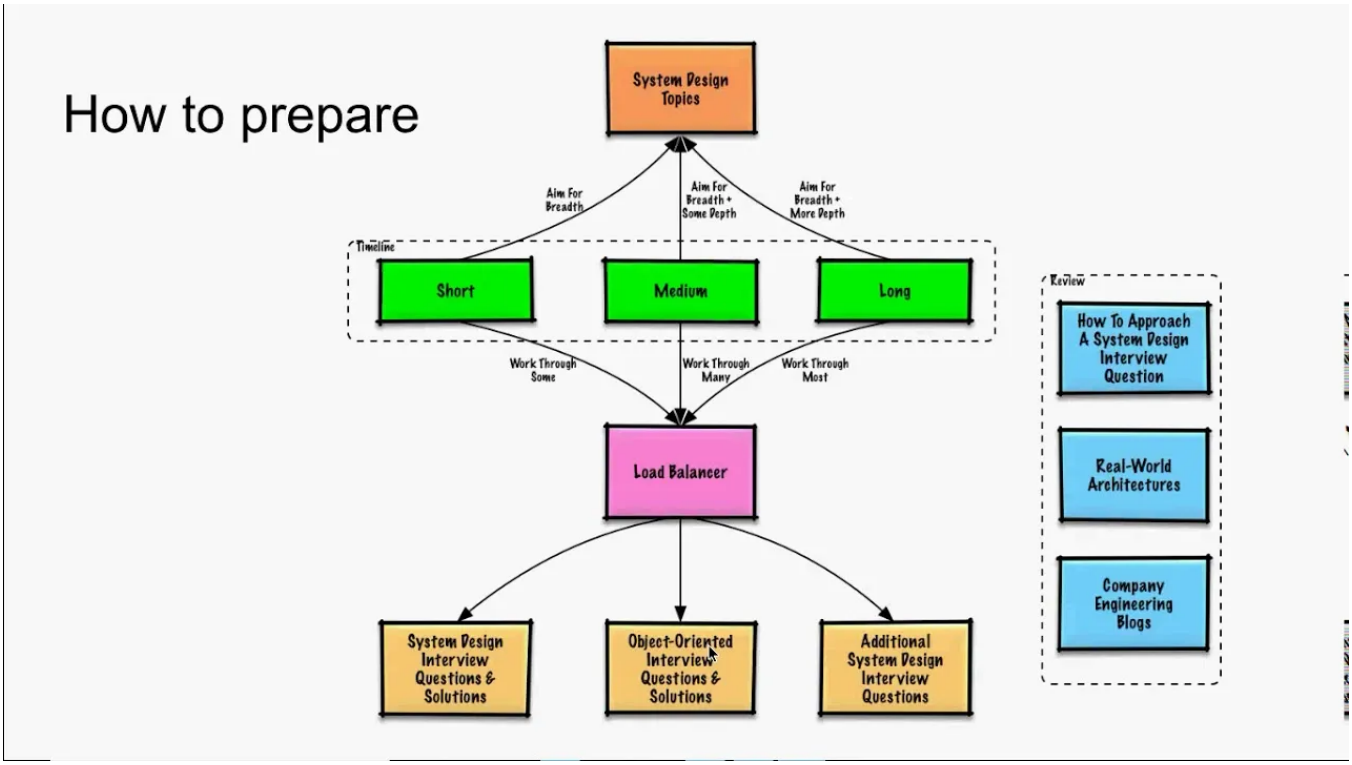
Member-only story

Top 10 System Design Concepts Every Programmer should learn

Essential System Design Concepts for Programmers: 10 Key Principles for Building Scalable, Reliable, and High-Performing Software Systems

Soma · Follow
Published in Javarevisited
16 min read · Apr 15



Hello folks, as a programmer, understanding system design concepts is crucial for developing software systems that are scalable, reliable, and high-performing. **System design** involves designing the architecture and components of a software system to meet specific requirements and achieve desired performance characteristics.

With the rapid advancement of technology and increasing complexity of software applications, mastering system design concepts has become essential for programmers to build efficient and effective systems.

In the past, we have learned about **Microservice design principles** and **Patterns** and In this article, we will explore 10 key system design concepts that every programmer should learn.

These concepts provide a solid foundation for designing software systems that can handle large-scale data processing, accommodate concurrent users, and deliver optimal performance.

Whether you are a beginner or an experienced programmer, understanding these system design concepts will empower you to create robust and scalable software systems that meet the demands of modern applications. So, let's dive in and explore these essential system design principles!

But, if you are not a Medium member then I highly recommend you to join Medium and read great stories from great authors from real field. You can **join Medium here**

10 System Design and Software Architecture Concepts Every Developer Should Know

Here are 10 important system design concepts that every programmer should consider learning:

1. **Scalability**
2. **Availability**
3. **Reliability**

4. **Fault Tolerance**
5. **Caching Strategies**
6. **Load Balancing**
7. **Security**
8. **Scalable Data Management**
9. **Design Patterns**
10. **Performance Optimization**

Understanding and applying these system design concepts can help programmers and developers build robust, scalable, and high-performing software systems that meet the needs of modern applications and users.

Now, let's deep dive into each of them and understand what they are how to achieve them in your application.

1. Scalability

Scalability refers to the ability of a system or application to handle increasing amounts of workload or users without a significant degradation in performance or functionality.

It is an important concept in system design as it allows a system to grow and adapt to changing requirements, **such as increased data volume, user traffic, or processing demands,** without experiencing performance bottlenecks or limitations.

Scalability is critical in modern computing environments where systems need to handle large and growing amounts of data and users. For example, popular websites, mobile apps, and cloud-based services need to be able to handle millions or even billions of requests concurrently, while distributed databases and big data platforms need to scale to handle petabytes or exabytes of data.

Scalability is also important in systems that need to accommodate peak loads, such as online shopping during holiday seasons or sudden spikes in user activity due to viral events.

> *There are two main types of scalability:* ***vertical scalability and horizontal scalability****. Vertical scalability involves adding more resources, such as CPU, memory, or storage, to a*

