HOTPATCH OTA

Over-The-Air Update System for React Native

Complete System Architecture & Technical Reference

Version 1.0 | 2025

Rust CLI Build & Deploy	**Go Backend** API & Storage	**React Native SDK** App Integration

1. Executive Summary

HotPatch OTA is a self-hosted, production-grade Over-The-Air (OTA) update system purpose-built for React Native applications. It enables development teams to push JavaScript bundle updates directly to end users without requiring a full app store release cycle, significantly reducing time-to-fix for critical bugs and accelerating feature delivery.

The system is composed of three tightly integrated components: a Rust-based Command Line Interface (CLI) for developers to build and publish updates, a Go-powered backend API that manages release distribution and device tracking, and a React Native SDK embedded in the mobile application that silently downloads, verifies, and applies updates.

Core Value Proposition
Teams using HotPatch OTA can deliver a critical hotfix to 100% of active users within minutes of discovery, rather than waiting 24-72 hours for app store review and user-initiated updates. The system supports phased rollouts, mandatory updates, automatic crash rollback, and release channels (production / staging / beta).

1.1 Key Capabilities

- Instant JS bundle delivery without app store involvement
- SHA256 integrity verification and Ed25519 cryptographic signing

- Crash detection with automatic rollback to last stable bundle
- Phased rollout control (e.g. release to 5% of devices, then 50%, then 100%)
- Release channels: production, staging, beta, and custom
- Full audit trail of releases, device installations, and rollback events
- Cross-platform support: Android (Kotlin) and iOS (Swift)
- Future-proof differential patch engine via bsdiff / xdelta

1.2 Technology Stack Summary

Component	Technology	Reason
CLI Tool	Rust + Tokio + Clap	Performance, safe async, excellent crypto crates
Backend API	Go + Gin/Fiber + GORM	High concurrency, low latency, simple deployment
Database	PostgreSQL 15+	Relational integrity, JSON support, mature ecosystem
Object Storage	AWS S3 / Cloudflare R2	Scalable, CDN-friendly bundle distribution
Android SDK	Kotlin	Modern Android, coroutines, strong typing
iOS SDK	Swift	Native performance, Swift Package Manager support
Cache (optional)	Redis	Session caching, update check rate limiting

2. System Architecture

2.1 High-Level Architecture

The system follows a three-tier architecture with clear separation of concerns between the developer toolchain, the server infrastructure, and the mobile SDK runtime.

```
                    DEVELOPER MACHINE
                          |
               [Rust CLI  rn-ota release]
                          |
                  Build RN Bundle (npx)
                  Compress + Hash
                  Sign with Ed25519
                          |
              POST /releases  (multipart)
                         |
             ┌──────────────────────────────┐
             │       GO BACKEND API          │
             │   (Gin + PostgreSQL + GORM)   │
             │                               │
             │  Auth  │  Releases  │  Devices│
             └──────────────────────────────┘
                  /         \
          PostgreSQL    Object Storage
          (metadata)    (S3 / R2 bundles)
                         |
                 CDN Distribution
                         |
             ┌──────────────────────────────┐
             │     REACT NATIVE APP          │
             │  (iOS Swift / Android Kotlin)│
             │                               │
             │  SDK: Check → Download →      │
             │  Verify → Apply → Restart     │
             └──────────────────────────────┘
```

2.2 Update Flow (End-to-End)

The following sequence describes the complete lifecycle from a developer issuing a release command to a user's app updating silently in the background:

