American Journal of Computer Architecture 2024, 11(5): 66-74
DOI: 10.5923/j.ajca.20241105.03

Impact of Scalability in Computer Architecture

Pradeep Jain

Principal Application Architect, Discover Financial Services, Pittsburgh, PA, USA

Abstract Whenever a software architect designs an application, then he/she needs to first and foremost understand and plan for functional requirements of the software. But nonfunctional requirements like Scalability, availability or security hold the same importance without which software application will lose its shine as customers wouldn't be able to perform intended functions as desired. The following article would throw light on the ways software applications can be made scalable so that the software application can grow itself as the customer base grows.

Keywords Kubernetes, Vertical scaling, Horizontal scaling, Replication, SOA, Monolithic, Microservices

1. Introduction

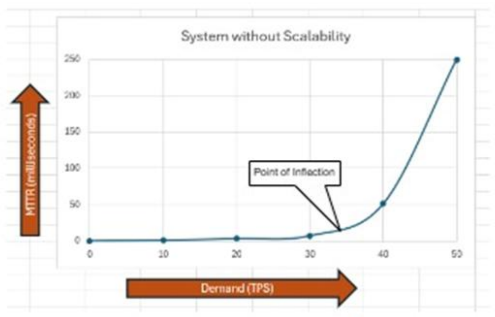

Figure 1. Shows software MTTR behaviour with demand for Non scalable systems

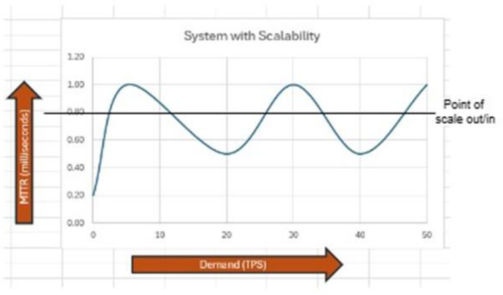

Figure 2. Shows software MTTR behaviour with demand for scalable systems

Most operating businesses in today's world operate using software applications that have been designed and developed based on the business functional requirements. For Ex: a financial institution would have various payment applications like to make ACH, Wires, Zelle payments for its customers. At first, the payment application could have been designed for a limited number of concurrent customers but what if customers making payments grow during festive season or due to bank popularity. If the application has not been thoughtfully designed to be scalable then elevated payments will lead to application performance degradation and hence would impact the business. Scalability is the property of software so that it can adapt to increase or decrease the application traffic leading to higher stability and availability. In nutshell, scalability is the ability of a system to handle an increased workload by adding resources without losing performance. This article will dive through the options through which a software application can be architected to be made scalable leading to operational resiliency of the applications and improving the software performance during unanticipated higher traffic loads. The graph below explains that MTTR (Mean Time to Response) increases as load increases if system is not scalable. Conversely, the MTTR remains cyclical if the system is scalable.

2. Type of Scalabilities

There are two types of scalabilities which can be implemented by software systems. One of them rely on increasing the computing power on single machine while other relies on increasing the number of machines for carrying the software intended functions.

2.1. Vertical Scalability/Scaling up

Vertical scalability, also addressed as Scaling-Up, is achieved by increasing the computational power of the underlying hardware (e.g. Database, Memory etc.) and platform on which application is hosted i.e. increasing the capacity of a single instance (e.g., adding more CPU, RAM, or storage).

* Corresponding author:
pradeep3j@gmail.com (Pradeep Jain)
Received: Nov. 3, 2024; Accepted: Nov. 18, 2024; Published: Nov. 22, 2024
Published online at http://journal.sapub.org/ajca

As an illustration, banking payment application would need to have the underlying relational database management system (RDMS) in which records of the customers along with payments are stored. If the traffic increases and the database struggles to handle the elevated workloads then the read/write efficiency of the database will decrease and therefore, the database storage, CPU processors and memory (RAM) will need to be increased so that application users doesn't face unexpected delays. Let’s say bank web application was running on a server with 4 CPU cores and 8GB of RAM earlier. To address increase in traffic, an Architect might decide to vertically scale database servers by upgrading them with 8 CPU cores and 16GB of RAM.

