Report 2

1.

What the issue is:

The scalability issue is that the logo uploads are handled “statefully” on each API server instance: when a user uploads a logo, the API server both does the CPU-intensive image processing and stores the resulting file on that same server`s local disk, and later serves the image from that instance. This tightly couples compute + storage + serving to a single web node, instead of treating the API servers as interchangeable instances that can be scaled horizontally behind a load balancer. This conflicts with the goal of scaling out by adding more equivalent servers.

Why it is a problem in the current system:

This design becomes a problem because horizontal scaling assumes any request can go to any server (via load balancing). When uploaded files live on local disk, these are some problem that could occur:

Inconsistent reads / missing file problem: If the load balancer routes a later request to a different API instance, that instance does not have the file locally, so the logo may not be found or served correctly.

Fragile deployments: Restarting, replacing, or autoscaling nodes can lose access to locally stored files because local disks are not shared across instances.

Throughput bottleneck: CPU-heavy image processing happens on the same nodes that must also respond to regular API traffic, so uploads compete for CPU/memory/disk I/O with other requests, limiting overall throughput. This is exactly the kind of limitation that can reduce scalability when you try to add more parallel workers due to non-parallelizable parts or shared-resource overhead.

Overall, the API layer becomes stateful and no longer easily replaceable/replicable as intended in scalable architectures.

Under what conditions it would become noticeable (e.g. higher load, failures):

First, because images are processed and stored on the API server’s local disk, we can’t safely run more than one backend instance. That means we’re stuck scaling vertically instead of horizontally. In practice, this puts a hard cap on how much traffic we can handle. With around 50–100 active users, we’d likely start seeing slower responses as the CPU becomes a bottleneck.

Second, image uploads make this worse. Image processing is CPU-heavy, and when users upload images, it ties up the same resources that handle regular API requests. During upload-heavy periods, the backend can slow down noticeably for everyone. In those cases, the system might only handle something like 5–20 concurrent users before lag becomes obvious.

There’s also a cost issue. Storing images on the server means we’re using high-performance local SSD storage from AWS or Azure, which is one of the more expensive options. As more images are uploaded over time, storage costs will keep growing, and scaling disk size just adds more pressure to our infrastructure budget.

Reliability is another concern. Since everything runs on a single instance, if that instance goes down, the entire service goes down with it. We can try to automatically restart it, but if that fails for any reason, we’re completely offline until it’s fixed.

Finally, if we tried to ignore this and scale horizontally anyway, we’d run into serious data consistency issues. Images would only exist on the instance that handled the upload, so users would randomly see missing or broken images depending on which backend instance their request hit. And the instance would return an error to the client. If the client got lucky and can see the image, then try to replace the image, they would have to get lucky and hope their request goes to the right machine.

2.
Our changes are to add 2 key components to resolve this scalability issue. First of all we are going to add a load balancer to distribute the load across all instances, otherwise only one instance is going to be used and the rest will just sit idle. Now the second change is that we add a storage bucket via cloudflare r2. Adding the bucket will resolve the scalability issue and expose images via our cloudflares API safely. After this change we can now scale horizontally without issues and cloudflare r2 has free egress requests meaning each time a user tries to fetch the image, we incur little to no costs.

However, while cloudflare does provide faster image fetching than what we could offer before the change, there is an aspect which we missed to discuss. Uploading the images would have to go with the following path,

client > cloudflare > load-balancing > instance > cloudflare r2

compared to,

client > cloudflare > instance

This will more than likely increase the upload time, however we do not think this time will be significant for the user. Another issue is that we are even more reliant on cloudflare now, this brings us back to fault tolerance. If cloudflare were to go down due to a failure, which happens more often than one thinks, then our service would also be completely down.

3.

- Install the library for cloudflare R2
- Start writing code to replace the old upload mechanism while keeping the old code active. Basically don’t change the current code just start writing the new functions to replace the old ones. In the clubcontroller.js
- Add a “replaceImage” function so that we don’t just add in images each time. This would probably cause issues in the future otherwise.