

iOS app

Database pair and launch integrity

🕒 4 min read · Updated 25 September 2026 · Verified against 417c92f on 2026-09-25

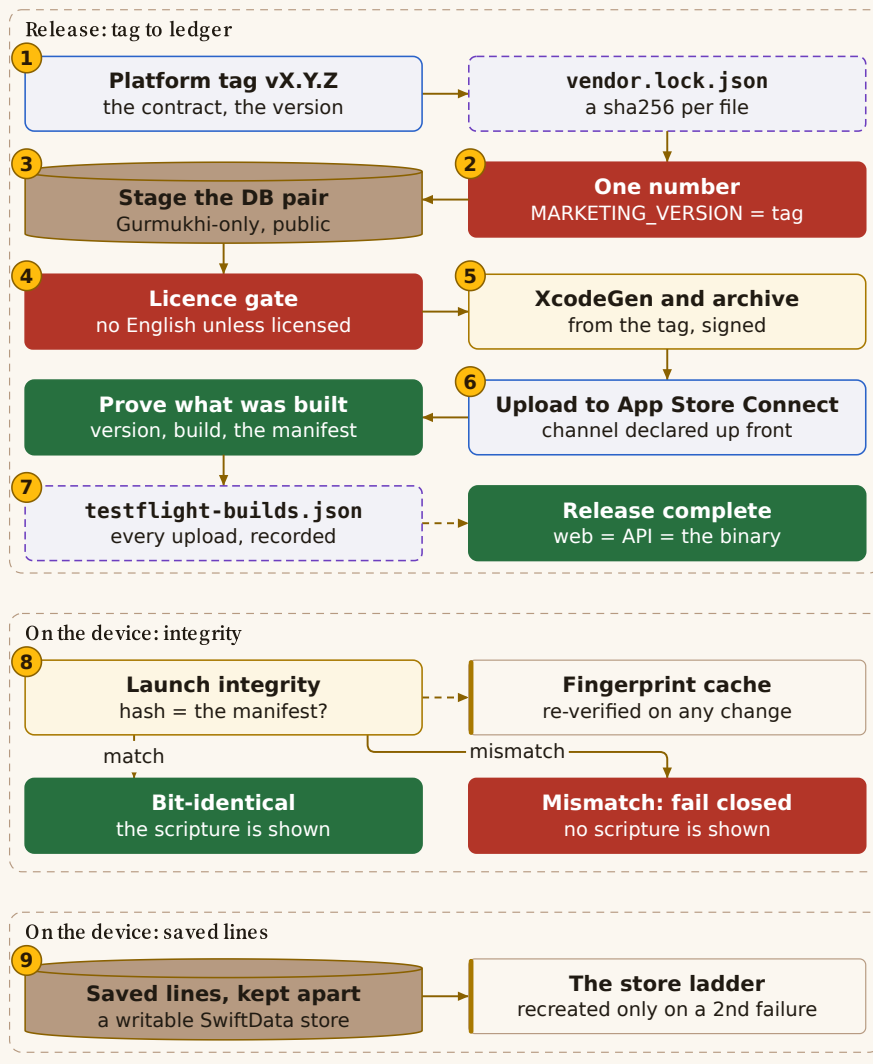
The app bundles a database and its certified manifest in two profiles; a licence gate decides which may ship; at launch the app hashes the database and refuses scripture on a mismatch.

<https://docs.gurbanisoul.com/ios/db-pair-and-launch-integrity/>

The app carries the scripture with it. Two files travel together — `sggs-ios.sqlite` and `sggs-ios.manifest.json` — and the manifest's checksums are what every gate, from the archive script to the app's own launch, verifies against. The poster follows a release from the vendor sync to the ledger, then the two ladders that run on the device.

