

Tech Stack Recommendations

Last Updated: 5 November 2026, 10:17

Prepared for Tailwind

Project Tailwind App

© We Are Affective Ltd 2026. All rights reserved.
Private and Confidential.

Contents

Introduction 3
Summary 3
Tech Stack Recommendations, Summary Decision Table 3
Mobile 7
iOS 7
Android 8
Why Native, Not Cross-Platform 9
Backend 11
Runtime 11
Framework 12
Language and Type Safety 14
Database 15
Primary Database 15
Caching Layer 18
Infrastructure 20
Cloud Provider 20
Containerisation 21
Infrastructure as Code 23
CI/CD 24
Third-Party Services 26
Observability 26
Analytics 28
Push Notifications 29
Content Management 30
Feature Flags 31
What Is Not Recommended 33

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Introduction

Every technology decision made for Tailwind carries the same underlying question: does this serve the product, or does it serve a preference? The recommendations in this document are not a wish list. They are a set of considered positions, arrived at by holding each option against three specific tests. First, whether it genuinely fits what this product needs to do, including delivering a single living itinerary, surfacing real-time status with honest confidence levels, and keeping a named travel partner visibly present throughout. Second, whether it matches the realistic capabilities of the team building and maintaining it. A technically superior choice that nobody on the team can operate with confidence is not superior at all. Third, whether it can be maintained well over time, without accumulating the kind of quiet technical debt that eventually forces a rebuild nobody planned for.

The product Tailwind is building is not especially complex in its surface behaviour. A client opens the app, sees a calm line about where things stand, and puts it down. That simplicity is the point, and it is harder to build than it looks. Underneath it sits real-time data from airlines and ground handlers, partner-authored updates that must never be confused with automated ones, strict privacy constraints on every notification and lock screen surface, and a household data model that keeps several people's information appropriately separate. Each of these has meaningful implications for stack choices, and they are treated here as constraints rather than preferences.

Where a significant trade-off exists, it is named. Some recommendations involve accepting a real cost in exchange for a greater benefit, and this document does not paper over those moments. A team that understands what it is giving up when it makes a choice is far better positioned to manage that choice in practice than one that was handed a clean recommendation with the difficulty left out. The goal is not a perfect stack. It is a stack that is honest about what it is, maintainable by the people responsible for it, and genuinely built around what this product needs to do.

Summary

Tech Stack Recommendations, Summary Decision Table

The table below is the authoritative reference for all technology choices across the Tailwind stack. Every later section elaborates on these decisions but does not depart from them. Where a status of "Decision pending" appears, the condition that needs to be resolved before confirming is noted.

Layer	Recommended	Status
iOS client	Swift + SwiftUI	Confirmed
Android client	Kotlin + Jetpack Compose	Confirmed

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Layer	Recommended	Status
Shared business logic	None (native-first, no cross-platform layer)	Confirmed
Backend language and runtime	Node.js (TypeScript)	Confirmed
Backend framework	NestJS	Confirmed
Real-time transport	WebSockets via Socket.IO, with server-sent events as fallback	Confirmed
REST API layer	NestJS REST controllers, OpenAPI documented	Confirmed
Primary database	PostgreSQL	Confirmed
In-memory cache and pub/sub	Redis	Confirmed
Search and preference indexing	PostgreSQL full-text search (no dedicated search service at this scale)	Confirmed
File and media storage	AWS S3	Confirmed
Push notifications	Apple Push Notification Service (APNs) and Firebase Cloud Messaging (FCM), routed via a single internal notification service	Confirmed
Authentication	Auth0, with biometric device binding handled natively on each platform	Confirmed
Secrets management	AWS Secrets Manager	Confirmed
Container orchestration	AWS ECS with Fargate	Confirmed
Infrastructure provisioning	Terraform	Confirmed
CI/CD	GitHub Actions	Confirmed
Observability, logging	Datadog	Confirmed

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Layer	Recommended	Status
Observability, error tracking	Sentry	Confirmed
Observability, APM	Datadog APM	Confirmed
Flight and schedule data	Cirium (primary), with a secondary feed from FlightAware for real-time position	Decision pending, commercial terms to be confirmed before build begins
Ground handler and supplier messaging	Internal partner-authored update model; no third-party supplier messaging platform at v1	Confirmed
Mapping	Mapbox (used only for the one-step-behind location view, never as a first-screen hero)	Confirmed
Communication, voice and SMS fallback	Twilio	Confirmed
Communication, in-app messaging	Custom-built on the existing WebSocket layer; no third-party chat SDK	Confirmed
Analytics	Amplitude, scoped strictly to product behaviour signals; no personally identifiable data in event payloads	Confirmed
Feature flags	LaunchDarkly	Confirmed
Cross-platform framework (React Native, Flutter)	Not recommended	Not recommended
GraphQL API layer	Not recommended at v1	Not recommended
MongoDB or document-store primary database	Not recommended	Not recommended
Serverless functions as primary backend compute	Not recommended	Not recommended
Firebase Realtime Database or Firestore	Not recommended	Not recommended
Third-party in-app chat SDK (e.g. SendBird, Stream)	Not recommended	Not recommended

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Why native iOS and Android, not a cross-platform framework

The decision to build natively on both platforms is the one most likely to draw scrutiny, because it doubles the surface area of client-side code and requires two teams working in parallel rather than one. The trade-off is real and it is accepted deliberately.

Tailwind's iOS client will make heavy use of biometric authentication, background delivery of push notifications with precise lock screen control, and deep integration with system-level privacy surfaces. SwiftUI gives direct, idiomatic access to all of these. The same applies to Jetpack Compose on Android. React Native and Flutter have both improved significantly in recent years, but neither offers the same level of control over notification payload rendering, lock screen content suppression, and per-app biometric binding that this product requires. For a product where a single notification preview showing the wrong content is a serious trust failure, "good enough" access to native APIs is not good enough.

The maintainability argument also runs in favour of native. The team building Tailwind is likely to be small. A codebase that each mobile engineer can reason about completely, without a cross-platform abstraction layer between them and the platform SDK, is one they can change quickly and debug without guesswork.

Why NestJS on Node.js, not an alternative backend

NestJS provides a structured, opinionated framework on top of Node.js that supports the kind of team discipline a product like this requires. The module system, dependency injection, and decorator-based routing are familiar to any TypeScript engineer and they make onboarding predictable. Node.js's event loop handles WebSocket connections and concurrent I/O efficiently, which matters for the real-time update model at the core of the product.

The alternative most seriously considered was a Go service for the real-time layer with a separate Node.js layer for partner-facing APIs. That separation adds operational complexity that is not yet justified by the scale. A single NestJS service, with the real-time and REST concerns separated at the module level rather than the service level, keeps the team's operational surface manageable without sacrificing the structure needed to maintain the codebase well.

Why PostgreSQL as the primary database, not a document store

The household data model, where several people share a record but each must have their individual information kept appropriately separate, is a relational problem. Trips have multiple legs, each leg has multiple participants, each participant has preferences, and preferences have a history. The relationships between these entities matter, and they need to be enforced at the database level, not managed in application code. PostgreSQL handles all of this well, and it handles the audit trail requirements, where every change to a status line needs to be stamped with who made it and when, without a separate eventing system at v1.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

MongoDB and other document stores were considered and rejected. The flexibility of a document model is not an asset here; it is a liability. A schema that can hold anything tends to drift towards holding everything, and the discipline of a defined relational schema is what protects the household privacy model from quiet accidental violations as the product evolves.

The flight data decision

Cirium is the recommended primary source for schedule and status data, with FlightAware providing the real-time positional layer. This combination gives the honest confidence levels the product requires. A flight status from Cirium, timestamped and attributed, supports the "confirmed at 5.52" model. FlightAware adds the positional accuracy needed for the arrival estimate that tells a client when Michael should be at the kerb.

