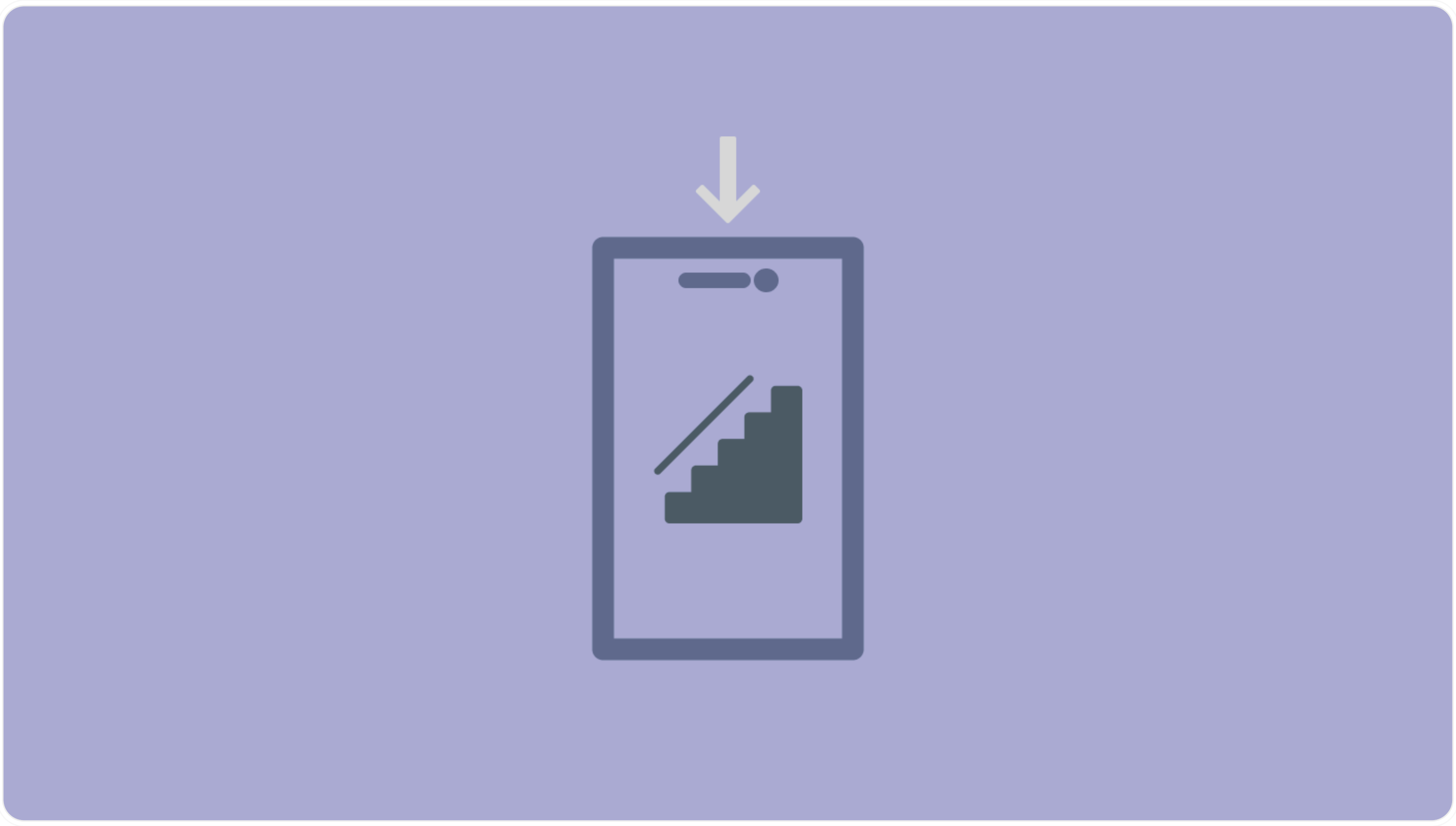
ENGINEERING

Levels.fyi's Over-The-Air Mobile Updates

How we keep our mobile app updated everyday

Adithya Viswamithiran
June 4, 2025



💡 Check out the Levels.fyi app for salaries & more!

