A Crash Course on Architectural Scalability

BYTEBYTEGO

AUG 15, 2024 · PAID

♡ 253 ↻ 13 Share ···

In today's interconnected world, software applications have a global reach, serving users from diverse geographical locations.

With the rapid growth of social media and viral content, a single tweet or post can lead to a sudden and massive surge in traffic to an application. The importance of building applications with scalable architecture from the ground up has never been higher.

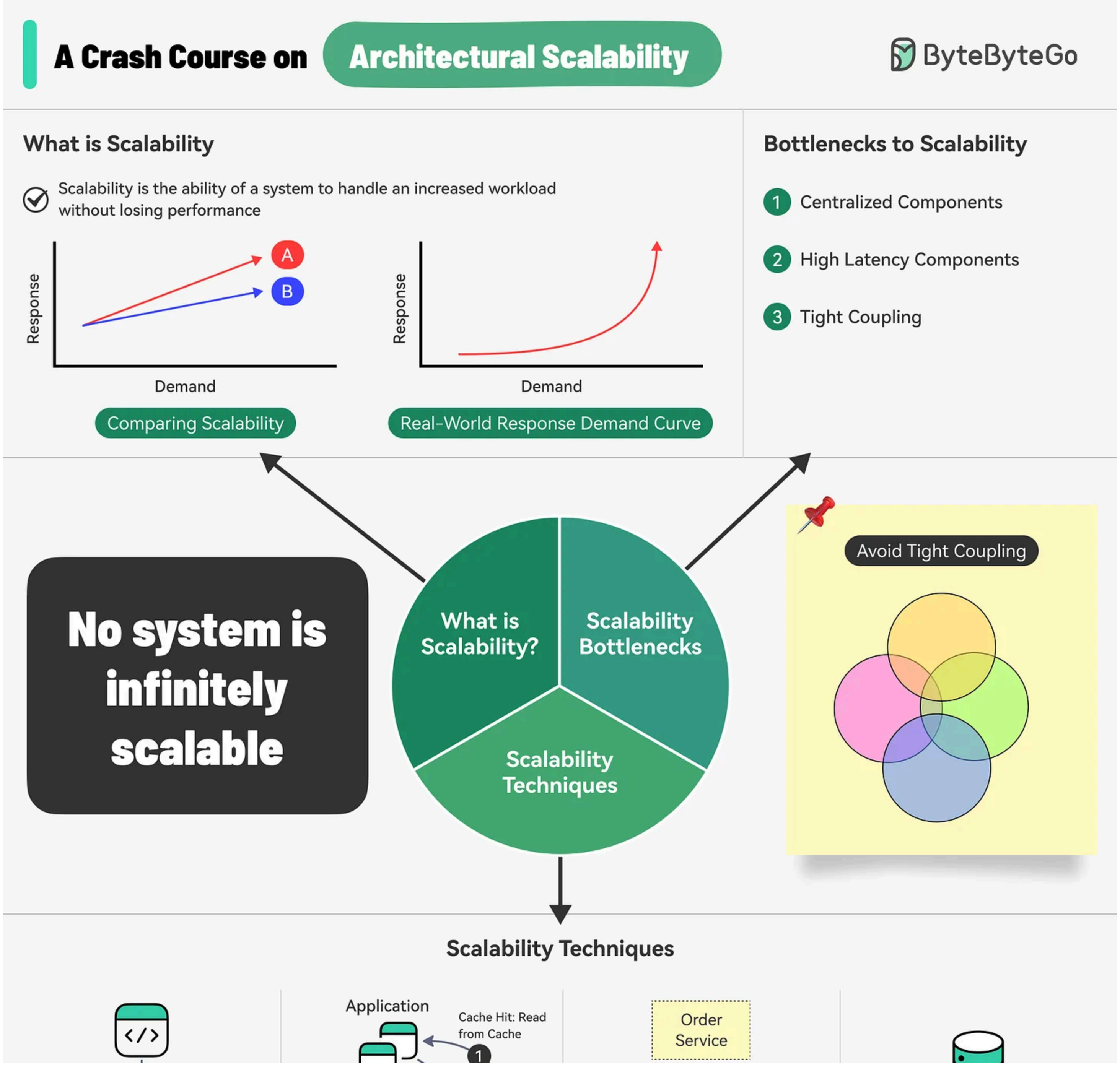
Being prepared for unexpected traffic spikes is indispensable for development teams building applications. A sudden increase in user demand may be just around the corner. Not being prepared for it can put immense pressure on the application's infrastructure. It not only causes performance degradation but can also, in some cases, result in a complete system failure.

To mitigate these risks and ensure a good user experience, teams must proactively design and build scalable architectures.

But what makes scalability such a desirable characteristic?

Scalability allows the application to dynamically adapt to changing workload requirements without compromising performance or availability.

In this post, we'll understand the true meaning of scalability from different perspectives followed by the various techniques and principles that can help you scale the application's architecture. Also, in subsequent posts in the coming weeks, we'll take deeper dives into the scalability of each layer and component of a typical architecture.



A Crash Course on Architectural Scalability

ByteByteGo

What is Scalability

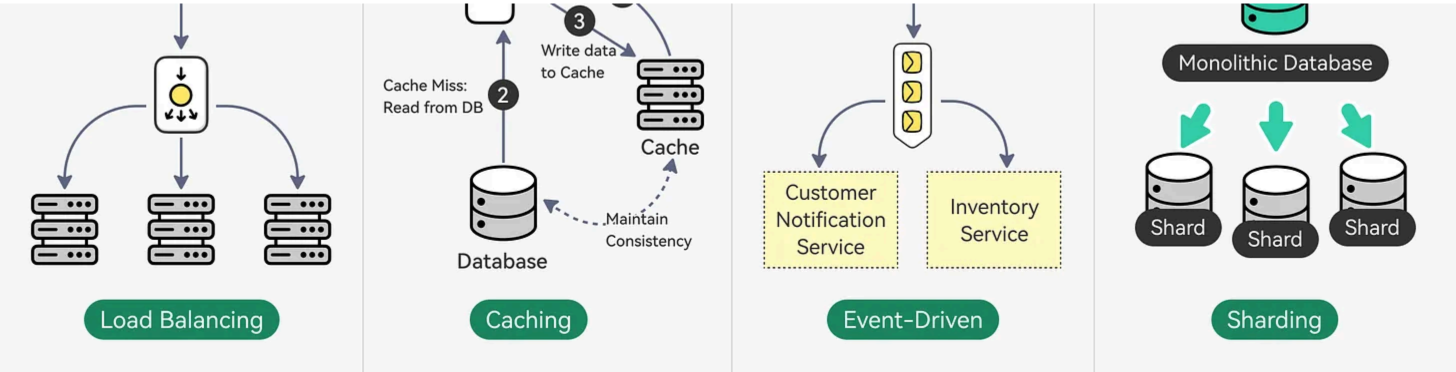
✓ Scalability is the ability of a system to handle an increased workload without losing performance

