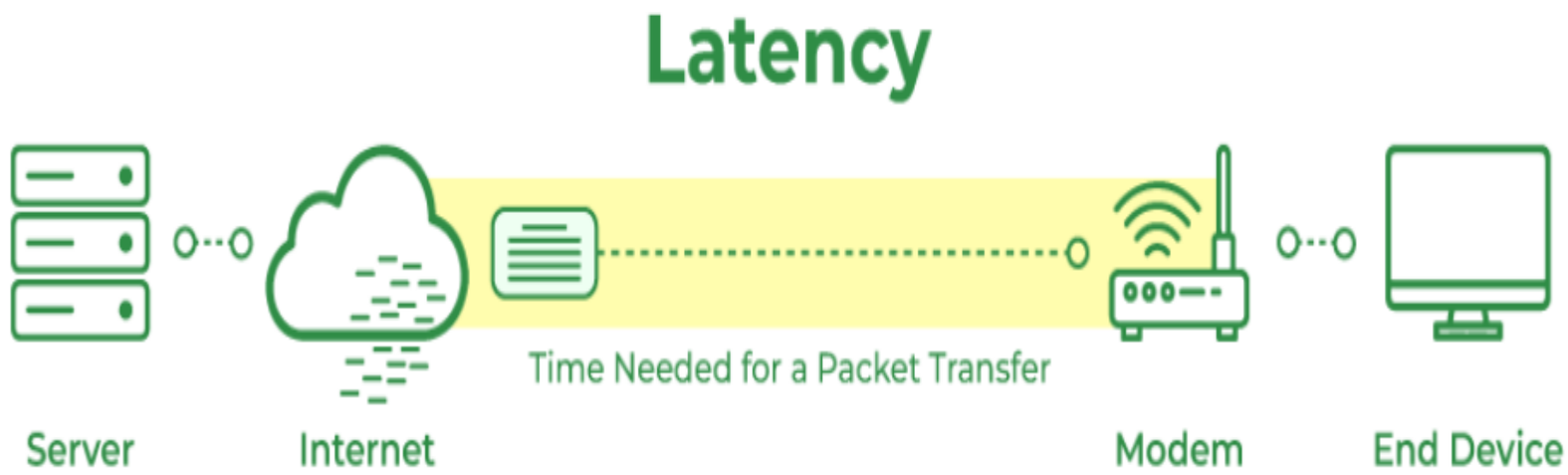
Throughput

Throughput is defined as the **measure of amount of data transmitted** successfully in a system, in a certain amount of time. In simple terms, throughput is considered as **how much data is transmitted** successfully over a period of time. The unit of measure for throughput is **bits per second or bps**.

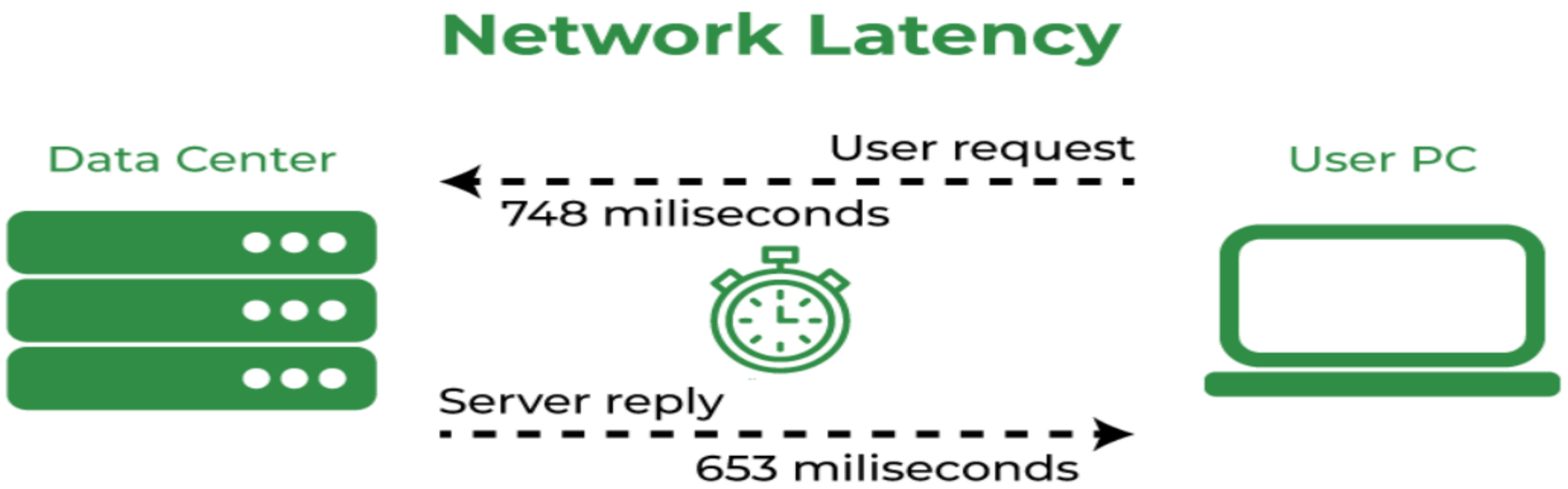


Latency

Latency is defined as the **amount of time** required **for a single data to be delivered** successfully. Latency is measured in **milliseconds (ms).**



Example: The **delay between user input and web application response to the same input is known as latency**.



Reasons for high Latency
Now you must be wondering about the factors that are responsible for delays. So high latency mainly depends on 2 factors:
Network Delays
Mathematical Calculation Process Delays

In **monolithic architecture**, as we know there is only a single block and all network calls are local within hence network delay is zero and hence latency equals computational delays only (which if not latency equals zero in monolithic systems)

In **distributed systems**, there is a networks over which signals are passed to and fro hence there will for sure be network delay.

Components Affecting Latency:
Packet Size: Smaller the packet chunk size faster the transmission and the lower the latency.
Packet Loss: Transmission of huge packets of various sizes in medium losses to very few losses in packets.
Medium of transmission: Optical fiber is the fastest way of transmission.
Distance between Nodes: Poor signal will increase latency and great connectivity decreases to a greater extent.
Signal strength: Good signal strength reduces latency.
Storage delays: Stored information in a database and fetching from it requires very little time which supports increasing latency.

How to Reduce latency:
Use a content delivery network (CDN): CDNs help to cut down on latency. In order to shorten the distance between users and reduce the amount of time that data must travel over great distances, CDN servers are situated at various locations.
Upgrading computer hardware/software: Improving or fine-tuning mechanical, software, or hardware components can help cut down on computational lag, which in turn helps cut down on latency.
Cache: A cache is a high-speed data storage layer used in computers that temporarily store large amounts of transient data. By caching this data, subsequent requests for it can be fulfilled more quickly than if the data were requested directly from its original storage location. This lessens latency as well.

Availability is the **percentage of time the system is up** and working for the needs.

How availability is measured?

Now you must be thinking about what are these levels and how they are measured. Levels in availability are measured via downtime per year via order of 'nines'. More 'nines' lead to lesser downtime.

It is as shown below via table as follows:

Availability(%)	Downtime/Year
90	~36.5 days
99	~3.65 days
99.9	~8.7 Hours
99.99	~52 Minutes
99.999	~6 Minutes

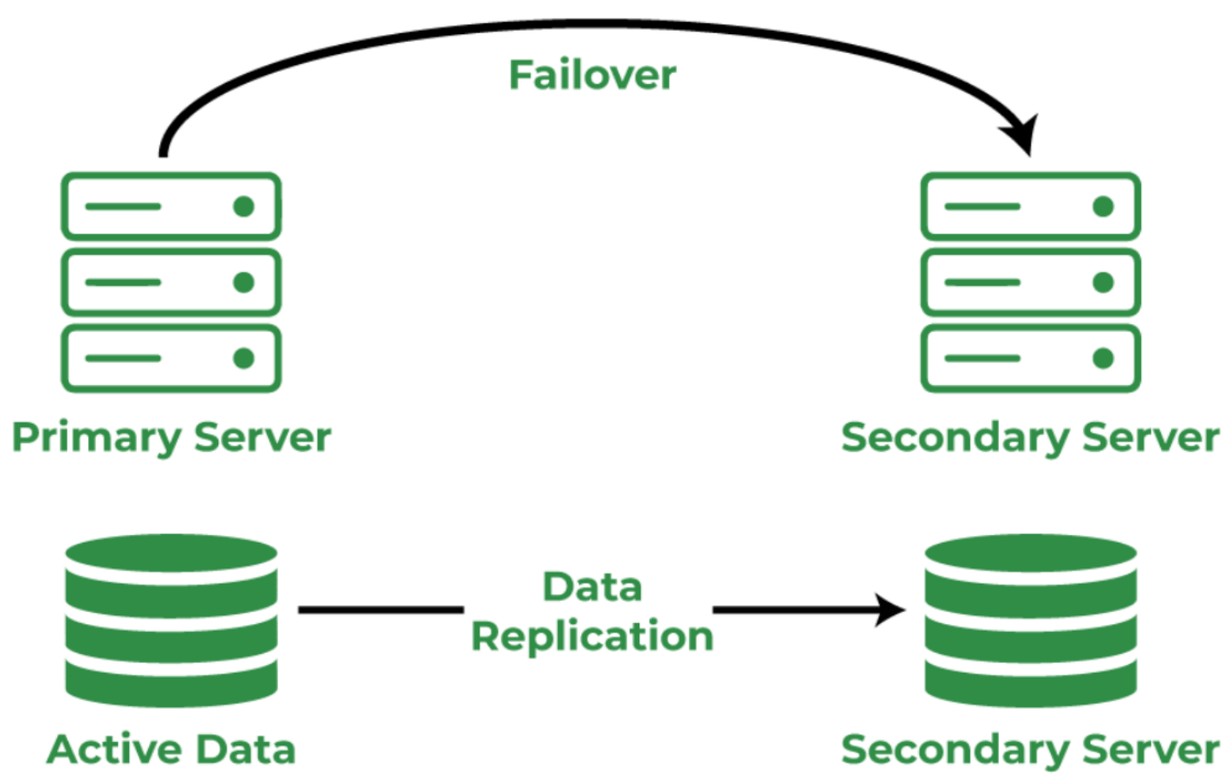
$$\text{Availability} = \frac{\text{Uptime}}{(\text{Uptime} + \text{downtime})}$$

How to increase Availability?
Eliminate SPOF(major and important)
Verify Automatic Failover
Use Geographic Redundancy
Continue upgrading and improving

From the above understanding, we can land up with two conclusions:
Availability is low in monolithic architecture due to SPOF.
Availability is high in distributed architecture due to redundancy.