When Levels.fyi started out, we were using a simple google sheet as a backend. With time, our scale grew to millions of users. With scale, our range of offerings has been growing and we have been entering newer territories everyday.

Mobile @ Levels.fyi has had a similar journey as well. The initial Levels.fyi app had little resemblance to our website and was meant to cater to a niche subset of our overall product—the Levels.fyi community. Over time, we have come to realize that mobile tackles a very important aspect of our business, which is user stickiness. We have been focusing on continuously improving our mobile app to pass on delight to our users.

Over-the-air updates have played a very important role in our development cycle, helping us push newer features to users without requiring them to download newer versions of the app from Google Play/Apple Store.

Finding a Codepush alternative

The Levels.fyi app is built using react-native. Until the initial days of February 2025, we were using Microsoft appcenter (and codepush) for error diagnostics and pushing over-the-app (OTA) updates to our users. With the impeding sunset of appcenter on March 31 2025, we were faced with the problem of choosing an alternative for OTA.

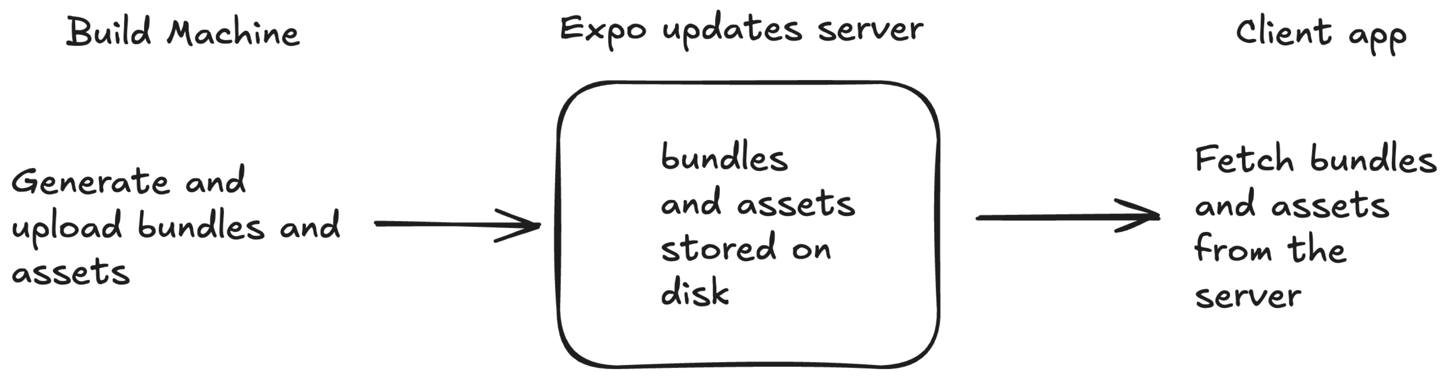
We initially explored similar managed services for OTA with Expo Application Services (EAS) at the top of the list. Keeping a close eye on spend sinks is important for a growing startup like Levels.fyi. EAS is a great service, but at the time, it didn't make much sense for us to go with EAS.

EAS would incur a cost at least $350 per month. $350 is ~7% of our ENTIRE spend on AWS currently, and AWS is the biggest spend source for us. For every additional 10k active users of the Levels.fyi app in a month, our cost would increase by $50. So, spend can snowball pretty quickly.

Self-hosting isn't something new at Levels.fyi. Plausible and PostHog are examples of services we have been self-hosting/have tried self-hosting. A self-hosted version of the code push server is available as open-source, but the codebases for the server and the OTA react native module weren't going to be actively maintained post the sunsetting of Microsoft appcenter. Considering maintainability issues, we decided to go ahead with a self-hosted server implementing the expo updates protocol for OTA updates.