This decision is marked pending because commercial terms with both vendors need to be confirmed before the build begins. If Cirium terms prove prohibitive, OAG is the closest alternative for schedule data. The architecture does not change; only the data provider behind the abstraction layer does.

Mobile

iOS

Language and UI framework: Swift 5.9 or later, with SwiftUI as the primary UI layer. UIKit is available for the handful of cases where SwiftUI's abstractions are insufficient, specifically custom notification content extensions and certain biometric flow integrations, but the default for all new screens is SwiftUI. Minimum deployment target is iOS 16, which gives the team access to the full SwiftUI navigation stack, improved sheet presentation APIs, and the notification permission controls introduced in that release.

Key architectural pattern: MVVM, with view models written as `ObservableObject` conforming classes and state driven entirely from the server. The iOS client does not construct its own view of the journey from assembled signals. It receives a finished, attributed status object from the API and renders it. This is not just a preference; it is a constraint that follows directly from the product's core promise. A status line that reads "Confirmed by Michael at 5.40" must have been confirmed by Michael at 5.40, and that fact must originate in the application service layer, not in a client-side inference from a push event and a cached response.

Local persistence: Minimal and scoped. The Keychain stores authentication tokens and biometric binding data. A small on-device cache, implemented with SwiftData, holds the current journey state for offline rendering during brief connectivity gaps, a lounge, an aircraft, a basement car park. This cache is bounded to the active journey window, cleared on session expiry, and never used as the source of truth for a status line. It is a rendering convenience, not a data store.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Push notification handling: APNs with notification service extensions. The extension intercepts incoming payloads before display and applies the lock screen suppression logic at the device level. A payload arrives with the full structured data. The extension renders only the permitted surface string, "Your car is ready", before it reaches the lock screen. Nothing identifying, no name, no destination, no companion, ever appears in a notification preview. This is enforced in the extension, not relied upon in the payload configuration alone. The distinction matters because payload-level suppression can be bypassed by certain MDM configurations on corporate devices.

Biometric authentication: Local Authentication framework, with Face ID and Touch ID handled natively. Auth0 manages the session token, but device binding is a native concern. The flow is: biometric success unlocks the Keychain-stored session token, which is presented to the API gateway. There is no password fallback presented to the client in normal operation. If biometric authentication is unavailable, the session ends and the client re-authenticates through Auth0's standard flow. This is a deliberate constraint, not an oversight. A product carrying this kind of data should not offer a weaker authentication path as a convenience fallback.

Real-time connection: A persistent WebSocket connection, managed by a dedicated network layer class, handles live journey status updates. The connection is established after authentication and maintained across foreground sessions. On reconnection after a gap, the client requests a state reconciliation from the server rather than trusting its local cache as current. The server response is a complete current-state object, not a diff. Diffs are faster but introduce the risk of a client that has missed an event silently displaying stale information. For Tailwind, silent staleness is worse than a slightly larger payload.

Minimum iOS version trade-off: iOS 16 excludes approximately eight to ten percent of active iOS devices at the time of writing. That figure will fall as the iOS 17 and 18 install bases grow. For Tailwind's client profile, the proportion of active users on iOS 15 or below is likely to be materially lower than the global average, as the product's user base skews towards newer hardware. The decision is accepted on those grounds. Building against iOS 15 would require significant workarounds in the navigation stack and SwiftUI sheet APIs that would complicate maintenance without serving a meaningful portion of the actual user base.

Android

Language and UI framework: Kotlin, with Jetpack Compose as the primary UI layer. The minimum SDK is API level 29 (Android 10), giving access to Biometric API v1.1, the notification channel controls needed for lock screen suppression, and the foreground service behaviour required for reliable background state management. Android 10 covers approximately ninety-three percent of active Android devices, making the exclusion negligible in practice.

Key architectural pattern: MVVM with unidirectional data flow, using Kotlin Flows for reactive state and Hilt for dependency injection. As with iOS, the ViewModel receives complete, server-authoritative

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

state objects and exposes them to the Compose UI layer. It does not assemble journey state from local signals.

Local persistence: Room for the bounded journey cache, EncryptedSharedPreferences for session tokens, and the Android Keystore system for biometric credential binding. The same principles apply as on iOS: the cache exists to render during brief offline gaps, it is bounded, and it is never treated as authoritative.

Push notification handling: Firebase Cloud Messaging with a custom `FirebaseMessagingService` implementation. Notification payloads are high-priority data messages, which wake the app's background service and allow the lock screen suppression logic to run before the notification is posted. The same principle as iOS applies: the full payload is received by the service, the display string is constructed there, and only the permitted surface content is posted to the notification tray. Tailwind's FCM implementation never uses `notification` type payloads for journey-related content, because those bypass the application layer on Android and render directly from the FCM payload, which would remove the ability to apply suppression logic reliably.

Biometric authentication: The Biometric API handles fingerprint, face, and device credential authentication in a unified flow. As on iOS, the biometric success event unlocks a Keystore-backed key, which decrypts the session token for presentation to the API gateway. No weaker fallback is offered in normal operation.

Real-time connection: Managed identically to iOS at the protocol level. The WebSocket connection is handled by OkHttp's WebSocket implementation, wrapped in a Kotlin coroutine-based layer that manages reconnection, exponential back-off, and state reconciliation on reconnect.

Why Native, Not Cross-Platform

This is the decision with the most visible cost. Two codebases, two teams, two release pipelines. The case for React Native or Flutter is not frivolous. Both have matured considerably, both have production deployments at meaningful scale, and the argument for a single shared codebase on a small team is a reasonable one to make. It was made in the evaluation for this project. It did not hold.

The rejection comes down to two categories of requirement that neither cross-platform framework handles without friction.

Notification payload processing and lock screen control. On iOS, a notification service extension is the mechanism that intercepts a push payload before display and applies suppression logic. This is a native extension point, invoked by the operating system, that runs in a separate process from the main app. React Native's push notification libraries wrap this in varying degrees, but the extension itself must be written in Swift or Objective-C. A React Native app that needs reliable, fine-grained lock screen control ends up with a native extension anyway, plus the overhead of bridging it into the JS layer. Flutter's situation is comparable. For a product where a single notification preview showing the

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

wrong content is a material trust failure, this is not a place to accept additional abstraction between the product code and the platform behaviour.

On Android, the distinction between `notification` and `data` message types in FCM, and the precise control over when and how a notification is posted, requires reliable access to `FirebaseMessagingService` and the notification posting APIs. Expo and the most common React Native push libraries handle the common cases well. The edge cases, particularly around guaranteed background wakeup behaviour across Android OEM variants, are where the abstraction layer introduces unpredictability. Tailwind's disruption model depends on a push notification arriving reliably at 2am and triggering the right behaviour. That is not a case where "usually works" is acceptable.

Biometric binding and Keychain/Keystore access. Both platforms provide system-level secure storage backed by hardware security modules where available. Auth0's native SDKs integrate with these directly. Cross-platform frameworks access these through bridge layers and plugins of varying quality and maintenance status. For a product that stores session tokens in the Keychain or Keystore and binds them to biometric credentials, the security model depends on the behaviour of that storage being exactly what the platform SDK documents say it is. A plugin maintainer's interpretation of that behaviour is a layer of risk that is unnecessary when the native SDK is available.