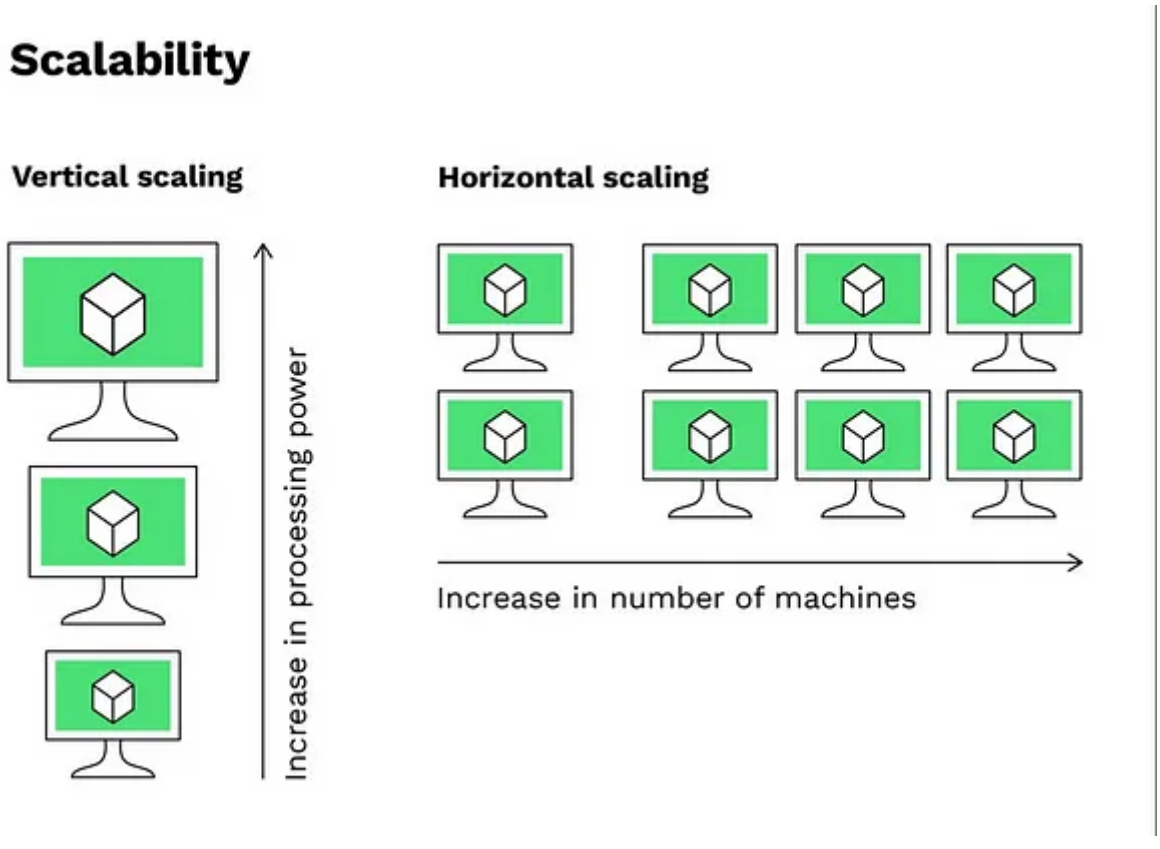
> *single server or node to handle increased workload. Horizontal scalability, on the other hand, involves adding more servers or nodes to a system to distribute the workload and handle increased demand. Horizontal scalability is often achieved through techniques such as load balancing, sharding, partitioning, and distributed processing.*

Achieving scalability requires careful system design, architecture, and implementation. It involves designing systems that can efficiently handle increasing workloads, efficiently utilize resources, minimize dependencies, and distribute processing across multiple nodes or servers.

> *Techniques such as caching, asynchronous processing, parallel processing, and distributed databases are often used to improve scalability. Testing and performance monitoring are also crucial to ensure that the system continues to perform well as it scales.*

Scalability is an essential consideration in building robust, high-performance systems that can handle growth and adapt to changing requirements over time. It allows systems to accommodate increasing demand, provide a seamless user experience, and support business growth without encountering performance limitations or downtime.

Here is a nice diagram which shows the *difference between Vertical Scaling and Horizontal Scaling*



2. Availability

Availability refers to the ability of a software system to remain operational and accessible to users even in the face of failures or disruptions.

High availability is a critical requirement for many systems, especially those that are mission-critical or time-sensitive, such as online services, e-commerce websites, financial systems, and communication networks. Downtime in such systems can result in significant financial losses, reputational damage, and customer dissatisfaction. Therefore, ensuring high availability is a key consideration in system design.

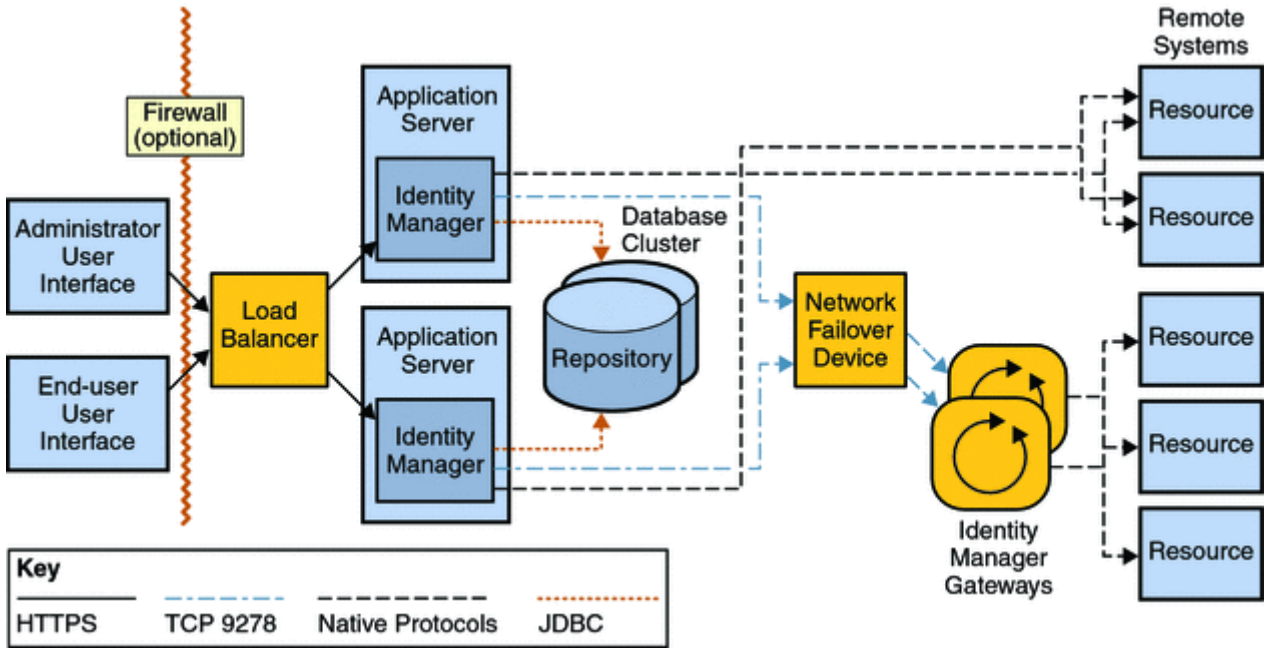
Availability is typically quantified **using metrics such as uptime**, which measures the percentage of time a system is operational, and downtime, which measures the time a system is unavailable.

Achieving **high availability involves designing systems with redundancy,** fault tolerance, and failover mechanisms to minimize the risk of downtime due to hardware failures, software failures, or other unexpected events.

> *In system design, various techniques and strategies are employed to improve availability, such as load balancing, clustering, replication, backup and recovery, monitoring, and proactive maintenance.*

These measures are implemented to minimize single points of failure, detect and recover from failures, and ensure that the system remains operational even in the face of failures or disruptions.