1. Developer runs rn-ota release --platform android --version 1.2.0 --channel production
2. CLI builds the React Native JavaScript bundle via npx react-native bundle
3. CLI zips the bundle + assets, computes SHA256 hash, signs with Ed25519 private key
4. CLI uploads the signed bundle to the Go backend via POST /releases
5. Backend stores metadata in PostgreSQL and uploads the bundle to S3/R2
6. Backend returns a release ID and confirms storage success
7. App launches → SDK calls GET /update/check with current version, platform, device ID
8. Backend compares version against release table, respects channel + rollout percentage
9. If update available, backend returns bundle URL, hash, mandatory flag
10. SDK downloads bundle in background, verifies SHA256 hash locally
11. SDK verifies Ed25519 signature against embedded public key
12. SDK atomically replaces bundle file on disk in app's private storage
13. On next app start (or immediate restart if mandatory), new bundle is loaded
14. If app crashes on first launch of new bundle, SDK reverts to previous bundle
15. Crash event is reported to backend, triggering automated rollback logging

2.3 Component Interaction Diagram

Each component communicates over well-defined interfaces. The CLI communicates with the backend exclusively over HTTPS REST. The SDK communicates with the backend for update checks and directly with object storage CDN URLs for bundle downloads, minimizing backend load.

From	To	Protocol / Interface
Rust CLI	Go Backend	HTTPS REST + JWT Bearer Token
Go Backend	PostgreSQL	TCP (GORM / SQLX connection pool)
Go Backend	S3 / Cloudflare R2	HTTPS (AWS SDK v2 for Go)
React Native SDK	Go Backend	HTTPS REST (update check, analytics)
React Native SDK	S3 / Cloudflare R2	HTTPS direct download via signed URL
Go Backend	Redis (optional)	TCP (in-memory cache for update check)

COMPONENT 1 — RUST CLI

3. Rust CLI — rn-ota

The CLI is the developer-facing tool that manages all aspects of publishing OTA updates. Built in Rust for performance and reliability, it handles bundle building, compression, hashing, signing, and upload in a single command. It is distributed as a standalone binary requiring no runtime dependencies.

3.1 Responsibilities

- Build React Native JavaScript bundles (Android + iOS) via shell subprocess
- Compress bundle + assets into a zip archive
- Compute SHA256 hash of the compressed bundle for integrity verification
- Sign the bundle with an Ed25519 private key for authenticity
- Upload bundle to backend API with version metadata
- Manage release channels (production, staging, beta, custom)
- Execute rollback commands to designate a previous version as active
- Generate differential patches between bundle versions (Phase 3)
- Authenticate with the backend using API tokens stored in local config

3.2 Tech Stack & Dependencies

The Cargo.toml dependencies below are the recommended set for the MVP and Phase 2 builds:

```
[package]
name = "rn-ota"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = { version = "4", features = ["derive"] }  # CLI argument parsing
reqwest = { version = "0.11", features = ["json", "multipart", "stream"] }
tokio = { version = "1", features = ["full"] }   # Async runtime
serde = { version = "1", features = ["derive"] } # Serialization
serde_json = "1"
indicatif = "0.17"    # Progress bars for upload/download
zip = "0.6"           # Bundle compression
sha2 = "0.10"         # SHA256 hashing
ed25519-dalek = "2"   # Bundle signing
walkdir = "2"         # Recursive directory traversal
anyhow = "1"          # Error handling
dirs = "5"            # Platform-specific config paths
toml = "0.8"          # Config file parsing
colored = "2"         # Terminal output coloring
```

3.3 Project Structure

```
rn_ota_cli/
├── src/
│   ├── main.rs                   # Entry point, command dispatch
│   ├── commands/
│   │   ├── release.rs            # rn-ota release (main publish flow)
│   │   ├── patch.rs              # rn-ota patch (diff uploads, Phase 3)
│   │   ├── rollback.rs           # rn-ota rollback (revert a version)
│   │   └── login.rs              # rn-ota login (store API token)
│   ├── bundle/
│   │   ├── builder.rs            # Shell: npx react-native bundle
│   │   ├── compressor.rs         # ZIP creation + SHA256
│   │   └── diff.rs               # bsdiff / xdelta wrapper (Phase 3)
│   ├── api/
│   │   └── client.rs             # HTTP client, auth, retry logic
│   ├── config/
│   │   └── config.rs             # Read/write ~/.rn-ota/config.toml
│   ├── signing/
│   │   └── signer.rs           # Ed25519 key management + signing
│   └── utils/
│       └── hash.rs             # SHA256 file hashing utilities
└── Cargo.toml
```