Consideration	Native (Swift/Kotlin)	React Native	Flutter
Lock screen notification suppression	Full control via native extension points	Requires native extension; bridging adds complexity	Same as React Native
FCM data message handling on Android	Complete, direct API access	Varies by library; background wakeup reliability inconsistent	Varies by plugin
Keychain/Keystore biometric binding	Direct SDK access, well-documented behaviour	Plugin-dependent; varies in security model fidelity	Plugin-dependent
Debugging at 2am for a live client issue	One layer: platform SDK	Two layers: JS bridge plus platform SDK	Two layers: Dart runtime plus platform SDK
Team onboarding	iOS or Android engineers working in familiar toolchains	Requires React Native familiarity across the team	Requires Dart/Flutter familiarity
Long-term maintainability	Platform updates ship with direct SDK access	Dependent on framework and library update cadence	Dependent on Flutter update cadence

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

The maintainability row deserves a specific note. Both React Native and Flutter have demonstrated that they can track platform changes with reasonable speed. But "reasonable speed" introduces a window between when Apple or Google ships a change and when the framework and its dependencies absorb it. For most products, that window is an acceptable cost of a shared codebase. For Tailwind, any window in which notification suppression behaviour, biometric API behaviour, or background delivery behaviour is uncertain is a window in which the product's core trust properties are uncertain. That is not a trade-off worth making for the convenience of a single codebase.

The cost of the native decision is real. It is managed by ensuring that the API contract between the mobile clients and the backend is defined precisely and documented in the blueprint, so that the two mobile teams are building against the same server behaviour, and by keeping the business logic where it belongs, in the application service layer rather than duplicated across platforms. What the iOS and Android clients share is not code. It is a clear, version-controlled API contract and a shared understanding of what the server will always tell them.

Backend

Runtime

Node.js is the confirmed runtime. The decision rests on three properties that matter specifically for Tailwind rather than for backend services in general.

The first is how Node.js handles concurrent I/O. The event loop model is not the right choice for every workload, but it is well suited to this one. The Tailwind backend spends most of its time waiting: for flight data feeds to respond, for partner messages to be dispatched, for WebSocket events to be fanned out to connected clients. These are I/O-bound operations, and Node.js handles them efficiently without thread-per-connection overhead. A backend managing several hundred concurrent WebSocket connections from clients in lounges and terminals around the world will not be under significant CPU pressure from Tailwind's workload at v1 scale. The event loop serves that pattern well.

The second is the relationship between the backend and the mobile clients through TypeScript. Defining shared types for the API contract, the status object shapes, confidence level enums, household permission structures, in TypeScript and using them in both the NestJS backend and, where the tooling allows, the API integration layers of the mobile clients, substantially reduces the category of error where a server sends something the client does not understand. That is not a theoretical benefit for Tailwind. The status object that carries "Confirmed by Michael at 5.40" has a specific shape with specific required fields. A type mismatch between what the server sends and what the client expects is a silent rendering failure, not a crash, and silent failures on a product built around the trustworthiness of its status lines are the worst kind.

The third is the size and familiarity of the Node.js ecosystem for the integrations this product needs. The libraries for WebSocket management, JWT validation, PostgreSQL connectivity, Redis pub/sub,

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

and AWS SDK interactions are mature, well-documented, and widely understood. The team inheriting this codebase should not need to become specialists in unusual tooling to maintain it confidently.

The trade-off with Node.js is CPU-bound performance. If a future requirement involves heavy computation, image processing, or complex cryptographic operations on the main thread, Node.js is the wrong tool for that work. The architecture handles this through the same principle applied elsewhere: the integrations boundary normalises and transforms data from external sources before it enters the application layer. Any computationally intensive transformation happens at that boundary, and if it ever needs to be extracted into a separate service, the module boundary is already there.

Framework

NestJS is the confirmed framework. It is worth being direct about what this choice buys and what it costs.

What it buys is structure. NestJS is opinionated in a way that Tailwind's team, and any team that inherits this codebase, will benefit from. The module system forces a clear separation between concerns, the dependency injection container makes services testable without significant mocking infrastructure, and the decorator-based approach to routing, middleware, and guards produces code that is readable without being obvious about how much convention it depends on. For a product where the partner-authored update pathway must be architecturally distinct from the automated status update pathway, that structural clarity is not a nicety. It is the thing that prevents those two pathways from quietly merging under deadline pressure.

What it costs is the framework's own abstraction layer. NestJS sits on top of Express by default, and that layer introduces indirection that occasionally makes debugging less direct than it would be in a bare Express or Fastify application. When something behaves unexpectedly, the question of whether the issue is in the application code, the NestJS decorator, or the underlying Express middleware chain takes a few extra steps to answer. For most engineers familiar with NestJS this is a minor friction. For an engineer encountering it for the first time at 2am with a client in disruption, it is worth knowing in advance.

The primary alternatives assessed were Fastify with a manual module structure, Express with a custom architecture, and Go with a lightweight HTTP framework.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Framework	Structural enforcement	WebSocket support	Team familiarity (TypeScript engineers)	Debugging transparency	Long-term maintainability
NestJS on Express	Strong, built-in module and DI system	Native support via `@nestjs/websockets` and Socket.IO adapter	High; NestJS is the dominant TypeScript backend framework	Moderate; abstraction adds one layer to trace	Strong; opinionated structure resists drift
Fastify with manual structure	Weak without convention; depends on team discipline	Available via plugins	High; Fastify is well known	High; thinner abstraction	Moderate; structure only as good as the team maintains it
Express with custom architecture	None; entirely team-dependent	Available via `ws` or Socket.IO directly	High; Express is ubiquitous	High; no abstraction between code and framework	Low; tends to drift without disciplined review
Go (net/http or Gin)	Moderate; Go's type system and package model enforce discipline	Available but less ecosystem support	Low for a TypeScript-first team	High	High if the team knows Go; high risk if they do not

The Go option deserves specific comment because it was seriously considered for the real-time layer. Go's goroutine model handles concurrent WebSocket connections with lower memory overhead than Node.js, and its compiled binary has predictable performance characteristics that are easier to reason about under load. The reason it was set aside is team fit. A small team maintaining two native mobile codebases alongside a backend should not also be context-switching into a third language with materially different idioms. The performance ceiling Node.js reaches before becoming a concern is well above Tailwind's v1 scale, and the complexity cost of a polyglot backend is not justified by a headroom argument that may never become relevant.

Fastify with a manual structure was set aside for a different reason. Its performance advantage over NestJS on Express is real but modest at this scale. The absence of a built-in module system and DI

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

container means the team imposes its own structure, and teams under pressure tend to let imposed structure drift. NestJS's opinions are a constraint during development and an asset during maintenance. For a product that needs to be handed to engineers who were not in the room when it was designed, that asset matters more than a few percentage points of request throughput.

Language and Type Safety

TypeScript is non-negotiable, and strict mode is non-negotiable within that. The `tsconfig.json` for the NestJS application will enable `strict: true` from the first commit. This is not a preference; it is a requirement, for the following specific reasons.

The status confidence model depends on it. The `ConfidenceLevel` type, which distinguishes "confirmed", "expected", and "being checked", must be an exhaustive union that the compiler enforces at every callsite. If a function can return a `ConfidenceLevel` and a developer adds a fourth state without updating every place that consumes it, the compiler must catch that, not a code reviewer on a pull request two weeks later.

The household permission model depends on it. The data structures that govern which household member can see which segment of which itinerary are the most sensitive data relationships in the product. TypeScript's strict null checks and discriminated unions make it possible to model those permission boundaries in a way that makes an accidental exposure of one person's data to another a compiler error rather than a runtime bug. The alternative is a test suite that has to exhaustively cover every possible combination of household configuration. That suite will never be comprehensive enough.

The partner-versus-system message distinction depends on it. Every message in the Partner Channel carries an `author` field that is either a `PartnerId` or the literal string `"tailwind"`. That distinction is the architectural expression of the product's most important promise. TypeScript's type system can make it impossible for a function to construct a message with a `PartnerId` author without that author being a real, looked-up partner record. Without strict typing, that invariant lives in documentation and developer intention. With it, it is enforced by the compiler.

