shorebird

A COMPLETE TECHNICAL GUIDE

Flutter Code Push

The complete technical guide to over-the-air Flutter updates.

Ship fixes in minutes — not review cycles.
A field guide for Flutter engineering teams.

FOR DEVELOPERS & MOBILE TEAMS

SHOREBIRD
Production-ready Code Push for all Flutter apps

shorebird.dev
2026 EDITION

Contents

ABOUT THIS GUIDE

This guide explains why Flutter teams historically couldn't ship over-the-air updates, how Shorebird's Code Push solves it at the runtime level, and the exact sequence to take a production Flutter app from no OTA capability to shipping a patch in under 30 minutes.

AUDIENCE	READING TIME	LEVEL
Flutter & mobile engineers	~18 minutes	Beginner → Advanced

INTRODUCTION

The 48-hour problem

A critical bug ships to production. Users report crashes in your payment flow. Your team has a fix committed and tested within the hour. Then you submit for App Store review and wait. Twenty-four to forty-eight hours on a good day, longer if it lands on a weekend. By the time the fix reaches your users, the damage to retention and app store ratings is already done.

React Native teams sidestep this problem by swapping a JavaScript bundle. Flutter teams have historically had no equivalent option. This guide explains why that's true, how Shorebird's Code Push solves it at the runtime level, and how to get the full implementation running in your production pipeline.

> ALLAN WRIGHT · SENIOR MOBILE DEVELOPER · PUSHPRESS
> "Our speed to release updates has skyrocketed. It used to take four days to two weeks depending on urgency. Now it's way faster. The reliability of shipping fixes is night and day."

01 Why Flutter can't do what React Native does

Flutter compiles Dart code to native ARM machine code at build time using ahead-of-time (AOT) compilation. The output is a compiled snapshot baked directly into the app binary. At runtime, there's no interpreter between your Dart logic and the hardware.

React Native works differently. The JavaScript logic runs inside a JS engine (Hermes or JSC), loaded from a bundle file at startup. Expo's EAS Update ships a new bundle to devices over the air because swapping the JS file at the path the runtime reads from is all it takes. The app's logic layer is just a file on disk.

Flutter has no equivalent file to swap. The `libapp.so` on Android (and its iOS counterpart) is compiled Dart code in binary form. You can't diff a `.dill` snapshot against a new one and swap it at runtime the way you'd replace a JavaScript bundle. The runtime isn't built to load updated compiled code dynamically.

> **KEY INSIGHT**
> This isn't a missing feature waiting to be added. It's a direct consequence of AOT compilation — the same property that makes Flutter fast is what makes swapping code hard.

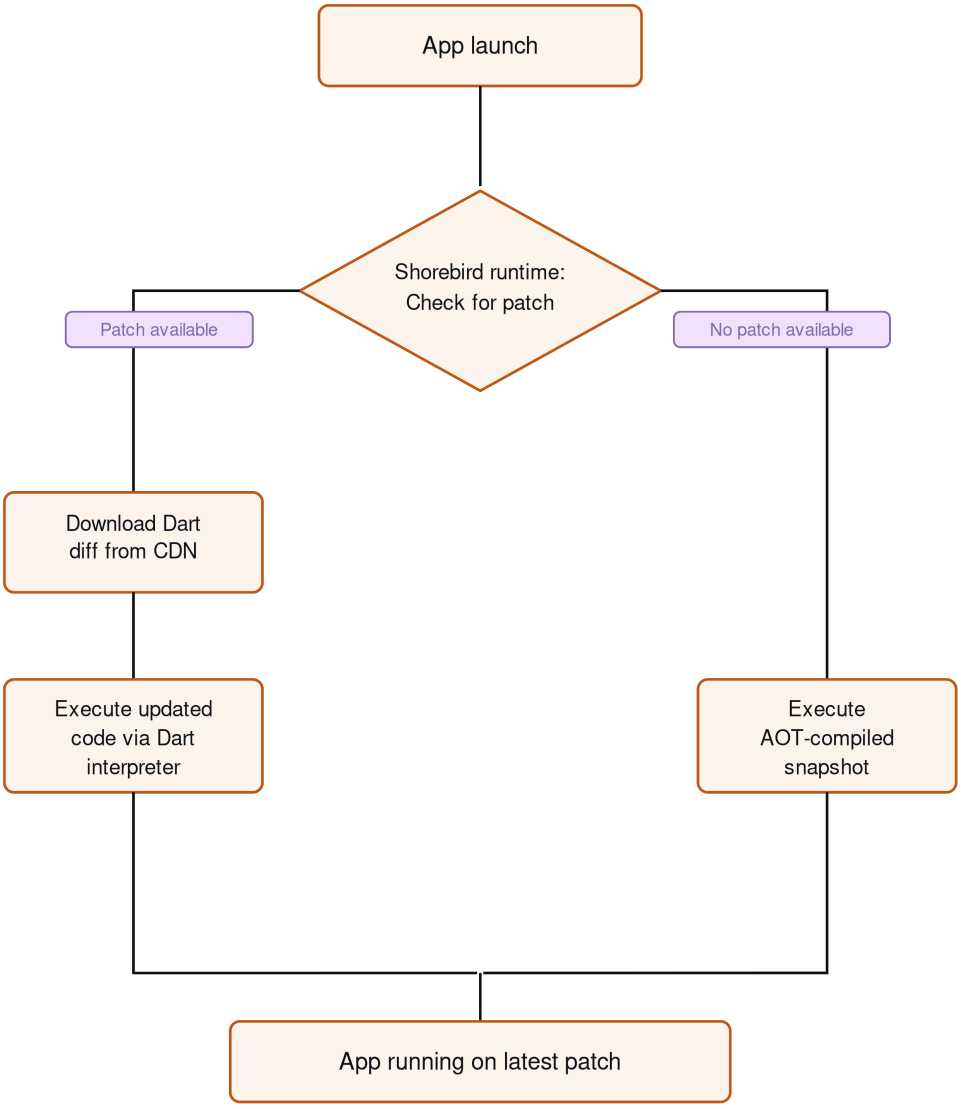
Solving this requires modifying the Dart runtime to support some form of dynamic code execution, which is what Shorebird does. We modified both the Dart VM and the Flutter engine to make genuine diff-based OTA code updates possible on Android and iOS.