By designing systems with high availability, engineers can ensure that the systems are accessible and operational for extended periods of time, leading to improved customer satisfaction, reduced downtime, and increased business continuity.



3. Reliability

Reliability refers to the **consistency and dependability of a software system** in delivering expected results. Building systems with reliable components, error handling mechanisms, and backup/recovery strategies is crucial for ensuring that the system functions as intended and produces accurate results.

Reliability is a critical factor in system design, as it directly impacts the overall performance and quality of a system. Reliable systems are expected to consistently operate as expected, without experiencing unexpected failures, errors, or disruptions.

High reliability is often desired in mission-critical applications, where system failures can have severe consequences, such as in aviation, healthcare, finance, and other safety-critical domains.

Reliability is typically quantified using various metrics, such as **Mean Time Between Failures (MTBF)**, which measures the average time duration between failures, and Failure Rate (FR), which measures the rate at which failures occur over time.

> *Reliability can be achieved through various techniques and strategies, such as redundancy, error detection and correction, fault tolerance, and robust design.*

In system design, achieving high reliability involves careful consideration of various factors, including component quality, system architecture, error handling, fault tolerance mechanisms, monitoring, and maintenance strategies.

By **designing systems with high reliability**, engineers can ensure that the systems perform consistently and as expected, leading to improved customer satisfaction, reduced downtime, and increased system performance and availability.

4. Fault Tolerance

Fault tolerance refers to the ability of a system or component to continue functioning correctly in the presence of faults or failures, such as hardware failures, software bugs, or other unforeseen issues. A fault-tolerant system is designed to detect, isolate, and recover from faults without experiencing complete failure or downtime.

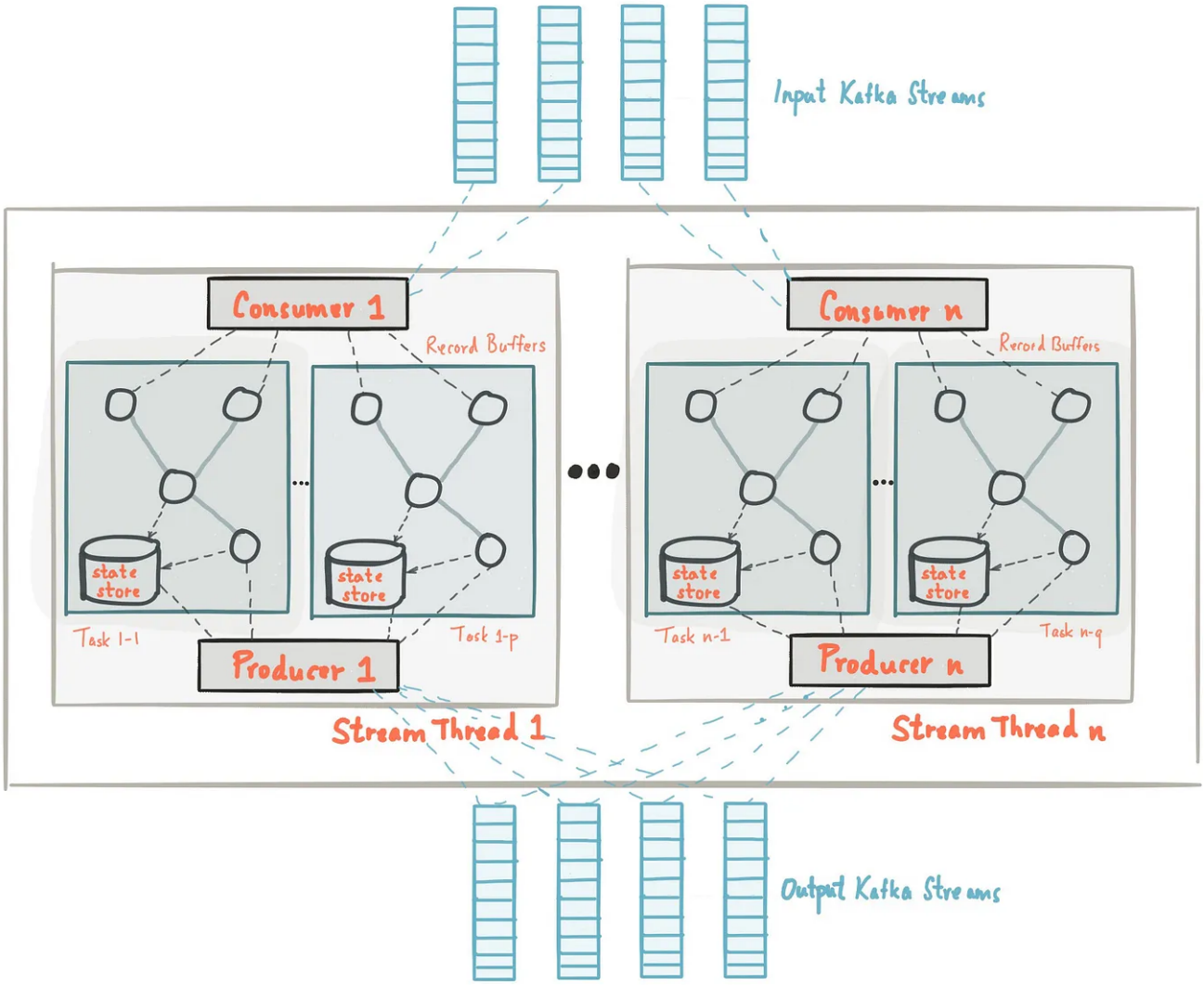
Fault tolerance is an important concept in system design, particularly in distributed systems or systems that need to operate reliably in challenging environments.

It involves implementing redundancy, error detection, error correction, and error recovery mechanisms to ensure that the system continues to operate even when certain components or subsystems fail.

> *There are various techniques and strategies for achieving fault tolerance, such as **replication**, where multiple copies of the same data or service are maintained in different locations, so that if one fails, others can take over; checkpointing, where system states are periodically saved, so that in case of failure, the system can be restored to a previously known good state; and graceful degradation, where the system can continue to operate with reduced functionality in the presence of failures.*

Fault tolerance is crucial for ensuring high availability, reliability, and resilience of systems, particularly in mission-critical applications where system failures can have severe consequences.

By incorporating fault tolerance mechanisms in system design, engineers can create robust and dependable systems that can continue to operate and deliver expected results even in the face of unexpected failures.



5. Caching Strategies

Caching strategies are techniques used to optimize the performance of systems by storing frequently accessed data or results in a temporary storage location, called a cache, so that it can be quickly retrieved without needing to be recalculated or fetched from the original source. There are several common caching strategies used in system design:

1. **Full Caching**
   In this strategy, the entire data set or results are cached in the cache, providing fast access to all the data or results. This strategy is useful when the data or results are relatively small in size and can be easily stored in memory or a local cache.