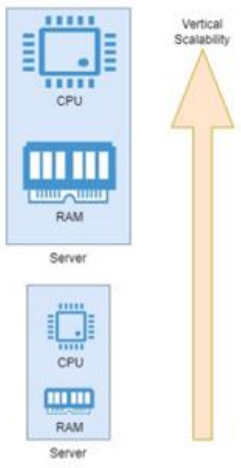

Figure 3. Vertical scalability representation on single machine

2.2. Horizontal Scalability/Scaling out

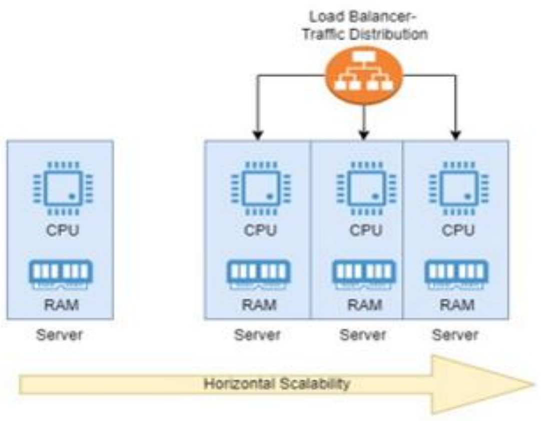

Figure 4. Horizontal scalability representation via multiple machines

Horizontal scalability, also addressed as Scaling-out, is achieved by diverting the application traffic to multiple servers of generally similar configurations so that the traffic can be distributed equally among multiple servers as needed i.e. adding more instances to handle increased load, distributing the workload across multiple servers. The horizontal scalability also requires a load balancer which has the responsibility of distributing the traffic based on the server availability and its performance. As an illustration, banks will need to have a web application deployed on web server that will accept the payment requests from users. If the workload increases due to elevated users, then another web server can be launched which will start accepting the traffic so that instead of one server, now its two servers who will accept the traffic leading to better response times to users. Scaling out provides resiliency because if one server fails then you still have other servers to take the load and do the function. Resiliency is ability to function if one or more pieces fail.

3. Implementation of Scalability

There are different ways of implementing scalability. The implementation of the approach taken by architect depends on the IT architecture of the organization as explained below in each strategy.

3.1. Microservices Architecture

Modular architectures like Microservices and SOA (Service Oriented Architecture) guides the design methodology through which each logical business functionality would be defined as an individual service and will be deployed independently so that each individual service/functionality can be scaled up/down as needed. This design methodology also makes sure that if one service is down then it does not bring the whole system down. For ex: a bank can have independent services of makePayment, retrievePayment, modifyPayment, cancelPayment etc so that if makePayment is getting failures then it should not impact the customer cancelling the payment. The deployment of services can be done using Kubernetes environment which would automatically scale up and scale down the pods making software application highly adaptive. There are also challenges that are faced with microservice architecture which are data consistency among service databases, increased network overhead and complexity of managing distributed systems.

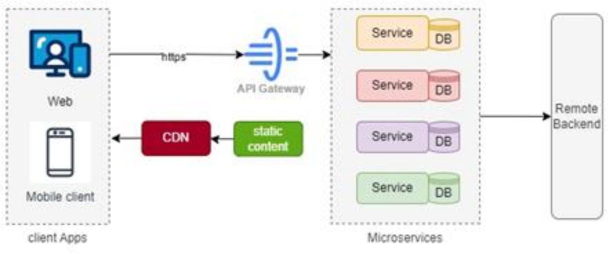

Figure 5. Microservices Architecture

3.2. Load Balancing

The load balancer is placed behind the webservers and distributes application traffic among them so that one specific server doesn't get overburdened and failure of one server doesn't bring down the whole system. In software technology architecture, an architect should place in load balancer to evenly distribute the traffic. The load balancing will make software architecture scalable as if workload increases, then another server will be provisioned which would start taking requests via load balancer making system more available and stable. The load balancer can adopt various algorithms like Round Robin, least connections, geographical or performance-based metrics to choose the server.

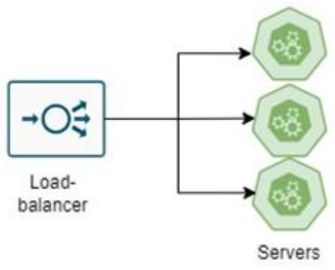

Figure 6. Load Balancing between multiple servers

3.2.1. Round Robin Algorithm

Round Robin algorithm is the simplest and widely adopted algorithm in which requests are distributed among servers in a sequential manner. This algorithm assumes that all servers are operating in similar capacity and doesn't account the current workload on any server before sending out the request to them. If the server is not accepting the requests due to being nonoperational at all then load balancer will not send out request to that server. This algorithm is successful if all servers are equally capable and processing requirements of each request is predictable.

