Large Scale Services: What it takes

Tushar Bammi | Lead Software Engineer
Abhijeet Kulkarni | Sr. Lead Software Engineer

Adobe Systems India Private Limited
Abstract

If you are working on a large-scale web service with large user base, which receives millions of calls per minute, deployed in cloud, has huge business impact, then you surely understand what it takes to test and deploy it. Deployment and verification of cloud based services changes drastically when we consider large scales. There are certain factors which seem to be ignorable for small scale services but becomes utmost important in case of large scale. So, scale matters a lot.

This paper covers various challenges faced while designing, testing and deployment different phases of a large-scale web service, and strategies to handle them impeccably.

Strategies should be created for each phase from design to deployment, from monitoring to roll-back. These strategies could be broadly categorized into following buckets:

Design phase: Memory management, Caching architecture, Microservices style architecture
Testing phase: Functionality, and its impact on existing services, Capacity and Resource planning, Production environment simulation for each scenario, Longevity runs
Deployment phases: Multi Geo deployment, Phased deployment in controlled manner, Auto scaling strategy, Roll-back strategy, Monitoring and alerts management.

These may seem familiar, but when viewed with large scale lens, they create a huge difference
What do we mean by large scale services?

Scale is subjective. If a team has built small scale applications in the past and need to make it large, as it handles more data and more traffic, they might call it a large-scale service. But if you compare that application with ones like LinkedIn, Facebook, or Google then it will still be a small scale application. People have different notions of what large scale is. What is large for some might be medium size for others and small scale for some others.

HipChat has about 1.2 billion messages/documents stored, Sales force deals with 1.3 billion daily transactions with over 24,000 database transactions per second and over 22PB of raw SAN storage capacity, CinchCast has over 50 million page views a month, Pinterest has over 18 million visitors with a 10X growth rate, Amazon has over 55 million active customer accounts, Flickr has over 4 billion queries per day, Netflix has 48 million members with over 50,000 requests per second and the list goes on. These numbers should be good enough to explain how large a scale could be.

So, scale could be defined by requests per second any service is serving, or number of concurrent users active on an application or by the its eye catching growth rate.

What are the problems faced when services designed for small scale starts handling large scale load?

Following are the problems faced if large scale is not considered while creation of the service –

1. **High Latency**: There is always an acceptable limit for a service in which it should respond. High latency occurs if the service is not able to serve the data in the acceptable range.

2. **Scalability Issues**: Scalability could be of 2 types:
   a. Vertical: Hardware of existing server is enhanced by adding more hardware resources.
   b. Horizontal: Existing server could be multiplied so that load could be divided among them

   Scalability issues occurs if service is not able to do any of these scaling.

3. **Frequent service downtime**: If a service is facing above mentioned issues, then it is bound to encounter downtime as well. It could be because of scalability issues at peak load times or due to very high latency. Reason could be any one of the two but end result is same, our users are suffering.
What are the major bottlenecks faced while scaling?

Scaling bottlenecks are formed due to following component

1. **Centralized component**: A shared component in application architecture which cannot be scaled out adds an upper limit on number of requests that entire architecture or request pipeline can handle, for example Database or central cache system, which is being shared by multiple server instances.

2. **High latency component**: A slow component in request pipeline puts lower limit on the response time of the application like Database, as with many applications database writes are slowest part.

   Usual solution to fix this issue is to make high latency components into background jobs or executing them in asynchronous manner with queuing.

3. **Limited individual component**: There are certain resources which are dedicated in nature like memory assigned to each server. But simultaneous failure of such resource on multiple server instances, can disrupt proper functioning of service. For instance, if half of my running instances gets Out of Memory at same time, then remaining half healthy instances will collapse under double load exerted on them due to failure of other half.
What are the different parameters which should be taken care for large scale web services?

Different parameters should be taken care for large scale web services in different phases:

1. **Design & Development Phase:**
   a. **Memory Management**: Memory management is the process of controlling and coordinating computer memory, assigning portions called blocks to various running programs to optimize overall system performance. Memory management resides in hardware, in the OS (operating system), and in programs and applications. Efficient memory management techniques are required for managing used and unused blocks of memory as well as for cleaning unused memory. Situations like memory leak could prove costlier when dealing with large scale services.
   
   b. **Caching architecture**: Since we are dealing with large scale, it becomes essential that calls to data sources should be minimized as they impact response times as well as could prove counter economic in case third-party data sources are used. So, not every incoming request should perform all the logic and fetch all the data. Caching the data to use it for further calls could reduce the response time and save the system resources. Exact cache duration and data to be cached could depend on case to case basis.
   
   c. **Microservices style architecture**: Monolithic applications where all the logic is running from receiving a request to serving the response could work well for small scale, but not for large scale. In large scale microservices style architecture work, where different operations/tasks are created as microservices, which could then be used in different order for different types of requirements.
   