Bottlenecks to Scalability

1. Centralized Components
2. High Latency Components
3. Tight Coupling

No system is infinitely scalable

Scalability Techniques

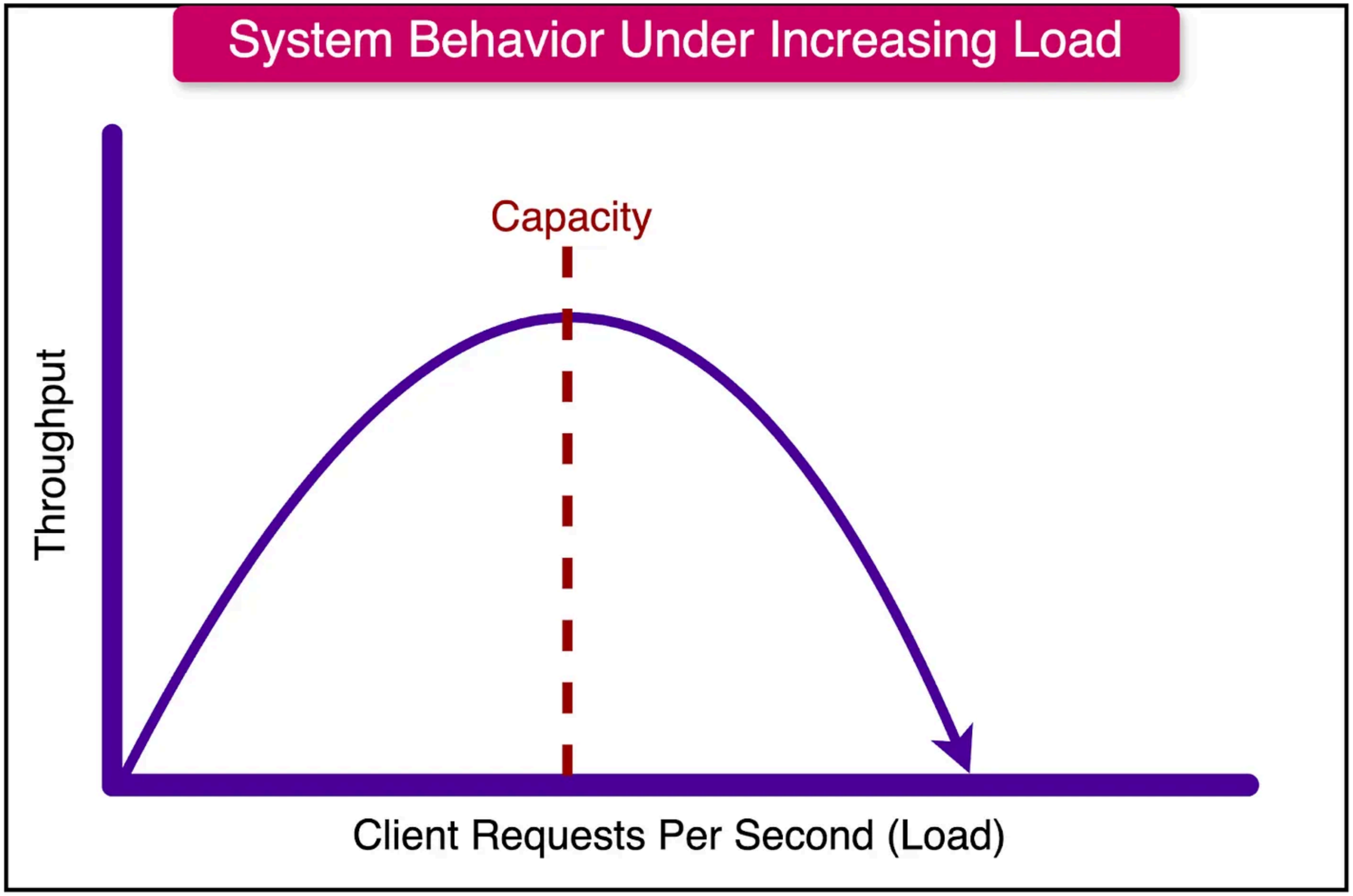


What is Scalability?

Scalability is a critical aspect of system design that determines how well a system can handle increased workload without compromising performance.

A system is considered scalable if it can accommodate growth by adding resources, such as additional servers or processing power, without experiencing a significant degradation in performance.

The primary focus for developers is to determine an interval of demand within which the system will perform at an acceptable level. This interval represents the range of workload that the system can handle without experiencing significant performance degradation. Beyond this level, the performance drops off sharply.



In real-world scenarios, developers of actively used systems continuously monitor the workload levels. They employ various monitoring tools and techniques to track key metrics such as:

- CPU utilization
- Memory usage

- Network bandwidth consumption
- Response times
- Throughput (requests per second)

However, there is another slightly overlooked aspect of understanding scalability. This aspect becomes clearer with another definition of scalability.

The Alternative Definition of Scalability

Scalability is a system’s ability to handle an increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity.

This definition shifts the focus from merely handling more work to the efficiency and effectiveness of the scaling approach.

Several important questions arise:

- If we add more processors to increase capacity, what is the method for coordinating work between the added processors? The coordination method plays a crucial role in determining the effectiveness of the scaling strategy.
- Will the coordination method consume additional processing cycles? If the coordination overhead is significant, the benefit of adding more processors can be diminished. In such cases, adding processors beyond a certain point may not be cost-effective for improving scalability.

By emphasizing the repeated application of a scaling strategy, this definition increases the awareness around choices and their long-term impact on the system's scalability.

For example, consider the following scenario:

Replacing an O(n^2) algorithm with an O(nlogn) algorithm can enable the system to process a larger workload in the same amount of time. In other words, improving the algorithm's efficiency enhances the system's scalability.

