BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUTE OF TECHNOLOGY

FACULITY OF ELECTRICAL AND COMPUTER ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

INTRODUCTION TO DISTRIBUTED SYSTEM

Individual Assignment

	NAME	*ID*
1.	*Yordanos Abebe*	*BDU1308350*

Submission Date: 16/05/2017E.C

Submited to:Ms.Temsgen

Cloud Infrastructure Provider: Google Cloud Platform (GCP)

How GCP Works:
GCP *is a suite of cloud computing services offered by Google. It provides a wide range of tools for data storage, computing, data analytics, and machine learning. GCP leverages Google's cutting-edge infrastructure, including its global fiber optic network and powerful AI/ML capabilities.*

GCP Services:
1. ***Compute Engine:*** *Provides virtual machines (VMs) for running various operating systems and applications.*

2. ***Google Kubernetes Engine (GKE):*** *A managed Kubernetes service for deploying and managing containerized applications.*

3. ***Cloud Storage:*** *Offers various storage options, including object storage, file storage, and cloud SQL.*

4. ***Cloud Functions:*** *A serverless compute platform for executing code in response to events.*

5. ***Cloud SQL:*** *A managed relational database service supporting various database engines (e.g., MySQL, PostgreSQL).*

6. ***Cloud Spanner:*** *A globally distributed, scalable, and strongly consistent database.*

7. ***Cloud Pub/Sub:*** *A real-time messaging service for streaming data between applications.*

8. ***Cloud Dataflow:*** *A fully managed service for batch and stream data processing.*

9. ***Cloud Vision API:*** *A powerful image analysis API for tasks like object detection, image classification, and optical character recognition.*

*10.****Cloud Natural Language API:*** *An API for understanding and analyzing human language, including sentiment analysis, entity recognition, and text summarization.*

Designing, Implementing, and Deploying Distributed Systems on GCP:

1. Define Requirements and Objectives:
- ***Scalability:*** *Determine the system's ability to handle increasing workloads and data volumes.*
- ***Availability:*** *Define the required uptime and fault tolerance mechanisms.*
- ***Performance:*** *Establish desired response times and latency levels.*
- ***Data Consistency:*** *Determine the level of data consistency required (e.g., strong, eventual).*
- ***Security:*** *Define security requirements, including data encryption, access control, and threat protection.*

2. Choose an Architectural Style:
- ***Microservices:*** *Decompose the system into small, independent services.*

● ***Serverless:*** *Utilize serverless functions for event-driven processing and scaling.*
● ***Data-intensive:*** *Leverage GCP's big data and analytics services for data processing and analysis.*

3. Select GCP Services:

● ***Compute:*** *Compute Engine, GKE, Cloud Functions.*
● ***Storage****: Cloud Storage, Cloud SQL, Cloud Spanner.*
● ***Networking****: Virtual Private Cloud (VPC), Cloud Load Balancing, Cloud DNS.*
● ***Messaging****: Cloud Pub/Sub.*
● ***Data Processing****: Cloud Dataflow, Cloud Datastore.*
● ***AI/ML****: Cloud Vision API, Cloud Natural Language API.*

4. Design Data Flow and Communication:

● ***Data Flow****: Define how data will be ingested, processed, stored, and accessed.*● *Communication: Determine the communication patterns between services (e.g., synchronous, asynchronous, message queues).*

5. Implement and Test:

● *Develop and test individual components.*
● *Containerize applications using Docker and deploy them to GKE.*
● *Utilize infrastructure-as-code tools like Terraform or Google Cloud Deployment Manager for automated deployments.*
● *Conduct thorough testing, including load testing, performance testing, and security testing.*

6. Monitor and Optimize:

● *Utilize Cloud Monitoring to collect metrics and logs.*
● *Analyze data to identify performance bottlenecks and areas for improvement.*
● *Continuously monitor and adjust system configuration to meet changing requirements.*

Example System: Social Media Platform

Architecture:

● ***Frontend****: Cloud Run for deploying a containerized web application.*
● ***Backend Microservices:***
○ ***User Service:*** *GKE for managing user accounts, profiles, and social graphs.*
○ ***Post Service:*** *GKE for handling post creation, storage, and retrieval.*
○ ***Notification Service:*** *Cloud Pub/Sub for real-time notifications.*
● ***Database:*** *Cloud Spanner for globally distributed data, Cloud SQL for relational data.*
● ***Storage:*** *Cloud Storage for storing user-generated content (images, videos).*
● ***AI/ML:*** *Cloud Natural Language API for sentiment analysis, Cloud Vision API for image moderation.*