Redundancy in System Design

Redundancy is defined as a concept where certain entities are **duplicated with aim to scale up the system and reduce over all down-time**. For example, as seen in the image below, we are duplicating the server. So if one server goes down, then we have a **redundant server** in our system to balance the load.



Load Balancer

A load balancer works as a “traffic cop” sitting in front of your server and routing client requests across all servers. It simply distributes the set of requested operations (database write requests, cache queries) effectively across multiple servers and ensures that no single server bears too many requests that lead to degrading the overall performance of the application. A load balancer can be a physical device or a virtualized instance running on specialized hardware or a software process.

Consider a scenario where an application is running on a single server and the client connects to that server directly without load balancing. It will look something like the one below.

How to handle the unavailability of a Load Balancer?
If the load balancer becomes unavailable, then the corresponding server will become unavailable and the system will go into downtime. In order to handle such cases, we do opt for two techniques:
Way 1: Using backup load balancer technique: It contains primary and secondary load balancers involving concepts of 'floating ID' and 'Health check'.
Way 2: Using DNS Server: For now newbies to understand associate this works quite similar to the redundancy principle.

Consistency in System Design

Consistency is referred to as **data uniformity in systems**.
When a user requests data, the system always returns the same data, regardless of the user's location, time, etc. Before a user receives data from any node, the server at which the data is updated or changed should successfully replicate the new data to all the nodes.
In order to understand consistency in simpler terms:
•Consider three nodes **X, Y, and Z** in a system, where very little **t1** time is taken for data transmission for replication from **X** to **Y** and **t2** from **X** to **Z**.
•Now the maximum among **t1** and **t2** is taken where no user is supposed to fetch info so as to avoid to get inconsistency in data by providing older records.
Example: Account Transactions

Time is a measure of sequences of events happening which is measured here in seconds in its SI unit.
It is measured using a clock which is of two types:
Physical Clock: responsible for the time between systems.
Logical Clock: responsible for the time within a system.

Performance vs Scalability

- Performance vs Scalability in System Design explores how systems balance speed (performance) and ability to handle growth (scalability). Imagine a race car (performance) and a bus (scalability). The car zooms quickly but can't carry many passengers, while the bus carries lots but moves slower.

- Similarly, in tech, a system may be super fast but crash with too many users (like the car), or handle many users but slow down (like the bus).
- Designing systems requires finding the right balance that is, fast enough for current needs, yet flexible to grow with demand.

What is Performance?

Performance in system design refers to how well a system executes tasks or processes within a given timeframe. It encompasses factors like speed, responsiveness, throughput, and resource utilization.

•For instance, a high-performance system might process a large amount of data quickly, respond to user inputs rapidly, and efficiently utilize system resources such as CPU, memory, and network bandwidth.
•Performance optimization involves techniques such as code optimization, caching, load balancing, and hardware upgrades to ensure that a system meets its performance requirements and delivers a smooth user experience.

Performance Optimization Techniques

Performance optimization techniques in system design involve various strategies aimed at improving the speed, efficiency, and resource utilization of a system. Some common techniques include:

- **Code optimization:**
  - Refining algorithms and code structures to minimize execution time and resource consumption. This can involve eliminating redundant operations, reducing algorithmic complexity, and optimizing loops and data structures.
- **Caching:**
  - Storing frequently accessed data or computed results in fast-access memory (cache) to reduce the need for repeated computations or database queries. Caching can significantly improve response times for frequently requested data.
- **Load balancing:**
  - Distributing incoming requests or tasks evenly across multiple servers or resources to prevent overloading any single component. Load balancers can dynamically adjust resource allocation based on current demand to optimize performance.
- **Parallelism and concurrency:**
  - Leveraging multiple threads or processes to execute tasks simultaneously, thereby utilizing available resources more efficiently and reducing overall processing time. Techniques such as parallel processing, asynchronous programming, and multi-threading can enhance system performance.
- **Database optimization:**
  - Optimizing database queries, indexing, and schema design to improve data retrieval speed and reduce latency. Techniques like query optimization, index optimization, and denormalization can enhance database performance.
- **Resource pooling and reuse:**
  - Reusing existing resources, connections, or objects rather than creating new ones for each request, reducing overhead and improving efficiency. Techniques like connection pooling in database connections or object pooling in object-oriented programming can help conserve resources.

What is Scalability?

Scalability in system design refers to a system's ability to handle increasing amounts of work or users without compromising performance. It involves designing a system so that it can easily accommodate growth in terms of data volume, user traffic, or processing demands without significant changes to its architecture.

•Scalable systems can seamlessly expand by adding more resources or components, such as servers or databases, to distribute the workload efficiently.

•This ensures that the system can continue to deliver high performance even as demands increase. Scalability is crucial for ensuring that a system remains responsive and reliable as it grows in size or usage.

Performance vs. Scalability

Aspect	Performance	Scalability
Definition	Focuses on optimizing speed and responsiveness	Focuses on handling increasing workload or users
Goal	Achieve maximum efficiency for current tasks	Accommodate growing demands without slowdown
Concerns	Speed, latency, throughput, resource utilization	Capacity, availability, distribution of workload
Key Techniques	Code optimization, caching, load balancing	Horizontal scaling, stateless architecture, microservices
Scaling Approach	Vertical scaling (scaling up)	Horizontal scaling (scaling out)
Impact of Growth	May degrade with increased workload	Maintains performance with increased workload
Resource Allocation	May require hardware upgrades for improvement	Adds more instances or nodes for improvement
Maintenance Complexity	Generally lower complexity	May involve higher complexity due to distributed nature
Example	A high-performance gaming server	A scalable social media platform

Latency

Latency refers to the time it takes for a request to travel from its point of origin to its destination and receive a response.
•Latency represents the delay between an action and its corresponding reaction.
•It can be measured in various units like seconds, milliseconds, and nanoseconds depending on the system and application.

What does it involve?
Latency involves so many things such as processing time, time to travel over the network between components, and queuing time.
•**Round Trip Time**: This includes the time taken for the request to travel to the server, processing time at the server, and the response time back to the sender.
•**Different Components:** Processing time, transmission time (over network or between components), queueing time (waiting in line for processing), and even human reaction time can all contribue to overall latency.

How does Latency works?
The time taken for each step—transmitting the action, server processing, transmitting the response, and updating your screen—contributes to the overall latency.
Example: Let see an example when player in an online game firing a weapon.

If your latency is high: You press "fire."
•The command travels through the internet to the server, which takes time.
•The server processes the shot.
•The result travels back to your device.
•Your screen updates the result.

The working of Latency can be understood by two ways:

1. **Network Latency**

In system architecture, network latency is a sort of latency that describes how long it takes for data to move between two points in a network. Using email as an example, we can consider it to be the time lag between sending an email and the recipient actually getting it. For real-time applications, it is measured in milliseconds or even microseconds, just like total latency.

2. System Latency

System latency refers to the overall time it takes for a request to go from its origin in the system to its destination and receive a response. Think of Latency as the "wait time" in a system. The time between clicking and seeing the updated webpage is the system latency. It includes processing time on both client and server, network transfers, and rendering delays.

Factors that causes High Latency

High latency can severely impact the performance and user experience of distributed systems. Here are key factors that contribute to high latency within this context:

Network Congestion: High traffic on a network can cause delays as data packets queue up for transmission.

Bandwidth Limitations: Limited bandwidth can cause delays in data transmission, particularly in data-intensive applications.

Geographical Distance: Data traveling long distances between distributed nodes can increase latency due to the inherent delays in transmission.

Server Load: Overloaded servers can take longer to process requests, contributing to high latency.

Latency in Database Queries: Complex or inefficient database queries can significantly increase response times.

How to measure Latency?

There are various ways to measure latency. Here are some common methods:

Ping: This widely used tool sends data packets to a target server and measures the round-trip time (RTT), providing an estimate of network latency between two points. **(RTT = 2 * one-way latency)**.

Traceroute: This tool displays the path data packets take to reach a specific destination, revealing which network hops contribute the most to overall latency.

MTR (traceroute with ping): Combines traceroute and ping functionality, showing both routing information and RTT at each hop along the path.

Performance profiling tools: Specialized profiling tools track resource usage and execution times within a system, providing detailed insights into system latency contributors.

Application performance monitoring (APM) tools: Similar to network monitoring tools for networks, APM tools monitor the performance of applications, including response times and latency across various components.

Problem Statement:
Calculate the round-trip time (RTT) latency for a data packet traveling between a client in New York City and a server in London, UK, assuming a direct fiber-optic connection with a propagation speed of 200,000 km/s.
•**Distance:** Distance between NYC and London: 5570 km
•**Propagation speed:** 200,000 km/s
•**Constraints:** Assume no network congestion or processing delays.
•**Desired Output:** RTT latency in milliseconds.

1. **Calculate One-Way Latency**: One-way latency is the time taken for the data to travel from the client to the server:
One-way latency = Distance / Propagation speed = 5570 KM / 200,000 Km/s = **27.85 ms**

2. Calculate RTT: The RTT is twice the one-way latency:
RTT = 2 × 27.85ms = **55.7ms**

Throughput

The rate at which a system, process, or network can move data or carry out operations in a particular period of time is referred to as throughput. Bits per second (bps), bytes per second, transactions per second, etc. are common units of measurement. It is computed by dividing the total number of operations or objects executed by the time taken.

For example, an ice-cream factory produces 50 ice-creams in an hour so the throughput of the factory is **50 ice-creams/hour**.

$$\text{Throughput} = \frac{\text{Number of Units Produced}}{\text{Time Periods}}$$

Here are a few contexts in which throughput is commonly used:

1.Network Throughput: Throughput in networking is the quantity of data that can be sent via a network in a specific amount of time. When assessing the effectiveness of communication routes, this measure is important.

2.Disk Throughput: In storage systems, throughput measures how quickly data can be read from or written to a storage device, usually expressed in terms of bytes per second.

3.Processing Throughput: In computing, especially in the context of CPUs or processors, throughput is the number of operations completed in a unit of time. It could refer to the number of instructions executed per second.

Problem: A web server handles 30,000 requests in 10 minutes.

Solution:

$$\text{Throughput} = \frac{30,000 \text{ requests}}{600 \text{ seconds}} = \boxed{50 \text{ RPS}}$$

Let's say your **Order Service** in an e-commerce system processes 1000 orders in 5 minutes.