2. **Partial Caching**
   In this strategy, only a subset of the data or results are cached, typically based on usage patterns or frequently accessed data. This strategy is useful when the data or results are large in size or when not all the data or results are frequently accessed, and it is not feasible to cache the entire data set.

3. **Time-based Expiration**
   In this strategy, the data or results are cached for a specific duration of time, after which the cache is considered stale, and the data or results are fetched from the original source and updated in the cache. This strategy is useful when the data or results are relatively stable and do not change frequently.

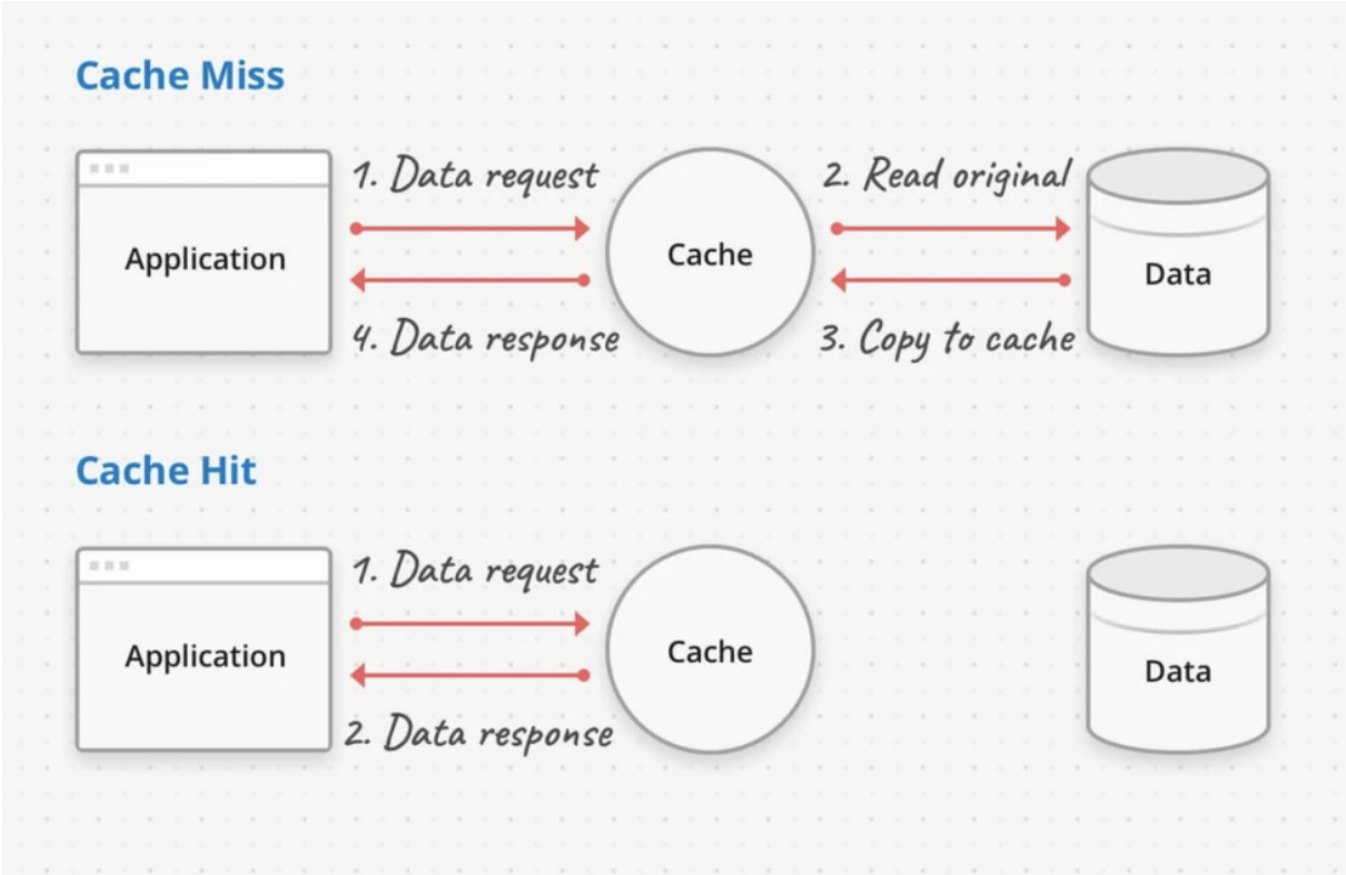
4. **LRU (Least Recently Used) or LFU (Least Frequently Used) Replacement Policy**
   In these strategies, the data or results that are least recently used or least frequently used are evicted from the cache to make room for new data or results. These strategies are useful when the cache has limited storage capacity and needs to evict less frequently accessed data to accommodate new data.

5. **Write-through or Write-behind Caching**
   In these strategies, the data or results are written to both the cache and the original source (write-through) or only to the cache (write-behind) when updated or inserted. These strategies are useful when the system needs to maintain consistency between the cache and the original source or when the original source cannot be directly updated.

6. **Distributed Caching**
   In this strategy, the cache is distributed across multiple nodes or servers, typically using a distributed caching framework or technology. This strategy is useful when the system is distributed or scaled across multiple nodes or servers and needs to maintain consistency and performance across the distributed cache.

7. **Custom Caching**
   Custom caching strategies can be implemented based on the specific requirements and characteristics of the system or application. These strategies may involve combinations of the above-mentioned strategies or other custom approaches to suit the unique needs of the system.

The selection of the *appropriate caching strategy depends on various factors such as the size of data or results, frequency of access, volatility of data or results, storage capacity, consistency requirements, and performance goals of the system*. Careful consideration and implementation of caching strategies can significantly improve system performance, reduce resource utilization, improve scalability, and enhance user experience.