Specific compiler settings that are non-negotiable alongside `strict: true`:

- `noImplicitAny: true` is covered by `strict` but worth stating explicitly. Every value has a declared type. No `any` except at the integrations boundary, where external data arrives, and even there, it must be narrowed to a typed internal representation before it enters the application layer.
- `strictNullChecks: true` is likewise covered by `strict`. A `string | null` is not a `string`. The distinction matters when a partner's name, a departure time, or a gate reference might not yet be known.
- `noUncheckedIndexedAccess: true` is not part of `strict` and must be set explicitly. Array and record access returns `T | undefined`, not `T`. A gate array that might be empty and a

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

preference map that might not contain a given key should not be treated as guaranteed to return a value.

- ESLint with `@typescript-eslint/no-explicit-any` set to `error`. The compiler setting and the linter rule together mean that `any` introduced at the integrations boundary is immediately visible and must be accompanied with a type assertion that narrows it. It cannot propagate silently.

The discipline this requires is real. Engineers who are accustomed to looser TypeScript configurations will find strict mode resistant at first, particularly around null handling and indexed access. That resistance is the point. The moments where TypeScript strict mode is most inconvenient are the moments where a runtime error would otherwise be most likely, and for a product where a status line that says the wrong thing costs a client their trust in the whole system, that inconvenience is the right trade.

Database

Primary Database

PostgreSQL is the confirmed primary database. The choice is not primarily about performance, though PostgreSQL handles Tailwind's workload comfortably. It is about what the data model actually is, and what properties the database needs to enforce.

Tailwind's data is relational at its core. A household contains members. A member has preferences, each with a history. A journey has segments. A segment has participants, and each participant's visibility to others is governed by an explicit permission record. A status line has an author, a confidence level, a timestamp, and a parent segment. These relationships are not incidental to the data; they are the data. An entity that exists in the wrong relationship, a preference attached to the wrong household member, a status update attributed to the wrong partner, is not just a bug. It is a trust failure, and in some cases a privacy failure.

PostgreSQL enforces referential integrity at the database level. A `preference` row cannot exist without a valid `account_id` foreign key. A `status_update` row cannot carry a `partner_id` that does not correspond to a real partner record. These constraints are not duplicated in application code and then hoped for in the database. They live in the schema, which means they are enforced on every write, from every path, including the ones the team has not thought of yet.

Consideration	PostgreSQL	MongoDB	DynamoDB	Firebase Firestore
Relational data model	Native; enforced at schema level	Not supported natively; managed in application code	Not supported; key-value and document only	Not supported; document model only

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Consideration	PostgreSQL	MongoDB	DynamoDB	Firebase Firestore
Household permission enforcement	Foreign keys and row-level security policies	Application-layer enforcement only	Application-layer enforcement only	Security rules, but limited join capability
Audit trail with attribution	`created_at`, `created_by` columns on every write table; straightforward	Achievable but requires discipline; no schema enforcement	Achievable but complex to query across time ranges	Achievable; Firestore has audit log but limited query flexibility
Preference history and free-text origin phrases	`preferences` table with structured and `text` columns side by side	Document model suits mixed types but drifts without enforcement	Possible but schema-less nature makes consistency fragile	Similar to DynamoDB
Complex queries across household	Joins and CTEs; PostgreSQL handles these well	Requires multiple reads and application-side assembly	Joins not supported; requires application-side assembly or denormalisation	Limited join support; designed for shallow reads
Full-text search on preferences and notes	Built-in `tsvector` and `tsquery`; adequate for Tailwind's volume	Atlas Search available but adds cost and operational surface	Not supported natively; requires external service	Not supported natively
Operational familiarity	Very high; most backend engineers have PostgreSQL experience	High	Moderate; AWS-specific	Moderate; Google Cloud-specific
Managed hosting on AWS	Amazon RDS for PostgreSQL or Aurora PostgreSQL	MongoDB Atlas (external) or DocumentDB (partial compatibility)	Native AWS service	Google Cloud only
Migration tooling	Mature; TypeORM or Prisma migrations are well understood	Schema migrations are informal by design	No formal migration tooling	No formal migration tooling

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

uncertain, or where the flexibility of a document model genuinely reflects how the data varies across records. Neither condition applies to Tailwind.

Every preference record has the same structure: a structured field, a free-text phrase, a creation date, an account association, and a confirmation status. Every status update has the same structure: an author, a confidence level, a timestamp, a segment association, and a previous value. The schema is well known. Flexibility would not help; it would introduce the risk that records silently diverge from the expected shape, and that divergence would only surface when a client sees something wrong on their screen.

The specific concern raised about MongoDB was this: a schema that can hold anything tends to drift towards holding anything. The preference data model, where inferred preferences must be kept structurally separate from confirmed ones, depends on that separation being enforced. In PostgreSQL, confirmed preferences and unconfirmed signals live in different tables with different access patterns, and the TypeORM entity definitions make the separation visible in the application code. In MongoDB, the separation is a convention. Conventions erode. The household privacy model cannot afford that erosion.

DynamoDB was not seriously considered for the primary store, though it is appropriate for certain ancillary workloads. Its key-value model with limited query capability is the wrong fit for a data model that requires multi-dimensional queries: all preferences for a given account, all status updates for a given segment within a given time range, all household members with access to a given journey. Achieving these in DynamoDB requires either significant denormalisation, which introduces consistency risks, or multiple reads assembled in application code, which moves complexity out of the database and into the service layer where it is harder to enforce.

Aurora PostgreSQL is the recommended hosting choice over standard RDS. The managed clustering gives automatic failover with sub-thirty-second recovery, read replica support for the partner-facing query workload, and storage that scales without manual intervention. The cost premium over RDS single-instance is modest relative to the operational simplicity it provides, and the recovery time objective for a product where a partner is actively working a disruption at 2am needs to be short.

Schema discipline: two rules that are not optional. Every table that records a change to any client-facing data carries `created_at` and `updated_at` timestamps, and where a human actor is responsible, a `created_by` or `updated_by` foreign key to the `partners` table. This is the database-level expression of the product's status attribution model. The second rule is that soft deletes are used for preference and instruction records, with a `removed_at` timestamp and a `removed_by` key. A preference the client has asked to be removed is not hard-deleted; the content is cleared and the row is marked removed. The audit log retains the fact that it existed. This satisfies both the client's expectation that it is gone, and the internal requirement to maintain an accurate record of what was held and when.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Caching Layer

Redis is the confirmed caching layer, and also the pub/sub mechanism for distributing WebSocket events across application service instances. These are two distinct roles running on the same Redis infrastructure, and it is worth being clear about how they are separated and why each belongs there.

What belongs in Redis, and what does not.

The distinction is simple in principle and occasionally tempting to violate in practice: Redis holds data that can be regenerated from PostgreSQL without loss, and data that needs to move between service instances in real time. It does not hold data that is the source of truth for anything a client sees.

Data type	Lives in	TTL	Rationale
Current journey status object	Redis	5 minutes	Served on Journey View cold start; regenerated from PostgreSQL on expiry or on any status update write
Partner record (name, photo URL, reply window)	Redis	10 minutes	Read on every Journey View open; infrequently updated; safe to serve from cache with short TTL
Active session tokens	Redis	Session duration	Enables fast token validation at the API gateway without a PostgreSQL read per request
WebSocket room membership	Redis	Session duration	Socket.IO adapter uses Redis pub/sub to fan out status update events to the correct client room across multiple service instances
Notification dispatch queue	Redis	Until consumed	Short-lived queue for notifications pending lock screen content review; consumed by the notification service
Itinerary segment detail	Not cached	N/A	Fetched from PostgreSQL on tap; low read frequency; caching adds staleness risk without meaningful performance benefit
Preference records	Not cached	N/A	Updated infrequently; accessed infrequently; the risk of a stale preference being applied is not worth the minor read saving
Household permission records	Not cached	N/A	Permission decisions must always reflect the current database state; a stale cache here is a potential privacy failure

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Data type	Lives in	TTL	Rationale
Audit log entries	Not cached	N/A	Write-only from the application layer; read from PostgreSQL directly for partner tooling and compliance queries

The five-minute TTL on the journey status object deserves specific comment, because it is the most sensitive caching decision in the product. The Journey View must render within 1.5 seconds from a cold start. Without a cache, that requires a PostgreSQL read, potentially involving joins across the segment, status update, and partner tables, on every open. The cache makes that target achievable. The risk is that a client opens the app and sees a status that was current five minutes ago.