02 How Shorebird Code Push actually works

When you build a release with Shorebird, the resulting binary contains two components: the standard AOT-compiled Dart snapshot and a modified Dart interpreter that Shorebird built and maintains. On startup, the Shorebird runtime checks Shorebird's servers for a patch scoped to that specific release version. If no patch exists, the AOT snapshot runs exactly as it would in a standard Flutter build. If a patch is available, the runtime executes the updated Dart code through the interpreter instead, while the AOT snapshot remains in place as the fallback.

> **MICHAEL GALLEGO · FOUNDER · WAGUS**
> "We patch fast. One release had 60 patches. Most of them are small quality-of-life updates requested by users. With Shorebird, we could ship them the same day."

What happens on every app launch



At startup, the Shorebird runtime checks for a patch scoped to the installed release. If one is available, it downloads the Dart diff from the CDN and runs the updated code through the Dart interpreter; otherwise the original AOT-compiled snapshot runs unchanged.

The operational model comes down to two concepts you need to keep distinct: releases and patches.

Releases vs. patches

A **release** is a full app build that includes the Shorebird runtime. You build it with the Shorebird CLI, register it with Shorebird's servers, and release the build to your users as you do today. Once users install it, that install is tied to a specific release version in Shorebird's system.

A **patch** is a Dart code diff computed against that release. Shorebird compares the new Dart snapshot against the original, produces a binary diff, and uploads only the changed bytes to its CDN. Patches are version-scoped: a patch built against release `1.2.0` only applies to devices running `1.2.0`.

The hard boundary: OTA vs. store release

This model has a firm boundary. Pure Dart can go OTA. Anything else goes back through the store.

CAN UPDATE OVER THE AIR	REQUIRES A NEW STORE RELEASE
All Dart and Flutter widget code	Native Kotlin, Swift, Java, Objective-C code
UI layout, business logic, routing	Native plugin platform channel implementations
Strings, state management, providers	AndroidManifest.xml / Info.plist changes
Dart-layer bug fixes	New runtime permissions
Pure-Dart Flutter package upgrades	Native binary assets
App configuration managed in Dart	Changes to native dependencies

If you need a new camera permission, you need a new release. Anything outside pure Dart goes back through the store.

03 Setup: from init to first patch

≈ 15 MINUTES

The setup from an existing Flutter project to your first deployed patch takes about fifteen minutes. Five steps, all on the CLI.