6. Load Balancing

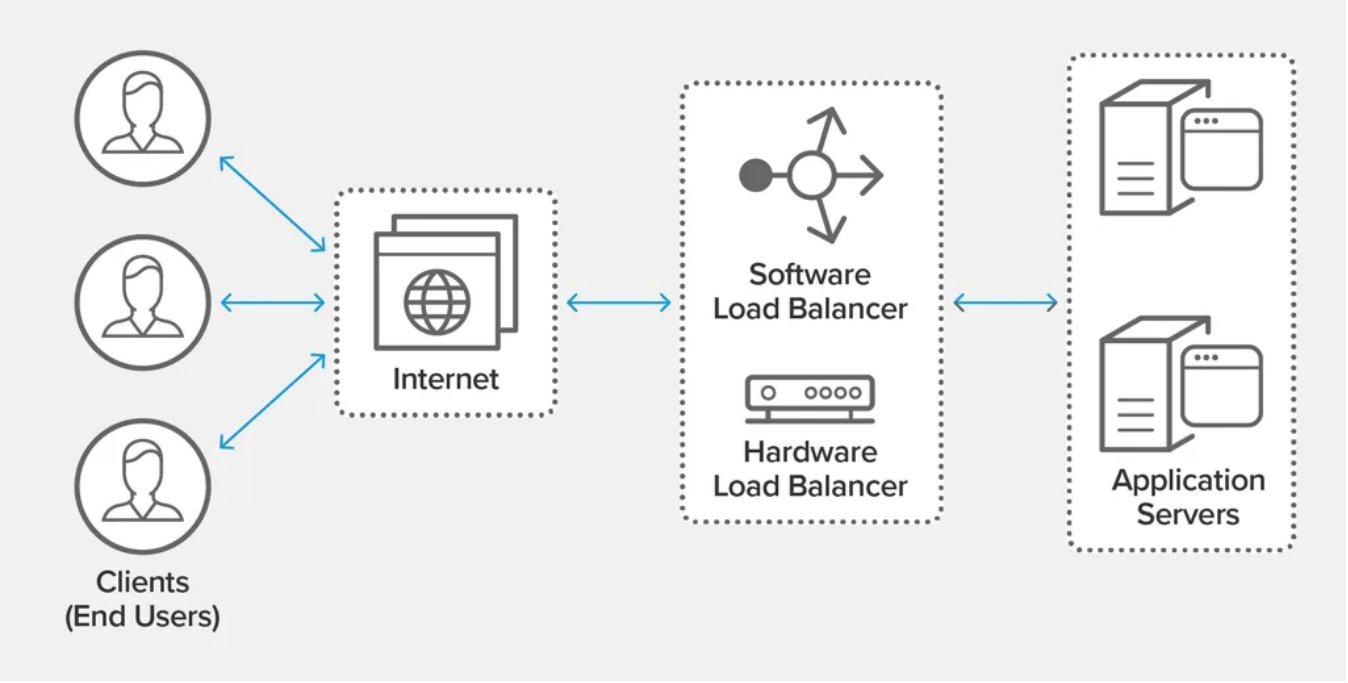
Load balancing is a technique used in distributed systems or networks to distribute incoming network traffic or workload evenly across multiple servers or resources, ensuring that no single server or resource is overwhelmed with excessive traffic or workload.

The purpose of load balancing is to optimize resource utilization, maximize system availability, and improve overall system performance and reliability. In Microservice architecture, *Load balancing and API Gateway* are often refer same but they are not, API Gateway can do a lot more as explained in this article.

Load balancing can be achieved through various algorithms or methods, such as:

1. **Round Robin:** Incoming requests are distributed sequentially to each server or resource in a rotating manner, ensuring an equal distribution of traffic across all servers or resources
2. **Least Connection:** Incoming requests are distributed to the server or resource with the least number of active connections, ensuring that the server or resource with the least load receives new requests.
3. **Source IP Affinity:** Incoming requests from the same client IP address are directed to the same server or resource, ensuring that requests from a specific client are consistently handled by the same server or resource.
4. **Weighted Round Robin:** Incoming requests are distributed based on predefined weights assigned to each server or resource, allowing for different traffic distribution ratios based on server or resource capacity or capability.
5. **Adaptive Load Balancing:** Load balancing algorithms dynamically adjust the distribution of traffic based on real-time monitoring of server or resource health, performance, or other metrics, ensuring optimal resource utilization and system performance.

Load balancing can be implemented using hardware-based load balancers, software-based load balancers, or cloud-based load balancing services. It plays a critical role in distributed systems or networks with high traffic loads or resource-intensive workloads, enabling efficient utilization of resources, enhancing system availability and reliability, and providing seamless user experience.



7. Security

Security in system design refers to the consideration and implementation of measures to protect a system from potential security threats, vulnerabilities, or attacks. It involves designing and implementing systems with built-in security features and practices to safeguard against unauthorized access, data breaches, data leaks, malware attacks, and other security risks.

Security in system design typically involves the following principles:

1. **Authentication:** Ensuring that users or entities are verified and granted appropriate access privileges based on their identity and credentials.
2. **Authorization**: Enforcing access controls and permissions to restrict users or entities from accessing unauthorized resources or performing unauthorized actions.
3. **Encryption**: Protecting sensitive data by using encryption techniques to prevent unauthorized access or data breaches.
4. **Auditing and Logging**: Implementing mechanisms to track and log system activities and events for monitoring, auditing, and forensic purposes.
5. **Input Validation:** Validating and sanitizing all input data to prevent common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and

cross-site request forgery (CSRF) attacks.

6. **Patching and Updates**: Keeping the system up-to-date with the latest security patches and updates to address known security vulnerabilities.
7. **Defense in Depth**: Implementing multiple layers of security controls, such as firewalls, intrusion detection systems, and antivirus software, to provide a multi-tiered defense against security threats.
8. **Principle of Least Privilege**: Limiting users or entities' access privileges to the minimum necessary to perform their tasks, reducing the potential impact of a security breach or attack.
9. **Secure Communication**: Using secure communication protocols, such as HTTPS or SSL/TLS, to protect data in transit from interception or eavesdropping.

Security in system design is critical to protect the integrity, confidentiality, and availability of data and resources, and to ensure the overall security posture of a system. It should be considered and incorporated into the design, development, and deployment phases of a system to mitigate potential security risks and safeguard against security threats.

8. Scalable Data Management

Scalable data management refers to the ability of a system or application to effectively handle growing amounts of data without experiencing performance degradation or loss of functionality. It involves designing and implementing data management practices and technologies that can handle increasing data volumes, user loads, and processing requirements, while maintaining acceptable levels of performance and reliability.

Scalable data management typically involves the following principles:

1. **Data Partitioning:** Splitting large datasets into smaller, manageable chunks or partitions to distribute the data across multiple storage or processing resources. This helps in reducing the load on individual resources and allows for parallel processing and improved performance.

2. **Distributed Database Systems:** Using distributed databases or data storage solutions that can distribute data across multiple nodes or servers, enabling horizontal scaling and improved performance.

3. **Data Replication**: Replicating data across multiple nodes or servers to ensure data availability and fault tolerance. This can involve techniques such as data mirroring, data sharding, or data caching to improve performance and reliability.

4. **Caching and In-Memory Data Storage:** Caching frequently accessed data or storing data in memory for faster retrieval and processing, reducing the need for expensive disk I/O operations and improving performance.

5. **Indexing and Query Optimization**: Using efficient indexing and query optimization techniques to speed up data retrieval and processing operations, especially in large datasets.

6. **Data Compression:** Implementing data compression techniques to reduce the storage footprint and improve data transfer efficiency, especially for large datasets.

7. **Data Archiving and Purging**: Implementing data archiving and purging practices to remove or archive old or infrequently accessed data, reducing the storage and processing overhead and improving performance.

8. **Scalable Data Processing Frameworks:** Using scalable data processing frameworks such as Apache Hadoop, Apache Spark, or Apache Flink, that can handle large-scale data processing and analytics tasks in a distributed and parallelized manner.

9. **Cloud-based Data Management**: Leveraging cloud-based data management services, such as Amazon S3, Amazon RDS, or Google Bigtable, that provide scalable and managed data storage and processing capabilities.