12 · iOS release and integrity

From the platform tag to an App Store binary with the same number — every step a gate — and what the app proves before it shows a line



Poster 12 — iOS release and integrity: vendor sync, one version number, the archive script's gates, the ledger, the launch-integrity ladder and the bookmarks store ladder

- Vendor the release.** make vendor-sync-platform REF=vX.Y.Z copies the platform's golden contract and contributors roster into the app repository at their upstream paths and records the commit, a sha256 per file and the platform's APP_VERSION in vendor.lock.json. The dataset is the same sggs-data pin the platform serves; the iOS database builder is vendored from it. CI proves every vendored file equals its lock entry and its source.

2. **One number.** `MARKETING_VERSION` must equal the vendored platform version; `CURRENT_PROJECT_VERSION` stays 1 and every upload passes `BUILD=N`, kept monotonic within a marketing version by the ledger before any archive starts. A release is re-archived at the same number even for a web-only change.
3. **Stage the database pair.** From a clean, detached worktree at the tag, `make testflight` derives the database for the chosen profile — public, Gurmukhi-only, by default — into a staging directory under `ios/App/build`, with its manifest: `profile`, `db_sha256`, `scripture_sha256`, `en_bundled`, `size`, `tables`. The developer's own pair in `ios/Resources` is never read or written.
4. **The licence gate.** `check_release_license.sh` runs on that exact artifact: English may be bundled only with `LICENSED: true` in the attestation; `scripture_sha256` must equal the certified corpus value; the artifact must hash to its manifest; every golden suite must be present; and on the App Store channel the Nitnem non-SGGS text must carry the scholar's `REVIEWED: true`.
5. **XcodeGen and archive.** A temporary sibling spec differs from `project.yml` in exactly three lines (the name and the two database resource paths, pointed at the staging directory); XcodeGen generates a throw-away project; `xcodebuild` archives the Release configuration with the Team ID and build number from the command line under automatic signing. Xcode must be at or above Apple's SDK floor.
6. **Upload and prove.** The archive is exported for App Store Connect; with `UPLOAD=1` the export is the upload. Then the script proves what was built: the `Info.plist` version and build and the manifest inside the `.app` equal what was staged. `CHANNEL=appstore` is declared up front, because a `TestFlight` binary and an App Store binary are the same upload.
7. **The ledger, and “complete”.** The build is appended to `ios/testflight-builds.json` — version, build, `source_commit`, `platform_commit`, `dataset_commit`, `profile`, `channel`, `db_sha256`, Xcode and SDK — and committed with the release; `make appstore-preflight` submits only a build recorded on the App Store channel. `scripts/release/check_release_complete.py X.Y.Z` then proves production's web and API report `X.Y.Z` from the tag's commit and that the ledger has an appstore build of `X.Y.Z` with that `platform_commit`.
8. **On the device: launch integrity.** Before any scripture is shown, `LaunchIntegrity` hashes the bundled database and compares it with the manifest's `db_sha256`; a match means the database is bit-identical to the proven corpus. After one full streaming verify a fingerprint (sha, file size and mtime, bundle version, manifest sha) lets later launches skip the re-hash — but any change forces a full re-verify and a failure is never cached. On any failure the app refuses to present scripture and shows the report under `About → Integrity`. The threat model is accidents; anti-tamper is code signing's job.
9. **On the device: the bookmarks store.** Saved verses live in a separate writable `SwiftData` store — never the read-only corpus — holding verbatim Gurmukhi without saroop markup, under a versioned schema (`SavedLineSchemaV1`) with a migration plan. Opening it climbs a ladder: the versioned schema, a retry, and only on a second consecutive failure destroy-and-recreate, so a transient error never costs a reader their bookmarks. Nitnem progress is its own JSON file in the App Group with a frozen v1 schema.