3.2.2. Least Connection Algorithm

Least connection algorithm is the strategy to divert the request traffic to the server which has the least active connections. This algorithm assumes that the server having the least connections are most available and therefore, can handle the requests efficiently. This strategy also doesn't take in account the processing power of each server though its mostly used in the situations where traffic is unpredictable.

3.2.3. Geographical Algorithm

Geographical algorithm adopts the strategy of diverting the traffic based on the geography of the originator of the request. This is used in the situations where servers are distributed across various regions in the worlds. This strategy assumes that if originator is from specific region (i.e. Europe) then the closest server would be the best option to handle the request. This strategy doesn't look out for the current workload on the server before traffic diversion.

3.2.4. Weighted Distribution Algorithm

Weighted distribution algorithm considers the server capacity before sending the request which was lacking in Least Connection algorithm. This strategy requires a prerequisite to assign weight parameter to each server based on the server capacity. If server capacity is higher then assigned weight will be higher too. The requests will be diverted based on the relative weight of the server.

3.2.5. Sticky Session Algorithm

Sticky session algorithm is used in applications where the request will need to be forwarded to the same server which has served the earlier request. This is very common need as its widely used in the applications like E-Commerce sites which needs to hold user session for the cart management or also used in banking where external account needs to be added on to the customer profile and there is session holding the information for validation of customer identity.

3.2.6. IP Hash Algorithm

The IP Hash algorithm uses the IP address of the client and server for determination of server to handle the request. Adopting this algorithm means that if the client is same and number of servers remains same then request from that specific client will always be processed by same server. This can also be beneficial for applications that need to maintain state between requests. However, it can lead to imbalances if a large number of requests come from a small number of IP addresses. Therefore, it may not be the best choice for environments with a wide distribution of client IP addresses.

3.3. Asynchronous Architecture

Asynchronous architecture suggests using decoupled architecture methodology so that components operate with each other using queues or notifications. If the workload increases, then the requests will be parked in queue so that provider systems don't get overwhelmed. This architecture can be followed only for situations when a live consumer is not waiting for a response. For Ex: application to receive credit card shipping updates can be implemented using asynchronous architecture. Serverless messaging systems like Amazon SQS and Amazon SNS are frequently used to support high scalability and are considered one of the best solutions when workload is unpredictable. The challenges that are faced with Asynchronous architecture are error handling, debugging and data consistency.

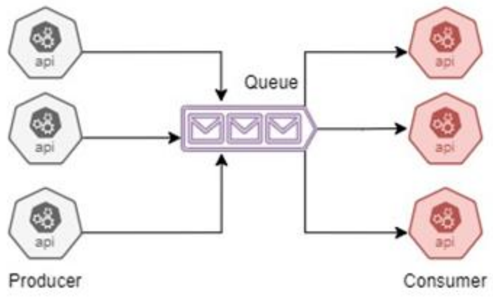

Figure 7. Asynchronous Architecture

3.4. Database Replication

Any database would offer read and write operations, and to make systems scalable, the database read replica can be created which would serve read only traffic. The application logic ensures that all read only queries are directed to read-replica and all database updates are directed to master/ primary database which are then replicated asynchronously to read replica. For Ex: all business reporting and data ware housing workloads can be shifted to read replica while master database can be used only for active production work loads. Data replication is the process of copying the same data across multiple locations or systems to ensure availability, fault tolerance, and scalability. However, each replication strategy—whether real-time (synchronous), near real-time, or scheduled (asynchronous)—introduces its own set of challenges and considerations.

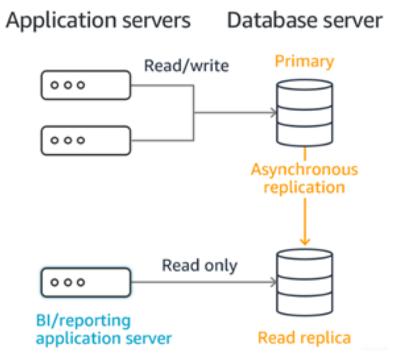

Figure 8. Database Replication for read queries and high availability

3.4.1. Synchronous Replication

Synchronous replication involves instantly copying data from the primary node to the read replica as soon as a write operation occurs on the primary node. This strategy is used by applications requiring immediate failover capabilities and low Recovery Time Objectives (RTO) to prevent data loss. However, it also means that a write operation cannot be completed until the update is mirrored in the read replica, introducing latency to the system. While this approach increases latency, it ensures strong consistency, meaning every read operation—regardless of the node—returns the same result. Example of applications needing synchronous replication are financials as balance inquiry should always result the consistent response to the user independent of the accessed node.