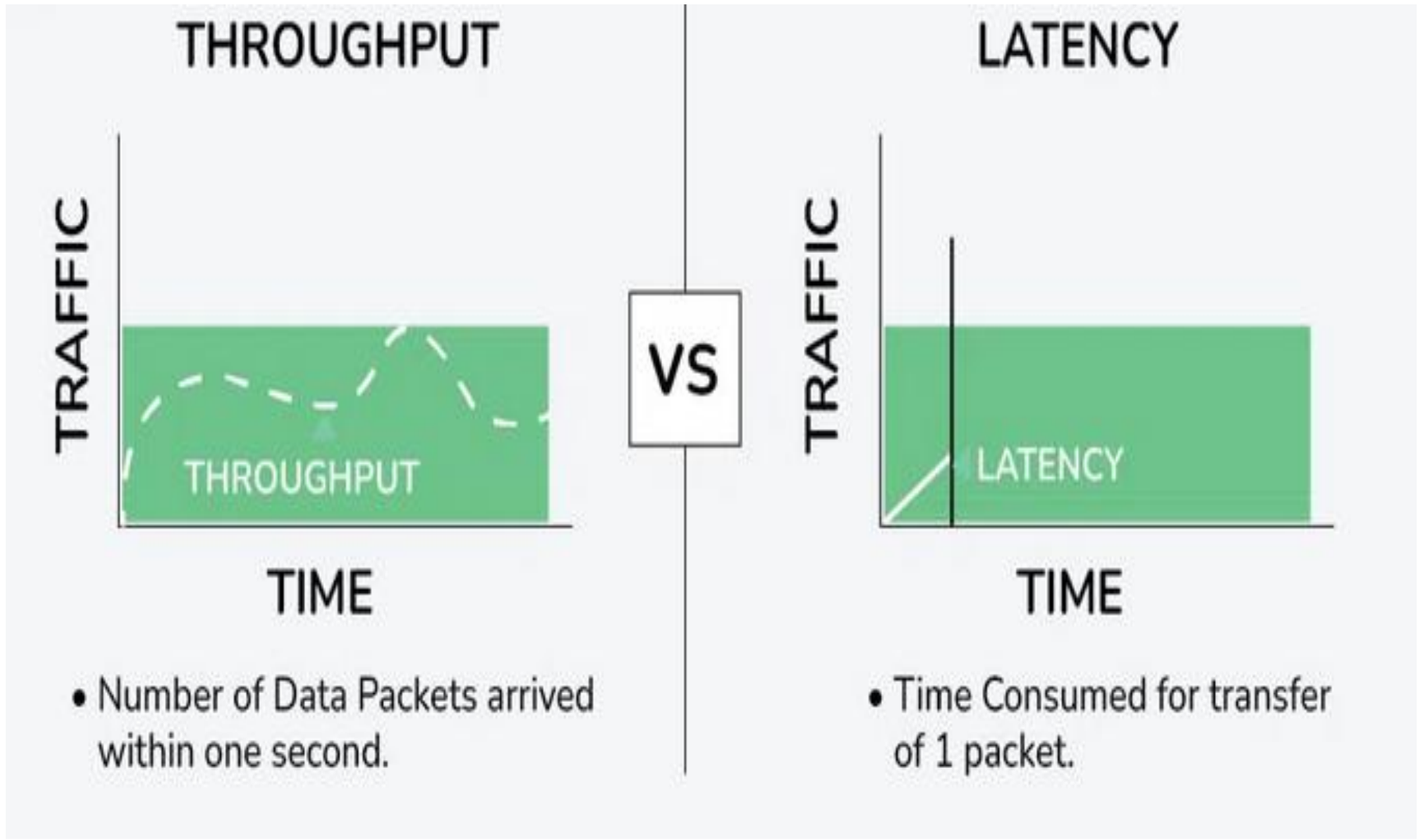
Throughput:

$$\frac{1000}{300 \text{ seconds}} = \boxed{3.33 \text{ orders/second}}$$

Problem: A NoSQL database handles 18,000 read operations in 60 seconds. Ans. 300 rps

Problem: You run a load test and observe the following log:
Total requests: 12,000
Test duration: 3 minutes
Average latency: 200 ms
Ans. 66.67 rps

Problem. A system processes 5000 files in 2 hours. Ans. 41.67 filesp minute


THROUGHPUT
TRAFFIC
THROUGHPUT
TIME
VS
LATENCY
TRAFFIC
LATENCY
TIME
• Number of Data Packets arrived within one second.
• Time Consumed for transfer of 1 packet.

Aspect	Throughput	Latency
Definition	The number of tasks completed in a given time period.	The time it takes for a single task to be completed.
Measurement Unit	Typically measured in operations per second or transactions per second.	Measured in time units such as milliseconds or seconds.
Relationship	Inversely related to latency. Higher throughput often corresponds to lower latency.	Inversely related to throughput. Lower latency often corresponds to higher throughput.
Example	A network with high throughput can transfer large amounts of data quickly.	Low latency in gaming means minimal delay between user input and on-screen action.
Impact on System	Reflects the overall system capacity and ability to handle multiple tasks simultaneously.	Reflects the responsiveness and perceived speed of the system from the user's perspective.

Factors affecting Throughput

•**Network Congestion:** High levels of traffic on a network can lead to congestion, reducing the available bandwidth and impacting throughput.
•**Bandwidth Limitations:** The maximum capacity of the network or communication channel can constrain throughput. Upgrading to higher bandwidth connections can address this limitation.
•**Hardware Performance:** The capabilities of routers, switches, and other networking equipment can influence throughput. Upgrading hardware or optimizing configurations may be necessary to improve performance.
•**Software Efficiency:** Inefficient software design or poorly optimized algorithms can contribute to reduced throughput.
•**Latency:** High latency can impact throughput, especially in applications where real-time data processing is crucial.

Methods to improve Throughput

1.Network Optimization: 1. Utilize efficient network protocols to minimize overhead.
2. Optimize routing algorithms to reduce latency and packet loss.
2.Load Balancing: 1. Distribute network traffic evenly across multiple servers or paths.
2. Prevents resource overutilization on specific nodes, improving overall throughput.
3.Hardware Upgrades: 1. Upgrade network devices, such as routers, switches, and NICs, to higher-performing models.
2. Ensure that servers and storage devices meet the demands of the workload.
4.Software Optimization: 1. Optimize algorithms and code to reduce processing time.
2. Minimize unnecessary computations and improve code efficiency.
5.Compression Techniques: 1. Use data compression to reduce the amount of data transmitted over the network.
2. Decreases the time required for data transfer, improving throughput.
6.Caching Strategies:
1. Implement caching mechanisms to store and retrieve frequently used data locally.
2. Reduces the need to fetch data from slower external sources, improving response times and throughput.

Consistency

When a system is said to be consistent, every time there is a write then a read, the returned value will always be the most recent written value. This ensures that the data is consistent from one node to another in the distributed environment.

•**Strong Consistency:** Ensures that if a write is totally done, from that point on any reads will result to the write. This is somewhat similar to the ACID properties that are found in the traditional database systems where any transaction, or set of operations, is required to be affirmed as whole and correct before other transactions can begin.

•**Eventual Consistency:** Guarantees always that if any two nodes are allowed enough time to make computations, they will in the long run have the same value, although in the short run, they can have dissimilar values. This is normally used in systems where accuracy is not as important compared to the availability and performance of the system.

•**Causal Consistency:** Makes sure that all nodes get to view operations that are related in terms of cause and effect at the same time. This is weaker than strong consistency but stronger than eventual consistency.

Availability

In a distributed system, availability gives every request a response even if it is a failed one. This is because, in an available system, all the working nodes have to be able to respond to queries and answer even if it is with wrong or old data. Some of the importance of availability include; The importance of availability cannot be over-emphasized when it comes to IT service delivery since some systems will require to be up and responsive always. Some types of availability include:

High Availability: Makes sure that the system is up and running for most of the time needs. High availability systems are those systems, which are intended to work even if they fail and usually work at this level through redundancy and fail-over techniques.

Partial Availability: It is not correct to consider that this system is either available or unavailable because some sections may be available while others are not depending on the failure situation. This can be observed in systems that support a degraded mode of operation.

Feature	Consistency	Availability
Definition	Ensures all nodes have the same data simultaneously.	Ensures every request receives a response.
Primary Goal	Data accuracy and integrity.	Service continuity and responsiveness.
Response Behavior	May delay responses to ensure data is up-to-date.	Always provides a response, even if data is stale.
Trade-offs	May sacrifice availability for data correctness.	May sacrifice consistency for higher uptime.
Typical Use Cases	Banking systems, transaction processing.	Web services, online applications.
CAP Theorem Focus	Consistency and Partition Tolerance.	Availability and Partition Tolerance.
Complexity	Higher complexity due to synchronization.	Lower complexity, easier to implement.
Failure Handling	May reject requests to ensure data consistency.	Always responds to requests, even during failures.

Benefits of Consistency

•**Data Integrity:** Checks the integrity of data in all the nodes that has been sent to it by other nodes. This is essential especially in an environment where the presence of wrong data, or outdated data may cause severe problems, for instance, in banking and or in medical records.

•**Reliability:** Avoids having irregularities in the data set. More so where data accuracy is critical, having the entire system in a consistent state all the nodes are guaranteed that the behavior of the system will be as expected.

•**Predictability:** Ensures that all read and write operations are easily determinable by a user. End users may always be assured of acquiring the latest data, thereby easing the computational concepts pertaining to data in an application.

Use Cases of Consistency

•**Banking Systems:** Where transaction processing is important. It is necessary to maintain proper records and intact security measures that address the problems such as double-spend or undesired balance.

•**Inventory Management:** To make certain that inventory is correctly recorded in other premises. For businesses that use multiple warehouses or Stores when it comes to inventory management consistency helps them maintain accurate stock status to evade such problems as overselling.

•**Distributed Databases:** When data correctness and entry standardization are critical. In systems where data is stored and retrieved at any of the nodes in the network, consistency means that all nodes mirror each other's state and important to prevent data inconsistencies and anomalies.

Benefits of Availability

Service Continuity: Maintains the system to function well at all times as expected. High availability is required for applications that have to be always available in the application space to the users regardless of failures or high loads.

User Experience: Works within the framework of user demands requirement, and assurance of timely feedback to users thereby improving the user satisfaction. Availability-oriented systems guarantee that the client is constantly able to use the application with no disruptions of any sort.

Resilience: Can work with one or more nodes of a cluster being unavailable or the communication between nodes being severed without affecting service. Making the system fault tolerant enhances the availability aspect of the software and makes the internet application more reliable.