10. **Monitoring and Scalability Testing**: Regularly monitoring system performance and conducting scalability testing to identify and address performance bottlenecks, resource limitations, or other scalability challenges, and ensuring that the data management practices can effectively handle increasing data volumes and loads.

In short, Scalable data management is crucial for modern applications and systems that need to handle large amounts of data, user loads, and processing requirements. It enables systems to grow and adapt to changing needs without sacrificing performance or reliability, ensuring that the data management practices can effectively handle increasing data volumes and loads.

9. Design Patterns

Design patterns in system design refer to reusable solutions or best practices that are used to address common design challenges or problems encountered during the development of a software system. Design patterns are widely accepted and proven approaches to design and architect software systems, and they provide a set of well-defined guidelines for designing efficient, maintainable, and scalable systems.

Design patterns in system design can be categorized into various types, including:

1. **Creational Patterns:** These patterns focus on object creation mechanisms and provide ways to create objects in a flexible and reusable manner. Examples of creational patterns include Singleton, Factory Method, Abstract Factory, Builder, and Prototype patterns.
2. **Structural Patterns:** These patterns focus on the organization of classes and objects to form a larger structure or system. Examples of structural patterns include Adapter, Bridge, Composite, Decorator, and Facade patterns.
3. **Behavioral Patterns:** These patterns focus on the interaction and communication between objects or components within a system. Examples of behavioral patterns include Observer, Strategy, Command, Iterator, and Template Method patterns.
4. **Architectural Patterns:** These patterns provide high-level guidelines and strategies for designing the overall architecture of a system. Examples of architectural patterns include Model-View-Controller (MVC), Model-View-ViewModel (MVVM), Layered architecture, Microservices, and Event-Driven architecture patterns.

Design patterns are important in system design as they provide proven and standardized ways to address common design challenges, improve code quality, and ensure maintainability and scalability of software systems.

They promote code reusability, separation of concerns, and encapsulation of functionality, making it easier to manage complex systems and adapt them to changing requirements.

By using design patterns, developers can leverage existing knowledge and best practices to design robust and efficient systems that meet the needs of the users and stakeholders.

In the last few articles I have also talked about common Microservice design patterns like **Event Sourcing**, **CQRS**, **SAGA**, **Database Per Microservices**, **API Gateway**, **Circuit-Breaker** and also shared *best practices to design Microservices* , you can also check these article to learn more about Microservice communication, including synchronous and asynchronous communication.

Difference between Synchronous vs Asynchronous Communication in Microservices

Choosing the Right Communication Pattern for Microservices Architecture: Exploring Synchronous and Asynchronous...

medium.com

10. Performance

While we already know what performance means, remember slow laptop? In System Design, Performance refers to the speed, responsiveness, and efficiency of a software system in processing data and delivering results.

Optimizing system performance through efficient algorithms, caching, indexing, and other techniques is essential for creating systems that can handle large-scale data processing and deliver optimal response times.

Performance in system design refers to the ability of a software system or application to efficiently and effectively carry out its intended functions or tasks, while meeting performance requirements and expectations. It encompasses various aspects such as response time, throughput, resource utilization, scalability, and efficiency of the system.

Performance is a critical factor in system design as it directly affects the user experience, system reliability, and overall system efficiency. A poorly performing system can lead to slow response times, low throughput, high resource utilization, and inefficient use of system resources, resulting in degraded system performance and user dissatisfaction.

System designers need to consider various performance-related factors during the design process, such as choosing appropriate algorithms and data structures, optimizing code, minimizing unnecessary overheads, managing system resources efficiently, and ensuring proper system configuration and tuning. Performance testing and profiling techniques can also be used to identify and address performance bottlenecks and optimize system performance.

Optimizing performance in system design requires a careful balance between functionality, complexity, and resource utilization. It involves making informed design decisions, using best practices, and continuously monitoring and optimizing system performance to ensure that the system meets its performance requirements and delivers a smooth and efficient user experience.

Conclusion

In conclusion, **understanding and mastering these key system design concepts is crucial for programmers** to build robust, scalable, and efficient software systems. These concepts, including fault tolerance, reliability, availability, caching strategies, load balancing, security, scalable data management, design patterns, and performance, play a critical role in ensuring that software systems meet their intended objectives, perform optimally, and deliver a superior user experience.

By having a solid understanding of these system design concepts, you can make informed design decisions, choose appropriate technologies and techniques, and optimize system performance. It also allows you to design systems that are resilient, scalable, secure, and efficient, capable of handling the challenges of modern-day software development and meeting the expectations of end-users.

And, if you are not a Medium member then I highly recommend you to join Medium so that you can not only read these articles but also from many others like Google Engineers and Tech experts from FAANG companies. You can **join Medium here**

Join Medium with my referral link - Soma

Read this and every story from thousands of great writers on Medium). Read more from great writers on Medium like...

medium.com

Other **System Design and Microservices articles** you may like

What is CQRS (Command and Query Responsibility Segregation) Pattern?

Why use CQRS Pattern? What changes in your Microservices architecture by using CQRS Pattern

medium.com

What is Event Sourcing Design Pattern in Microservices Architecture? How does it work?

An Overview of Event Sourcing Design Pattern and Its Implementation in Microservice Architecture.

medium.com

What is Database Per Microservices Pattern? What Problem does it solve?

Breaking Down Monoliths: How the Database Per Microservice Pattern Can Transform Your Architecture

medium.com

Programming Software Engineering Microservices Development Technology

Follow

Written by Soma

4.1K Followers · Editor for Javarevisited

Java and React developer, Join Medium (my favorite Java subscription) using my link 👉
https://medium.com/@somasharma_81597/membership

More from Soma and Javarevisited

Implementation	implementation	an implementation	features
Persistence API	Hibernate provides its own API	Defines a set of standard APIs for ORM	Builds on JPA APIs, adds more functionality
Database Support	Supports various databases through dialects	Depends on the JPA implementation used	Depends on the JPA implementation used
Transaction Management	Provides its own transaction management	Depends on the JPA implementation used	Depends on the JPA implementation used
Query Language	Hibernate Query Language (HQL)	JPQL (Java Persistence Query Language)	JPQL (Java Persistence Query Language)
Caching	Provides first-level and second-level caching	Depends on the JPA implementation used	Depends on the JPA implementation used
Configuration	XML, annotations, or Java-based configuration	XML, annotations, or Java-based configuration	XML, annotations, or Java-based configuration
Integration	Can be used independently or with JPA	Works with any JPA-compliant implementation	Works with any JPA-compliant implementation

Soma in Javarevisited

Difference between Hibernate, JPA, and Spring Data JPA?

Hello folks, if you are preparing for Java Developer interviews then part from preparing Core Java, Spring Boot, and Microservices, you...