The benefits of strong consistency which is achieved by synchronous communication are simpler application logic, data durability and consistent data across systems.

The drawbacks of strong consistency are reduced availability, higher latency, and network intensive.

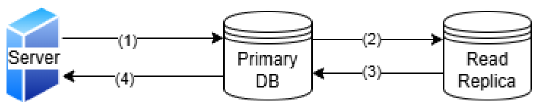

Figure 9. Synchronous Replication

3.4.2. Asynchronous Replication

Asynchronous replication involves copying data from the primary node to the read replica after a delay. This allows the application performing the write operation to receive an immediate response, as the primary node does not wait to update the read replica, resulting in lower latency. However, this approach comes at the cost of data consistency: the delay in updates means that data on the primary node may differ from that on the read replica. Consequently, this replication strategy offers eventual consistency, meaning that at any given moment, the same request might yield different responses depending on the node accessed. Example of asynchronous replication are property listing sites or social media sites which can operate with eventual consistency.

The benefits of eventual consistency which is achieved by asynchronous communication are simple implementation, high availability, scalability, and lower latency.

The drawbacks of eventual consistency are potential data loss, higher data conflicts, and higher data inconsistency across multiple systems.

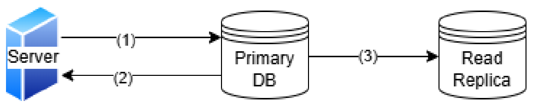

Figure 10. Asynchronous Replication

3.5. Database Sharding

Database sharding is the concept through which data is distributed among various smaller databases called shards. This improves scalability as data is divided among various servers which would eliminate the overwhelming of one specific database leading to higher availability, efficiency and stability. Sharding also needs the record keeper who will decide where the traffic needs to be routed based on the key.

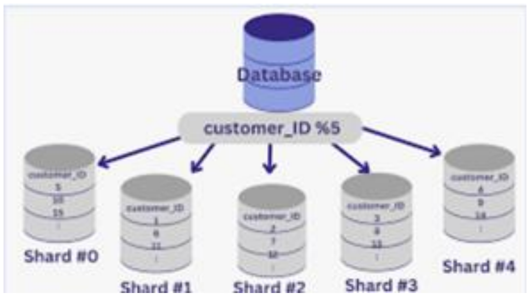

Figure 11. Database Sharding for distributed load

There are certain challenges that are associated with database sharding.

3.5.1. Uneven Distribution

There is possibility of a particular database i.e. shard receiving higher load than others due to incorrect sharding logic or the data. For Ex: In above image, if 80% of records are having customer_id%5 as 0 then 1st shard will receive 80% of traffic causing uneven distribution. The solution would be to optimize the sharding logic based on the data.

3.5.2. Functional Complication

The functional aspect of the application gets complicated with sharding because of multiple databases having data. Instead of one single database, the application owner will need to retrieve data from multiple shards and then have to merge before generating the response. With emergence of cloud capabilities, the functional complications have reduced but still not to the limit of making them non-noticeable.

3.5.3. Cost

The shards are expensive to maintain. Each shard is a standalone database and therefore, the cost to provision and maintain the database rises exponentially as data size increases and it gets more diversified. With more diversified data, the architect will need to bring in more shards and thus cost will increase too.

3.5.4. Complex Replication

The shards are relatively difficult to be replicated. In case of single database, the read replica is relatively easier to create and manage rather than of multiple databases as is the case in sharding. Therefore, the complexity of creating and managing multiple read replicas of all shards becomes too complex.

3.6. CDN and Caching

Content delivery networks (CDN) and caching must be used, wherever applicable, in software architecture to make the system scalable. Using CDN improves scalability by caching and delivering content from servers that are geographically closer to users, reducing latency and improving performance. Caching improves scalability by keeping frequently used data in memory eliminating the need to access the original source of the data thus improving latency of the application.