Use Cases of Availability

E-commerce Websites: Where continuity as well as response time is an essential factor that determines the use of the application. As for the online retailing, availability directly affects a range of user activities such as, products and services' viewing, order placing or even payments making, thus influencing the sales and customers' satisfaction.

Social Media Platforms: That users can always upload and download material. Consumers of content shared via the social media do expect constant availability of this content, high availability is central in the sustaining of interest and overall satisfaction.

Content Delivery Networks (CDNs): Offering maximum access of information to the world population. CDNs provide the content in multiple servers globally to enable quick and accurate delivery to the user no matter the region as they rely on high availability to provide performance and reliability.

Modularity

The process of breaking down a complex system into smaller, more manageable components or modules is known as modularity in system design. Each module is designed to perform a certain task or function, and these modules work together to achieve the overall functionality of the system.

To enhance the system's flexibility and dependability.

For example: In an object-oriented programming language like Java, a module might be represented by a class, which defines the data and behavior of a particular type of object.

```
// Module 1: Addition module
public class AdditionModule {
    public static int add(int a, int b) { return a + b; }
}

// Module 2: Subtraction module
public class SubtractionModule {
    public static int subtract(int a, int b)
    {
        return a - b;
    }
}
```

```
// Module 3: Multiplication module
public class MultiplicationModule {
    public static int multiply(int a, int b)
    {
        return a * b;
    }
}

// Module 4: Division module
public class DivisionModule {
    public static double divide(int a, int b)
    {
        if (b != 0) {
            return (double)a / b;
        }
        else {
            System.out.println("Cannot divide by zero");
            return Double.NaN; // Not a Number
        }
    }
}

// Main program
public class Main {
    public static void main(String[] args)
    {
        int num1 = 10;
        int num2 = 5;

        // Using addition module
        int resultAdd = AdditionModule.add(num1, num2);
        System.out.println("Addition result: " + resultAdd);

        // Using subtraction module
        int resultSubtract
            = SubtractionModule.subtract(num1, num2);
        System.out.println("Subtraction result: "
                           + resultSubtract);

        // Using multiplication module
        int resultMultiply
            = MultiplicationModule.multiply(num1, num2);
        System.out.println("Multiplication result: "
                           + resultMultiply);

        // Using division module
        double resultDivide
            = DivisionModule.divide(num1, num2);
        System.out.println("Division result: "
                           + resultDivide);
    }
}
```

Examples

1. Ecommerce website
2. Patient monitoring system
3. Learning management system

Characteristics of Modularity

The key characteristics of modularity include:

Flexibility: Allows for easy customization and adaptation to changing requirements.

Abstraction: Modules provide clear, high-level interfaces abstracting complex functionality.

Collaboration: allows teams to operate independently on various modules, which promotes parallel development.

Testing: Modular systems are easier to test as each module can be tested separately, promoting robustness.

Documentation: Encourages better documentation practices as module interfaces need to be well-defined and documented.

Interchangeability: Modules can be swapped or upgraded without affecting the overall system functionality, promoting interoperability.

Key Components of Modular Design

Below are the key components of Modular Design:

Modules: These are the smaller, separate components that comprise the system as a whole. Every module is self-contained, has clearly defined interfaces to other modules, and is made to carry out a specific task.

Interfaces: These are where modules can communicate with one another. Interfaces, which can be software, mechanical, or electrical connections, specify how the modules communicate with one another.

Subsystems: These are groups of modules that work together to perform a specific function within the overall system.

Integration: This involves integrating the various modules to form an integrated unit and testing the system as a whole to make sure everything is operating as it should.

Maintenance: To make sure the system keeps functioning properly, this involves maintaining an eye on it and updating it as necessary. In some cases, this may involve changing or swapping out certain modules.

Documentation: This includes all of the technical and operational information about the system, including schematics, manuals, and instructions for use.

Benefits of Modularity

Some of the important benefits of modularity are:

•**Improved flexibility:** Modular designs allow individual components or modules to be easily added, removed, or replaced, making it easy to modify the product to meet changing needs or requirements.

•**Increased efficiency**: Modular designs enable different parts of the product to be developed and tested independently, allowing for faster development and more efficient use of resources.

•**Improved quality**: By enabling more extensive testing of individual parts and making it easier to employ better materials and building methods, modular designs can raise a product's overall quality.

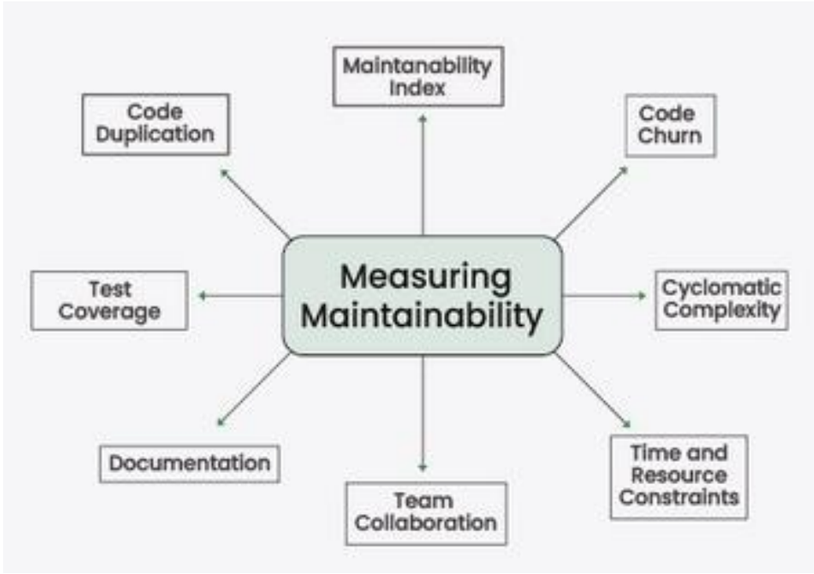
•**Enhanced** **scalability**: Because modules can be added or removed as needed, modular designs may ease the process of scaling a product up or down in terms of size, capacity, or capability.

Maintainability

Maintainability determines how easy and profitable it will be to maintain, update, and do upgrades in that software system. Along with Scalability, trust ability, and Security, Maintainability is also a pivotal factor of a system software.

A largely justifiable system contains the following characteristics:
Modularity: It's organized into different factors or modules, allowing maintainers to understand and modify individual pieces without affecting the rest of the system.
Readability: Its Codebase is clear, terse, and readable, making it easier for maintainers or other team members to understand and modify
Error Handling: It's designed to handle problems or issues effectively, furnishing meaningful error dispatches and avoiding disastrous failures.
Utilizes VCS: It Utilizes VCS similar to Git for effective shadowing of changes and making collaboration easier.
Testability: It's effective in testability and helps in relating implicit issues snappily.

Measuring Maintainability
While measuring maintainability is subjective, here are some metrics to measure it: -



Code Duplication: The percentage of duplicated code present in the system can be an indicator of maintainability. Code duplication occurs when the same or very similar code appears in multiple places within the system. In that case, changes made to one component of duplicated code may need to be applied to other areas too. Code duplication can be measured with static code analysis tools that identify duplicated code segments.

Maintainability Index: It provides an overall score that represents the maintainability of a specific component/Code module/Entire system. The formula for calculating the Maintainability Index varies depending upon the IDE you are using, it basically combines complexity, duplication, and other factors to produce a single value. Higher Maintainability Index scores indicate better maintainability and many people in tech use this metric to identify sections of the codebase that require enhancements.

Test Coverage: It measures the extent to which automated tests cover the codebase, It helps prevent regressions from being introduced. Test coverage tools assess the extent of code exercised by automated tests, expressed as a percentage.

Documentation: Good documentation reduces the learning curve for new Devs and helps the existing team understand it better during maintenance. A Good documentation is the one, which makes a newcomer understand the project and also covers not only code comments but also architectural decisions, System design, API references, as well.

Team Collaboration: A strong collaborative culture within the development team helps them share knowledge with each other, perform Knowledge Transfer programs (KT), mentor newcomers, and work together on maintenance tasks, it helps team members grow together and makes sure someone won't struggle in doing a particular task.

Time and Resource constraints: It's important to balance between speed of development as well as speed of maintainability. A rush by the development team to complete a tight deadline without considering maintainability can lead to complicated future maintenance efforts.

Code Churn: It measures the frequency of changes to a code module over time using Version Control's data. Active maintenance is essential for improvements and bug fixes, but excessive code churn may also indicate instability and the need for more thorough testing. By monitoring, focus areas of the system that may require more attention can be identified during maintenance.

Cyclomatic Complexity: It measures the complexity of a code module. Higher complexity is an indication that the code is more complicated to understand and there may be potential easier ways available to break the complexity of a particular code module.

Formula for Calculating Cyclomatic Complexity

Mathematically, for a structured program,

Case 1::: the directed graph inside the control flow is the edge joining two basic blocks of the program as control may pass from first to second.