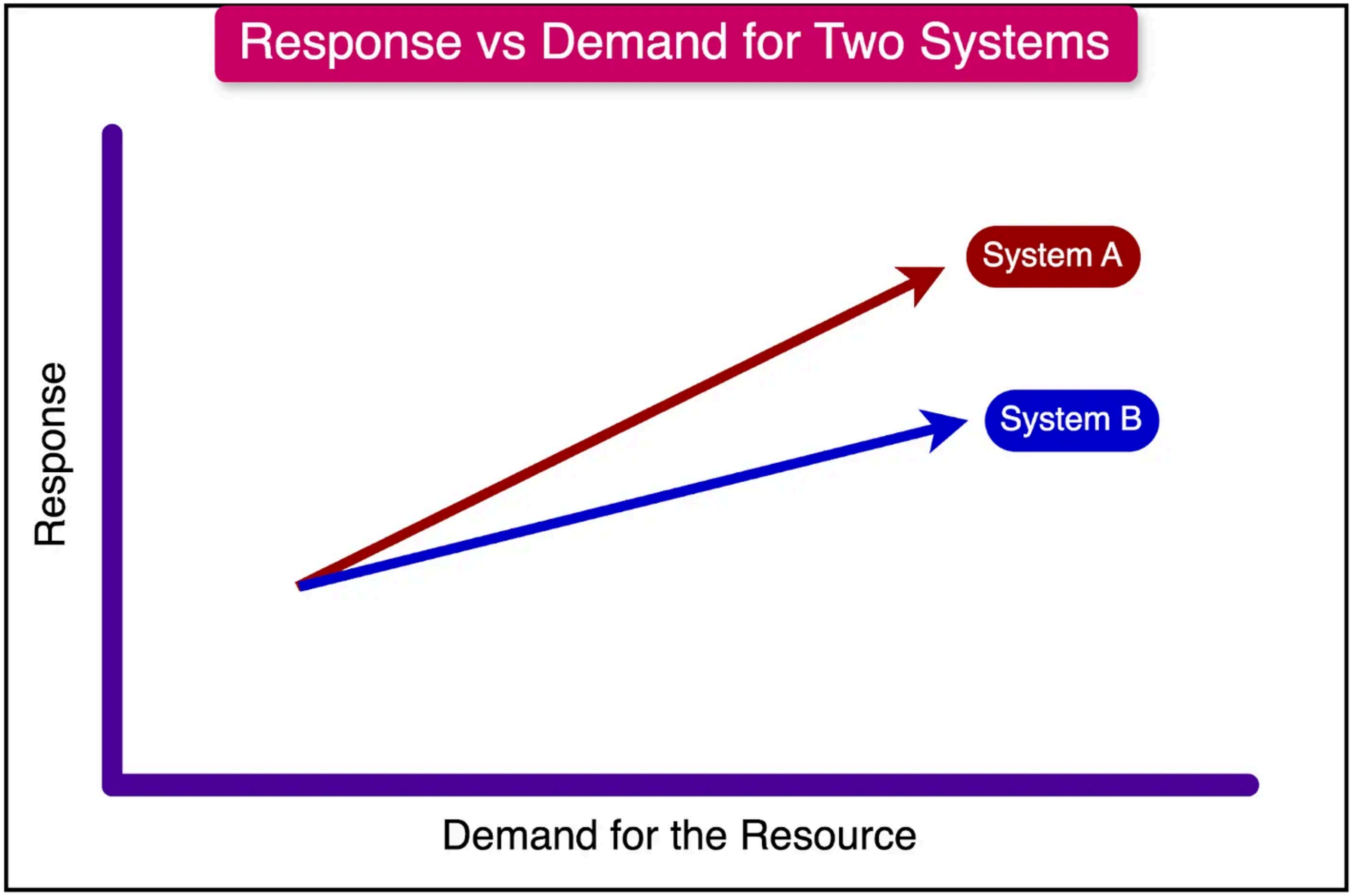
However, once we have implemented the most efficient algorithm, the algorithmic replacement strategy ceases to be effective. In other words, you cannot apply the strategy multiple times and boost a system's scalability.

Considering all this, there is another angle by which we can look at scalability.

Comparing the Scalability of Systems

When evaluating the scalability of systems, it is more meaningful to compare the scalability of two different systems rather than labeling individual systems as scalable or not. A useful approach is to analyze the response vs. demand curves of the systems under consideration.

Consider the example in the graph below, which depicts the response vs. demand curves for two hypothetical systems, A and B.



In this graph, we observe that for any given demand level, the response metric is worse for System A compared to System B. This means that System B can handle the same level of demand with better performance than System A.

If there's a maximum tolerable value for the response metric, System A will cross that threshold earlier than System B. In other words, System B can accommodate a higher level of demand before its response metric becomes unsatisfactory.

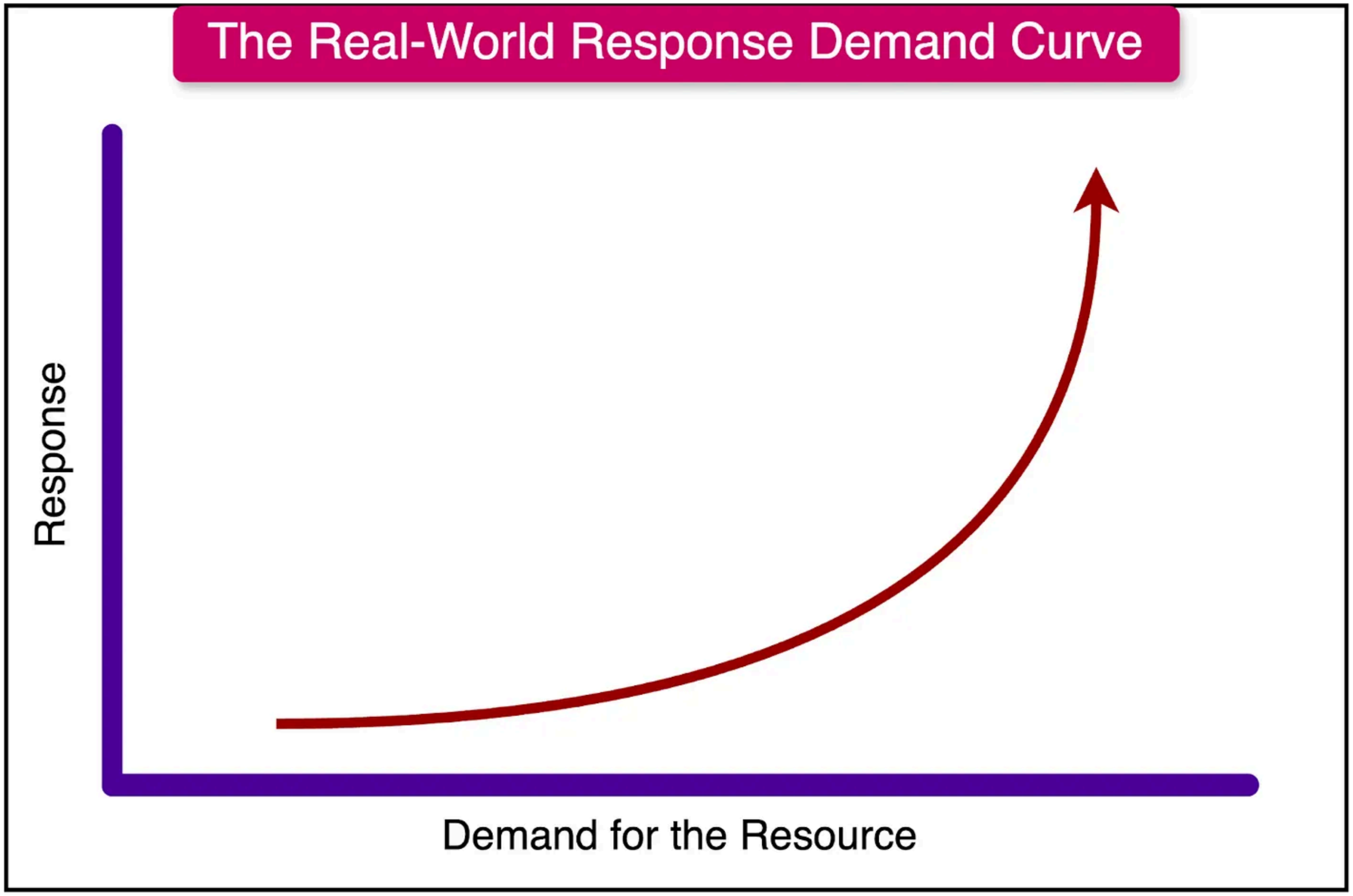
Based on this analysis, the conclusion is that System B is more scalable than System A.