The mitigation is the stale confidence indicator, which is documented in the API layer specification. When the application service detects that the cached status is approaching its TTL without a fresh write, it applies the stale state automatically before the TTL expires. The client renders "Last checked at 5.40, Charlotte is updating" rather than a confident status that happens to be ageing. The cache still serves the response quickly. The confidence level accurately represents what is known. This is the architectural expression of the product principle that honesty about what we know is more reassuring than false confidence.

The household permission records and audit log entries are explicitly excluded from caching, and that exclusion should be treated as a hard rule rather than a starting position to be relaxed under performance pressure. A cached permission record that is five minutes stale can serve a household member content they were revoked from seeing during those five minutes. The performance cost of a PostgreSQL read for permission checks is acceptable. The trust cost of a privacy failure is not.

Redis topology. A managed ElastiCache for Redis cluster with replication is the recommended hosting configuration. A single Redis instance is a single point of failure for the WebSocket fan-out mechanism, and a disruption to Redis during an active journey disruption would mean status updates cannot be delivered to connected clients. The replication overhead is modest. The recovery time from a primary failure in a replicated ElastiCache cluster is short enough not to affect a client who is watching a status line update during a rebooking.

The Socket.IO Redis adapter is the mechanism that distributes WebSocket events across NestJS service instances. When a status update is written to PostgreSQL, the application service publishes the update to a Redis channel. All service instances are subscribed to that channel. Each instance fans out the event to the WebSocket rooms it is managing. This means the product can run multiple NestJS instances behind the load balancer, as it will under normal operating conditions on ECS, without a client's WebSocket connection needing to be pinned to the instance that processed the originating write. Sticky sessions are not required, which simplifies the load balancer configuration and removes a category of failure where a service instance restart drops all WebSocket connections that were pinned to it.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Infrastructure

Cloud Provider

AWS is the confirmed cloud provider. The choice reflects three practical realities rather than a preference for any one vendor.

The first is that the confirmed stack is already AWS-native. Aurora PostgreSQL runs on RDS. Redis runs on ElastiCache. Files live in S3. Secrets live in AWS Secrets Manager. Container orchestration runs on ECS with Fargate. Choosing AWS as the cloud provider is not a separate decision layered on top of these; it is the same decision, stated once at the appropriate level. Running these services on a different provider would mean substituting each managed service individually, and the operational familiarity argument that runs in favour of the individual choices runs equally in favour of the provider.

The second is operational surface. A small team maintaining two native mobile codebases and a NestJS backend should not also be managing provider boundaries. AWS's managed services handle the operational concerns, patching, failover, certificate renewal, backup retention, that would otherwise require dedicated infrastructure engineering time. The team's time is better spent on the product.

The third is the data residency and compliance posture. Tailwind's clients include people for whom location and travel information is sensitive. AWS's EU regions provide GDPR-compliant data residency without requiring a separate compliance layer.

Primary region: eu-west-1 (Ireland). This is where all production workloads run. The rationale is straightforward: the majority of Tailwind's client base and its travel partners are European, Ireland has strong data protection infrastructure, and eu-west-1 is one of AWS's most mature regions with the full service catalogue available. Latency from the UK and continental Europe is low. Latency from the US and Asia is acceptable for a product where the primary real-time interaction is a WebSocket connection maintained over minutes rather than milliseconds.

Secondary region: eu-west-2 (London). Disaster recovery targets are defined against this region. Database snapshots are replicated cross-region. In a scenario where eu-west-1 becomes unavailable, the recovery posture is restore-from-snapshot rather than active-active failover. Active-active cross-region adds significant complexity, and the recovery time objective does not demand it at v1 scale. The decision should be revisited if the client base grows to the point where eu-west-1 downtime during a live journey wave would be materially unacceptable.

The case for GCP was briefly considered, given its strengths in data analytics and managed Kubernetes. It was set aside because the confirmed stack has no components that favour GCP specifically, and splitting providers to capture marginal advantages on individual services would fragment the operational surface without a proportionate benefit.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Containerisation

AWS ECS with Fargate is the confirmed orchestration choice. The decision table below documents what was assessed and why Fargate was selected over the alternatives.

Option	Operational overhead	Team familiarity requirement	Scaling model	Suitability for Tailwind's v1 workload	Verdict
AWS ECS with Fargate	Low; no node management, no cluster infrastructure to maintain	Moderate; ECS concepts are well-documented and straightforward	Task-level autoscaling via Application Auto Scaling	Good; workload is I/O-bound and scales horizontally without state concerns	Confirmed
AWS EKS (Kubernetes)	High; cluster management, node pools, networking configuration, certificate management	High; Kubernetes requires dedicated operational knowledge	Flexible and powerful; overkill at this scale	Overengineered for v1; the complexity is not justified by the workload	Not recommended
AWS App Runner	Very low; fully managed, no task or cluster configuration	Low	Limited; less granular control over scaling triggers and concurrency	Insufficient control over networking, VPC placement, and security group configuration for Tailwind's requirements	Not recommended
Self-managed EC2 with Docker Compose	High; instance management, OS patching, manual scaling	Low for initial setup, high for ongoing maintenance	Manual or custom autoscaling	Carries significant maintenance burden with no compensating benefit over Fargate	Not recommended

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Option	Operational overhead	Team familiarity requirement	Scaling model	Suitability for Tailwind's v1 workload	Verdict
AWS Lambda as primary compute	Low for deployment; high for architectural constraints	Moderate	Event-driven, per-invocation scaling	Fundamentally incompatible with persistent WebSocket connections; confirmed as not recommended in the summary decision table	Not recommended

Kubernetes deserves a direct response because it is often assumed to be the correct choice for any production backend. The assumption is wrong for this product at this stage. Kubernetes is the right tool for organisations that need fine-grained control over scheduling, multi-cluster deployments, custom resource definitions, or workloads that cannot be mapped onto the container abstractions a managed service provides. Tailwind has none of those requirements at v1. What it has is a small team that needs to move quickly and maintain confidently. ECS with Fargate gives that team a deployment target that handles node provisioning, OS patching, and placement without requiring a Kubernetes operator to manage it. The operational simplicity is not a compromise. It is appropriate for the scale.

Service topology on ECS. Three task definitions cover the full backend workload.

- The API service runs the NestJS application, handling REST requests and WebSocket connections. It runs behind an Application Load Balancer with target group health checks on the `/health` endpoint. Autoscaling is configured against CPU and memory metrics, with a minimum of two tasks running at all times to ensure no single-task failure drops the service.
- The worker service runs BullMQ consumers for background jobs: flight status polling, notification dispatch, "nothing needs you" evaluation, and post-trip note reminders. It runs independently of the API service so that a spike in background job processing does not compete with request handling for CPU time.
- The integrations service handles the connections to external data sources: Cirium, FlightAware, and Mapbox. Keeping this as a separate task definition means that a degraded external feed, or a need to update the integration logic, does not require redeploying the core API service.

All three task definitions run in private subnets within the Tailwind VPC. The Application Load Balancer sits in public subnets. No ECS task is directly addressable from the internet. Outbound internet access for the integrations service routes through a NAT gateway, which allows IP-based allowlisting with external data providers where required.