Implementation and Deployment:

1. Develop and test each microservice independently.
2. Containerize each microservice using Docker.
3. Deploy microservices to GKE.
4. Configure load balancing and auto-scaling for each service.
5. Utilize Cloud Pub/Sub for asynchronous communication between services.
6. Deploy the frontend application to Cloud Run.
7. Implement security measures using Cloud IAM and VPC.
8. Monitor system performance and health using Cloud Monitoring and logging.

9. Continuously optimize the system based on performance data and user feedback.

Key Considerations:

● ***Data Consistency:*** *Choose the appropriate database (Cloud Spanner for strong consistency, Cloud SQL for eventual consistency) based on application requirements.*

● ***Scalability:*** *Leverage auto-scaling, load balancing, and horizontal scaling of GKE clusters to handle traffic fluctuations.*

● ***Fault Tolerance:*** *Utilize regional and zonal redundancy, along with failover mechanisms, to ensure high availability.*

● ***Security****: Implement robust security measures, including data encryption, access control, and intrusion detection.*

● ***Cost Optimization:*** *Utilize cost-effective services like Cloud Functions and optimize resource utilization to minimize costs.*

Data-Centric Consistency Models:

Data-centric consistency models govern how data is accessed and updated in a distributed system, ensuring that all nodes maintain an agreed-upon view of the shared data. Here's a detailed look at each model:

1. Strict Consistency

○ *The strongest and most demanding consistency model.*

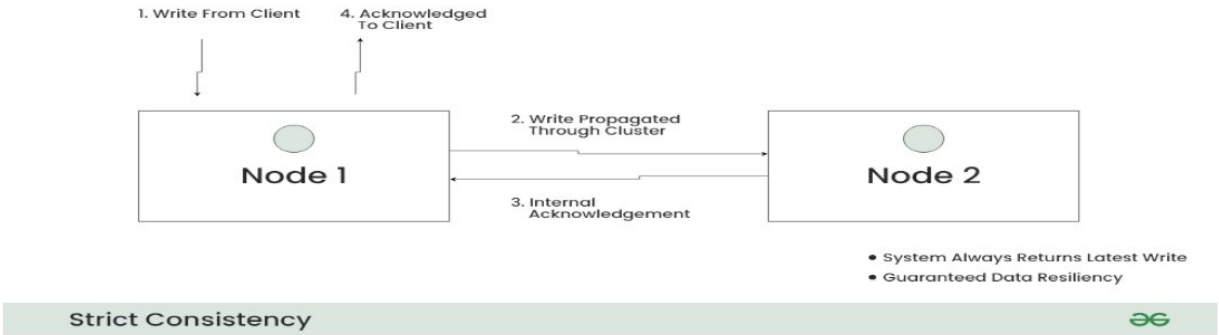
○ *A write operation is considered complete only when all replicas of the data item have been updated.*

○ *Reads must always return the value of the most recent write.*

● ***Analogy****: Imagine a single, perfectly synchronized clock across all nodes. Writes are broadcast simultaneously to all nodes at the exact same moment, and reads always reflect the latest written value.*

● ***Implementation****: Extremely challenging to achieve in practice, especially in large-scale distributed systems with varying network latencies. Requires complex mechanisms for ensuring instantaneous updates across all nodes.*

● ***Use Cases:*** *Systems with critical real-time requirements where the absolute latest data is paramount (e.g., high-frequency trading systems).*