3.4 CLI Commands Reference

3.4.1 Login

Authenticates with the backend and stores the API token to the user's local config file at ~/.rn-ota/config.toml. All subsequent commands read the token from this file.

```
rn-ota login

Prompts for:
  API Endpoint: https://api.yourserver.com
  API Token: <paste token from dashboard>

Stores to: ~/.rn-ota/config.toml
```

3.4.2 Release (Primary Command)

The release command performs the full publish pipeline in sequence. Each step is shown with a progress indicator. The command is idempotent — re-running with the same version will return a 409 Conflict from the server.

```
rn-ota release \
  --platform android \
  --version 1.2.0 \
  --channel production \
  --mandatory false \
  --rollout 10

Options:
  --platform     android | ios
  --version      Semver string (must match your app's native version)
  --channel      production | staging | beta | <custom>
  --mandatory    Force update on check (default: false)
  --rollout      Rollout percentage 1-100 (default: 100)
  --entry-file   JS entry point (default: index.js)
  --build-dir    Local output dir (default: ./build)
```

Internal pipeline executed by release:

16. Run npx react-native bundle for the target platform

17. Recursively walk build directory and zip all files preserving paths
18. Compute SHA256 hash of the resulting zip file
19. Sign the hash with the developer's Ed25519 private key
20. POST multipart/form-data to /releases: zip + metadata JSON + signature
21. Print confirmation with release ID and CDN URL

3.4.3 Rollback

Instructs the backend to mark a previously published version as the active release for a given channel. All devices checking for updates will receive this version until overridden.

```
rn-ota rollback \
  --platform android \
  --version 1.1.5 \
  --channel production
```

3.4.4 Patch (Phase 3 — Differential Updates)

Generates and uploads a binary diff between two versions, reducing download size for incremental updates. Uses bsdiff or xdelta3 under the hood, making Rust's subprocess handling ideal for this workflow.

```
rn-ota patch \
  --from 1.1.5 \
  --to 1.2.0 \
  --platform android
```

3.5 Bundle Building Logic

The CLI shells out to the React Native bundler using std::process::Command. The Rust process captures stdout and stderr, streaming them to the terminal while checking the exit code for errors.

```
// builder.rs (simplified)
pub async fn build_bundle(platform: &str, entry: &str, out_dir: &str) -> Result<()>
{
    let status = Command::new("npx")
        .args([
            "react-native", "bundle",
            "--platform", platform,
            "--dev", "false",
            "--entry-file", entry,
            "--bundle-output",
            &format!("{}/index.{}.bundle", out_dir, platform),
            "--assets-dest", out_dir,
        ])
        .status()?;
    if !status.success() {
        anyhow::bail!("React Native bundle command failed");
    }
    Ok(())
}
```

3.6 Bundle Signing

Security is implemented via a two-step process: SHA256 for integrity (detecting corruption or tampering) and Ed25519 for authenticity (verifying the update was published by a trusted party).

File	Purpose
bundle.zip	Compressed JavaScript bundle and assets
signature.sig	Ed25519 signature over SHA256 hash of bundle.zip
metadata.json	version, platform, channel, hash, mandatory, rollout_percentage

The Ed25519 private key is stored in ~/.rn-ota/signing_key.pem and must never be committed to source control. The corresponding public key is embedded at compile time into the React Native SDK. Key rotation requires an SDK update and new native app release.

COMPONENT 2 — GO BACKEND API

4. Go Backend API

The Go backend is the central coordination layer of the OTA system. It authenticates CLI requests, stores release metadata in PostgreSQL, manages bundle files in object storage, and serves the high-throughput update-check endpoint used by every app launch across your user base.