So, **cyclomatic complexity M** would be defined as,

M = E - N + 2P

where

E = the number of edges in the control flow graph

N = the number of nodes in the control flow graph

P = the number of connected components

In case 2:::, when exit point is directly connected back to the entry point. Here, the graph is strongly connected, and cyclometric complexity is defined as

M = E - N + P

where

E = the number of edges in the control flow graph

N = the number of nodes in the control flow graph

P = the number of connected components

In the case 3::: of a single method, P is equal to 1. So, for a single subroutine, the formula can be defined as

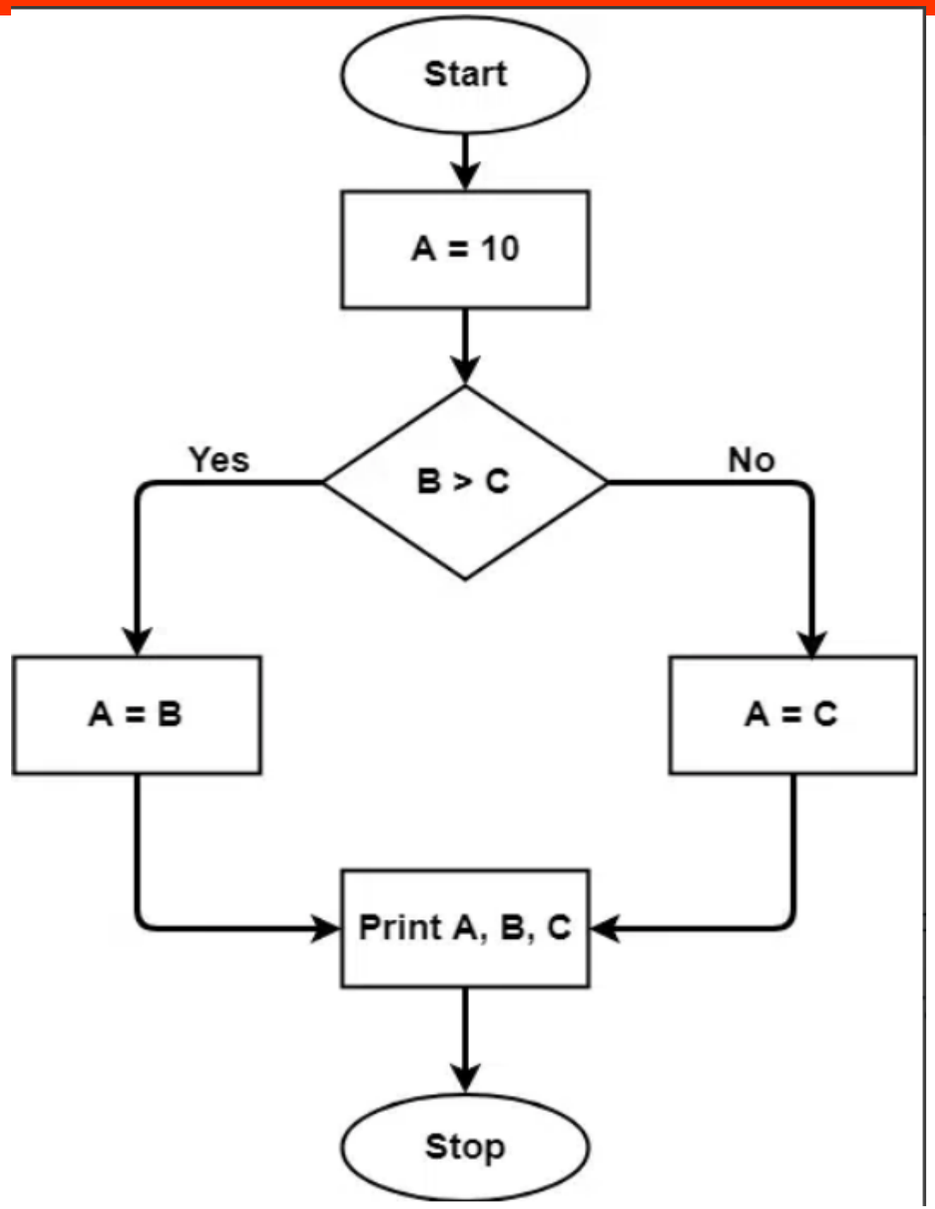
M = E - N + 2

where

E = the number of edges in the control flow graph

N = the number of nodes in the control flow graph

P = the number of connected components



M=7-7+2=2

NOTE: if the source code contains no control flow statement then its cyclomatic complexity will be 1, and the source code contains a single path in it. Similarly, if the source code contains one if condition then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.

How to achieve high Maintainability
Designing a highly maintainable system requires a proactive approach during the development process. Here are some strategies to achieve and improve maintainability in system design: -

1. Follow Design Patterns:
Design patterns, such as Model-View-Controller and SOLID principles, promote modularity, and flexibility. Following these patterns enhances overall architecture while making it easier to maintain.

2. Code Consistency:
Well-written, readable code with meaningful variables, comments, and documentation makes maintenance tasks even smoother. So, as much as possible while writing code or documentation make it simple in design and implementation.

3. Conduct Code Reviews:
Code reviews by peers, and maintainers can help in identifying potential issues early on, while making sure that code adheres to maintainability standards and preferred code style or not.

4. Test-Driven Development (TDD):
Adopting TDD ensures that test cases are written before the code implementation, helping developers understand the codebase easily while troubleshooting an issue.

5. Documentation:
Good documentation reduces the learning curve for new Devs and helps the existing team understand it better during maintenance. Maintain thorough documentation of the system's architecture, APIs, modules, and dependencies.

6. Plan for Change:
Design the system in such a way, that makes it easier to add new features and adapt to evolving requirements of daily enhancements.

7. Automate Testing and Deployment:
Automating testing and deployment processes reduces the chance of human error helps maintain the stability of the system during updates and also saves time and effort of team members, allowing them to work on important issues.

Clean Code Principles

- Clean code is more than just correct code — it's readable, maintainable, and elegant.
- Prioritizes clarity so that others (and your future self) can understand it easily.
- Avoids unnecessary complexity; focuses on doing one thing well.
- Uses consistent formatting, meaningful naming, and modular design.
- Clean code is structured to accommodate changes with minimal risk.
- Aligns with industry conventions like SOLID principles, DRY, and KISS.
- Easier for teams to work on, debug, and extend collaboratively.
- Well-written code often reduces the need for excessive comments.
- Demonstrates a developer’s discipline, thought process, and respect for the next coder.

Why Is Clean Code Important?

Enhanced Maintainability: Clean code is inherently easier to maintain and update, reducing the time and resources needed for future modifications.

Improved Readability: Clean code is easy for new developers to understand and contribute to a project without a steep learning curve.

Increased Efficiency: Well-organized codebases are less prone to bugs and issues, facilitating a smoother development process and quicker troubleshooting.

Better Collaboration: Clean code principles encourage practices that make collaborative work more seamless, allowing teams to work more effectively together.

Principles

- **Meaningful Names:** Use descriptive, specific names for variables, functions, and classes that clearly convey their purpose.
- **Keep It Simple, Stupid (KISS):** Avoid unnecessary complexity. Aim for simplicity in your solutions.
- **Don’t Repeat Yourself (DRY):** Minimize duplication in your codebase to ensure that every piece of knowledge has a single, unambiguous representation.
- **Single Responsibility Principle (SRP):** Each module, class, or function should have one reason to change, encapsulating a single responsibility.
- **Readability Over Cleverness:** Code should be straightforward and easy to read rather than overly clever or complex.

How to Write Clean Code

Writing clean code is an iterative process that involves constant refinement and adherence to best practices. Following points must be kept in mind:

Refactor Regularly: Review and refine your code to improve its structure and readability.

Follow Style Guidelines: Adhere to your language's style guidelines and conventions for formatting and structuring your code.

Code Reviews: Participate in code reviews to receive feedback and learn from others' approaches to problem-solving.

To further embed clean code principles in development process, consider these best practices:

Write Unit Tests: Ensure your code is testable and covered by unit tests to maintain functionality and prevent regressions.

Document Thoughtfully: While clean code should be self-explanatory, judicious use of comments can clarify complex logic or decisions.

Use Version Control Wisely: Leverage version control systems to manage changes and collaborate more effectively with others.

Tips to Write Cleaner Code

Enhancing the cleanliness of your code is a continuous journey. Here are some tips to guide you along the way:

Start Small: Focus on improving one aspect of your code at a time.

Learn from the Masters: Study code from experienced developers and open-source projects to see clean code principles.

Practice, Practice, Practice: Like any skill, writing clean code gets easier with practice. Challenge yourself with new problems and projects.

Clean code principles

Clean code principles in **system design** go beyond just writing readable code—they focus on building systems that are **maintainable, scalable, testable, and easy to understand** at a high level.

1. Single Responsibility Principle (SRP)

Each component, module, or service should have one and only one reason to change.

In code: Each class or function does one thing.

In system design: Each service or microservice handles a single domain (e.g., Auth Service, Payment Service).

2. Separation of Concerns (SoC)

Different parts of the system should handle distinct concerns.

•UI Layer, Business Logic, Data Access should be separated.

•Example: MVC (Model-View-Controller), layered architectures.

3. Modularity

Design your system in independent, interchangeable modules.

Microservices, plug-ins, or independently deployable packages.

Encourages reuse, parallel development, and easier testing.

4. DRY (Don't Repeat Yourself)

Avoid code and logic duplication.

Abstract common logic into shared services or libraries.

Don't repeat logic across services—use shared contracts or service calls.

5. High Cohesion & Low Coupling
High cohesion: Related logic is grouped together.
Low coupling: Components should have minimal dependencies

6. Well-Defined Interfaces & APIs
Define clear contracts between components or services.
Use API versioning and documentation (e.g., OpenAPI/Swagger).

7. Defensive Design
Validate all inputs and handle failures gracefully.
Fail fast when necessary but degrade gracefully.
8. Observability
•Design for **logging**, **monitoring**, **metrics**, and **tracing**.
•Use tools like Prometheus, Grafana, OpenTelemetry, etc.

9. Consistency
Use consistent naming, patterns, and technologies across the system.
Follow shared conventions for configuration, error handling, etc.

10. Scalability & Performance Awareness
Design components to be scalable from the start (e.g., stateless services, caching layers).
Use asynchronous messaging, load balancing, and horizontal scaling where needed

11. Testability
Design components that can be unit tested and integration tested independently.
Use mocking, dependency injection, and test doubles where applicable.

Principle	Benefit
SRP	Easier maintenance
Separation of Concerns	Better modularity
Modularity	Scalability and testability
DRY	Reduces redundancy
High Cohesion, Low Coupling	Flexibility and resilience
Defined Interfaces	Reliable communication
Defensive Design	Fault tolerance
Observability	Easier debugging
Consistency	Developer friendliness
Scalability Awareness	Performance-ready systems
Testability	Reliable code changes
Documentation	Team clarity

CI/CD Pipeline

- CI/CD is a DevOps strategy that automates software development and deployment.
- Integrates and automates every stage—from coding to testing to deployment.
- **Continuous Integration (CI)**:

➢ Developers frequently merge code into a shared repository.

➢ Automated tests run on each change to ensure quality and detect issues early.

- **Continuous Delivery (CD)**:

➢ Automates application release processes.

➢ Ensures software can be deployed to production at any time with minimal manual intervention.

- Teams work on a single codebase with automated pipelines for consistency.

Continuous Integration

CI automates the build and testing process whenever developers commit code to version control.

Shared Repository: All developers merge changes—big or small—into a central branch regularly (main/trunk).

Automated Pipeline:

- Triggered with every commit.
- Builds the latest code, runs tests, and validates integrity.

Benefits of Continuous Integration

- **Smaller and Easier Code Changes:** With every code change being pushed to version control immediately, the CI/CD pipeline has to deal with smaller code changes and integrations at a time.
- **Easier Debugging**
  A CI-based pipeline facilitates fault isolation AKA the practice of formulating systems in which errors lead to limited negative consequences.
- **Faster Product Releases**
  A CI-powered pipeline is a continuously moving system in which failures are detected and debugged faster.
- **Lighter Backlog**
  As explained above, CI enables quicker bug identification and debugging, all within the early testing stages of code changes and integration.
- **Increased Transparency and Accountability**
  Frequent code commits lead to immediate and frequent feedback from the automated system as well as the team.