2. Sequential Consistency

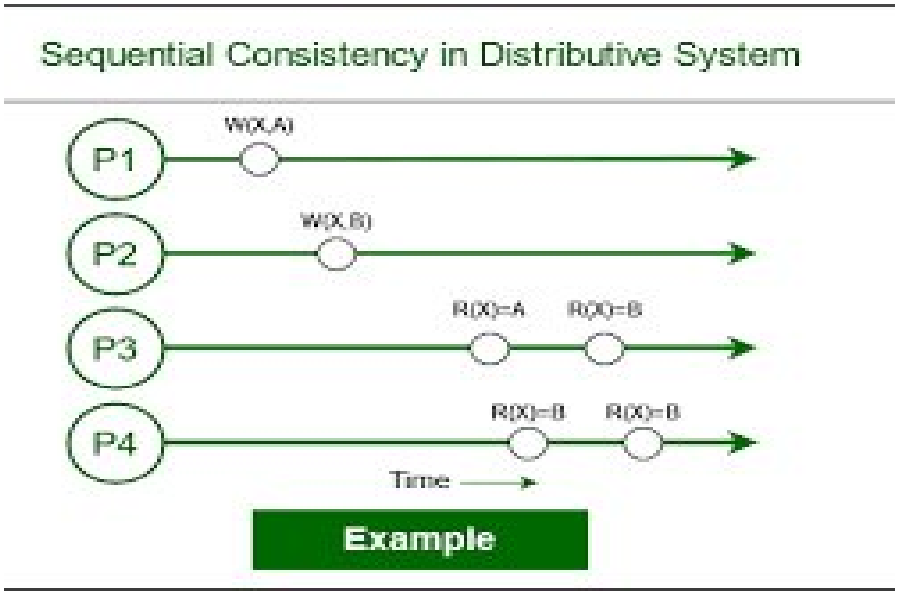
○ *All operations (reads and writes) appear to execute in a single total order, as if they were executed on a single processor.*

○ *The order of operations within each process is preserved.*

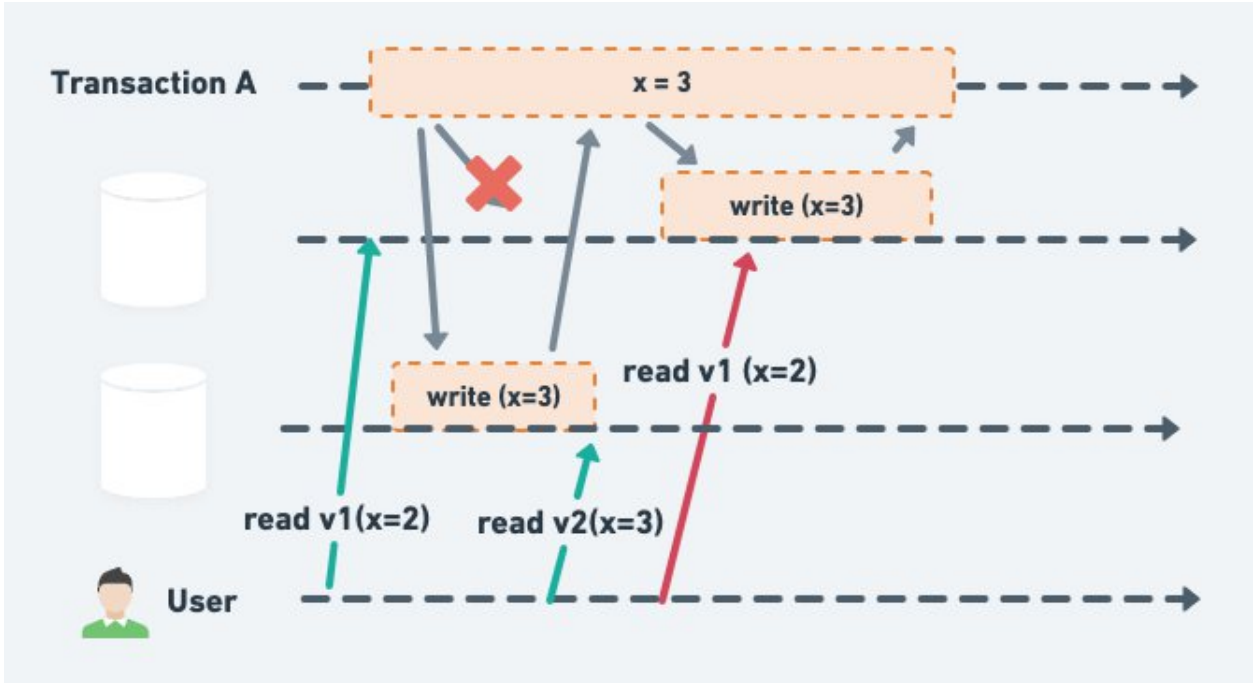
● ***Analogy****: Imagine a global event log where all operations are recorded in a specific order. Each process observes this log, but the exact order of operations within the log may differ from the actual physical execution order.*