4.1 Responsibilities

- JWT-based authentication for CLI and (optionally) dashboard access
- Accept bundle uploads from the CLI and store them in S3 / Cloudflare R2
- Manage releases: create, list, activate, rollback, archive

- Serve the update check endpoint: evaluate version, channel, rollout cohort
- Track device registrations, current version, and last-seen timestamps
- Record installation events (success, failure, rollback) from the SDK
- Enforce rollout percentage using stable device-based cohort bucketing
- Expose metrics and analytics endpoints for the Phase 2 dashboard

4.2 Tech Stack

Library / Tool	Purpose	Notes
Go 1.22+	Runtime	Latest stable, generics support
Gin or Fiber	HTTP framework	Gin for maturity; Fiber for performance
GORM or SQLX	Database ORM/query	GORM for rapid dev; SQLX for control
golang-jwt/jwt/v5	JWT authentication	Access + refresh token support
aws/aws-sdk-go-v2	S3 / Cloudflare R2	R2 is S3-compatible, use endpoint override
go-redis/redis/v9	Redis cache (optional)	Cache update check responses
golang-migrate	DB migrations	Version-controlled schema changes
Docker + docker-compose	Deployment	Single-command local + production setup
testify	Testing	Assert helpers for unit + integration tests

4.3 Project Structure

```
server/
├── cmd/
│   └── main.go                   # Server entry: config load, DI, start
├── internal/
│   ├── api/
│   │   ├── handlers/
│   │   │   ├── auth_handler.go    # POST /auth/token
│   │   │   ├── release_handler.go # POST/GET /releases
│   │   │   ├── update_handler.go  # GET /update/check
│   │   │   └── device_handler.go  # POST /devices
│   │   ├── middleware/
│   │   │   ├── auth.go            # JWT validation middleware
│   │   │   └── rate_limit.go      # Per-device rate limiting
│   │   └── routes.go              # Route registration
│   ├── services/
│   │   ├── release_service.go     # Business logic: create, rollback
│   │   ├── update_service.go      # Update check + cohort logic
│   │   └── device_service.go     # Device registration + tracking
│   ├── repository/
│   │   ├── release_repo.go       # DB queries for releases
│   │   └── device_repo.go        # DB queries for devices
│   ├── models/
│   │   ├── app.go                # App model
│   │   ├── release.go            # Release model
│   │   └── device.go             # Device + Installation models
│   └── storage/
│       └── s3.go                 # S3/R2 upload, presigned URL gen
├── migrations/
│   ├── 001_create_apps.sql
│   ├── 002_create_releases.sql
│   └── 003_create_devices.sql
├── docker-compose.yml
└── go.mod
```

4.4 Database Schema

4.4.1 apps table

```
CREATE TABLE apps (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name        VARCHAR(255) NOT NULL UNIQUE,
  platform    VARCHAR(10)  NOT NULL CHECK (platform IN ('android', 'ios')),
  created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

4.4.2 releases table

```
CREATE TABLE releases (
  id                  UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  app_id              UUID NOT NULL REFERENCES apps(id),
  version             VARCHAR(50) NOT NULL,
  channel             VARCHAR(50) NOT NULL DEFAULT 'production',
  bundle_url          TEXT NOT NULL,
  hash                VARCHAR(64) NOT NULL,   -- SHA256 hex
  signature           TEXT NOT NULL,          -- Ed25519 base64
  mandatory           BOOLEAN NOT NULL DEFAULT false,
  rollout_percentage  SMALLINT NOT NULL DEFAULT 100,
  is_active           BOOLEAN NOT NULL DEFAULT true,
  created_at          TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  UNIQUE(app_id, version, channel)
);
CREATE INDEX idx_releases_active ON releases(app_id, channel, is_active);
```

4.4.3 devices table