Continuous Deployment

- Continuous Deployment is the next step after Continuous Delivery—automatically releases code to production.
- **Fully Automated Release**:
  - ➢ No manual approvals needed.
  - ➢ Code is deployed directly once it passes all pipeline checks.
- **Test-Driven Automation**:
  - ➢ Deployment depends on a series of pre-defined, automated tests.
  - ➢ Ensures only stable, tested code reaches production.
- **Zero Human Intervention**:
  - ➢ Eliminates delays from manual gatekeeping.
  - ➢ Increases release speed and reliability.

Benefits of Continuous Deployment

- Rapid delivery of new features and fixes.
- Reduced time-to-market.
- Consistent and repeatable deployments.
- Immediate user feedback from live environments.
- Best Fit For:

➢ Teams with mature testing practices.

➢ Products requiring frequent updates or continuous innovation.

Continuous Delivery vs Continuous Deployment

Aspect	Continuous Delivery	Continuous Deployment
Automation Level	Code is **built, tested, and made ready** for production	Code is **automatically released** to production
Approval Required	**Manual approval** (by Dev, PM, or Team Lead) is required	**No manual approval** needed; fully automated
Deployment Trigger	Release is **manually triggered** after validation	Deployment is **triggered automatically** post-validation
Use Case	Suitable when **control and oversight** are priorities	Suitable when **speed and rapid feedback** are essential
Tooling & Maturity	Needs **moderate automation and checks**	Requires **mature CI/CD pipeline with robust testing and rollback**
Risk Management	Easier to manage risks via staged rollouts	Requires strong **rollback & monitoring** strategies

Pipeline Flow:

1. **Code Commit** – Developer pushes code to the repository.
2. **Build** – Code is compiled and packaged automatically.
3. **Integration** – Merged into shared branch, resolving conflicts.
4. **Automated Testing** – Unit, integration, and UI tests run.
5. **Deployment** – Code is deployed to staging or production.
6. **Monitoring** – Performance and errors are continuously tracked.

CI/CD Tools

- **Configuration Management** - Ansible, Puppet, Chef
- **Code Management** - GitHub, GitLab, BitBucket
- **Build** - Jenkins, Bamboo, TeamCity
- **Testing** - Selenium, JUnit, SonarQube
- **Deployment** - Argo, Spinnaker, Octopus Deploy

Importance of CI/CD in Modern Software Development
Continuous Integration and Continuous Deployment (CI/CD) are vital in modern software development for several reasons:

Faster Time to Market
Quick Iterations: CI/CD allows for rapid iteration of software, enabling new features and updates to reach users faster.
Automated Processes: Automation reduces the time needed for manual testing and deployment, speeding up the development cycle.

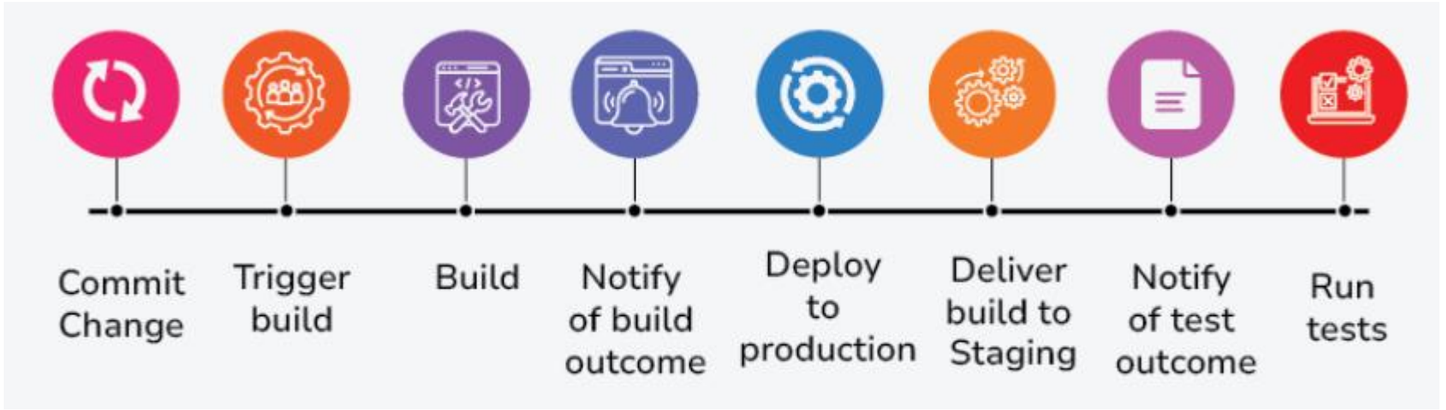
Improved Code Quality:
Automated Testing: Continuous testing ensures that new code changes do not introduce bugs or break existing functionality.
Consistent Integration: Regular integration of code helps in identifying and resolving conflicts early.

Increased Collaboration and Efficiency:
Shared Codebase: Developers work on a single shared codebase, improving collaboration and reducing integration issues.
Feedback Loops: Automated feedback on code changes helps developers fix issues quickly, enhancing overall productivity.

Reduced Risks:
Early Bug Detection: Frequent integration and testing catch bugs early in the development cycle, reducing the risk of critical failures in production.
Rollback Capability: Automated deployments often include easy rollback options, minimizing the impact of any issues that do make it to production.

Consistent and Reliable Releases:
Repeatable Processes: Automation ensures that the process of building, testing, and deploying software is consistent every time.
Reduced Human Error: Automation minimizes manual intervention, reducing the likelihood of errors.

Scalability and Flexibility:
Handling Load: CI/CD pipelines can handle multiple builds and deployments simultaneously, making it easier to scale development efforts.
Adapting to Changes: The flexibility of CI/CD pipelines allows teams to adapt quickly to changes in requirements or technology.



Components of CI/CD Pipelines

Steps for Designing a CI/CD Pipeline

Designing a CI/CD pipeline involves several key steps to ensure a smooth and efficient workflow from code commit to deployment. Here are the steps to design an effective CI/CD pipeline:

Step 1: Assess Current Development Process

- **Understand Current Workflow:** Map out your existing development, testing, and deployment processes.
- **Identify Bottlenecks:** Identify areas where automation can save time or reduce errors.
- **Set Objectives:** Define the goals of your CI/CD pipeline (e.g., faster releases, improved quality, reduced manual intervention).

Step 2: Choose Tools and Technologies

- **Version Control System (VCS):** Select a VCS like Git, GitHub, GitLab, or Bitbucket.
- **CI/CD Tools:** Choose CI/CD tools that fit your needs. Common options include Jenkins, GitLab CI, CircleCI, Travis CI, and Azure DevOps.
- **Build Tools:** Select build tools appropriate for your technology stack (e.g., Maven, Gradle, npm).
- **Testing Frameworks:** Choose frameworks for automated testing (e.g., JUnit, Selenium, pytest).

Step 3: Define Pipeline Stages

- **Commit Stage:** Ensure code changes trigger the pipeline automatically.
- **Build Stage:** Configure the build process to compile code and package artifacts.
- **Test Stage:** Automate running of unit tests, integration tests, and other relevant tests.
- **Staging Deployment:** Deploy to a staging environment for further validation.
- **Production Deployment:** Define the process for deploying to the production environment.

Step 4: Implement Automated Testing

- **Unit Tests:** Ensure every code change passes a suite of unit tests.
- **Integration Tests:** Validate the interaction between different modules.
- **End-to-End Tests:** Test the application from a user perspective.
- **Performance Tests:** Ensure the application meets performance requirements.

Step 5: Set Up Continuous Integration
Automate Builds: Set up the CI tool to automatically build the code upon each commit.
Run Tests: Integrate automated tests into the CI process to run with every build.
Provide Feedback: Ensure the CI tool provides immediate feedback to developers on build and test results.

Step 6: Implement Continuous Delivery
Automate Deployments to Staging: Configure automated deployment to a staging environment after successful builds and tests.
Manual or Automated Approvals: Define whether deployments to production require manual approval or are automated.

Step 7: Implement Continuous Deployment (Optional)
Automate Production Deployments: If desired, automate the deployment process to production, ensuring all tests pass and conditions are met.

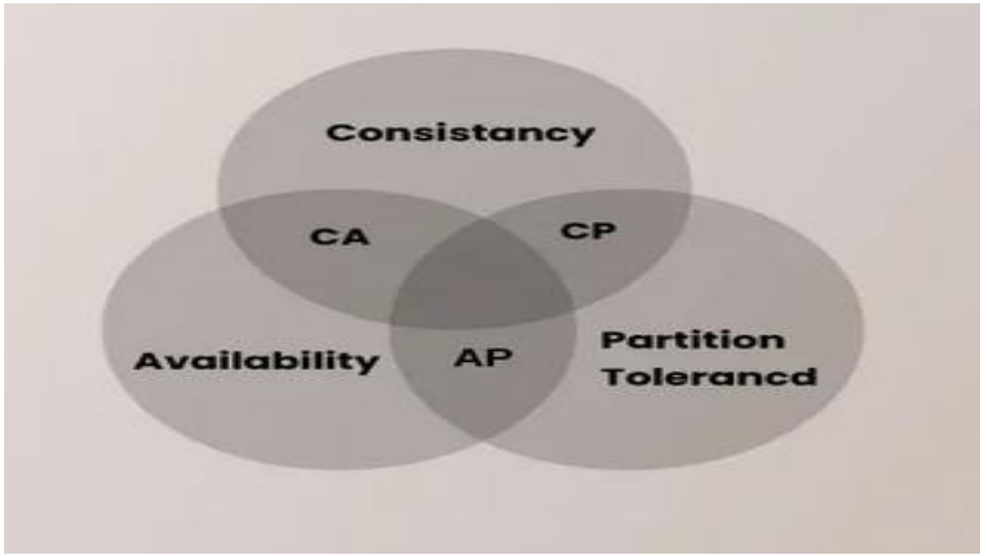
Step 8: Monitor and Improve
Logging and Monitoring: Set up logging and monitoring to track the health and performance of the pipeline and deployed applications.
Feedback Loops: Collect feedback from developers and stakeholders to continuously improve the pipeline.
Iterate and Optimize: Regularly review and optimize the CI/CD process for efficiency and effectiveness.