Ultimately it's important to understand that no system is infinitely scalable.

If both response vs. demand curves continue to rise at the same monotonic rate, they will eventually reach a point where the demand for resources exceeds their availability. At that point, the response times will become unsatisfactory for both systems, albeit at different demand levels.

The point at which the response metric becomes unsatisfactory signifies the limit of the system's scalability. Beyond this point, the system cannot handle the increased demand without compromising performance.

In reality, systems do not always follow a monotonic rate of increase in the response vs. demand metric. The curve often exhibits a non-linear shape, as shown in the example below.



In this hypothetical system, the response metric tolerates demand increases quite well until it hits a significant design limitation. This limitation manifests as a knee-like shape in the response vs. demand curve.

After crossing a particular demand level, the response metric rapidly deteriorates, indicating that the system has reached its scalability limit. Any further increase in demand will push the system down the hill, resulting in unacceptable performance.

Therefore, an important goal of system designers to improve scalability is to keep the knee of the response vs. demand curve as far to the right as possible. This means designing the system to handle higher levels of demand before reaching its scalability limit.

Architectural Bottlenecks to Scalability

Two primary issues typically contribute to the formation of scaling bottlenecks:

Centralized Components

A centralized component in an application architecture refers to a single point of processing or control that cannot be easily scaled out. Such components act as a bottleneck, placing an upper limit on the number of requests the entire architecture or request pipeline can handle.

For example, consider a web application that relies on a single database server to handle all data storage and retrieval operations.

As the application grows and the number of concurrent users increases, the database server becomes a centralized component that limits scalability. It can only handle a certain number of requests per second, regardless of how many application servers are added to the system.

High Latency Components

High latency components are components within the request pipeline that introduce significant delays or slowdowns, putting a lower limit on the application's overall response time.

For instance, consider a web application with a complex data processing task as part of its request handling. This task may involve time-consuming operations such as data transformation, external API calls, or resource-intensive computations.

If this high latency component is executed synchronously within the request pipeline, it will delay the client response and limit the system's throughput.

Key Tenets of Building a Scalable Architecture

There's no such thing as a universal scalable architecture model. Systems and architectures evolve organically over time and it is difficult to have fixed principles for scalability that can work in all of those cases.

However, some key principles apply in most situations.

Statelessness

In the world of scalable systems, statelessness is a highly desirable property. A system without a state is much easier to scale as compared to a stateful system.

Statelessness means that the server does not maintain any client-specific state between requests.

Imagine having a web server responsible for serving incoming requests. If this server follows RESTful principles, it will be stateless. Each request contains all the necessary information for the server to process that request without relying on any stored context from previous requests.

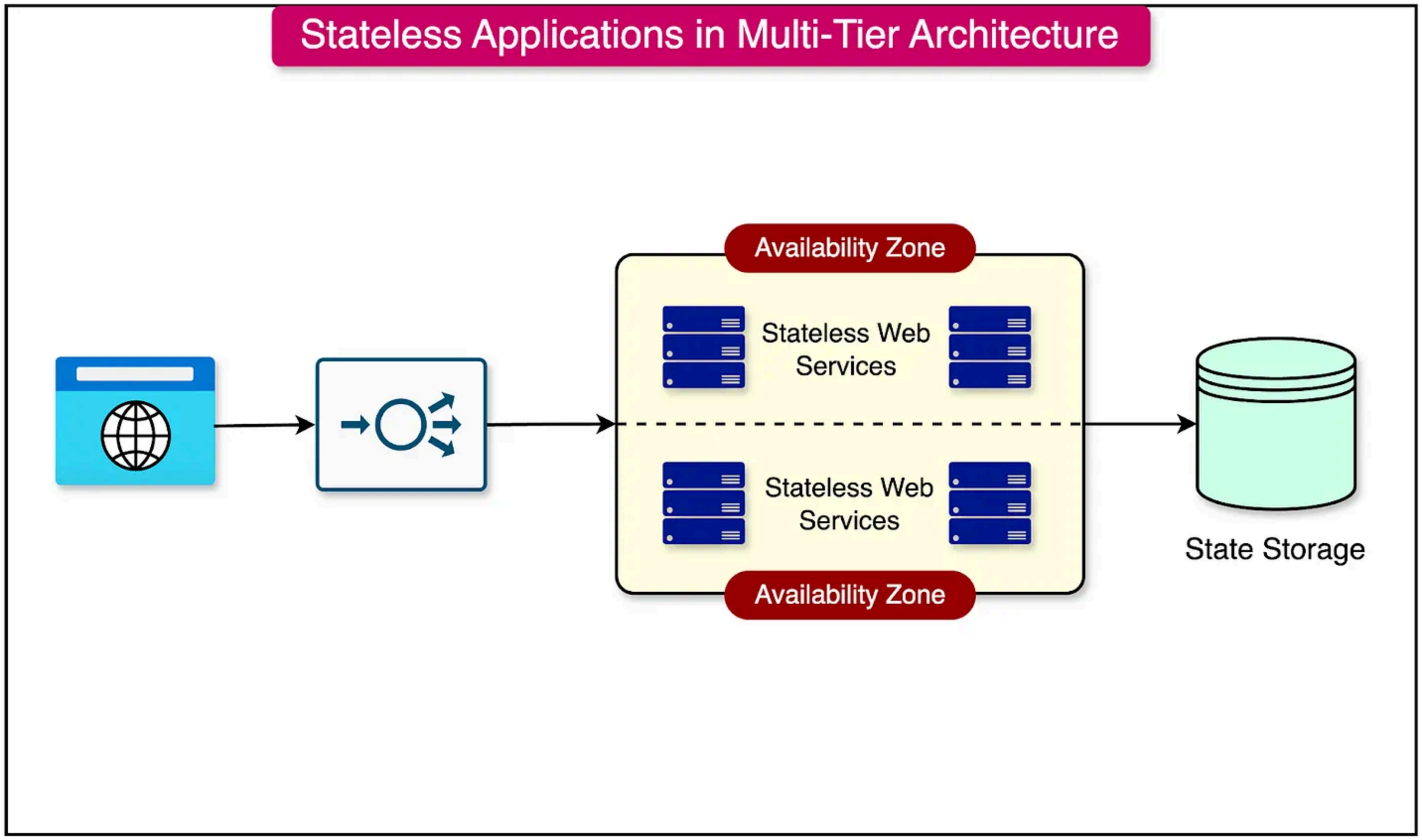
Stateless servers have several advantages:

- They are easier to scale horizontally.
- They are fault tolerant because one server instance going down does not impact the overall system.
- Stateless servers can be deployed and maintained independently since there is no need to maintain state consistency across instances.

While statelessness is ideal for scalability, it's important to note that most applications require some form of state management to provide meaningful functionality and user experience.

In a typical multi-tier architecture, the application is split into different layers, such as the presentation layer, backend layer, and data layer. The key is to keep the state in the data layer as much as possible while keeping the other layers stateless.

The diagram below shows such an architecture for reference.

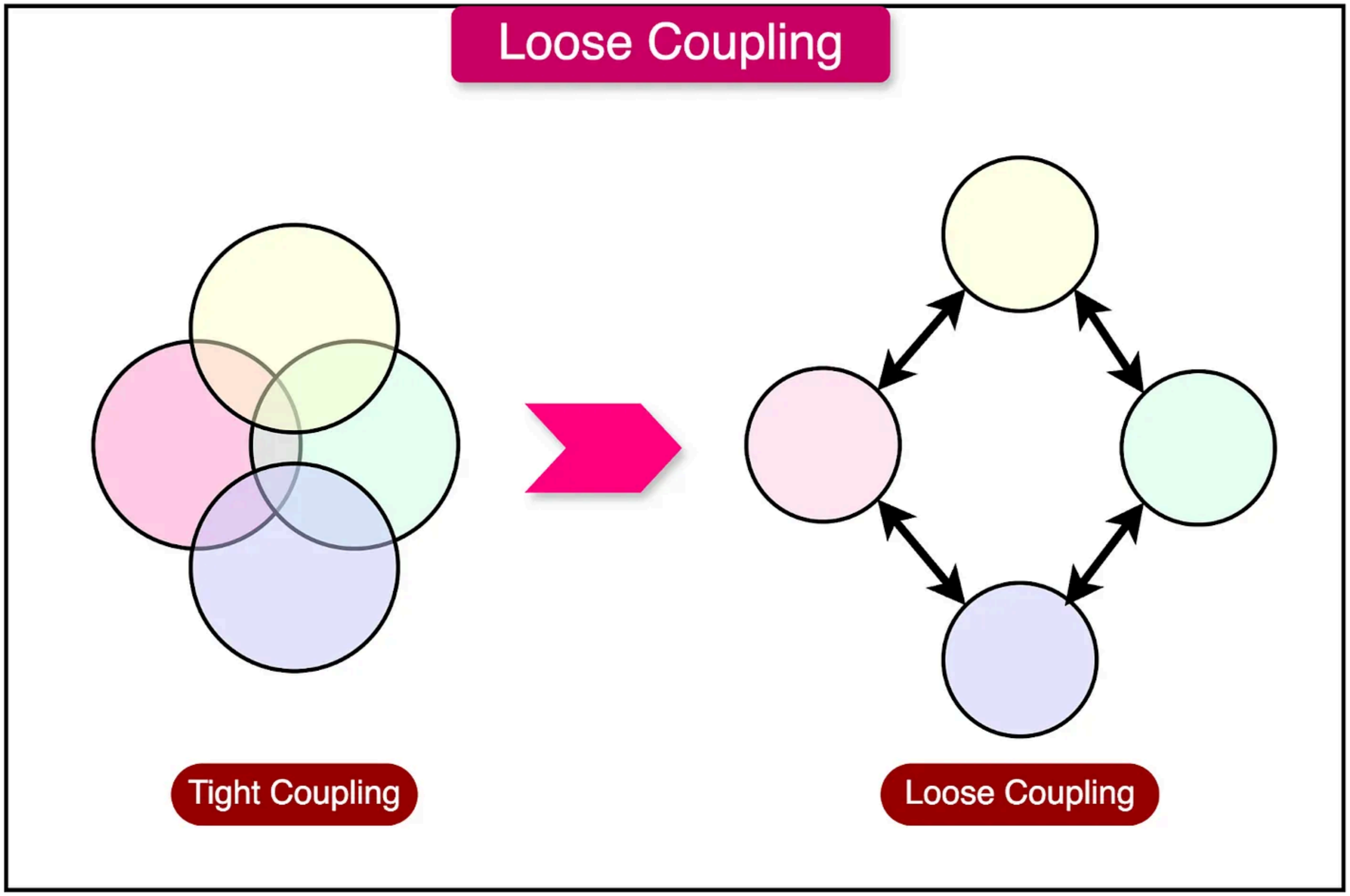


Loose Coupling

In a system with loosely coupled architecture, individual components can be modified or replaced without significantly impacting their dependencies. This modular approach offers several benefits, particularly in the context of scalability.

When components are tightly coupled, changing one component often requires extensive modifications to its dependencies. This tight interconnectedness can lead to a ripple effect, where a single change requires updates across a large portion of the application.

In contrast, loose coupling allows for targeted, granular changes to individual components without the need to rework their dependencies.



This modularity is especially valuable when it comes to scaling a system. By isolating the scalability requirements to specific components, developers can focus efforts on optimizing and scaling only those parts of the application.