```
CREATE TABLE devices (
  id                UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  device_id         VARCHAR(255) NOT NULL UNIQUE,  -- SDK-generated UUID
  app_id            UUID NOT NULL REFERENCES apps(id),
  platform          VARCHAR(10) NOT NULL,
  current_version   VARCHAR(50),
  last_seen         TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

4.4.4 installations table

```
CREATE TABLE installations (
  id            UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  device_id     UUID NOT NULL REFERENCES devices(id),
  release_id    UUID NOT NULL REFERENCES releases(id),
  status        VARCHAR(20) NOT NULL   -- 'applied' | 'failed' | 'rolled_back'
                CHECK (status IN ('applied', 'failed', 'rolled_back')),
  installed_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

4.5 API Endpoints

4.5.1 Authentication

Method	Path	Description
POST	/auth/token	Exchange API key for JWT access token
POST	/auth/refresh	Refresh an expired access token

4.5.2 Releases (CLI — JWT Required)

Method	Path	Description
POST	/releases	Upload new bundle (multipart/form-data)
GET	/releases	List all releases (with filters)
GET	/releases/:id	Get single release detail
PATCH	/releases/:id/rollback	Designate version as active (rollback)
DELETE	/releases/:id	Archive (soft delete) a release

4.5.3 Update Check (SDK — App Key Required)

This is the highest-traffic endpoint in the system — it is called on every app launch. It must respond in under 50ms at the 99th percentile. Redis caching is strongly recommended for production.

```
GET /update/check

Request body:
{
  "appId":   "550e8400-e29b-41d4-a716-446655440000",
  "deviceId": "a1b2c3d4-...",
  "version": "1.1.5",
  "platform": "android",
  "channel":  "production"
}

Response (update available):
{
  "updateAvailable": true,
  "bundleUrl":       "https://cdn.example.com/bundles/abc123.zip",
  "hash":            "e3b0c44298fc1c149afb...",
  "signature":       "base64-ed25519-sig",
  "mandatory":       false,
  "version":         "1.2.0"
}

Response (no update):
{
  "updateAvailable": false
}
```

4.5.4 Device & Installation Reporting (SDK)

Method	Path	Description
POST	/devices	Register device or update last_seen
POST	/installations	Report install result (applied/failed/rolled_back)

4.6 Rollout Percentage — Cohort Bucketing

The rollout_percentage field controls what fraction of devices receive an update. A naive random check would cause a device to flip in and out of rollout groups between app launches. Instead, the backend uses stable cohort bucketing:

```
// update_service.go
func isInRollout(deviceID string, rolloutPct int) bool {
    h := fnv.New32a()
    h.Write([]byte(deviceID))
    bucket := int(h.Sum32() % 100)  // 0-99, stable per device
    return bucket < rolloutPct
}
```

This ensures a device that receives the update on one launch continues to receive it on all subsequent launches, and devices outside the rollout consistently do not.

COMPONENT 3 — REACT NATIVE SDK

5. React Native SDK

The SDK is the most critical and complex component of the system. It runs silently inside every user's app, managing the entire update lifecycle from detection through application. It must handle edge cases like mid-download network loss, corrupted bundles, and app crashes after an update with zero user disruption.

5.1 SDK Structure

```
rn-ota-sdk/
├── android/
│   ├── src/main/java/com/hotpatch/
│   │   ├── OTAUpdateManager.kt    # Core update logic
│   │   ├── OTAModule.kt           # React Native bridge module
│   │   ├── OTAPackage.kt          # Package registration
│   │   ├── SignatureVerifier.kt   # Ed25519 verification
│   │   └── CrashDetector.kt       # Boot loop detection
│   └── build.gradle
├── ios/
│   ├── OTAUpdateManager.swift # Core update logic
│   ├── OTABridge.m            # Objective-C bridge header
│   ├── SignatureVerifier.swift # Ed25519 via CryptoKit
│   └── CrashDetector.swift    # Launch count / crash detection
├── src/
│   ├── index.ts               # JS API: OTAClient class
│   └── types.ts               # TypeScript type definitions
└── package.json
```

5.2 Android Implementation (Kotlin)

5.2.1 Override Bundle Path

The fundamental mechanism for OTA updates on Android is overriding the ReactActivity's bundle source. The OTA SDK intercepts this call and returns the path to the locally stored updated bundle if one exists and passes verification.