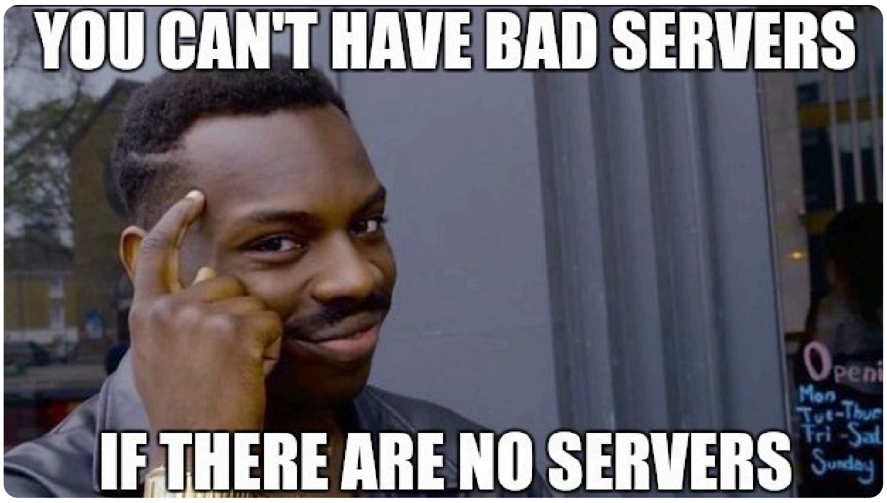
Why go serverless?

A custom server hosted implementation of the expo updates protocol is available as open-source. A simplified chart depicting how the custom updates server works, is shown below.


Build Machine
Expo updates server
Client app
Generate and upload bundles and assets
bundles and assets stored on disk
Fetch bundles and assets from the server

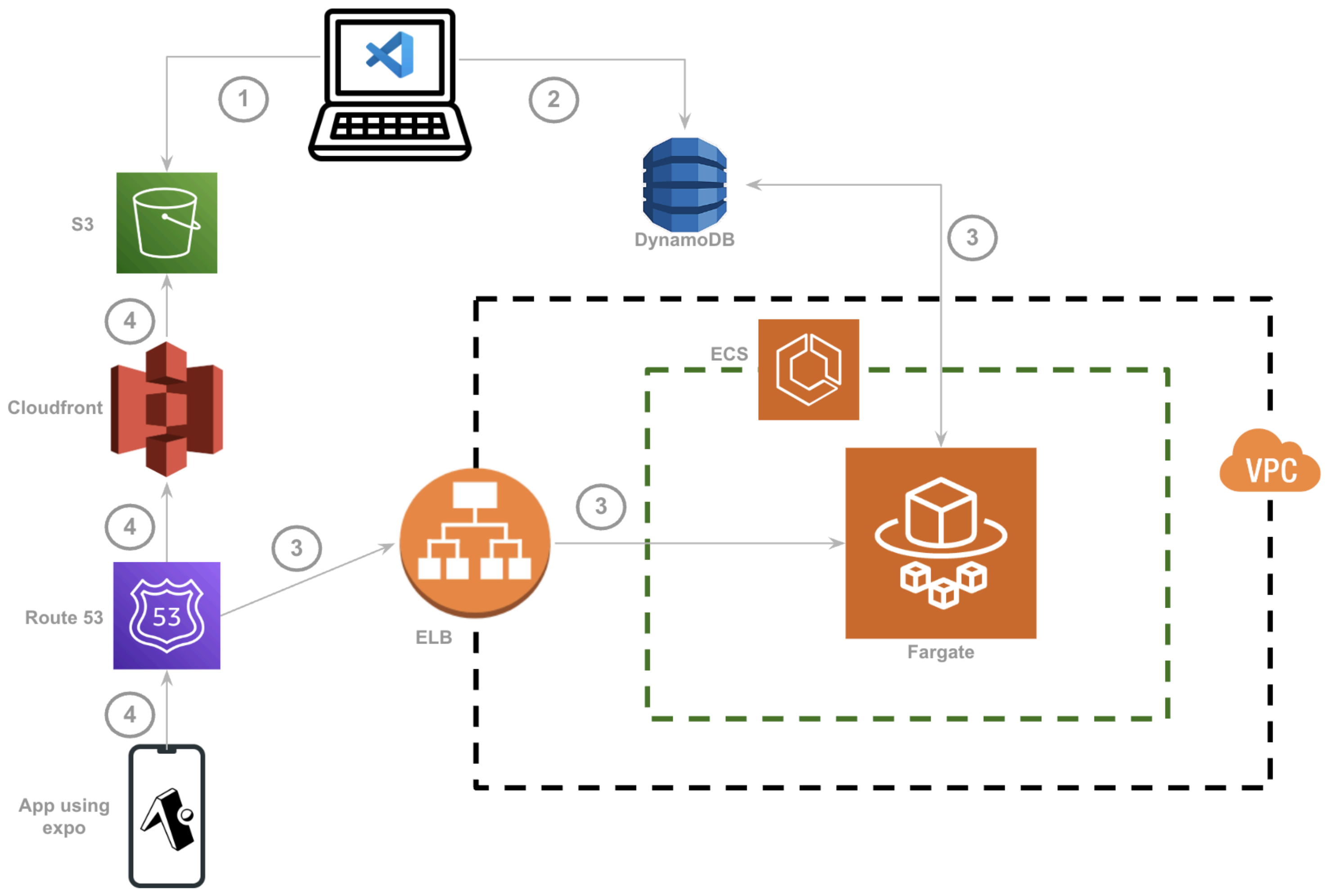
There were reasons as to why we thought serverless was a better option for us considering the following points