✨ · 10 min read · May 26

253 1

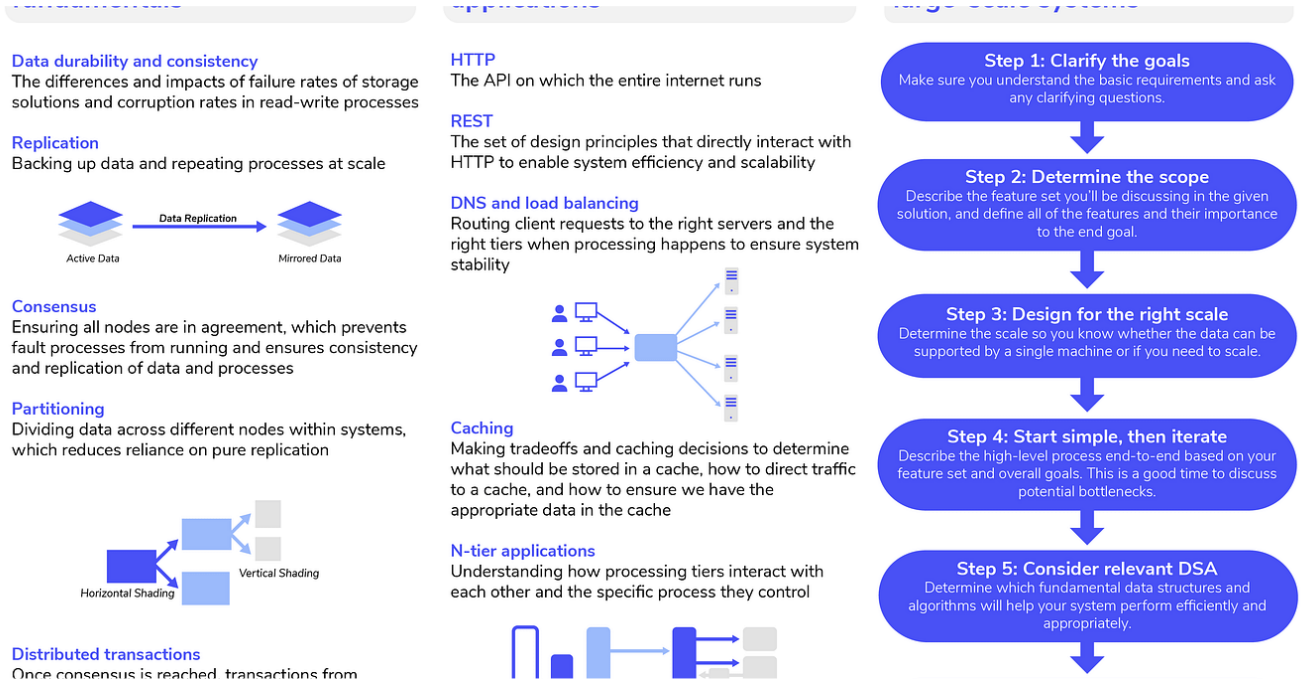
javinpaul in Javarevisited

Top 10 Websites to Learn Python Programming for FREE [2023]

Hello guys, if you are here then let me first congratulate you for making the right decision to learn Python programming language, the...

13 min read · Apr 22, 2020

328 6



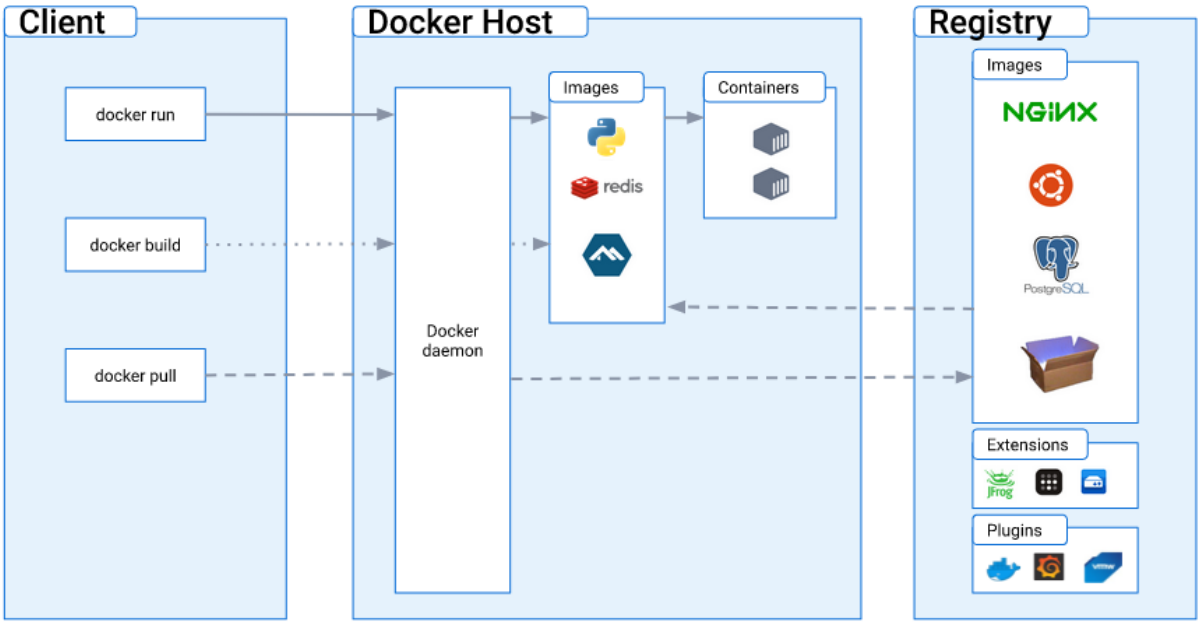
javinpaul in Javarevisited

Top 3 System Design Cheat Sheets for Developers

3 System Design Cheat Sheet you can print and put on your desktop to revise before Tech interviews

6 min read · Jul 19

204 1



Soma in Javarevisited

How Docker works internally? Magic Behind Containerization

Exploring the Inner Workings of Docker: Unveiling the Magic Behind Containerization

· 6 min read · Jun 30

184

See all from Soma

See all from Javarevisited

Recommended from Medium

Anto Semeraro in Level Up Coding

Microservices Orchestration Best Practices and Tools

Optimizing microservices communication and coordination through orchestration pattern with an example in C#

✨ · 10 min read · Mar 27

67

Syed Habib Ullah

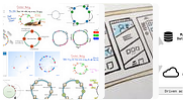
Top Most Asked Java Interview Questions at Accenture, Infosys, Capgemini, Wipro, Cognizant...

As Java is one of the most popular programming languages used in software development today, it's no surprise that Java skill is highly...

9 min read · Apr 1

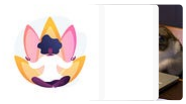
30 2

Lists



General Coding Knowledge
20 stories · 194 saves

It's never too late or early to start something
13 stories · 67 saves



Stories to Help You Grow as a Software Developer
19 stories · 265 saves

ChatGPT prompts
24 stories · 238 saves

Sowmya.L.R

Docker—Container era (PART-I)

In earlier days people rarely used web apps. But in this digital era, every single task is done by any particular app. Ex grocery buying...