Two profiles, one builder

`pipeline/build_ios_db.py` (owned by `sggs-data`, vendored at the dataset's commit) derives the iOS database from the pinned `db/sggs.sqlite` and writes the manifest beside it:

Profile	Contents	May ship publicly?
personal	every table, including the English translations and <code>fts_en</code>	only with <code>LICENSED: true</code> in <code>ios/Resources/TRANSLATION-LICENSE.md</code> — the Khalsa translation is personal-use until licensed in writing

Profile	Contents	May ship publicly?
public	Gurmukhi-only: the translation layer dropped, everything else verbatim (timing, banis, insights included)	yes — the profile the App Store build uses

The manifest records the profile, the database's `db_sha256`, the `scripture_sha256` (a checksum over the scripture rows alone, the same definition sggs-data's fingerprint uses), whether English is bundled, the byte size and the tables. On a developer's machine the pair is git-ignored for the database and tracked for the manifest; `make ios-db-check` proves they agree after a branch switch and `make ios-db-repair` re-derives the database without touching git.

The licence gate

gurbani-soul-ios/pipeline/check_release_license.sh · lines 1-25

```
#!/usr/bin/env bash
# check_release_license.sh – the mechanical NOTICE.md hard-gate for any PUBLIC iOS release.
#
# Fails (exit 1) if:
# (a) the manifest says en_bundled=true AND the licence attestation file
# (ios/Resources/TRANSLATION-LICENSE.md, override with $SGGS_LICENSE_ATTESTATION)
# does not contain the line `LICENSED: true` – the Khalsa English translation is
# PERSONAL, LOCAL, NON-COMMERCIAL use only until licensed in writing; a public /
# TestFlight / App Store build must either be built with `--profile public` or
# carry a filled-in attestation;
# (b) scripture_sha256 differs from the certified corpus value – scripture must be
# byte-identical to the proven corpus, always;
# (c) the DB artifact's sha256 doesn't match its manifest;
# (d) any golden-vector suite recorded in contract/_meta.json is missing or empty;
# (e) the bundled DB carries the Nitnem registry with NON-SGGS text (extra_lines) and
# either that table has an English column (never allowed) or the scholar review
# is not attested with `REVIEWED: true` in ios/Resources/NITNEM-REVIEW.md
# (override with $SGGS_NITNEM_ATTESTATION) – Sri Dasam Granth / Ardaas text is
# not covered by the SGGS reconcile proof. Unreviewed text is a WARNING for
# TestFlight builds (the app labels it "under review") and a hard FAIL when
# SGGS_RELEASE_CHANNEL=appstore (App Store submission).
#
# Usage: pipeline/check_release_license.sh [manifest] [db]
# defaults: ios/Resources/sggs-ios-public.manifest.json + its sibling .sqlite
# (pass the personal manifest to CONFIRM it is blocked – that's the negative test)
```

Read from the source at build time — [view these lines on GitHub at a386422](#).

Source: [pipeline/check_release_license.sh](#)² in gurbani-soul-ios.

The gate runs on the **exact artifact** the archive stages, never on the developer's pair. It fails a public build that bundles English without the attestation, a database whose `scripture_sha256` differs from the certified corpus value, an artifact that does not hash to its manifest, a missing golden suite, or — for an App Store channel — Nitnem's non-SGGS text without the scholar's `REVIEWED: true`.

The archive, gate by gate

`ios/tools/testflight_archive.sh` stops at the first failure:

1. **Refuse a dirty tree**, require Xcode at or above Apple's SDK floor, wait for green CI on the commit, check the version invariants and the ledger (`--strict` when uploading).
2. **Stage** the database for the chosen profile into `ios/App/build/stage-<version>-<build>/`.
3. **Licence gate** on that artifact.
4. **XcodeGen** a temporary project from a sibling spec that differs from `project.yml` in exactly three lines (the name and the two database resource paths → staging); the developer's own project is never repointed and neither project is committed.
5. **Archive** the Release configuration with the Team ID and build number from the command line, automatic signing; **export** for App Store Connect (with `UPLOAD=1` the export *is* the upload).
6. **Prove** what was built: the Info.plist version and build, and the manifest inside the `.app`, equal what was staged; then **record** the build in the ledger.