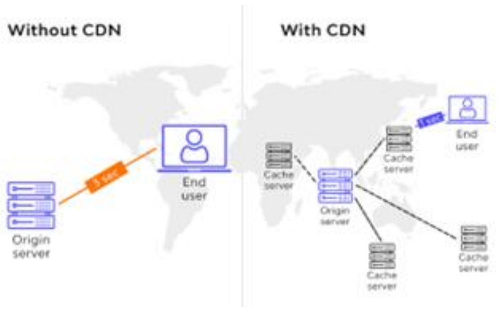

Figure 12. CDN representation

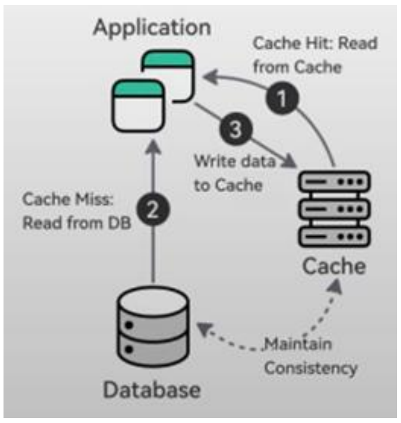

Figure 13. Caching representation for Cache-hit and Cache-miss

3.7. Fanout and FanIn

If the process is long running, then divide them into short, smaller independent tasks (Fan-Out) and run them parallel to collect back the responses (Fan-In) and deliver the functionality. The key is to allow the processing of functions in parallel which means to spun them per need to execute the activities, and at the end of processing the orchestrator waits for all functions to finish.

3.8. Right Platform and Sizing

The foundation of scalability is to choose the right platform for software applications. For Ex: If the application doesn't require strict schema, then NoSQL databases is the appropriate choice and that can be scaled out very efficiently by spreading the storage of data and the work to process the data over a large cluster of computers. Another example is to use Kubernetes platform using any cloud provider (GCP, AWS, Azure) that can make applications highly scalable without organization's worrying about underlying hardware.

4. Bottlenecks of Scalability

There are architectural mistakes that should be avoided while designing the software system as it leads to degraded software performance on elevated loads.

4.1. Centralized Components

Centralized components should be avoided while designing the system, like single database handling all requests independent of number of customers using the system. No matter how many instances are increased for Banking Card Application, the response time for customers will have latency due to central struggling database. However, sometimes centralized components are necessary due to business or technical constraints. In such cases, we need to adopt optimization strategies of Caching or database replication for routing of read only queries.

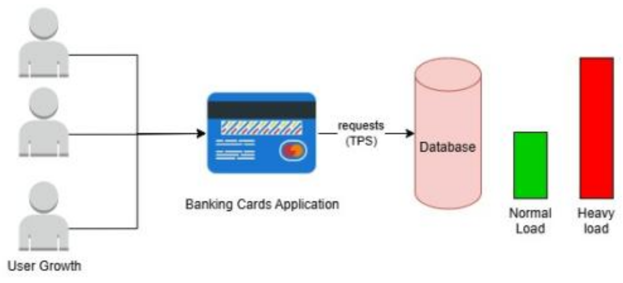

Figure 14. A sample application having centralized database

4.2. High Latency Components

If the system has high latency components, then it would be a blocker for scalability. In those cases, use caching or CDN. For Ex: if the component is defined to stream the video or bring images then it's worth to use CDN to cater the requests from the server closure to user.

4.3. Tight Coupling

Tight coupling is the nightmare for making systems scalable. If system components are created with dependencies among each other then it would hinder the possibility of scaling a specific component that's have unique demands. This modularity between system components is important for scalability because it allows us to scale specific parts of the system based on the usage. The ways to handle tight coupling are to adopt microservices architecture or event driven architecture.

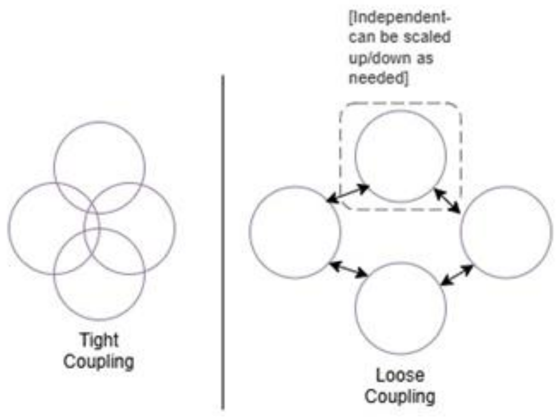

Figure 15. Tight vs Loose coupling

5. Benefits of Scalability

There are various architectural benefits that get realized if architect adopts the modern software architecture scalability principles.