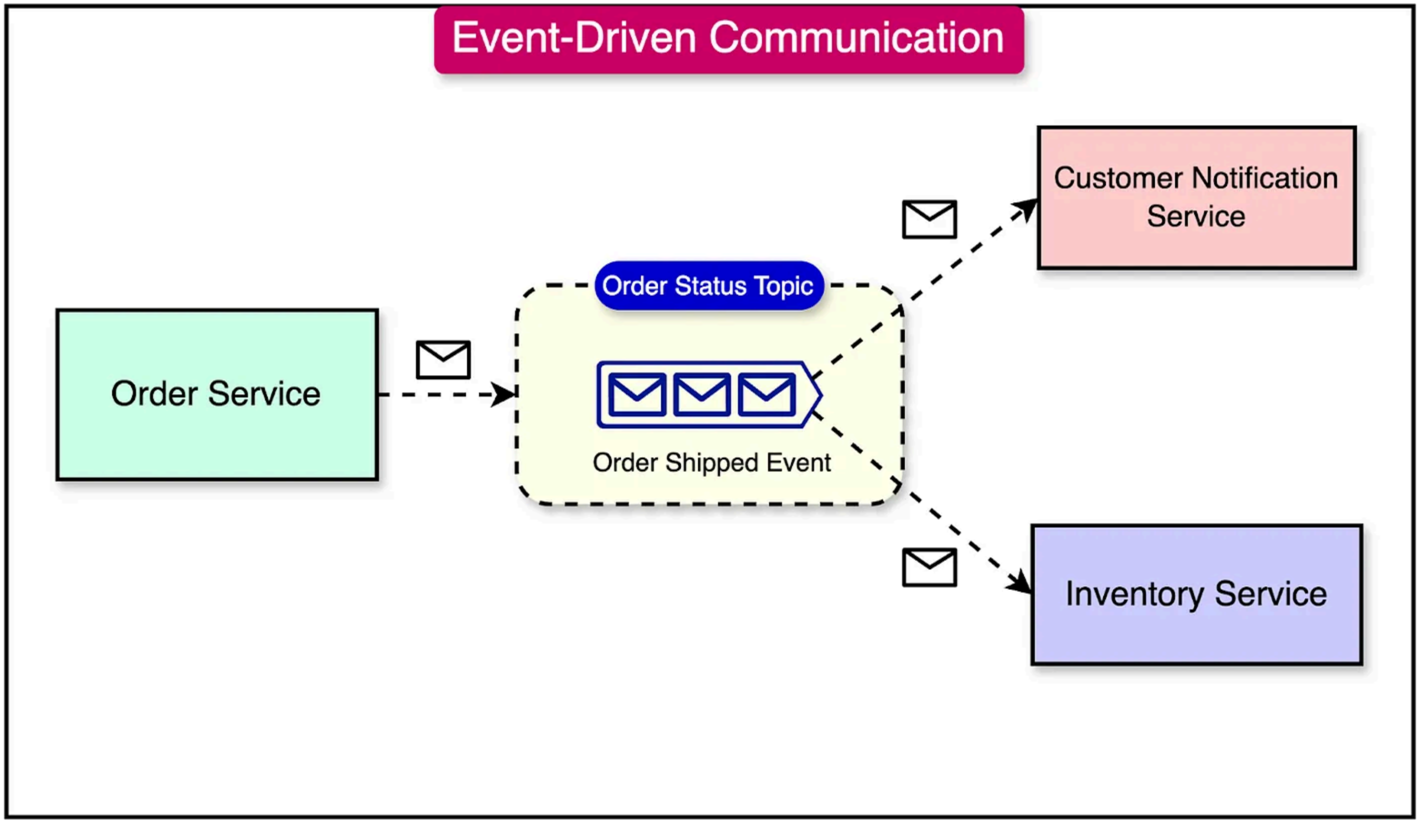
Asynchronous Processing

While breaking down a monolithic application into microservices offers benefits like modularity and scalability, the communication between these services can introduce dependencies such as:

- Tight coupling
- Synchronous Communication
- Cascading Failures

To mitigate these challenges, one solution is adopting an asynchronous communication approach using events. This pattern is known as event-drivrchitecture.

See the diagram below:



Choosing the Right Scaling Strategy: Vertical vs Horizontal

When selecting a scaling strategy, the first decision often involves choosing between vertical and horizontal scaling.

Both approaches have their use cases and benefits, and selecting the right strategy is crucial for efficiently managing resources without overuse or waste.

Vertical Scaling

Vertical scaling, also known as scaling up, involves increasing the capacity of a single resource. This can be achieved by upgrading to a larger server or instance size, providing more processing power, memory, or other resources within a single instance.

This approach is useful in the following scenarios:

- When the workload cannot be easily divided into smaller parts or the application architecture doesn't support horizontal scaling, vertical scaling is a suitable option.
- When the workload requires significant processing power, memory, or other resources, vertical scaling can provide the necessary capacity within a single instance.

Horizontal Scaling

Horizontal scaling, known as scaling out, involves adding more instances or resources to distribute the workload across multiple servers.

Instead of increasing the capacity of a single instance, horizontal scaling allows you to handle increased traffic or workload demands by distributing the load across multiple instances.

One of the key benefits of horizontal scaling is the ability to distribute instances across multiple Availability Zones (AZs). If one instance or even an entire Availability Zone experiences an outage, the other instances in different AZs can continue serving requests, minimizing the impact on the overall system.

This results in:

- Improved resiliency
- Increased capacity

To fully leverage the benefits of horizontal scaling, the application should ideally support a stateless design. By decoupling the state from the instances, the system can seamlessly scale out the application layer without worrying about synchronizing the state across instances.

It's better to opt for horizontal scaling when it is possible to divide the workload into smaller parts that run independently. This is usually the case with cloud-native applications that run on multiple nodes.