gurbani-soul-ios/ios/tools/testflight_archive.sh · lines 1-48

```
#!/usr/bin/env bash
# testflight_archive.sh – build the TestFlight / App Store candidate the *only* sanctioned
# way.
#
# What it does, in order (every step is a gate; the script stops on the first failure):
# 1. Derives the iOS DB for the chosen PROFILE from db/siggs.sqlite (scripture checksum
#    proven
#    by pipeline/build_ios_db.py) into a STAGING directory under ios/App/build/. The
#    developer's
#    ios/Resources/siggs-ios.sqlite + manifest pair is never read, written or restored.
# 2. Runs pipeline/check_release_license.sh on THAT artifact – a public build must be
#    Gurmukhi-only, or the English translation must be attested LICENSED: true.
# 3. Generates SIGGS-TestFlight.xcodeproj with XcodeGen from a temporary sibling spec that
#    is
#    project.yml with exactly three lines changed (name + the two DB resource paths →
#    staging).
#    Your own SIGGS.xcodeproj is not repointed; neither project is ever committed.
# 4. Archives the Release configuration with the Team ID and build number passed on the
#    command line (project.yml is not edited), automatic signing.
# 5. Exports for App Store Connect; with SIGGS_UPLOAD=1 the export *is* the upload.
# 6. Proves what was built: the Info.plist version/build and the manifest INSIDE the .app.
#
# Usage (macOS with Xcode ≥ 16, XcodeGen, python3; `make dataset` first; run from anywhere):
# SIGGS_TEAM_ID=ABCDE12345 SIGGS_BUILD_NUMBER=4 ios/tools/testflight_archive.sh
# SIGGS_TEAM_ID=... SIGGS_BUILD_NUMBER=5 SIGGS_UPLOAD=1 ios/tools/testflight_archive.sh
#
# Environment:
# SIGGS_TEAM_ID      (required) Apple Developer Team ID.
# SIGGS_BUILD_NUMBER (required) CFBundleVersion for this upload. Must be > every build
# already
#                    uploaded for this MARKETING_VERSION (App Store Connect rejects
# reuse).
# SIGGS_DB_PROFILE   public (default) | personal. `personal` bundles the English layer
# and is
#                    accepted only with LICENSED: true in ios/Resources/TRANSLATION-
# LICENSE.md.
# SIGGS_RELEASE_CHANNEL testflight (default) | appstore. A TestFlight binary and an App
# Store binary are
#                    the same upload, so the channel is declared up front: `appstore`
# makes
#                    pipeline/check_release_license.sh REQUIRE the scholar review of the
# Nitnem
#                    non-SIGGS text (ios/Resources/NITNEM-REVIEW.md → REVIEWED: true) and
# requires
#                    the in-app "under review" label to be off. Only a build recorded
# with
#                    channel=appstore may be submitted (make appstore-preflight checks
# the ledger).
# SIGGS_UPLOAD        1 → export destination "upload" (straight to App Store Connect).
#                    0 (default) → write an .ipa under ios/App/build/ for Transporter.
# SIGGS_ASC_KEY_PATH / SIGGS_ASC_KEY_ID / SIGGS_ASC_ISSUER_ID
#                    App Store Connect API key (all three, or none). Needed for CI / an
#                    unattended upload; a logged-in Xcode account suffices locally.
# SIGGS_SKIP_CI_CHECK 1 → with SIGGS_UPLOAD=1, skip waiting for green CI on the commit
# (offline /
#                    pipeline-outage use only; the candidate record notes it).
# SIGGS_ARCHIVE_DRY_RUN 1 → stop after step 3 (stage + gate + generate); nothing is signed,
#                    archived or uploaded. Used by ios/tests/test_ios_archive.py.
```

```
#  
# Tracked files this script may touch: only – and only with SGGs_UPLOAD=1 – the append-only  
# ios/testflight-builds.json ledger, which you then commit with the release. It runs no git  
# command that changes the working tree. Everything else it writes is git-ignored and lives  
# under  
# ios/App/build/ (staging, archive, export, lock) or is the temporary ios/App/SGGS-  
# TestFlight.yml  
# + SGGs-TestFlight.xcodeproj. NEVER commit ios/App/build/ or ios/Resources/*.sqlite.
```

Read from the source at build time — [view these lines on GitHub at a386422](#) .