5.1. Cost Efficient

Imagine a situation when an organization must shell out millions of dollars upfront on infrastructure considering its application usage of highest capacity while the application usage might not as high as expected. If the software application sizes itself based on the need of usage, then the cost can be kept minimal and to the most optimum level. For Ex: As payments grow during festive seasons or holidays, the application can size up itself to cater to elevated demand and as and when traffic reduces, then application can size down and thus bringing cost down.

5.2. High availability

The software application would be highly available if it can adapt itself based on the peak usage. The application workload due to high demand can be distributed among various servers, with horizontal scalability, so that the application remains responsive and available. If application wouldn't calibrate itself during peak usage, then it will bring risk of revenue loss due to downtime and can leave customers dissatisfied resulting in degraded future growth potential. Keeping high availability is critical and vital for any business.

5.3. Future Proofing

If software application architecture at first has been laid out of making application scalable, then organization can avoid future expenses which would be needed for redesigning and upgrades. The future proofing also provides competitive advantage to organizations, as they can quickly scale their software based on dynamic market demands.

5.4. Resource Optimization

The scalable design ensures that application efficiently allocate resources which leads to better utilization and reduce wastage. This also means that other software applications of the organization can leverage the existing and unused resources leading to better resource allocation.

6. Stateful and Stateless

The discussion for scalability would not be complete unless stateful and stateless design patters are touched upon. In stateful interaction, the server keeps track of the previous requests from the client and hence the response behaviour depends on the earlier request from the client. In contrast, the stateless interaction doesn't track any information from previous requests and each request is treated independently.

In traditional stateful applications, the client session data is stored on the server and therefore, it would be a mandate to divert the request from same client to same server as client session is stored on specific server only. The stick session algorithm, explained above, takes care of this requirement. Since the stateful applications has the dependency to be directed to same initial sever that served the request has scalability challenges as server might be overloaded or not available at all rendering the requestor to start from the beginning.

In modern architecture, there is shift of storing the session on the caching (Ex. Redis) instead of server. This paradigm shift will enable any server to process the request and get the session information from caching. Though this strategy has its own challenges because caching has to be highly available, should not have data overlapping and would have scalability problems across regions. With increased load, the caching can be overwhelming and also turn to be highly expensive.

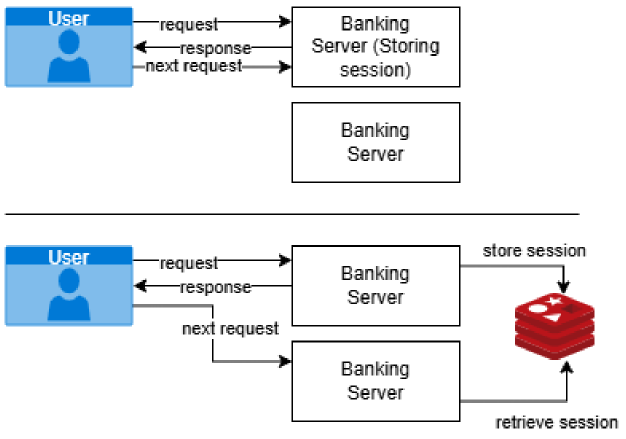

Figure 16. A sample application showing shift of storing user sessions

Due to the challenges above, the shift occurred towards stateless design pattern, which means that state of the request will be maintained with client itself rather than on either the caching or the least preferable option of server itself. With that said, the norm is that every HTTP request will need to include all the information necessary to process the request and therefore, eliminate dependency on the session stored on the remote environment. This enables load balancer to divert request to any server regardless of state, and hence brings more scalability.