● ***Implementation****: Less stringent than strict consistency, making it more practical for many*

distributed systems.
*● **Use Cases:** Systems where the overall order of operations is important, but not necessarily the exact timing of each operation.*

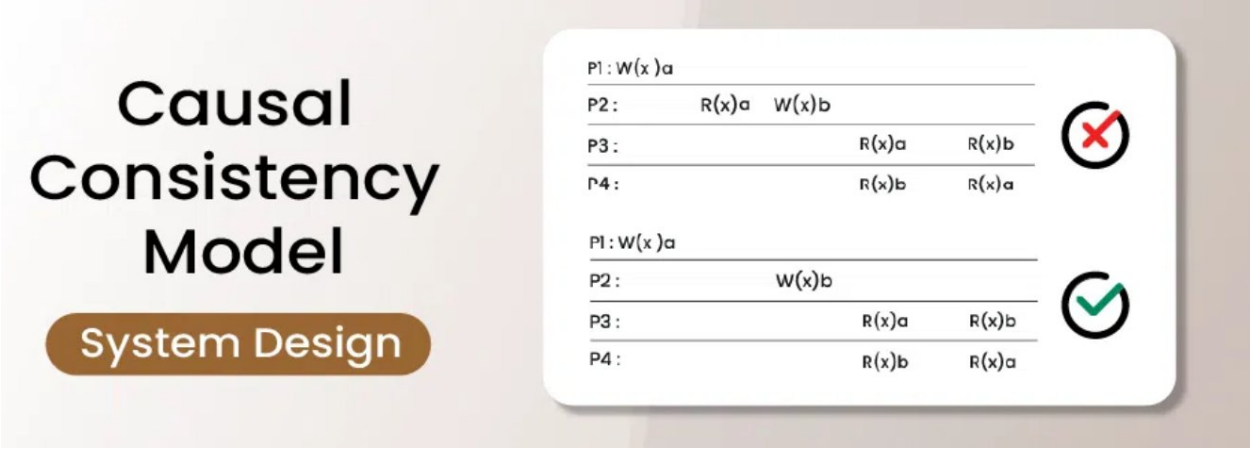


3. Linearizability
○ A stronger form of sequential consistency.
○ Each operation appears to take effect instantaneously at some point between its invocation and its response.
○ Operations appear to be atomic, meaning they happen indivisibly and instantaneously.
*● **Analogy**: Imagine each operation as a single, indivisible event with a well-defined start and end time. The order of these atomic events must be consistent across all processes.*
*● **Implementation**: More challenging to achieve than sequential consistency, often requiring complex synchronization mechanisms.*
*● **Use Cases:** Systems where precise timing of operations is crucial, such as in real-time systems and financial applications.*

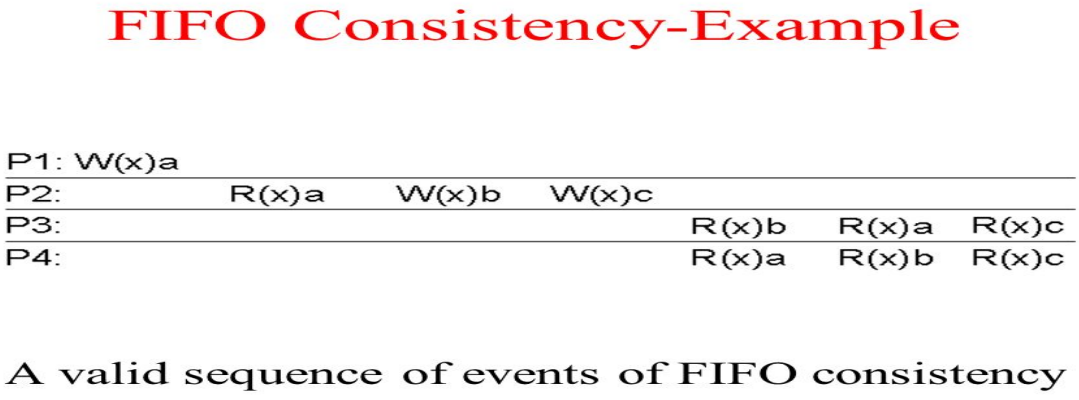


4. Causal Consistency

○ If operation A causally precedes operation B (i.e., A's effects can influence B), then all processes must see A before B.
○ Preserves the causal order of operations.● Analogy: Imagine a directed acyclic graph (DAG) where nodes represent operations and
edges represent causal dependencies. All processes must see the operations in an order that respects these dependencies.
*● **Implementation**: More relaxed than linearizability, allowing for some flexibility in the order of operations that are not causally related.*
*● **Use Cases**: Systems where preserving the causal order of events is important, such as in social media feeds where replies should appear after the original posts.*