Container images. All images are built from a minimal Node.js base image (node:20-alpine), with multi-stage builds separating the TypeScript compilation step from the production runtime. The production image contains only the compiled JavaScript, the `node_modules` for production dependencies, and the necessary runtime. No development dependencies, no TypeScript compiler,

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Secrets and environment configuration. No secrets live in environment variables passed directly to task definitions. All sensitive configuration, database credentials, Auth0 client secrets, Cirium API keys, Twilio authentication tokens, is retrieved from AWS Secrets Manager at task startup via the ECS task definition's `secrets` block. The task execution role has the minimum IAM permissions required to retrieve those specific secrets and nothing else. Rotating a secret requires updating the value in Secrets Manager and triggering a task redeploy; no code changes are required.

Infrastructure as Code

Terraform is the confirmed infrastructure provisioning tool. All AWS resources are defined in Terraform, version-controlled alongside the application code, and applied through the CI/CD pipeline rather than manually.

The specific reasons for Terraform over the AWS-native alternatives are worth stating briefly. AWS CloudFormation and AWS CDK were both considered.

CloudFormation is mature and deeply integrated with AWS, but its YAML and JSON syntax is verbose, error messages when a stack fails are frequently unhelpful, and the feedback loop between a change and a deployed state is slower than Terraform's plan and apply model. For a team making infrastructure changes regularly during an active development phase, that feedback loop matters.

AWS CDK addresses the syntax problem by letting engineers define infrastructure in TypeScript, which fits naturally with the rest of the stack. The reason it was set aside is that it generates CloudFormation under the hood, which inherits CloudFormation's slower apply cycle and its drift behaviour. CDK is also a relatively thin abstraction over the AWS resource model, which means the productivity gain over well-written Terraform HCL is modest, and CDK introduces its own construct versioning and compatibility concerns.

Terraform's provider ecosystem is also broader than either AWS-native option. Auth0, Datadog, LaunchDarkly, and GitHub are all managed via Terraform providers, which means the entire infrastructure surface, not just the AWS resources, is in version control and applied consistently.

State management. Terraform state lives in an S3 bucket with DynamoDB-based state locking. The state bucket is versioned. A failed apply that leaves state in a partial condition can be recovered from a previous version without manual intervention. State is not stored locally on any engineer's machine.

Module structure. Infrastructure is organised into modules that map to logical boundaries rather than AWS service types. A `networking` module owns the VPC, subnets, NAT gateways, and security groups. A `compute` module owns the ECS cluster, task definitions, and load balancer configuration. A `data` module owns Aurora PostgreSQL, ElastiCache, and the S3 buckets. An `observability` module owns the Datadog integration, CloudWatch log groups, and Sentry configuration. This structure means a change to the database configuration does not require reviewing or applying the compute module, and a new engineer can understand the infrastructure by reading one module at a time.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Environment promotion. Three environments exist: development, staging, and production. Each is a separate Terraform workspace with its own state file. Development uses smaller instance sizes and shorter backup retention than production. Staging mirrors production configuration as closely as is economical. Infrastructure changes are applied to development first, then staging, then production, via the CI/CD pipeline. No environment is ever modified manually through the AWS console during normal operation. Console access is available for incident investigation, but any change made through the console is treated as drift and will be overwritten on the next Terraform apply.

CI/CD

GitHub Actions is the confirmed CI/CD platform. The pipeline is structured around three concerns that are distinct but sequential: validate the change, build and test it, and promote it through environments.

The overall pipeline structure is shown below, followed by the specific steps for each trigger type.

Pipeline	Trigger	Key steps	Promotion gate
Pull request checks	Open or update a pull request	Lint, type-check, unit tests, integration tests, Terraform plan	All checks must pass; no auto-merge
Staging deployment	Merge to `main`	Full test suite, Docker build, ECR push, ECS deploy to staging, smoke tests	Automated; staging deploy runs on every merge to `main`
Production deployment	Manual trigger from a passing staging build	Docker image promotion (same image, new tag), Terraform apply to production, ECS deploy, smoke tests, rollback on failure	Manual approval required; production deploy never runs automatically
Infrastructure changes	Pull request or merge to `main` affecting Terraform files	`terraform fmt`, `terraform validate`, `terraform plan` on PR; `terraform apply` on merge after manual approval	Manual approval on `terraform apply` to any environment above development
Dependency security scan	Scheduled, daily	`npm audit`, ECR image scan results, Dependabot alert review	Alerts to Slack; does not block deployment automatically, but critical vulnerabilities require resolution before the next production deployment

Pull request checks. Every pull request runs the full lint and type-check suite before any tests execute. A TypeScript error or an ESLint violation with `error` severity fails the pipeline at that step,

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

which is faster and cheaper than discovering the same issue after a Docker build. Unit tests run next, then integration tests against a PostgreSQL and Redis instance spun up as GitHub Actions service containers. The Terraform plan step runs if the pull request includes changes to any `.tf` file, producing a plan that is posted as a pull request comment so the reviewer can see the infrastructure impact alongside the code change.

Staging deployment. Merging to `main` triggers a Docker build. The image is tagged with the commit SHA and pushed to Amazon ECR. ECS is instructed to deploy the new task definition, using a rolling update strategy that keeps the previous version running until the new tasks pass their health checks. If health checks fail, ECS rolls back automatically. After a successful deploy, a suite of smoke tests runs against the staging environment: authentication, Journey View render, status line attribution, and notification dispatch. A smoke test failure triggers an alert and blocks the production promotion path for that build.

Production deployment. Production deployments are never automatic. A production deploy is initiated by a deliberate trigger, typically a button in the GitHub Actions UI, against a specific staging-deployed image tag. The same image that passed staging is promoted; nothing is rebuilt for production. Terraform applies any pending infrastructure changes first, then ECS deploys the application. Post-deploy smoke tests run against production. If they fail within a defined window, an automated rollback redeploys the previous task definition. The rollback is fast because ECS retains the previous task definition revision, and the previous image is still in ECR.

Rollback design. The rollback strategy is worth being explicit about, because the moments it matters are the moments where a client may be in disruption. ECS task definition revisions mean that rolling back the application layer is a matter of updating the service to point at the previous revision, which ECS handles in seconds. Database migrations are a separate concern. Migrations are additive at v1: columns and tables are added, never removed or renamed in a single deployment. A rollback of the application layer does not require a database migration rollback, because the previous application version is compatible with the current schema. This constraint is enforced in the migration review process: any migration that would make the previous application version incompatible with the new schema requires a two-phase deployment, where the schema change and the application change are deployed in separate steps with a compatibility window between them.

Observability integration in the pipeline. Datadog receives deployment markers from the CI/CD pipeline at the point of each production deploy. This means that a spike in error rate or latency in the Datadog APM view can be correlated against a specific deployment without manual annotation. Sentry receives the release identifier at deploy time, which associates new errors with the release that introduced them and suppresses duplicate alerting for errors that were already present before the deployment.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Third-Party Services

Observability

Datadog covers logging and APM. Sentry covers error tracking. The two serve different purposes and are not interchangeable, and running both is deliberate rather than redundant.

Datadog's strength is the operational picture: request throughput, WebSocket connection counts, background job queue depth, cache hit rates, and the correlation between a production deployment and a subsequent change in any of those metrics. The deployment markers sent from GitHub Actions at each production release mean that a latency spike appearing in the APM trace view can be pinned to a specific commit without manual investigation. For a product where the Journey View must render within 1.5 seconds and where a status update delayed by thirty seconds during a live disruption is a trust failure, that correlation matters. Datadog APM is configured with distributed tracing across the three ECS task definitions, so a slow response on the API service can be traced through to a slow query on Aurora or a degraded response from Cirium without switching tools.