4 min read · 2 days ago

10

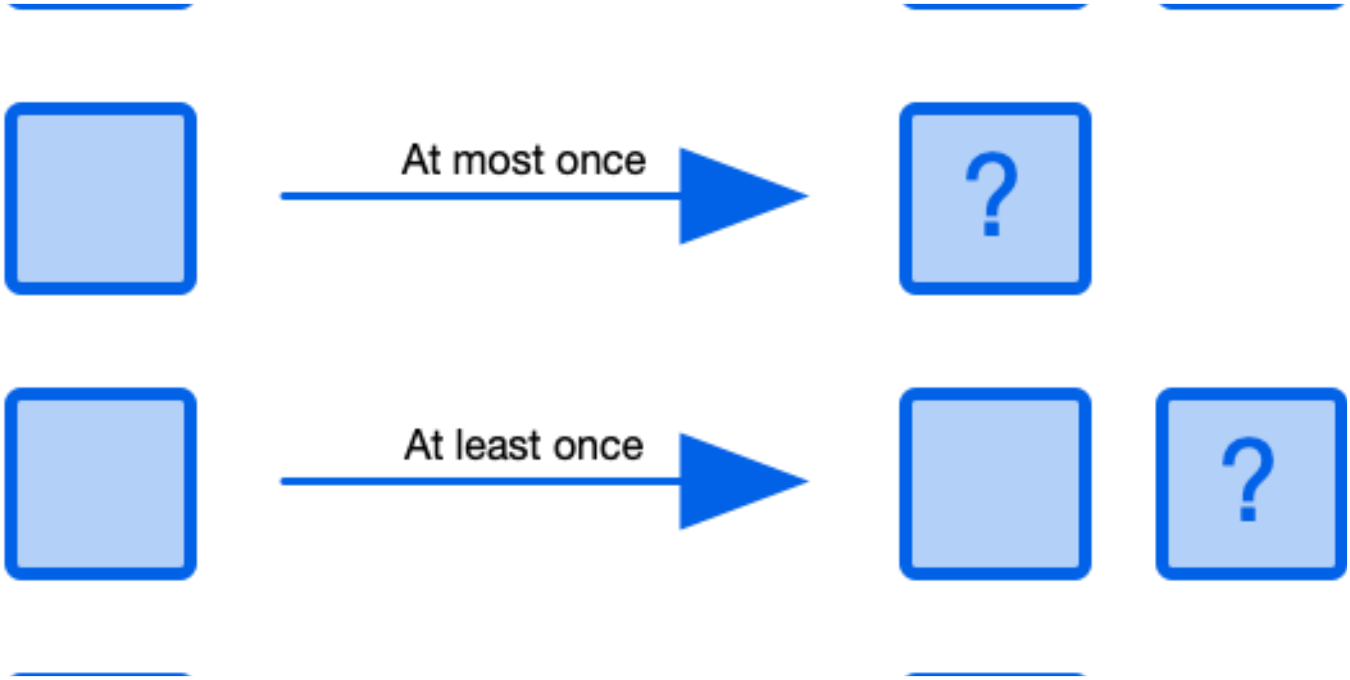
Rupert Waldron

Create a non-blocking REST Api using Spring @Async and Polling

Get the great asynchronous, non-blocking experience you deserve with Spring's basic Rest Api, polling and @Aysnc annotation.

12 min read · Feb 26

17



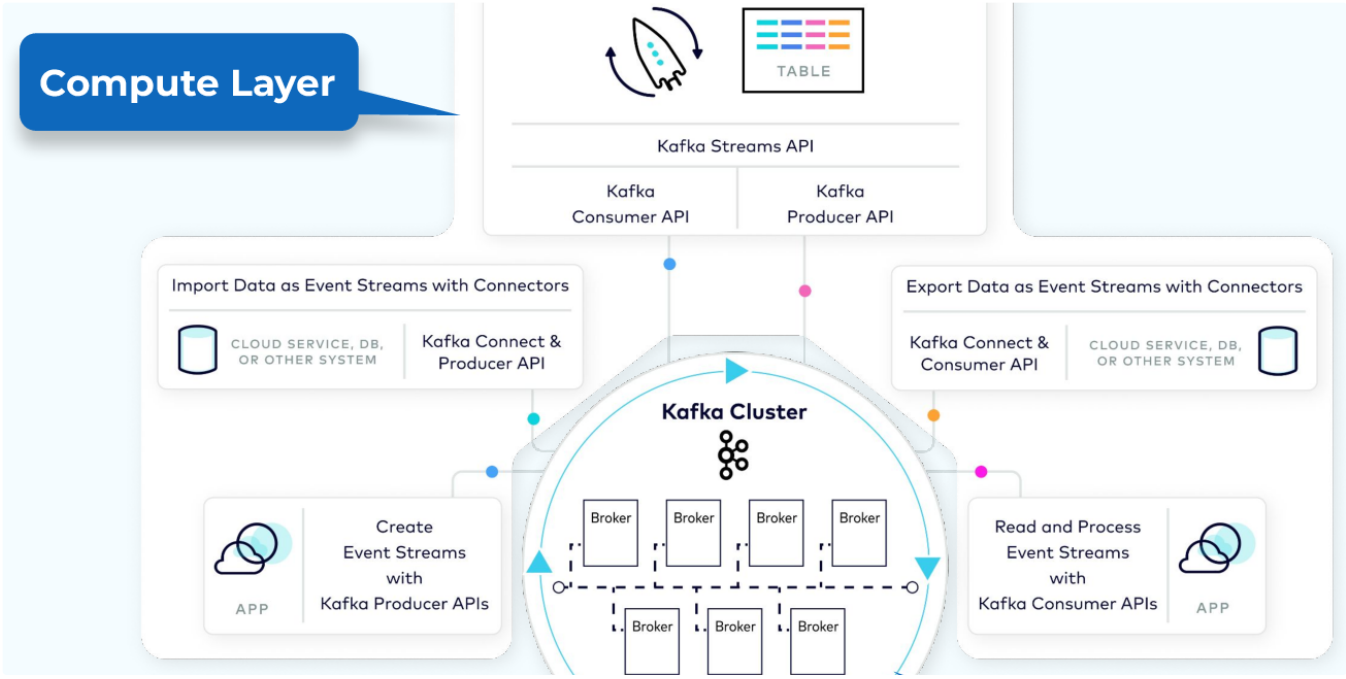
Andy Bryant

Processing guarantees in Kafka

Each of the projects I've worked on in the last few years has involved a distributed message system such as AWS SQS, AWS Kinesis and more...

21 min read · Nov 16, 2019

1.91K 5



Mahesh Saini

Foundational Concepts of Kafka and Its Key Principles

Apache Kafka is a distributed event store and stream-processing platform. The project aims to provide a unified, high-throughput...

7 min read · May 2

158

See more recommendations