```
// In your MainActivity.kt
class MainActivity : ReactActivity() {
    override fun getJSBundleFile(): String? {
        return OTAUpdateManager.getBundlePath(this)
    }
}

// OTAUpdateManager.kt
object OTAUpdateManager {
    fun getBundlePath(context: Context): String? {
        val otaBundle = File(context.filesDir, "ota/bundle/index.android.bundle")
        return if (otaBundle.exists() && isVerified(context)) {
            otaBundle.absolutePath
        } else {
            null  // Fall back to bundled asset
        }
    }
}
```

5.2.2 Bundle Storage Location

```
# App private storage (no permissions required)
/data/data/<your.package.name>/files/
  ota/
    bundle/
      index.android.bundle   # Active bundle
      assets/                # Images, fonts, etc.
    pending/
      bundle.zip             # Downloading
    previous/
      index.android.bundle   # Last stable (for rollback)
    metadata.json            # Version, hash, installed_at
    crash_count.json         # Boot loop detection counter
```

5.3 iOS Implementation (Swift)

5.3.1 Override Bundle URL

```
// AppDelegate.swift
override func sourceURL(for bridge: RCTBridge!) -> URL! {
    return OTAUpdateManager.bundleURL() ?? bundleURL()
}

// OTAUpdateManager.swift
class OTAUpdateManager {
    static func bundleURL() -> URL? {
        let otaPath = FileManager.default
            .urls(for: .documentDirectory, in: .userDomainMask)[0]
            .appendingPathComponent("ota/bundle/main.jsbundle")
        guard FileManager.default.fileExists(atPath: otaPath.path),
              isVerified() else { return nil }
        return otaPath
    }
}
```

5.4 Update Flow (SDK Internal)

The SDK executes the following flow on every app launch in a background coroutine / DispatchQueue to avoid blocking the UI thread:

22. Read current installed OTA version from metadata.json (or native bundle version if no OTA installed)
23. Check crash_count.json — if crash count >= 2, skip update check and flag for rollback reporting
24. Call GET /update/check with current version, platform, deviceId, channel
25. If updateAvailable == false, exit silently
26. If updateAvailable == true, download bundle.zip to pending/ directory using streaming (resume-capable)
27. Compute SHA256 of downloaded file — compare to hash field in response. Abort if mismatch.
28. Verify Ed25519 signature against embedded public key. Abort if invalid.
29. Copy current bundle/ to previous/ (atomic rename if possible)
30. Extract bundle.zip to bundle/ directory
31. Write new metadata.json with version and installed_at
32. Reset crash_count.json to 0
33. If mandatory == true: immediately trigger ReactContext reload. Otherwise: apply on next app launch.

5.5 Security Implementation

5.5.1 SHA256 Integrity

Every downloaded bundle is verified against the SHA256 hash returned by the server before being applied. A mismatch indicates either network corruption or a tampered bundle — both result in the download being discarded.

5.5.2 Ed25519 Signature Verification

The public key for signature verification is embedded at compile time in the SDK. On Android, it is stored as a raw resource. On iOS, it is embedded in the app bundle. The signature is verified after hash confirmation — if either check fails, the update is rejected and the error is reported to the server.