Sentry catches what Datadog does not. Datadog tells you that error rates have risen. Sentry tells you which line of code produced the error, in which release, for how many users, and whether it was happening before the last deployment. The release identifier sent from the CI/CD pipeline on each production deploy means Sentry can distinguish a regression introduced in the current release from a pre-existing error that has been present for three weeks. That distinction matters when something goes wrong at 2am and the first question is whether this is new.

The two tools integrate directly. A Sentry error alert can link to the corresponding Datadog APM trace for the same request, and a Datadog alert can surface the Sentry error context for the same time window. Engineers do not need to reconstruct the picture from separate tools; the correlation is built into the alerting.

Data handling in both tools. Neither Datadog nor Sentry receives personally identifiable information from Tailwind. This is a hard constraint, not a configuration preference, and it applies at the instrumentation layer rather than relying on either platform's data scrubbing features.

Datadog receives structured log events and metric data. Log events are structured JSON, and the schema for each event type is defined in the NestJS logging layer. Fields that could carry client-identifying information, account identifiers, partner names, journey identifiers, segment details, are replaced with opaque internal identifiers before the event is emitted. A log line that says a status update was written for account `acc_7f3a` at 14:32:07 with confidence `confirmed` is useful for debugging. A log line that says Charlotte updated Anna Hartley's flight to Geneva is not appropriate for a third-party logging platform. The opaque identifier can be resolved internally when a specific incident requires it, but it travels to Datadog as an opaque string.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Sentry receives exception data: the error type, the stack trace, the release identifier, and a set of structured tags that identify the service, the environment, and the broad category of operation that failed. It does not receive request payloads, response bodies, session tokens, or any value that could identify a client or their itinerary. The NestJS exception filter that feeds Sentry is reviewed as part of the security review process, with specific attention to ensuring that no middleware or interceptor is inadvertently attaching request context to the Sentry scope before the filter runs.

Both platforms are configured to process data within the EU region. Datadog's EU instance is used. Sentry's EU-hosted offering is used. This is consistent with the overall GDPR posture and the data residency decision that runs through the infrastructure choices.

Alerting thresholds and on-call posture. Datadog manages the alerting rules. The critical path thresholds are:

Metric	Alert threshold	Severity
Journey View API p95 latency	Above 800ms for 3 consecutive minutes	High
WebSocket connection error rate	Above 2% over 5 minutes	High
Notification dispatch failure rate	Above 1% over 10 minutes	High
"Nothing needs you" evaluation job failure	Any failure	High
Background job queue depth (worker service)	Above 500 unprocessed jobs	Medium
Aurora PostgreSQL connection pool saturation	Above 80% for 5 consecutive minutes	Medium
Cirium feed latency	p95 above 3 seconds for 10 minutes	Medium
ECS task restart rate	More than 3 restarts in 10 minutes on any task definition	High

High severity alerts page the on-call engineer immediately. Medium severity alerts go to the team Slack channel and are reviewed at the start of the next working day unless they worsen. The "nothing needs you" evaluation job has its own alert rather than being grouped with general background job failures, because a silent failure in that job means the phrase could appear on screen without a valid check behind it. That is a product correctness failure, not just an operational one.

Sentry alerts on new errors in production and on error rate spikes for existing errors. New errors in production always notify the on-call engineer. Error rate spikes are grouped by error type and notified once per grouping per hour to avoid alert fatigue during a broad incident.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Analytics

Amplitude is the confirmed analytics platform. The decision reflects a specific requirement: Tailwind needs to understand how clients move through the product, which screens they open, which moments prompt a message to Charlotte, and whether the product's core promise, that clients open the app, find what they need, and put it down, is being delivered in practice. Amplitude handles behavioural analytics at the level of precision this requires without imposing unnecessary complexity on the instrumentation.

The data handling constraints for Amplitude are strict and are worth stating before anything else, because they shape the instrumentation architecture entirely.

No personally identifiable information enters Amplitude. The client's name, their partner's name, their destination, their journey identifier, their household composition, none of these values appear in any Amplitude event payload. Events are identified by an opaque session identifier that is generated fresh on each app session and rotated on session end. There is no persistent user identifier in Amplitude that could be used to build a longitudinal profile of an individual client's behaviour across sessions. This is a deliberate constraint, not a limitation of the platform. The goal is product signal, not user tracking.

The events that Amplitude receives describe product behaviour at a categorical level. An event says that the Journey View loaded, not for whom. It says that a message was sent, not what it contained or who sent it. It says that the itinerary detail was opened, not which segment or which journey. The event schema is defined in the NestJS backend for server-side events and in the iOS and Android clients for client-side events, and both schemas are reviewed against this constraint before any new event type is added. A pull request that adds an event containing a value that could identify a client or their travel arrangements is rejected at review.

Event category	Examples	What is captured	What is not captured
Screen views	Journey View opened, Itinerary tapped, Partner Channel opened	Screen name, session identifier, timestamp, app version, platform	Account identifier, partner name, journey content
Key interactions	Message sent, Voice note played, Document opened, Preference removed	Interaction type, session identifier, timestamp	Message content, document type, preference content
Status transitions	Disruption state shown, Disruption resolved state shown, Nothing-needs-you shown	State name, timestamp, session identifier	Disruption details, resolution content, confidence level

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Event category	Examples	What is captured	What is not captured
Performance signals	Journey View load time, Status line render time, WebSocket reconnection	Timing value in milliseconds, platform, app version	Any content that was rendered
Error states	API error shown, Offline state shown, Notification not delivered	Error category, platform, app version	Error content, account context

The metrics Amplitude is specifically configured to surface are the ones that test whether the product's core promise is being kept. How often does a client open the Journey View and close the app without navigating elsewhere? That is the clearest signal that the first screen answered their question. How often does a client open the app more than three times during a single journey day? That is a signal that something is not being resolved for them. How often does the disruption state appear and how long does it remain before transitioning to resolved? That measures the product's actual disruption performance rather than assuming it.

Amplitude's feature flag integration with LaunchDarkly is used for cohort analysis during staged rollouts. When a feature is enabled for a subset of clients via LaunchDarkly, Amplitude can attribute behavioural differences to the flag state without knowing which specific clients are in which cohort. The cohort is identified by the flag key, not by individual session identifiers.

Amplitude data is retained for twelve months. After twelve months, event data is deleted. Aggregate metrics derived from that data, weekly Journey View open rates, median message response times by category, disruption resolution durations, are retained indefinitely as summarised figures, not as raw event records.

Push Notifications

APNs and FCM are the confirmed delivery mechanisms, routed via a single internal notification service rather than a third-party push platform. The summary decision table lists this as confirmed, and the reasoning is straightforward: introducing a third-party push platform would create an additional surface through which notification payloads pass before reaching the device. For Tailwind, those payloads contain the full structured journey data needed to assemble the lock screen string. Routing them through a managed platform adds a dependency on that platform's data handling practices and its infrastructure availability, neither of which can be controlled as precisely as the internal service.

The internal notification service sits within the ECS worker task definition. It consumes from the BullMQ notification queue, applies the lock screen privacy validation step, and dispatches to APNs or FCM directly using the respective SDKs. Each platform receives a distinct payload format.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

client appears in a rendered notification without passing through the extension's suppression logic first. The extension is the enforcement point, not the payload configuration alone, because payload-level suppression can be bypassed on certain MDM-managed corporate devices.

For Android, the payload is a high-priority FCM data message. The same principle applies. The `notification` object is not used for journey-related content because Android renders `notification` type messages directly from the payload, bypassing the application layer. High-priority data messages wake the application's `FirebaseMessagingService` regardless of battery optimisation state on most devices, and the service applies the suppression logic before posting the notification to the system tray.