Step 1 · Install the Shorebird CLI

Install the Shorebird CLI on your development machine:

```bash
curl --proto '=https' --tlsv1.2 https://raw.githubusercontent.com/shorebirdtech/install/main/install.sh -sSf | bash
```

Then authenticate with your Shorebird account:

```bash
shorebird login
```

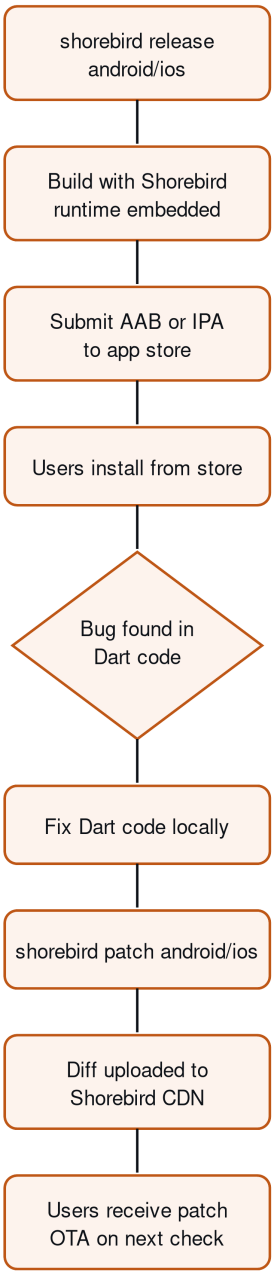
Step 2 · Initialize Shorebird in your project

Run this to set up Shorebird in your Flutter project:

```bash
shorebird init
```

`shorebird init` adds a `shorebird.yaml` file at the project root containing your `app_id`, and configures the Flutter engine binding to include the Shorebird runtime. Your existing Dart code and project structure stay unchanged.

The end-to-end release and patch lifecycle



A single Shorebird release can ship many Dart-only patches to the same users without another store review. The release path runs once per store submission; the patch path runs every time you fix a bug in Dart.

Step 3 · Create a release

In your CI build scripts, update your `flutter build` commands to:

```bash
shorebird release android
shorebird release ios
```

These commands build your app with the Shorebird engine embedded, save a copy of the release in a private bucket under your Shorebird account, and produce the artifacts you upload to the stores: an `.aab` for Play Console and an `.ipa` for App Store Connect. Your store submission workflow doesn't change.

Step 4 · Submit to the stores

Publish to Play Console and App Store Connect as normal. If you distribute through another channel, just use the artifacts Shorebird produces. Once a Shorebird-built version is live, you're ready for over-the-air updates.

Step 5 · Push a patch

After fixing a bug in Dart code, create and publish a patch:

```bash
shorebird patch android
shorebird patch ios
```

Shorebird diffs the new Dart snapshot against the registered release, uploads the delta, and makes it available immediately via its CDN. Users running that release version download the patch in the background on their next update check.

For teams running GitHub Actions, Shorebird provides documented CI integration that slots directly into existing workflows. Both `shorebird release` and `shorebird patch` run in CI with standard environment variables for authentication. Codemagic also has first-class Shorebird support.

> **PIPELINE TIP · RELEASE VS. PATCH**
>
> Treat shorebird release the same way you treat a store build — it runs on the same tags or release branches your existing pipeline already uses. shorebird patch is a separate, lighter workflow you can trigger on a hotfix branch or an ad-hoc dispatch, so engineers can ship a fix without touching the main release pipeline.

Authenticate with a `SHOREBIRD_TOKEN` generated by `shorebird login:ci`, store it as a secret in your CI provider, and export it for both the release and patch jobs. From there, your pipeline looks the same as any other mobile release — plus one extra job for patches.

04 Controlling update behavior in-app

By default, patches download in the background and apply on the next cold start. For most bug fixes, this is the right behavior. Users get the fix without any interruption, and you avoid forcing a restart mid-session.

The `shorebird_code_push` Dart package gives you programmatic control when you need it. Add the dependency:

```yaml
dependencies:
  shorebird_code_push: ^latest
```

Then check for and download updates explicitly:

```dart
import 'package:shorebird_code_push/shorebird_code_push.dart';

Future<void> checkForUpdate() async {
  // Create an instance of the updater class
  final updater = ShorebirdUpdater();

  final status = await updater.checkForUpdate();

  if (status == UpdateStatus.outdated) {
    try {
      // Perform the update
      await updater.update();
    } on UpdateException catch (error) {
      // Handle any errors that occur while updating.
    }
  }
}
```

Call `checkForUpdate()` from the `initState` of your root widget, or from an `AppLifecycleListener` callback on app resume. The root-widget approach catches users on every cold start; the lifecycle callback catches returning sessions.

For a patch addressing a security vulnerability or a crash hitting a significant fraction of your users, you can force a restart immediately after download. Wire this through `TerminateRestart` from the `terminate_restart` package. Reserve forced restarts for genuine incidents.