```
// Android: SignatureVerifier.kt
fun verify(bundleBytes: ByteArray, signatureBase64: String): Boolean {
    val publicKey = loadEmbeddedPublicKey()
    val signature = Base64.decode(signatureBase64, Base64.DEFAULT)
    return Ed25519.verify(bundleBytes, signature, publicKey)
}

// iOS: SignatureVerifier.swift (uses CryptoKit)
func verify(data: Data, signature: Data) throws -> Bool {
    let publicKey = try Curve25519.Signing.PublicKey(rawRepresentation:
embeddedKeyData)
    return publicKey.isValidSignature(signature, for: data)
}
```

5.5.3 Certificate Pinning (Optional)

For high-security environments, the SDK can be configured to pin the TLS certificate of the backend API, preventing man-in-the-middle attacks even on devices with compromised certificate authorities. This is implemented via OkHttp CertificatePinner on Android and URLSession with custom URLSessionDelegate on iOS.

5.6 Crash Rollback System

The crash rollback system protects users from being permanently stuck on a broken bundle. It uses a launch counter stored in a file (crash_count.json) that is incremented on launch and cleared on successful first render.

```
Launch #1 with new bundle:
  crash_count = 1
  App renders successfully -> crash_count = 0 (cleared)
```

```
Launch #1 with new bundle:
  crash_count = 1
  App crashes -> crash_count persists

Launch #2 (after crash):
  crash_count = 2 >= threshold
  -> Copy previous/ bundle back to bundle/
  -> Report installation status 'rolled_back' to server
  -> App loads last stable bundle
```

Important Implementation Note
The crash_count file must be written before ReactContext initializes — in the native Activity/AppDelegate onCreate equivalent — not in JavaScript. If written in JS, a crash in the native bridge would prevent it from being written at all, defeating the protection.

6. Security Architecture

Security is implemented in layers across all three components. The goal is to ensure that only bundles produced by authorized developers, uploaded to a trusted server, and delivered without tampering can ever be applied to a user's device.

Threat	Mitigation
Corrupted bundle (network error)	SHA256 hash verification on device before apply
Tampered bundle (MITM attack)	Ed25519 signature verification with embedded public key
HTTP downgrade attack	HTTPS enforced; optional certificate pinning
Unauthorized publish	JWT authentication on all CLI-facing API endpoints
Replay attack (old bundle)	Version field checked server-side; monotonic versioning enforced
Bundle decryption (IP theft)	Bundle encryption (Pro feature, Phase 3)
Boot loop	Crash counter with automatic rollback to previous bundle

7. Development Phases

Phase 1 — MVP (Estimated: 3-4 weeks)

Goal: A working end-to-end OTA update that a developer can publish and a real app can receive and apply.

- Rust CLI: login, release (bundle + upload only)
- Go Backend: POST /releases, GET /update/check, basic JWT auth
- PostgreSQL: apps, releases tables
- S3/R2 integration for bundle storage
- Android SDK: download, hash verify, apply bundle, getJSBundleFile override
- iOS SDK: download, hash verify, apply bundle, bundleURL override

Phase 2 — Production Ready (Estimated: 6-8 weeks)

Goal: Stable enough to run in production with real users.

- Rust CLI: rollback command, release channels, mandatory flag
- Go Backend: rollback API, device tracking, installations table, rollout % with cohort bucketing
- Android + iOS SDK: crash rollback system, mandatory update enforcement
- Next.js dashboard: release management, device overview, rollback button
- Redis caching for /update/check
- Rate limiting and DDoS protection

Phase 3 — Advanced (Estimated: 2-3 months)

- Differential patches: bsdiff / xdelta integration in Rust CLI
- Ed25519 signing enabled end-to-end
- Bundle encryption (Pro tier)
- A/B testing via named cohorts
- Full analytics: install rates, crash rates, rollback frequency, version distribution
- Self-hosted enterprise deployment guide + Terraform / Helm charts

8. Performance Considerations

8.1 Update Check Endpoint

The GET /update/check endpoint is the single most critical endpoint in the system. It is called on every app launch by every active user. At 100,000 daily active users, assuming an average of 3 app launches per day, this endpoint receives ~300,000 requests per day, or ~3.5 requests per second at steady state with peaks around morning commute hours.

- Target P99 latency: < 50ms

- Target P50 latency: < 10ms
- Redis cache: cache the response per (appId, channel, platform) tuple with a 60-second TTL
- Read replica: route /update/check queries to a PostgreSQL read replica
- Go's goroutine-based concurrency handles thousands of concurrent connections with a single small instance

8.2 Bundle Download Performance

Bundle downloads are served directly from S3 or Cloudflare R2 CDN — the Go backend is not in the download path. This means download performance scales horizontally with CDN capacity and geographic distribution.

- Recommend Cloudflare R2 over S3 for zero egress fees and global CDN
- Enable HTTP range requests for resume-capable downloads (important for large bundles on slow connections)
- Compress bundles with maximum zip compression to minimize download size
- Differential patches (Phase 3) can reduce update size by 80-95% for small code changes

8.3 SDK Performance

- Update check runs on a background thread — zero UI thread blocking
- Bundle verification (hash + signature) runs on background thread
- Atomic bundle replacement using file rename to prevent partial state
- Metadata reads (version check) use a simple JSON file — microsecond latency

9. Monetization Strategy

Tier	Features	Target
Free	Basic OTA, unlimited updates, 1 channel, no rollout control	Indie developers, side projects
Pro ($49/mo)	Differential updates, rollout %, analytics, 3 channels, priority support	Startups, small teams
Enterprise	Self-hosted, SLA, on-premise, SSO, unlimited channels, dedicated support	Large organizations

10. Deployment Guide

10.1 Backend Deployment