Source: [ios/tools/testflight_archive.sh](#)⁴ in gurbani-soul-ios.

On the device: the launch-integrity ladder

LaunchIntegrity is the iOS analogue of /api/health: a fail-closed self-test before any scripture is shown.

gurbani-soul-ios/ios/App/Sources/Data/LaunchIntegrity.swift · run · lines 91–152

```

static func run(corpus: CorpusActor, cacheSuiteName: String? = nil) async ->
IntegrityReport {
    #if DEBUG
    LaunchClock.mark(.integrityStart)
    #endif
    var checks: [IntegrityReport.Check] = []
    let defaults = cacheSuiteName.flatMap { UserDefaults(suiteName: $0) } ?? .standard
    let path = corpus.dbPath
    let expected = expectedSha()
    let attrs = fileFingerprint(path: path)
    let bundleVersion = bundleVersionString()

    let stored = defaults.data(forKey: cacheKey).flatMap { try?
JSONDecoder().decode(DBFingerprint.self, from: $0) }
    let cached = cacheHit(stored: stored, expectedSha: expected,
        fileSize: attrs?.size, fileMtimeNs: attrs?.mtimeNs,
bundleVersion: bundleVersion)

    let sha: String
    if cached {
        sha = expected ?? "" // full verify already proved this exact
file
    } else {
        #if DEBUG
        // Fault injection for the device checklist (D11): hold a cold verify open so a
force
        // quit can land during it. Debug only; check_no_test_hooks.sh keeps it out of
releases.
        if let hold =
ProcessInfo.processInfo.environment["SGGS_UITEST_HOLD_VERIFY"].flatMap(Double.init) {
            try? await Task.sleep(for: .seconds(min(hold, 15)))
        }
        #endif
        // Hash the ~100 MB DB OFF the main thread (this is awaited from a @MainActor
caller).
        sha = await Task.detached(priority: .userInitiated) { sha256(ofFileAt: path) ??
"" }.value
    }
    #if DEBUG
    LaunchClock.mark(.integrityHashed)
    #endif
    let hashOK = expected != nil && sha == expected
    checks.append(.init(name: "db_sha256 matches manifest", passed: hashOK))
    if hashOK && !cached, let expected, let attrs {
        let fp = DBFingerprint(dbSha256: sha, fileSize: attrs.size, fileMtimeNs:
attrs.mtimeNs,
            bundleVersion: bundleVersion, manifestSha: expected)
        if let data = try? JSONEncoder().encode(fp) { defaults.set(data, forKey:
cacheKey) }
    }
    if !hashOK { defaults.removeObject(forKey: cacheKey) } // never cache a failure

    // Structural checks run on EVERY launch (cached or not) – cheap defence in depth.
    // Mool Mantar prefix on Ang 1
    let mool = (try? await corpus.ang(1).lines.first?.gurmukhi)?
        .hasPrefix("ੴ ਸਤਿ ਨਾਮੁ ਕਰਤਾ ਪੁਰਖੁ ਨਿਰਭਉ ਨਿਰਵੈਰੁ") ?? false
    checks.append(.init(name: "Mool Mantar present", passed: mool))