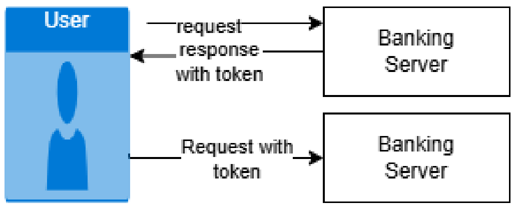

Figure 17. A sample application using session token

The example in financial world is of external account management for the customers. As and when user wish to add an external account on to its profile, the Bank needs to verify the user identity, account identity, account ownership, fraud decisioning. The earlier approach could have been to store the state of the application on to the server, so that, in case of network glitch, the customer application state would be maintained and there would not be need to restart the whole application. The storage of session on the server could have been a possibility if the server resides on-prem. But with cloud adoption, applications get deployed on multi region environments for disaster recovery and therefore, the application state would need to be maintained on to the caching environments like REDIS so that severs across multi-regions can access application state.

With architecture modernization, ideally, each state can be maintained in a token and can be provided to the client itself. For Ex: as soon as user identity is confirmed, then token can be updated with that, followed by update in token for account identity. Each request for progression of the application will result in token updates and at the final conclusion the client token would have encrypted confirmation of user identity, account identity, account ownership and fraud decision. This will make the use case of adding external account as stateless enabling horizontal scalability.

7. When to Use What Scalability

The approach to decide on type of scalability for the application depends on the nature of application, expected customer growth, transaction volumes and data size growth. One size doesn't fit all and that holds true for software architecture as well. There are certain rules of thumb, that monolithic applications which use in-memory databases and caching algorithms will need to adopt vertical scalability while applications having modern architectures of Microservices can adopt horizontal scalability.

7.1. Vertical

7.1.1. Monolithic

If the applications are monolithic, then adding more resources to single instance i.e. vertical scalability would be the choice.

7.1.2. Stateful

If application is of stateful nature i.e. previous request output will impact the response of the current request then Vertical scalability is the choice as if current request gets processed by a different server, then the request will fail or provide incorrect response. The way to handle it to store the stateful variables in the caching but it increases complexity.

7.2. Horizonal

7.2.1. Microservices

Applications built over modern architectural principles of Microservices can easily adopt Horizontal scalability.

7.2.2. Stateless Applications

If the earlier request doesn't depend on the next request, then application can be scaled horizontally as any server can handle any request and there is no need to maintain state and thus also eliminates need of fault tolerance as application can't lose state because of each request being independent.

8. Trade Off Assessment

There are certain trade offs that needs to be considered while determining the scalability strategy for applications. Each scaling strategy has it own pros and cons and architect needs to consider various other parameters like budget, complexity and IT infrastructure before designing the applications.

8.1. Upfront Cost

The cost incurred for provisioning and maintaining multiple nodes for horizontal scalability initially is higher than upgrading a single node with hardware. If the organization is tight on budget, then vertical scaling would be more feasible for the organization.

8.2. Early Limit

No application is scalable to infinite capacities but vertical scaling has more stringent limits because IT can upgrade infrastructure of a single machine up to a certain limit only. Therefore, if organization is having visibility of long-term scaling requirements, then vertical scaling would need to be downplayed.

8.3. Regional Distribution

If application demands to be served from different regions in the world, then its impractical to divert all the traffic to single machine at single location. This can add up way higher latency, incur single point of failure and SLA can be breached. Therefore, horizontal scaling would be winner in case regional distribution is needed.

8.4. Complexity

If application is simple, doesn't anticipate too much load in future then more simplistic is Vertical scaling to be adopted. The horizontal scaling improves availability, fault tolerance but it adds complexity, network overhead and coordination challenges.

8.5. Latency

Horizontal scalability adoption will need to have distributed load, caching, replication which in turn have the pitfall to introduce latency and data inconsistencies as explained above. In contrast, vertical scalability includes infrastructure, code, and network optimization but at cost of reduced flexibility and modularity.

8.6. Redesign Effort for Legacy Applications

Horizontal scaling most commonly needs redesigning of legacy applications, to adopt most aligned microservices architecture. This most of the times needs multiyear planning, resourcing, funding and execution which makes most organization awry of implementations. The risk involved might be not feasible for the organization and therefore, vertical scalability makes more choice considering the redesign efforts.

8.7. Real World Constraints

The IT architects will need to adopt a hybrid scalability strategy based on the business need, cost, future visions etc. For Ex: An online banking institution providing better CD rates always needs to manage external accounts tagged to customer profiles and most commonly needs relational database for that. If vision is that due to competitive CD rates, customer will make more deposits henceforth the organization will need to maintain and render more data on demand doesn't justify the need for provisioning read replica and application redesign to direct traffic towards it immediately. Rather architect should decide to increase the computation power for the database.

