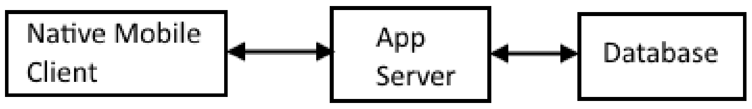
Web Architecture Design

Part 1

The below solution presents the architecture for the following:

a. Receive images from a native mobile app installed in users' mobile device
b. Process such images
c. Store results of the processing stage
d. Return processed images to users



Detailed Description:
The mobile client sends an http POST request as an asynchronous task (so the mobile app UI doesn't block) in the app. The image can either be encoded as base64 or sent directly as part of the POST data.

The web server (can use an EC2 instance) listens for requests from the mobile app. After receiving the raw image data, the server may need to decode the data and can then both store the image and start the processing – may need additional servers depending on the intenseness of this task. Then a POST response may be sent back to the client with the processed image. The mobile client then decodes the image data and displays it as necessary.

A MongoDB database (using 3 replica sets – 2 secondary's and a primary) can be used and the image data can be stored as Binary data (assuming images cannot exceed 16 MB – limit on a MongoDB document). For larger images, GridFS can be used to split a single image into multiple documents. Depending on how this architecture will scale (may need to think about sharding as user base grows), trade-offs need to be considered whether to go with a relational database like MySQL instead.

Part 2

What changes are needed when designing for:

a. 1000 users
b. 100k users

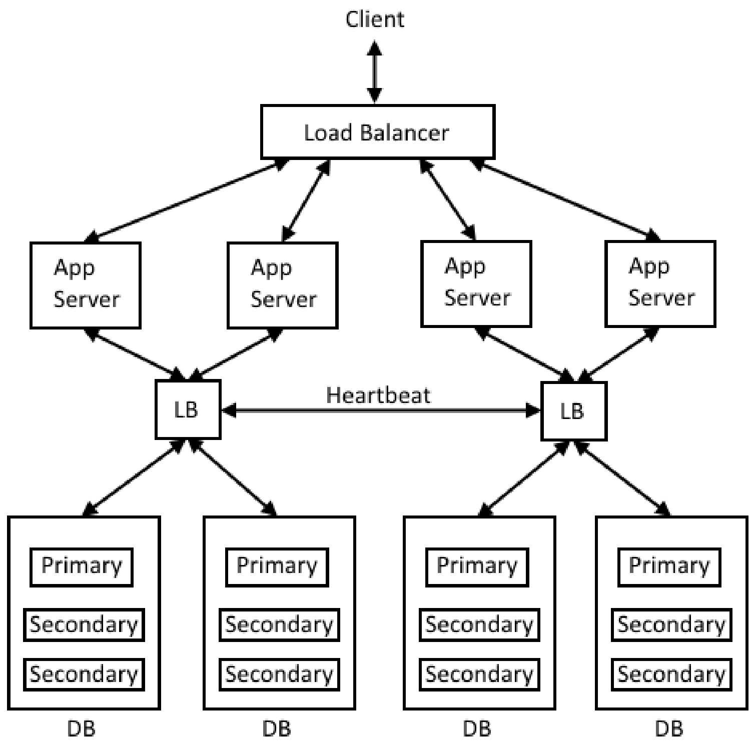
Timelines should also be included to implement functioning architecture.

For 1000 users:
No changes need to be made. Assuming 1000 requests happen at the same time from all the users (1 per user), web server can handle this by spawning the required number of processes. Also using AWS can help with costs as the EC2 instance needs to be powered up only during this one busy time (can use EBS for persistent storage).
Space Requirements:
Assuming maximum size of image (after being decoded) is 1 MB, need approximately 1 GB of space per day (1000 images per day). For flexibility and extra capacity, a single M1.large instance should suffice for both CPU / storage / RAM.

Timeline

	Monday	Tuesday	Wednesday	Thursday
Backend Development	■	■		
Unit Testing		■		
Load Testing Server / Database			■	■

For 100k users:
In this case, the architecture will need a lot of changes. Both the application servers and databases will need horizontal scaling since we're dealing with a lot more requests and a lot more storage.
To start out, the following architecture is proposed:

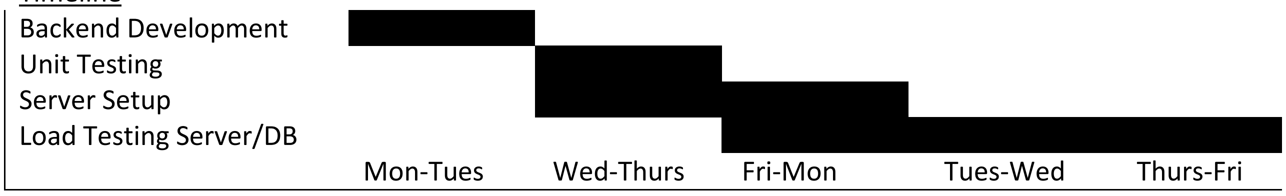


This is a theoretical proposition. Testing has to be performed to measure the exact performance of each component and only with experimental results scale as necessary.

The client will hit a load balancer (can use ELB or reverse proxy using HAProxy). A DNS round robin setup can be tested out initially. Using four application servers (EC2 M1 instances) should require about 25000 concurrent requests per server (100k/4). Once this setup is established, a tool like JMeter can be used to simulate the load to stress test and scale further as necessary.

For the database, MongoDB can continue to be used. One of the key features of MongoDB is it provides automatic sharding/replication so the application code doesn't have to be aware of the underlying database architecture structure. If a primary stops functioning, after holding an election, one of the secondary's will elect itself to be the new primary. Write performance is improved due to the presence of multiple primaries. Assuming 100 000 MB (or 100 GB/day, 1 MB per image), using AWS EBS provides sufficient IOPS to perform this operation.

Timeline



Part 3

If users from Part 2a/2b upload their images every day twice a day at the same time, does the architecture still function correctly?

Yes, only have the application instances running during the times this operation is to take place.
Theoretically:
Using EBS io1 volumes, there are 2000 IOPS and 256 kb/IO.
Using an image size of 1 MB, requires 4 IO/user, therefore 500 users per second, then 100 000 users would take 100 000 / 500 = 200 seconds or approximately 4 minutes. This is the bottleneck of using EBS io1 volumes. Thus, running this operation every 5 minutes is feasible theoretically (actual practical testing has to be done).

Part 4

a. Write code to implement the storage section of the backend
   See code

b. List endpoints of RESTful API needed to interact with the backend
   For part 1a:
   POST /getProcessed
   Using parameters: “user_id” and “image” as a file upload

c. Write code to do the following:
   - Receive image from client, convert it to grayscale, compute the histogram and store both the original image and histogram into a database
   - Extract the histograms of the current week for a single user
   - Extract the median histogram of the current day for all users (the output is a single histogram)

   See code

d. Write code that, given a user id as input, returns n user ids with the most similar histograms as the input user id (see code)
   See code