Step 9: Security and Compliance
Security Scans: Integrate security checks into the pipeline to identify vulnerabilities.
Compliance Checks: Ensure the pipeline enforces compliance with industry standards and regulations.

Step 10: Documentation and Training
Document Processes: Create clear documentation for the pipeline process, including setup, usage, and troubleshooting.
Training: Provide training for the development team on how to use the CI/CD pipeline effectively.

CAP Theorem

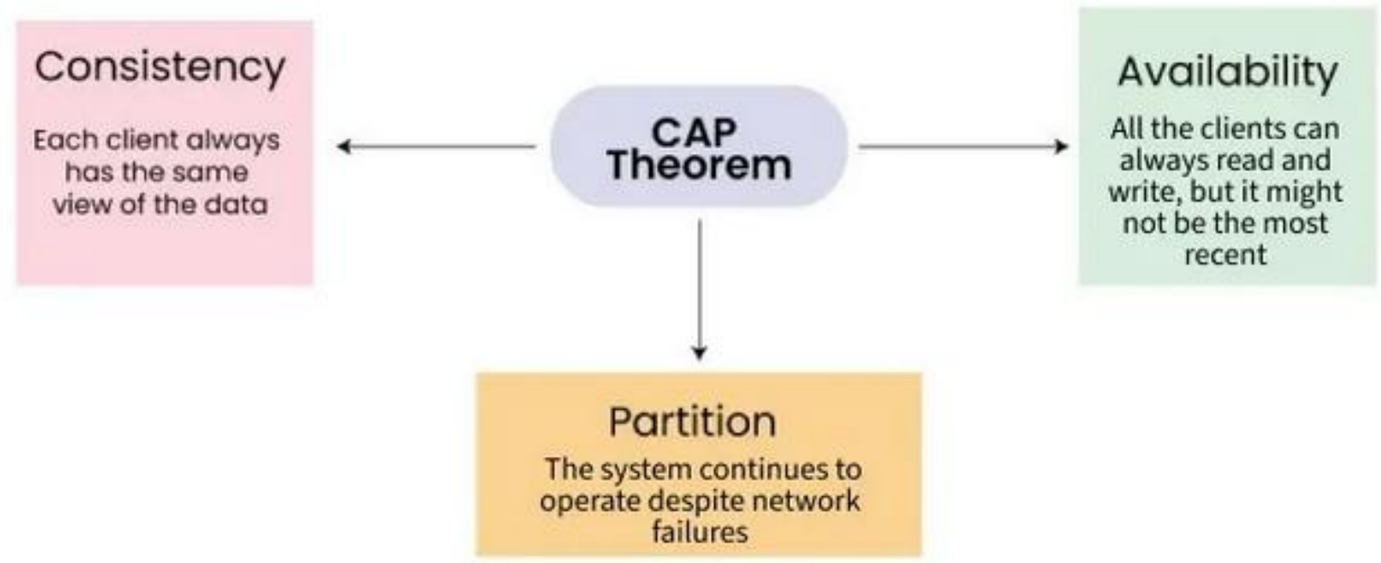
The CAP Theorem explains the trade-offs in distributed systems. It states that a system can only guarantee two of three properties: Consistency, Availability, and Partition Tolerance. This means no system can do it all, so designers must make smart choices based on their needs.



According to the CAP theorem, only two of the three desirable database characteristics—**consistency**, **availability,** and **partition tolerance**—can be shared or present in a networked shared-data system or distributed system.

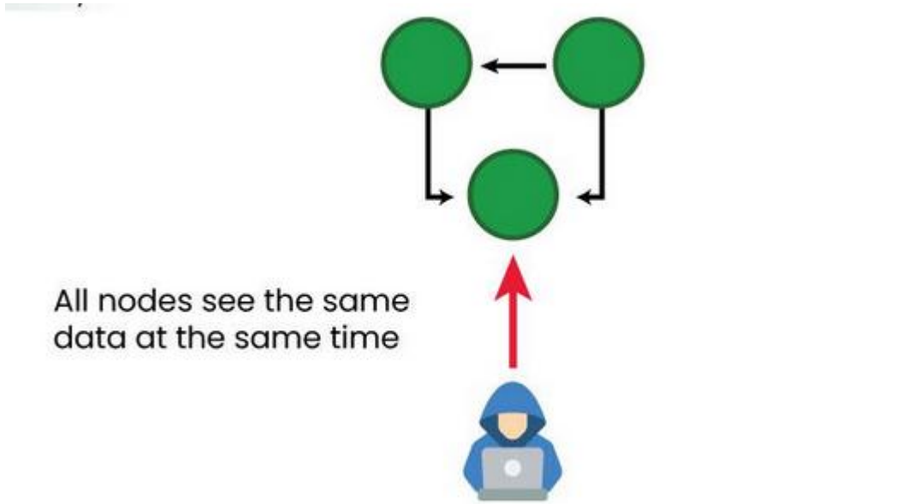
1.Consistency (C)− Every read receives the most recent write or an error.
2.Availability (A)− Every request receives a response, even if it is not the most recent data.
3.Partition Tolerance (P)− The system continues to function despite network partitions.

The computer scientist Eric Brewer presented the CAP Theorem, also called Brewer's theorem, at the Symposium on Principles of Distributed Computing in 2000.
The theorem provides a way of thinking about the trade-offs involved in designing and building distributed systems.
It helps to explain why certain types of systems may be more appropriate for certain use cases.
According to Brewer, the theorem states that a distributed system can have at most two of these guarantees.



Properties of CAP theorem

1. **Consistency:** Consistency defines that all clients see the same data simultaneously, no matter which node they connect to in a distributed system. For eventual consistency, the guarantees are a bit loose. Eventual consistency gurantee means client will eventually see the same data on all the nodes at some point of time in the future.



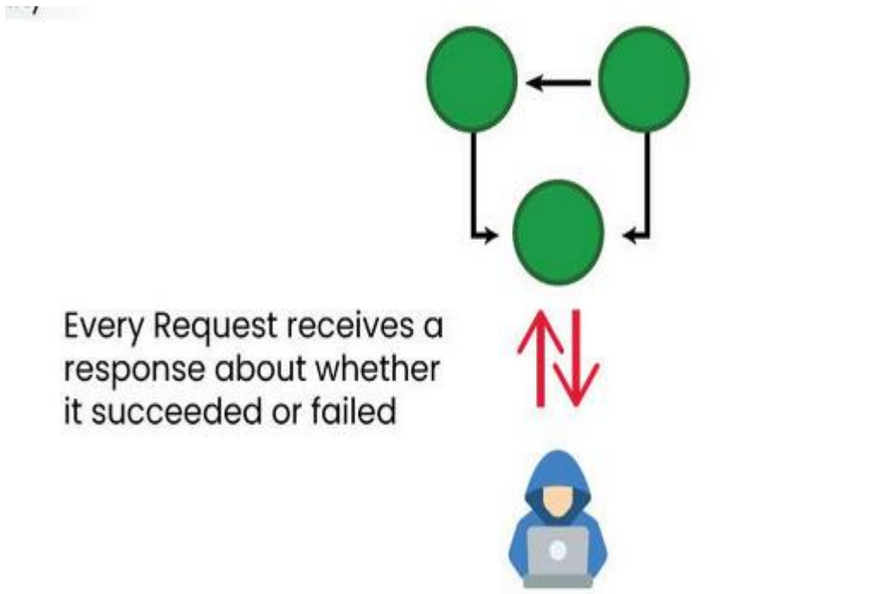
Example: If one node updates a value, other nodes in the system should reflect that updated value immediately.

Example− Traditional relational databases like MySQL prioritize consistency.

explaination of the above Diagram:

- All nodes in the system see the same data at the same time. This is because the nodes are constantly communicating with each other and sharing updates.
- Any changes made to the data on one node are immediately propagated to all other nodes, ensuring that everyone has the same up-to-date information.

2. Availability: Availability defines that all non-failing nodes in a distributed system return a response for all read and write requests in a bounded amount of time, even if one or more other nodes are down.

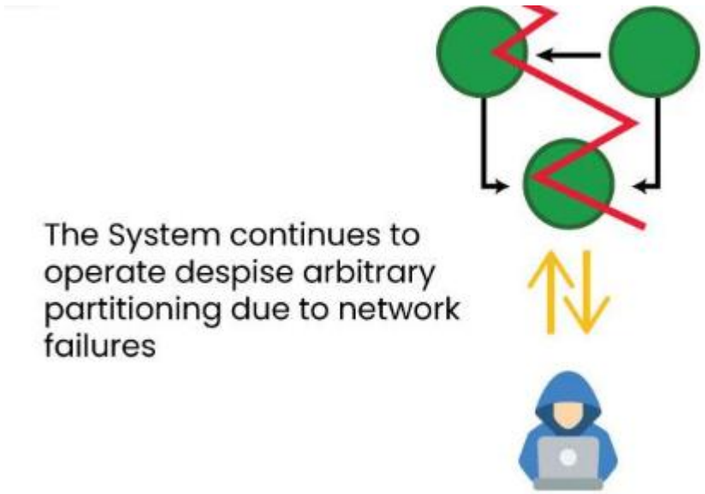


Example– Systems like DNS ensure high availability even at the expense of providing slightly outdated data.

Explanation of the above Diagram:

- User send requests, even though we don't see specific network components. This implies that the system is available and functioning.
- Every request receives a response, whether successful or not. This is a crucial aspect of availability, as it guarantees that users always get feedback.

3. Partition Tolerance: Partition Tolerance defines that the system continues to operate despite arbitrary message loss or failure in parts of the system. Distributed systems guranteeing partition tolerance can gracefuly recover from partitions once the partition heals.



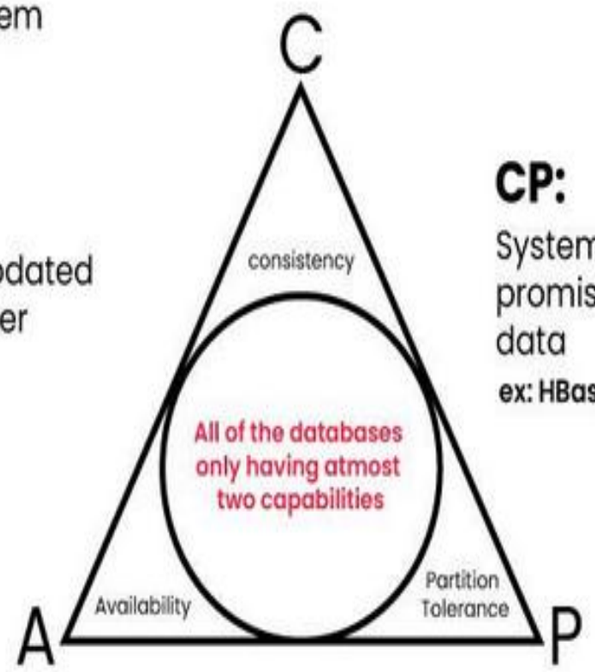
Example– Modern distributed databases like Cassandra and MongoDB prioritize partition tolerance.