You'll also need to set `auto_update` to false in `shorebird.yaml`:

```yaml
auto_update: false
```

05 Staged rollouts, rollbacks & patch signing

Shipping fast only pays off when you also ship safely. Shorebird gives you the same controls mature store-release pipelines rely on — gradual rollouts, one-click rollbacks, and cryptographic patch signing — applied to over-the-air updates. The result is a workflow where a regression caught at 5% user exposure never reaches the other 95%, and a compromised delivery path can't push code your users will actually run.

This section covers the three controls together: how to stage rollouts by percentage, how to roll a patch back when something slips through, and how patch signing keeps the release path honest end-to-end.

Percentage-based rollouts

Use percentage-based rollouts with Shorebird's "tracks" feature and a key-value store like Firebase. A safe progression looks like this:

5%	**Canary**	Monitor crash reporting for 60–90 min
25%	**Expanded**	Re-check error rates; hold on regressions
100%	**GA**	Roll out to entire active user base

Starting at 5% limits your blast radius. If a patch introduces a regression your test suite missed, you catch it at 5% user exposure rather than after it's affected your entire active user base.

Rollbacks

When a patch causes a regression, use the Shorebird Console to roll it back. Once rolled back, installed apps learn about the rollback as soon as a patch check occurs, and it takes effect on the next boot — no app-store involvement, no review queue.

> **HOW ROLLBACK WORKS**
> Rollback takes effect at the infrastructure level immediately. Users receive the previous patch the next time their app checks for updates, with no action required on their end.

Custom tracks

Shorebird supports named update tracks for staging and beta environments:

```bash
shorebird patch android --track=beta
```

Beta testers receive the patch immediately while production users remain on the current stable version. You can promote a patch from `beta` to `stable` once QA has signed off. The custom tracks documentation covers the full configuration.

Patch signing

Patch signing ensures that only code you explicitly authorize can run on your users' devices. Each patch is signed with a private key that you generate and control. The Shorebird runtime verifies the signature on-device before applying an update. If the signature doesn't match, the patch is rejected.

This follows the same model used by app stores: code must be signed by a trusted key before it runs. You control that key. Shorebird never has access to it, and neither does any infrastructure involved in delivering the patch.

Because verification happens on the device, a patch cannot be applied unless it was signed by your key. Even in unlikely scenarios — compromised CDN, upstream provider, or delivery network — an attacker still couldn't ship a valid patch without your signing key.

> TRANSPORT VS. CONTENT
>
> Transport security like TLS protects how patches are delivered. Patch signing protects what is delivered. Together they ensure that patches arrive securely and haven't been tampered with.

For teams in fintech, healthcare, or any regulated industry where code changes require an audit trail, patch signing is the recommended baseline — not an optional extra.

06 App Store & Play Store compliance

Shorebird's approach is compliant with both stores.

Apple App Store

Apple's App Store Review Guidelines section 2.5.2 prohibit apps from downloading and executing code that introduces or changes the app's features or functionality outside of the App Review process. The guideline includes a carve-out: apps may execute code that runs through a built-in interpreter, provided the code doesn't change the app's core purpose or add capabilities that would otherwise require App Review.

Shorebird's mechanism fits within this carve-out. Patched code runs through the Shorebird Dart interpreter, not as loaded native code. Shorebird maintains a documented compliance position covering the boundary in detail.

> **RULE OF THUMB**
> The practical rule: use OTA patches for bug fixes, performance improvements, and iteration on existing features. Avoid shipping changes that would surprise users or bypass the intent of App Review.

Google Play

Google Play's developer policies are more permissive. Play explicitly allows OTA code updates without the same interpreted-code carve-out requirement. Standard policy compliance applies — no loading code from untrusted external sources, no violations of content policies — but OTA patching itself carries no special compliance risk on Android.

Audit trails for regulated industries

If your team operates in a regulated industry, document each patch with a short description of what changed. The Shorebird console stores version history and audit logs for every patch across every release. Pair that record with your existing change management process and you have the audit trail most compliance frameworks require.

07 Your next step: ship a patch in 30 min

The full path from a clean Flutter project to your first OTA patch:

1. **Install the CLI**

   ```
   curl … | bash
   ```

2. **Log in & scaffold**

   ```
   shorebird login → shorebird init
   ```

3. **Cut a release**

   ```
   shorebird release android / ios
   ```

4. **Make a one-line Dart fix**

   ```
   update a string, fix a null check, adjust a param
   ```

5. **Push your first patch**

   ```
   shorebird patch android / ios
   ```

Shorebird's free tier supports the complete Code Push workflow, including the Shorebird console and patch delivery up to the free-tier patch limit. That's enough to run a full end-to-end proof of concept without a payment method on file.

GET STARTED IN MINUTES

Start shipping faster.

Free tier · No credit card required.

shorebird.dev

shorebird