1. Scalability: The easiest way to scale storage is storing files on a blob storage like S3. It is possible for multiple compute instances to achieve read/write to a single disk using technologies like Multi-attach. But if additional disks are to be attached, such a setup would require manual intervention again. We chose S3 as our managed blob storage service to better horizontal scale our storage needs.
2. Caching: Since we use S3 for storage, it is easy to bake in caching by creating a Cloudfront distribution that uses our S3 bucket as an origin.
3. Response Latency: The already available open-source server uses a mechanism to generate an Expo Updates manifest object on-the-fly per each request. So if we were to use S3 for storage, for every GET/api/manifest request made to the server, the server would need to download some files from S3 onto the disk and generate an Expo Updates manifest object. This to and fro causes extra latency. We decided to go with the approach of generating the Expo Updates manifest alongside the bundle generation step (build time). We store the generated Expo Updates manifest in DynamoDB.
4. Cost per usage: We were not not sure about the load traffic patterns for the OTA server. It made sense for us to go with serverless compute, which follows a pay-as-you-go billing model. We went with ECS fargate.



Architecture

💡 Check out my sample repo https://github.com/adithyavis/serverless-expo-ota-server/



1. On the build machine, the Expo CLI is used to generate the .hbc bundles and the assets for the OTA. The generated bundles and assets are then uploaded to a s3 bucket. Along with this, an Expo Updates manifest JSON file is also generated. This file contains the metadata related to the update, such as the id of the update, createdAt, URL of the launchAsset, etc. The launchAsset URL directly points to the location of the .hbc bundles and assets in the s3 bucket.
2. A new entry is added to the DynamoDB table. The partition key of the entry is a composite value consisting of the target platform and the target runtime version of the OTA update. It is something like `android-1.0.0`. The sort key is the update number for the given runtime version, like `1` or `2`. The Expo Updates manifest object is stringified and is stored in the `manifest` attribute of the DynamoDB item. The server can get the Expo Updates manifest from DynamoDB using a Query operation.

Partition Key	**Sort Key**	**id**	**platform**	**manifest**	**mandatory**	**runtimeVersion**	**otaUpdateVersion**	**activeDevices**	**ena**
android-1.0.0	1	fsfs133-14fdgds-13424dbg	android	{...}	true	1.0.0	1	100	tru
android-1.0.0	2	fsfs133-14fdgds-13424dbg	android	{...}	true	1.0.0	2	1240	tru

1. The react-native app queries the GET /api/manifest API endpoint on the server. The server makes a Query operation on the DynamoDB table to get the Expo Updates manifest object using the partition key and sort key. It prepares a manifest JSON and returns that to the client.
2. If there is a new update available for the react-native app, a fetch call is made and bundles and assets are downloaded through the Cloudfront distribution pointing to the S3 bucket, without having to reach the server.

Wrapping Up

The sunsetting of AppCenter could have been a major disruption, but it ended up being an opportunity for us to re-evaluate and improve how we ship updates to our users.

By going the self-hosted, serverless route, we now have an OTA pipeline that's cost-efficient, scalable, and fast. It fits right into our philosophy at Levels-fyi of owning our infrastructure where it counts, optimizing for performance and flexibility, and keeping a close eye on long-term maintainability.

As we continue to grow and iterate on our mobile experience, this setup gives us the control we need to keep delivering improvements seamlessly without waiting on app store release cycles.