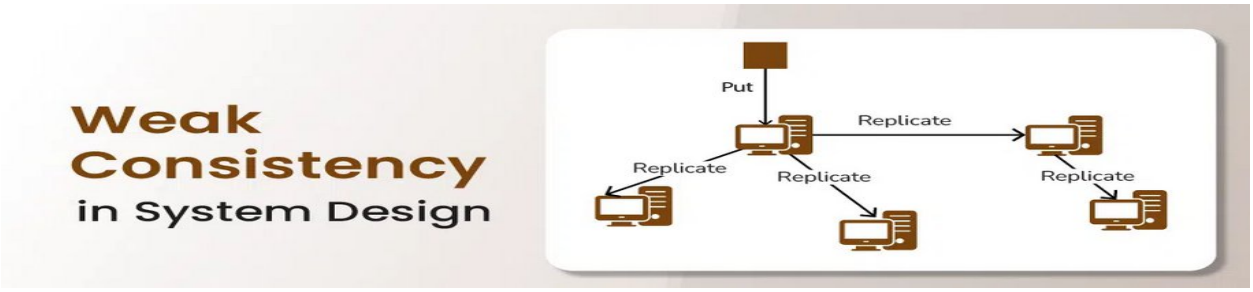


5. FIFO Consistency
○ Writes from a single process are seen by all other processes in the order they were issued.
*● **Analogy**: Imagine each process maintaining a queue for its own write operations. Other processes must see the writes in the order they appear in the queue of the issuing process.*
*● **Implementation**: Relatively easier to implement compared to stronger models.*
*● **Use Cases:** Systems where preserving the order of writes within a single process is crucial, such as in some database applications.*

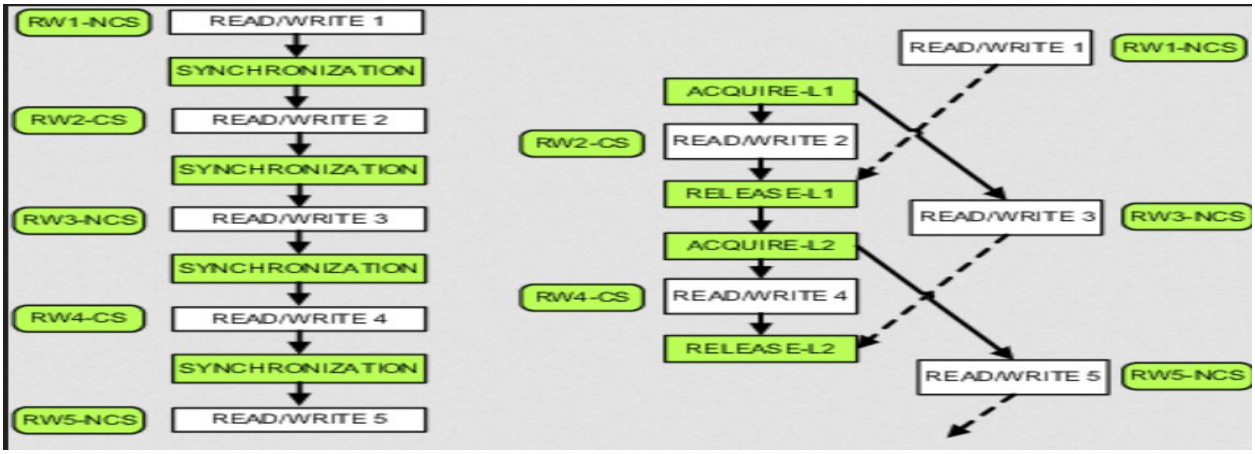


6. Weak Consistency
○ The weakest form of consistency.
○ No guarantees are made about the order of operations or the visibility of writes.
*● **Analogy**: Imagine a completely unordered set of operations. Processes may see writes in different orders, and some writes may not be visible to certain processes.*