Techniques for Reliable Scalability

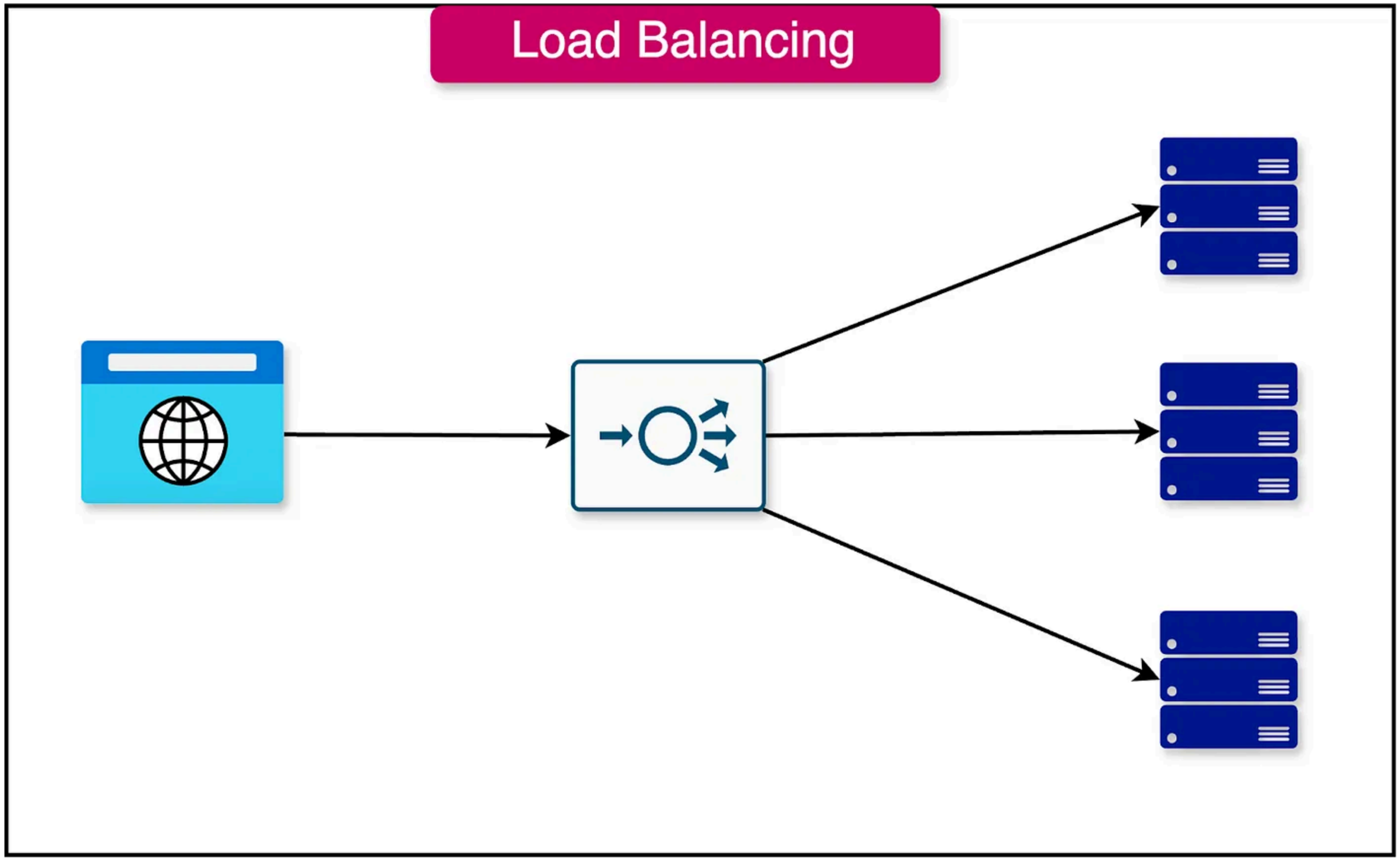
After exploring the fundamentals of scalability, let's look into some tried-and-tested techniques that can help you design and implement scalable systems.

Load Balancing

In a horizontally scaled system, one of the primary challenges is ensuring an even distribution of the workload across multiple servers. Without proper workload distribution, some servers may become overwhelmed while others remain underutilized, leading to suboptimal performance and potential bottlenecks.

Load balancing is a technique that addresses this challenge by intelligently spreading user requests across multiple nodes in the system. By evenly distributing the workload, load balancing ensures that no single server becomes a performance bottleneck.

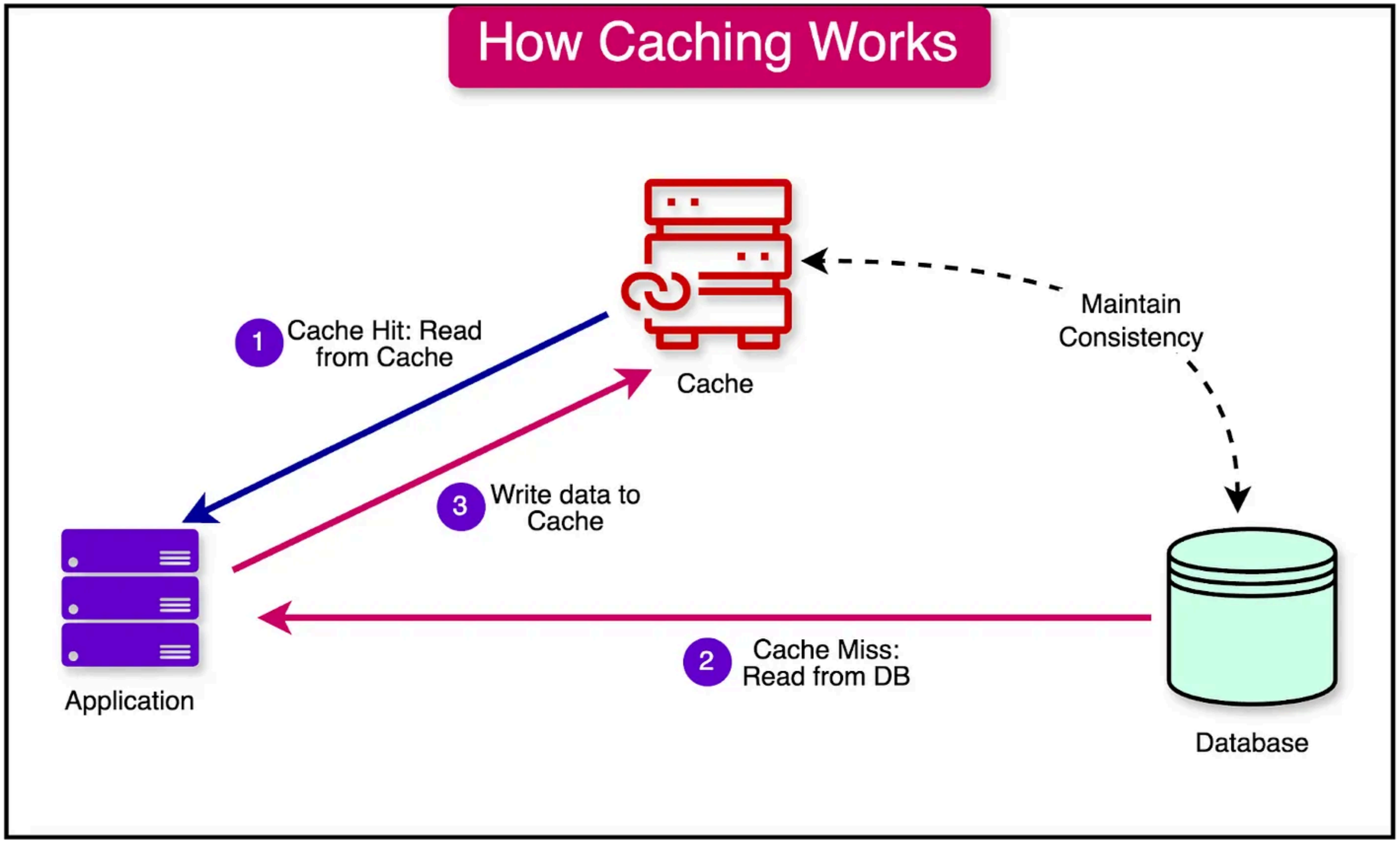
See the diagram below that shows a load balancer distributing traffic to multiple server instances.



Caching

Caching is a powerful technique that can significantly scale the system while enhancing performance.