   d. **Configurations Sharing**: Not every data in a service could be embedded in application code and should be kept outside so that changes could be made at runtime. Therefore, for large applications it becomes important to share the configurations so that a change made at single place should get reflected at all the servers.
2. Testing Phase:

a. **Functionality verification**: In large applications, even a small issue could get multiplied exponentially and can cause problem. So, it become necessary during functional verification to take care of its impact on already existing services, its upstream and downstream services.

b. **Capacity and Resource Planning**: There could be many resources which are used in the application and therefore their capacities should be planned to make sure that it does not becomes a bottleneck. One extra call to database may look small, but could result in millions of hits per minute in case of large scale

c. **Performance verification**: There are many issues which comes only under load. So, for large scale services, proper performance verification plays a crucial role in identifying critical bugs and helps in ensuring smooth deployment and running of the service

d. **Longevity Runs**: There are some resources which are acquired for processing the incoming request and then released upon the response serving by application continuously. Any anomaly in this process could exhaust those resources and then could create problems for the application in long run, therefore longevity runs ensure that there are no concerns with longer duration runs of the application. This process is also known as soak test or stress test by some organizations.

e. **Production environment simulation**: It is always said that production environment could not be replicated, it can only be simulated. Pattern and nature of incoming requests on production should be simulated in internal environments to ensure the behavior of application when deployed on Production. This also helps in smooth deployment of the application.
3. Deployment & Delivery Phase:

a. **Load Balancing:** Vertical scaling could only be done to a certain limit beyond as hardware could not be enhanced infinitely. So, horizontal scaling helps to scale the environment for handling high load. Load Balancer, therefore plays an important part in distributing the requests to different servers and fetching the response from them to serve it back to the caller.

b. **Scale up and scale down strategy:** Rarely services experience a continuous steady load at all the time. Often, it happens that load increases during few hours and decreases after it to increase again. So, it becomes necessary to have a strategy to increase more servers (scale up) on increase of load and then decrease extra servers (scale down) on decrease in load. This is quite necessary for large scale services as it could contribute a lot to the cost. Deciding auto scaling thresholds is equally important as having an auto scaling strategy. If you have a strategy but with very lenient threshold, it might not trigger when it is required and defeats whole purpose of having it. Similarly, highly conservative strategy may cost extra dollars by scaling the service unnecessarily.

c. **Content Distribution Network:** CDN plays an important role in minimizing response time and costs associated with the services. For services, which mostly serve static data that changes at low frequency, could host the data on CDN and save lots of money by not spending on the servers.

d. **Multi Geo deployment:** Large scale services receive enormously high number of requests, often distributed across geographies. Response time increases if requests are served from a server sitting in a different geography. Therefore, multi geo deployment not only helps in reducing response times but also ensuring that application is available in case of any disaster in a geography.

e. **Phased deployment:** Real test of any application is always the Production environment. Instead of deploying in one go, large scale services are required to be deployed in phases to mitigate the risk from any unknown issue in the application. Data monitoring could be done at different stages and expected numbers provide confidence to move ahead. Different strategies could be adopted for phased deployment of an application.
f. **Roll-back Strategy:** With large scale comes large responsibilities. Large services could not afford to go down or give errors even for short durations. So, they should always have a tested roll back strategy to go back to previous stable state quickly in case of any issue with new build deployment.

4. **Post-Deployment Phase:**

   a. **Monitoring:** Monitoring plays the most crucial role to ensure service is running smoothly. Various monitoring tools could be used to monitor and analyze different data:
   
      i. Response times
      ii. Response codes
      iii. CPU usage patterns
      iv. Memory usage patterns
      v. Error rate
      vi. Health check

   Assurance of expected numbers of these parameters ensure smooth functioning of the application and its availability 24x7.

   b. **Alerts Management:** Since we are dealing with large scale applications which involves many resources, monitoring parameters and tools, so it becomes humanly impossible to continuously monitor the service. Alerts and alarms are a way to let know the team that there is some deviation observed from the normal pattern. Alerts should be specified for soft and hard limits where soft indicates a slight aberration from a pattern and hard could indicate possible concern in running of the application. Both soft and hard limits should be placed in slightly aggressive manner so that we should have time to react before the problem is faced by users.

**Conclusion:**

Managing large scale service is an important task because of its impact magnitude. Even a small fluctuation in such service has potential to create a havoc.

So, we need to diligently plan and execute each phase for such service from design to implementation, from testing to deployment and monitoring.

We have discussed some of the techniques we are using, which are evolved while working on services of such scale, and importantly giving us very encouraging results.
References

1. https://web.stanford.edu/
2. https://stackoverflow.com
3. https://www.linkedin.com/pulse/architecting-extremely-large-scale-web-applications-panduranga
Author Biography

**Tushar Bammi** works as Lead Software Engineer with Adobe Systems India Private Limited, Noida. He has more than 8 years of industry experience, majorly in quality assurance of large scale web services.

**Abhijeet Kulkarni** works as Sr Lead Software Engineer at Adobe systems, Noida. He has 10 years of experience in Quality assurance for cloud based web services and server based applications. He is interested in new trends and advancements happening web services domain and cloud technologies.
THANK YOU!