Similarly, if the same online bank has higher deposits and have extended automated functionalities via different channels like IVR, Mobile, Web and Agents for trivial functions like balance inquiry, payment status check might lean towards adopting horizontal scaling for these services.

9. Scalability Assessment

The scalability of software applications can be determined using three step process.

9.1. Define Baseline and Target Metrics

First and foremost, application architect needs to define the baseline and target metrics of intended application performance. Most common ones that need to be defined and evaluated are TPS (Transactions per second) and Resource Utilization of CPU and Memory. The determination needs to be made in collaboration with the product owner and the business.

9.2. Define Scaling Policy

Architect will need to determine the scaling policy to be adopted based on the request protocols and the measurement of number of requests that are expected concurrently during business and non-business hours.

9.2.1. Target Scaling

It's the process of automatically adjusting the capacity of resources used by software application based on specific metrics or thresholds. For Ex: You can specify CPU metric and threshold of 60%. As and when CPU usage moves beyond 60% then another machine instance can be spin up. This is often used in conjunction with services like Amazon EC2 Auto Scaling, AWS Lambda, and Amazon ECS which automatically can scale out the resources based on the threshold.

9.2.2. Simple Scaling

An alarm can bet set for a range of higher and lower limits and not the average as in Target Scaling. For Ex: When the metric exceeds a certain threshold (e.g., CPU utilization above 80%), you can configure it to add a specified number of instances. Conversely, when the metric falls below a certain threshold (e.g., CPU utilization below 40%), it can terminate a specified number of instances.

9.2.3. Step Scaling

This is same as simple scaling, but multiple rules can be defined as a step like if CPU utilization above 60% then launch 2 instances, if CPU utilization above 80% then launch 4 instances etc.

9.2.4. Scheduled Scaling

Define the time when you must scale out and how many instances. This is applicable in scenarios where the software application receives files. The application doesn't need to keep resources running all the time when files only show up couple of times a day. In these scenarios, you can have policies which are time based and you can start instances some time before the files processing time.

9.3. Stress Testing

Evaluate the scalability of the system by first judging the system performance with baseline metrics and then start elevating the load till the time target metrics are achieved. You need to see if the scaling policies are being executed and how the performance of the applications are with target metrics. Architect needs to keep measuring the key metrics like CPU Usage, Memory Usage, Network Latency, Response Times, and network throughput to ensure that applications are performing efficiently at any given time irrespective of load. These metrics are invaluable for identifying the bottlenecks and making informed decisions about when and how to scale.

10. Case Study

10.1. Discover Financial Services

Discover Financial Services, LLC kicked off a multiyear plan to convert all monolithic applications to modern microservices architecture and deployed services on docker containers using Kubernetes which makes all applications highly scalable and manageable. Any new developments follow the similar design patterns and therefore, all banking services, including but not limited to Originations, Payments, check processing are efficient, resilient, reliable and scalable to cater load of millions of customers at same time.

10.2. Netflix

Netflix streaming service uses CDN, Caching, cloud computing and microservices architecture to deliver high quality content to millions of customers simultaneously using horizontal scalability.

11. Conclusions

Scalability is crucial in software application architecture and should not be overlooked while planning for functional requirements. By designing software applications scalable, businesses can have resilient and highly available systems along with cost saving while also ensuring that application will be able to adapt to higher demands without compromising performance and reliability. Scalability can be achieved using vertical or horizontal scaling. Architects should primarily focus on making applications stateless and avoid centralized components so that distributed architecture can be leveraged which in turn helps distribute workload efficiently and improves resilience.

ACKNOWLEDGEMENTS

The completion of this article would not have been possible without the support and guidance of Sriram Srinivasan, Discover Financial Services and Roland Sanguino, Discover Financial Services. His dedication and overwhelming attitude towards helping his colleagues, encouragement and insightful feedback were instrumental in accomplishing this task.

DISCLOSURE

REFERENCES

[1] Malik Syed, Mainframe Modernization Strategies, American Journal of Computer Architecture, Vol. 11 No. 1, 2024, pp. 1-9. doi: 10.5923/j.ajca.20241101.01.

[2] What is Scalability and How to achieve it [online], Available: https://www.geeksforgeeks.org/what-is-scalability/.

[3] Stateful vs Stateless Web App Design [online], Available: https://blog.dreamfactory.com/stateful-vs-stateless-web-app-design.