By keeping the most commonly requested information in memory, caching reduces the need for time-consuming and resource-intensive data retrieval operations, leading to faster response times and improved system efficiency



Different levels of caching can be implemented at various levels of a system:

- **Client-Side Caching:** Involves storing data on the client's device, such as the web browser or mobile app.
- **Server-Side Caching:** Refers to the practice of storing frequently accessed data (data, HTML, or computational data) on the server itself.
- **Distributed Caching:** Involves using a separate caching layer that is shared across multiple servers or nodes in a distributed system.

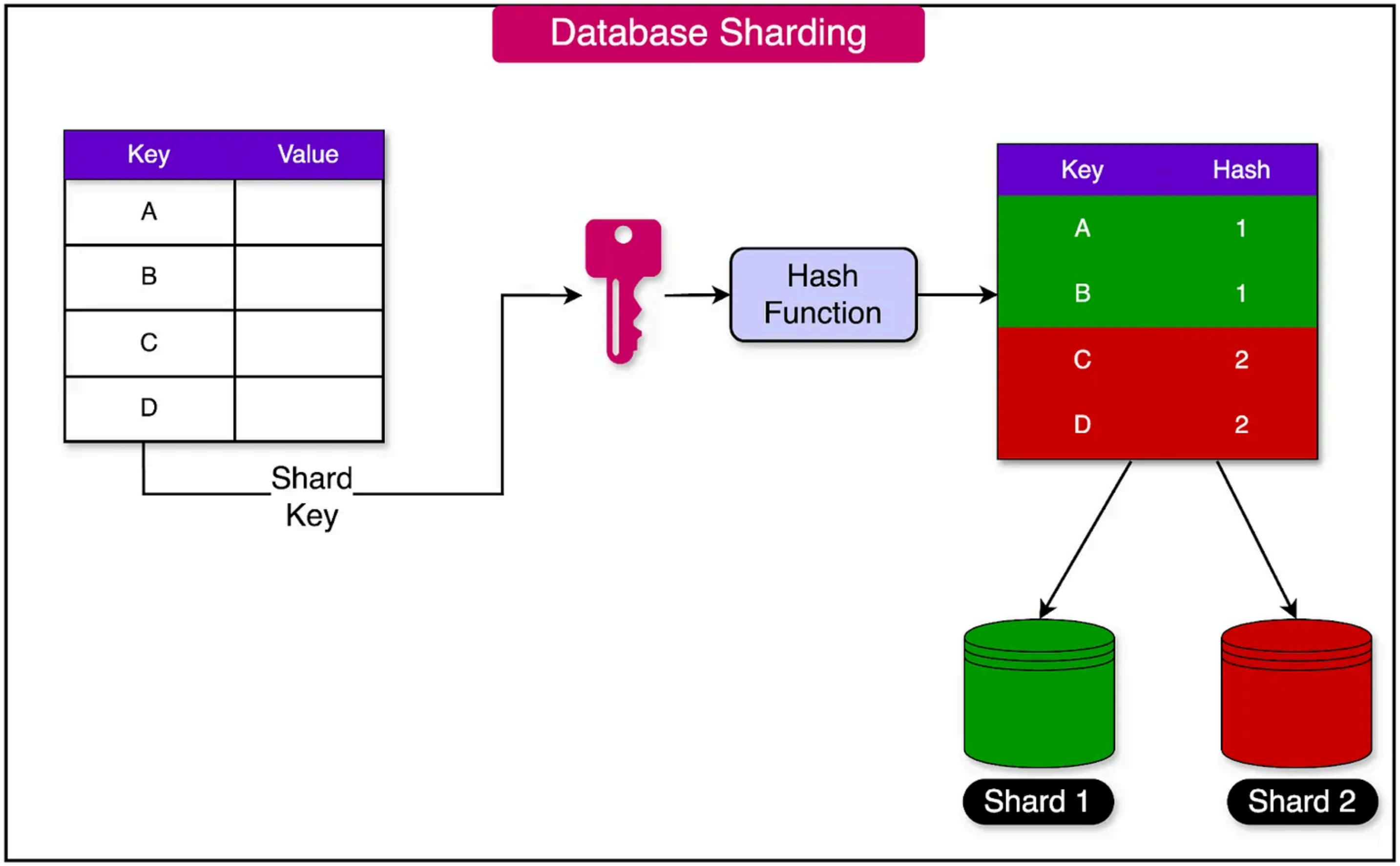
Sharding

As the data in a system grows, the performance of queries and data operations can degrade as the dataset expands, leading to slower response times and reduced scalability.

Sharding is a technique that addresses this challenge.

It involves splitting a large dataset into smaller, independent subsets called shards. Each shard is typically stored on a separate server or node, allowing the data to be distributed across multiple machines.

The diagram below shows database sharding using a hash function to distribute records into different shards.



Key	Value
A	
B	
C	
D	

Key	Hash
A	1
B	1
C	2
D	2

Queries and data operations are directed to the appropriate shard based on a sharding key or a partitioning strategy. This allows the system to parallelize data processing and distribute the workload across multiple nodes.

Avoid Centralized Resources

It's important to avoid using a single, centralized resource for the workload and distribute it across multiple resources for better scalability, fault tolerance, and performance.

Some specific scenarios worth considering are as follows:

- In a queue-based load leveling, consider partitioning the workload across multiple queues to distribute the processing load.
- Break long-running tasks into smaller ones that can scale better by taking advantage of parallelism.
- Design patterns such as fan-out, pipes, and filters can help avoid the creation of centralized resources in workflows.

Modularity

Modularity is a fundamental principle for managing complexity and enhancing scalability, security, and maintainability.

Modular design is a commonly used approach in modern application development, where an application's software is constructed using multiple, loosely coupled building blocks called functions or modules

The key idea behind a modular design is to promote the separation of concerns by establishing clear boundaries among the different components of the architecture. Each module encapsulates a specific functionality and communicates with other modules through well-defined interfaces or APIs.

Modular design helps avoid the pitfall of "fate sharing," which is commonly associated with monolithic architectures. In a monolithic application, all components are tightly coupled and deployed as a single unit, often running on a vertically scaled server. The problem with this approach is that any failure or issue with the server affects the entire application.

Summary

In this article, we've learned the fundamentals of architectural scalability and how it can help design scalable systems.

Let's summarize the learnings in brief:

- A system is considered scalable if it can accommodate growth by adding resources without compromising performance.
- Another way to look at scalability is in terms of the strategy. Scalability is the system's ability to handle an increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity.
- The main bottlenecks for scalability are centralized components and high latency components.
- Three key tenets to build a scalable system are statelessness, loose coupling, and asynchronous processing.
- The first choice of scaling strategy is between vertical and horizontal scaling.
- Some common techniques for reliable scalability are load balancing, caching, sharding, modularity, and avoiding centralized resources.

Reference:

- On System Scalability

253 Likes · 13 Restacks

Comments

Write a comment...