```
# docker-compose.yml (development)
version: '3.9'
services:
  api:
    build: .
    ports: ['8080:8080']
    environment:
      DATABASE_URL: postgres://user:pass@db:5432/hotpatch
      JWT_SECRET: your-secret-here
      S3_BUCKET: hotpatch-bundles
      S3_ENDPOINT: https://<accountid>.r2.cloudflarestorage.com
    depends_on: [db, redis]
  db:
    image: postgres:15-alpine
    volumes: ['pgdata:/var/lib/postgresql/data']
  redis:
    image: redis:7-alpine
volumes:
  pgdata:
```

10.2 SDK Installation

```
# Install SDK
npm install rn-ota-sdk

# iOS: link native modules
cd ios && pod install

# Configure in your app's initialization
import OTA from 'rn-ota-sdk';

OTA.configure({
  apiUrl: 'https://api.yourserver.com',
  appId: 'your-app-uuid',
  channel: 'production',
  checkOnLaunch: true,
});
```

10.3 Environment Variables Reference

Variable	Description	Required
DATABASE_URL	PostgreSQL connection string	Yes

JWT_SECRET	Secret for signing JWT tokens (min 32 chars)	Yes
S3_BUCKET	S3 or R2 bucket name	Yes
S3_ENDPOINT	Custom endpoint for Cloudflare R2	R2 only
AWS_ACCESS_KEY_ID	S3 / R2 access key	Yes
AWS_SECRET_ACCESS_KEY	S3 / R2 secret key	Yes
REDIS_URL	Redis connection string	No
PORT	HTTP server port (default: 8080)	No

11. Product Naming Recommendation

Based on the feature set, target audience, and brand clarity, the recommended product name is HotPatch. It clearly communicates the primary value (patching apps on-the-fly), has no existing major competitor with that exact name, and is memorable for developer audiences. Secondary recommendation is PatchFlow for a more enterprise-neutral feel.

Name	Pros	Cons
HotPatch	Clear, memorable, developer-friendly, conveys speed	May conflict with hotfix terminology
PatchFlow	Professional, process-oriented, enterprise-friendly	Less evocative of mobile/React Native
LiveNative	Emphasizes real-time + React Native connection	Less clear it's about updates specifically
ReactOTA	Very descriptive, SEO-friendly	Too generic, harder to brand

— End of Document —

HotPatch OTA | Technical Documentation v1.0 | Proprietary & Confidential