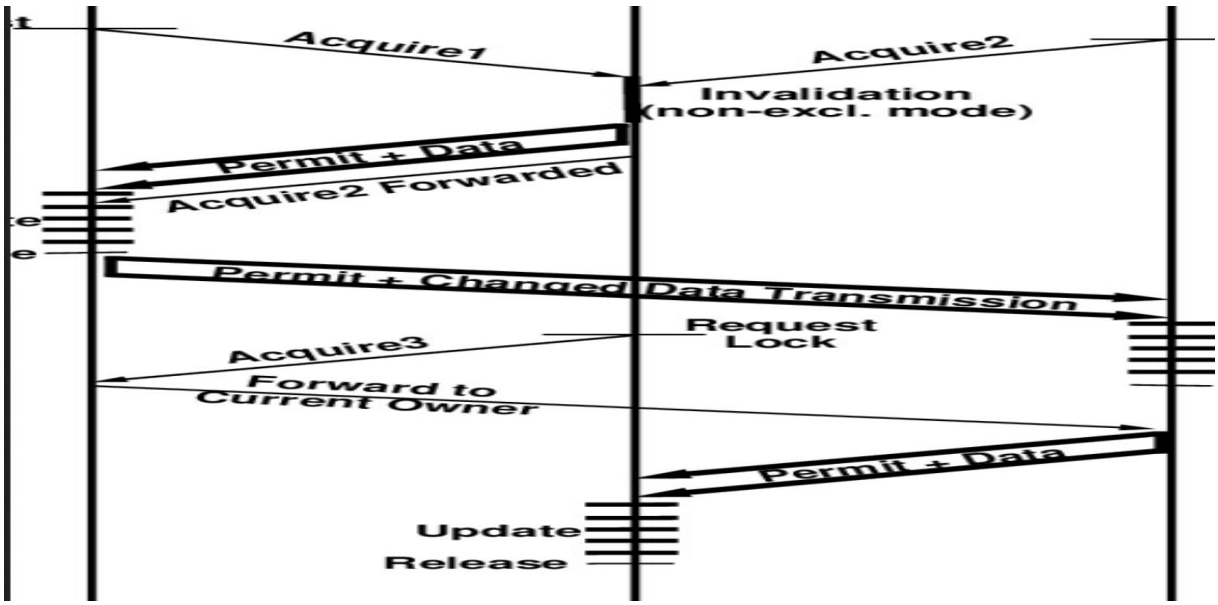
● ***Implementation****: Simplest to implement but offers the fewest guarantees.*
● ***Use Cases:*** *Systems where eventual consistency is acceptable and performance is prioritized (e.g., some distributed file systems).*



7. Release Consistency
○ *Used in distributed shared memory systems.*
○ *Writes to shared data are not visible to other processes until a "release" operation is performed.*
● ***Analogy****: Imagine a critical section where writes are buffered. Writes are only made visible to other processes after a "release" operation is executed, indicating the end of the critical section.*
● ***Implementation:*** *Requires synchronization mechanisms to ensure that writes are properly released.*
● ***Use Cases:*** *Systems that utilize critical sections for data synchronization.*



8. Entry Consistency
○ *Used in distributed shared memory systems.*
○ *Writes to shared data are not visible to other processes until the next critical section is entered.*
● ***Analogy****: Imagine a critical section where writes are buffered. Writes are only made visible to other processes before the next critical section is entered.*
● ***Implementation****: Similar to release consistency, but with the visibility window occurring before the next critical section.*
● ***Use Cases:*** *Systems that utilize critical sections for data synchronization, but with a different visibility window compared to release consistency.Key Considerations:*



● ***Trade-offs****: Stronger consistency models offer more guarantees but often come at the cost of performance and availability. Weaker models prioritize performance and availability but may introduce more complexity in ensuring data integrity.*
● ***Application Requirements:*** *The choice of consistency model depends heavily on the specific requirements of the distributed system, such as performance needs, data sensitivity, and fault tolerance requirements.*
● ***Implementation Challenges****: Implementing strong consistency models can be challenging due to factors like network latency, message ordering, and fault tolerance.*
By carefully considering these factors, developers can choose the most appropriate consistency model for their distributed system, balancing performance, availability, and data integrity.