Explanation of the above Diagram:

- Addresses network failures, a common cause of partitions. It suggests that the system is designed to function even when parts of the network become unreachable.
- The system can adapt to arbitrary partitioning, meaning it can handle unpredictable network failures without complete failure.

Trade-off in the CAP Theorem

CA:
system respond last updated data and promise higher availability
ex:RDBMS, PostgreSQL, etc



CP:
System can be distributed and promise to respond last updated data
ex: HBase, MongoDB, Redis

AP: System can be ditributed and promise to has high availability
ex: Cassandra, CouchDB, Riak, Voldemort, DynamoDB, etc

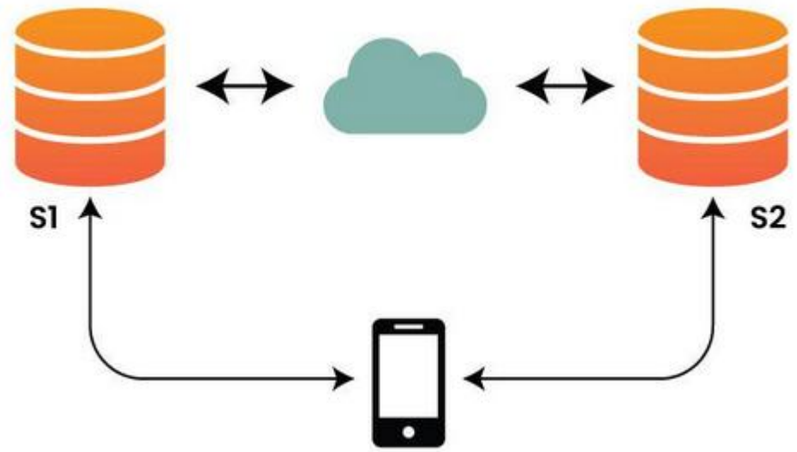
1. CA System
A CA System delivers consistency and availiability across all the nodes. It can't do this if there is a partition between any two nodes in the system and therefore does't support partition tolerance.

2. CP System
A CP System delivers consistency and partition tolerance at the expense of availability. When a partition occurs between two nodes, the systems shuts down the non-available node until the partition is resolved. Some of the examples of the databases are MongoDB, Redis, and HBase.

3. AP System
An AP System availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remains available, but those at the wrong end of a partition might return an older version of data than others. Example: CouchDB, Cassandra and Dyanmo DB, etc.

IMPLICATIONS::
CP (**C**onsistency, **P**artition tolerance) Systems− Focus on consistency and partition tolerance but sacrifice availability.
AP (**A**vailability, **P**artition tolerance) Systems− Focus on availability and partition tolerance but sacrifice consistency.
CA (**C**onsistency, **A**vailability) Systems− Focus on consistency and availability but cannot handle network partitions.

EXAMPLE



We have a simple distributed system where S1 and S2 are two server. The two server can talk to each other. Here, System is partition tolerant. Here We will prove that system can be either consistent or available.

- Suppose there is a network failure and S1 and S2 cannot talk to each other. Now assume that the client makes a write to S1. The client then send a read to S2.
- Given S1 and S2 cannot talk, they have different view of the data. If the system has to remain consistent, it must deny the request and thus give up on availability.
- If the system is available, then the system has to give up on consistency. This proves the CAP Theorem.

1. Banking Transactions (CP System)
Problem Statement:
Imagine a bank teller updating your account balance on a secure computer system. This system prioritizes consistency (C) and partition tolerance (P).

2. Social Media Newsfeed (AP System)
Problem Statement:
Think of your newsfeed on a social media platform constantly updating with new posts and stories. This system prioritizes **availability (A)** and **partition tolerance (P)**.

Advantages of CAP Theorem in System Design

1. Provides a Framework for Decision-Making
2. Promotes Understanding of Trade-offs
3. Guides System Architecture and Technology Selection
4. Enhances System Resilience and Performance

Disadvantages of CAP Theorem in System Design

1. Oversimplification
2. Abstract Trade-offs
3. Lack of Guidance for Hybrid Systems
4. Potential Misinterpretation

CONSISTENCY PATTERN

Consistency in system design refers to the property of ensuring that all nodes in a distributed system have the same view of the data at any given point in time, despite possible concurrent operations and network delays. In simpler terms, it means that when multiple clients access or modify the same data concurrently, they all see a consistent state of that data.

Importance of Consistency in System Design

1. Correctness
2. Trust and Reliability
3. Data Integrity
4. Concurrency Control
5. User Experience
6. Predictability

For example, if a bank account has a balance of $500, two users accessing the account from different locations should see the same balance.

Consistency in CAP

In systems prioritizing consistency (C), every read operation reflects the most recent write. However, this may come at the cost of availability during network partitions.

Types of Consistency Models

Consistency models define how and when updates to a distributed system become visible to users. They include−

Strong Consistency

Ensures that all clients see the most recent data after a write operation.

Example− Traditional relational databases like MySQL.

Eventual Consistency
Guarantees that all nodes will converge to the same state eventually, though not immediately.
Example– DNS systems or NoSQL databases like Cassandra.

Causal Consistency
Guarantees that causally related operations are seen by all nodes in the same order.
Example– Collaborative editing tools.

Read-Your-Writes Consistency
Ensures that after a user writes data, they will see their own updates in subsequent reads.
Example– User profile updates on social media.

Monotonic Consistency
1. ensures that if a client observes a particular order of updates (reads or writes) to a data item, it will never observe a conflicting order of updates.
2. prevents the system from reverting to previous states or seeing inconsistent sequences of updates, which helps maintain data integrity and coherence.

Monotonic Reads and Writes

1. ensure that if a client performs a sequence of reads or writes, it will observe a monotonically increasing sequence of values or updates.
2. ensure that clients never see older values in subsequent reads, while monotonic writes guarantee that writes from a single client are applied in the same order on all replicas.

Challenges with maintaining Consistency

Network Latency− Delays in communication between nodes can lead to stale data.
Partitioning− Network partitions can disrupt data synchronization.
Concurrency− Concurrent updates from multiple clients can lead to conflicts.
Scalability− Ensuring consistency across a large number of nodes can reduce system performance.
Cost− Consistency mechanisms, such as consensus algorithms, can be resource-intensive.

Techniques for Ensuring Consistency
Two-Phase Commit (2PC)
A protocol for coordinating transactions across multiple nodes−
Prepare Phase− Nodes vote on whether to commit the transaction.
Commit Phase− If all nodes agree, the transaction is committed.
Drawback− 2PC can block nodes during failures, impacting availability.

Quorum-Based Replication
In this method−
A quorum is a majority of nodes required to acknowledge a write or read.
Example−
Write Quorum− Data is written to W nodes.
Read Quorum− Data is read from R nodes.
Ensures consistency if R+W > NR + W > N (total nodes).

Consensus Algorithms
These algorithms help nodes agree on a single source of truth.
Paxos− Guarantees consensus in the presence of node failures.
Raft− Simpler and easier-to-understand alternative to Paxos.

Conflict Resolution Strategies
When concurrent updates conflict, strategies include−
Last Write Wins (LWW)− The most recent update overwrites older ones.
Custom Application Logic− Application-specific rules for resolving conflicts.
Vector Clocks− Track causality to identify conflicts.

Trade-offs in Consistency
Designing for consistency often involves trade-offs with other system properties−

Consistency vs. Availability
Systems prioritizing availability may sacrifice consistency during failures (e.g., eventual consistency).

Consistency vs. Performance
Strong consistency requires synchronization across nodes, increasing latency.

Consistency vs. Scalability
Maintaining consistency in highly scalable systems can lead to bottlenecks.

Real-World Examples

Example 1: Banking Systems
Strong consistency ensures that a user's bank balance is always accurate, even with concurrent transactions.
Techniques− Distributed transactions, 2PC.

Example 2: Social Media Platforms
Eventual consistency is often sufficient for non-critical operations like post visibility.
Techniques− Replication with conflict resolution.

Example 3: E-Commerce Websites
Product inventory must maintain strong consistency to prevent overselling.
Techniques− Consensus algorithms or locking mechanisms.

Differences

Aspect	Weak Consistency	Eventual Consistency
Synchronization	Lack of strict synchronization between replicas	Asynchronous replication of updates
Convergence Guarantee	No guarantee of convergence or synchronization	Guarantees eventual convergence of replicas
Divergence Tolerance	Allows significant divergence between replicas	Tolerates temporary divergence between replicas
Timing of Convergence	No specification on timing of convergence	Ensures eventual convergence but does not specify time
Ordering of Updates	No strict requirements on ordering of updates	Allows loose ordering of updates
Use Cases	Frequently used in caching systems and certain non-critical data scenarios	Commonly employed in distributed databases, cloud storage, and CDN
Impact on Performance	Offers potential performance benefits due to reduced synchronization overhead	May introduce temporary inconsistencies but provides eventual consistency and scalability benefits
Aspect	Weak Consistency	Eventual Consistency

Differences

Parameters	Strong Consistency	Eventual Consistency
Definition	Guarantees that all reads reflect the most recent write	Ensures that all replicas converge to the same value eventually
Data Freshness	Immediate consistency after a write	Temporary inconsistencies allowed, and eventual consistency
Latency	Higher latency due to synchronization	Lower latency due to asynchronous updates
Availability	Lower availability during network partitions (CAP theorem)	Higher availability even during network partitions (CAP theorem)
Partition Tolerance	Can be compromised for consistency	Prioritized alongside availability
Complexity	More complex to implement due to synchronization	Simpler implementation, fewer synchronization requirements
Use Cases	Financial transactions, inventory management, session management	Social media feeds, DNS, caching systems
Performance	Potentially slower due to synchronization overhead	Generally faster due to relaxed consistency
Scalability	More challenging to scale due to synchronization needs	Easier to scale across multiple nodes
Read/Write Operations	Synchronous, ensuring the latest data is read	Asynchronous, allowing for faster operations but with potential delays in consistency