```

```

// FTS works
let fts = ((try? await corpus.search("ਨਾਮ", mode: "gurmukhi", limit: 1, offset:
0))?.results.isEmpty == false)
checks.append(.init(name: "FTS search works", passed: fts))

// verify engine self-test → VERIFIED_EXACT (matched line id 5)
let verdict = (try? await corpus.verify("ਸੋਚੈ ਸੋਚਿ ਨ ਹੋਵਈ ਜੇ ਸੋਚੀ ਲਖ ਵਾਰ", ang:
1))?.verdict ?? ""
checks.append(.init(name: "Verify engine sane", passed:
verdict.hasPrefix("VERIFIED_EXACT")))

#if DEBUG
LaunchClock.mark(.integrityDone)
#endif
return IntegrityReport(ok: checks.allSatisfy { $0.passed }, dbSha256: sha,
expectedSha256: expected, sqliteVersion:
corpus.sqliteVersion,
checks: checks, usedCachedHash: cached)
}

```

Read from the source at build time — [view these lines on GitHub at a386422](#) .

Source: [ios/App/Sources/Data/LaunchIntegrity.swift](#)⁶ in gurbani-soul-ios.

- The SHA-256 of the bundled database is compared with the manifest's `db_sha256`; if it holds, the database is bit-identical to the proven corpus and every invariant holds by construction.
- After one **full** streaming verify the app stores a fingerprint (the sha, the file size and modification time, the bundle version, the manifest sha). At the next launch, if every field still matches, the ~100 MB re-hash is skipped and only the cheap structural checks run; **any** mismatch — a fresh install, an update, a repaired bundle, a manifest change — forces a full re-verify, and a failure is never cached.
- If anything fails the app refuses to present scripture (“Scripture integrity check failed”) and shows the report under More → About → Integrity.
- The threat model is **accidents** — corruption, a truncated copy, a bad build — not an attacker: anyone who can rewrite a file inside the signed bundle can rewrite the cache too; anti-tamper is the code-signing layer’s job.

Bookmarks: a separate store with its own ladder

Saved verses live in a **separate, writable SwiftData store**, never in the read-only corpus database, and store the verbatim Gurmukhi (no saroop markup). The schema is versioned from the first TestFlight build so a later change ships as a real migration stage:

gurbani-soul-ios/ios/App/Sources/Persistence/SavedLine.swift · SavedLineSchemaV1 · lines 29–32

```

enum SavedLineSchemaV1: VersionedSchema {
    static let versionIdentifier = Schema.Version(1, 0, 0)
    static var models: [any PersistentModel.Type] { [SavedLine.self] }
}

```

Read from the source at build time — [view these lines on GitHub at a386422](#) .

Source: [ios/App/Sources/Persistence/SavedLine.swift](#)⁸ in gurbani-soul-ios.

Opening the store climbs its own ladder: the versioned schema with its migration plan first; if that fails, a retry; the store is **destroyed and recreated only on a second consecutive failure**, so a transient error never costs a reader their bookmarks. Nitnem progress is a separate JSON file in the App Group with a frozen v1 schema ([Nitnem for engineers](#)).

Links

1. https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/a386422e90c2c681f9ecc4f7167b4deb6a35fb92/pipeline/check_release_license.sh#L1-L25
2. https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/main/pipeline/check_release_license.sh
3. https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/a386422e90c2c681f9ecc4f7167b4deb6a35fb92/ios/tools/testflight_archive.sh#L1-L48
4. https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/main/ios/tools/testflight_archive.sh
5. <https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/a386422e90c2c681f9ecc4f7167b4deb6a35fb92/ios/App/Sources/Data/LaunchIntegrity.swift#L91-L152>
6. <https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/main/ios/App/Sources/Data/LaunchIntegrity.swift>
7. <https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/a386422e90c2c681f9ecc4f7167b4deb6a35fb92/ios/App/Sources/Persistence/SavedLine.swift#L29-L32>
8. <https://github.com/Algorythmos-AI/gurbani-soul-ios/blob/main/ios/App/Sources/Persistence/SavedLine.swift>

Sri Guru Granth Sahib Ji — Knowledge Base wiki. Generated from commit c3d9360. Updated 25 September 2026. Verified against 417c92f on 2026-09-25. This copy may be out of date; the page is at <https://docs.gurbanisoul.com/ios/db-pair-and-launch-integrity/>