The internal service enforces one rule before any notification is dispatched: the assembled lock screen string is validated against a defined list of prohibited content patterns. The list includes client names, partner names, destination names, companion names, and any string that matches the format of a flight number or hotel name. A string that fails validation is held and a partner task is created rather than being sent with the prohibited content stripped out silently. Silent stripping produces a notification that may be unintelligible. A held notification gives the partner the opportunity to write a replacement that is both safe and useful. The held notification is flagged in the partner workflow view and in a Datadog alert.

The trade-off of the internal approach is operational ownership. When APNs or FCM introduces a breaking change, the internal service must be updated to accommodate it. Both Apple and Google publish deprecation notices well in advance of breaking changes, and both provide SDK versions that abstract the breaking change until a future deprecation cycle. The risk is manageable. A third-party push platform that handles this abstraction would need to be evaluated on its data handling practices, its availability SLA relative to the internal service, and its GDPR posture, each of which introduces a dependency that the internal service avoids.

The one scenario where a third-party push platform would warrant reconsideration is if the volume of notifications grew to a scale where the internal service's direct SDK integration became a maintenance burden. At v1 scale, with a client base of hundreds rather than hundreds of thousands, that threshold is not near.

Content Management

There is no CMS. The summary decision table does not list one, and the absence is intentional.

Tailwind has no editorial content. There are no articles, no destination guides, no promotional copy managed through a publishing workflow. The text the product delivers is operational: status lines, notification strings, itinerary change attributions, partner notes, and the system-generated phrases that appear when no partner-authored content is available. None of this is managed through a CMS, and introducing one would add operational complexity without serving the product.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

The system-generated phrases that do require structured management, status labels, notification body strings, the disruption update template format, and the reply window display text, are managed through a versioned configuration table in PostgreSQL and a purpose-built internal editing interface. This is documented in the backend services section under Content Management. The interface gives the operations team the ability to update these strings without a code deployment, with a two-person review step and a full audit log. It is not a CMS product. It is a small, deliberately scoped editing surface for a specific category of operational text.

The constraint that governs all system-generated text applies here as it does everywhere in the product. Any string that appears under a partner's name must have been written or explicitly approved by that partner. Anything the system generates speaks as Tailwind. The configuration interface enforces this by making the `authored_by` distinction visible in every string record, and by preventing any string marked `authored_by: partner` from being edited through the interface without a corresponding approval step. System strings are edited directly. Partner-attributed strings are not in the configuration table at all; they exist only as records created through the partner's own workflow.

Feature Flags

LaunchDarkly is the confirmed feature flag platform. Its role in the Tailwind stack is narrower than its full capability set. It manages the controlled rollout of new features to a subset of clients before full release, and it provides the circuit breaker mechanism for disabling specific functionality without a deployment.

The integration is straightforward. The NestJS backend evaluates LaunchDarkly flags at request time for server-side features, and the iOS and Android clients evaluate flags at session start for client-side features. Flag states are cached locally with a short TTL so that a LaunchDarkly availability issue does not affect the product's ability to serve requests. The default state for every flag, the value returned if LaunchDarkly cannot be reached, is the safe conservative option: the feature is off, not on. This means a LaunchDarkly outage does not accidentally enable a half-built feature for production clients.

Flags are defined in three categories. Release flags gate features that are in development and not yet ready for all clients. These are the most common type and are removed from the codebase once the feature has fully rolled out. Operational flags gate functionality that may need to be disabled quickly in response to a production issue, such as the flight status polling integration with a specific provider or the "nothing needs you" evaluation for a given segment type. Permission flags control access to features that are available only to a specific subset of clients during a trial period. These are the flags that integrate with Amplitude for cohort analysis.

No flag evaluation logic in the product uses client-identifying information as the targeting attribute. Flags are targeted by platform version, app build number, environment, and an opaque session cohort identifier. The cohort identifier is generated server-side and is not derivable from any client attribute.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

This means the flag system cannot be used to target a specific named client, which is consistent with the overall principle that the product does not single out individuals in its operational logic.

LaunchDarkly's data residency configuration is set to the EU region, consistent with the broader infrastructure posture. Flag evaluation data, including the cohort identifier and the flag keys evaluated during a session, does not leave the EU region. LaunchDarkly's audit log of flag changes is retained for the duration of the subscription and is accessible to the engineering team for incident investigation.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

What Is Not Recommended

Technology	Reason
React Native	Requires a native iOS extension for notification payload suppression regardless, adding a bridging layer between application code and the platform APIs that control lock screen content. Background wakeup behaviour across Android OEM variants is inconsistent through React Native's FCM libraries. Biometric credential binding to the Keychain and Keystore depends on plugin quality and maintenance, which is an unnecessary variable for a security-critical path. The abstraction layer adds a debugging step at every layer of the stack, including the moments that matter most.
Flutter	The same notification and biometric constraints apply as React Native. Flutter's Dart runtime sits between application code and the platform SDKs that govern lock screen rendering and FCM data message handling. The cross-platform convenience does not outweigh the loss of direct control over the surfaces where Tailwind's privacy model is enforced.
GraphQL API layer	Tailwind's API surface is well-defined, predictable, and not complex enough to justify the operational overhead GraphQL introduces. The queries are known, the payloads are shaped, and the mobile clients build against a fixed contract. GraphQL's flexibility becomes a cost rather than a benefit here, adding schema management, resolver complexity, and a more involved client-side query layer without a compensating gain. REST with a well-maintained OpenAPI specification serves this product better.
MongoDB	The household permission model is a relational problem. Confirmed preferences and unconfirmed signals must be structurally separate, enforced at the database level. A document store manages that separation in application code, which means it erodes under deadline pressure. The schema is well known across every entity type. Flexibility is not an asset; it introduces the risk that records silently diverge from the expected shape, and that divergence surfaces as a trust failure on a client's screen.
DynamoDB as primary database	The data model requires multi-dimensional queries across household, preference, and segment relationships that DynamoDB cannot serve without significant denormalisation or multiple application-side reads assembled into a single result. Neither approach is appropriate for a model where the relationships between entities carry the privacy constraints. DynamoDB is appropriate for certain ancillary workloads but not for the primary store.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Technology	Reason
Firebase Realtime Database	A Google Cloud product with no place in an AWS-native stack. Real-time delivery in Tailwind is handled by a WebSocket connection over Socket.IO with a Redis pub/sub adapter, which gives the team direct control over connection lifecycle, reconnection behaviour, and the state reconciliation model that prevents a client from silently displaying stale information after a connectivity gap. Firebase Realtime Database would introduce a separate vendor dependency and remove that control.
Firebase Firestore	The same Google Cloud dependency issue applies. Firestore's security rules provide limited join capability, making household-scoped permission enforcement difficult to express at the database level. Row-level security in PostgreSQL is the correct enforcement mechanism for this model.
AWS Lambda as primary backend compute	Fundamentally incompatible with persistent WebSocket connections. The Tailwind status model depends on a live server-to-client push channel that stays open for the duration of a foreground session. Lambda's invocation model closes the compute context after a response is sent. A workaround using API Gateway WebSocket integration is possible but introduces significant complexity in connection management, state reconciliation, and the fan-out model that distributes status updates to the correct client. ECS with Fargate handles this naturally.
AWS EKS (Kubernetes)	Overengineered for v1. Tailwind's workload is I/O-bound, scales horizontally without state concerns, and does not require the scheduling flexibility, custom resource definitions, or multi-cluster capability that Kubernetes exists to provide. A small team managing two native mobile codebases should not also be operating a Kubernetes cluster. ECS with Fargate provides the deployment target this product needs without the operational overhead.
AWS App Runner	Insufficient control over VPC placement, security group configuration, and networking for a product that requires private subnet placement of all compute, a NAT gateway for allowlisted outbound access to flight data providers, and precise control over how the load balancer interacts with WebSocket connections. App Runner abstracts these concerns away in a manner that does not suit Tailwind's requirements.
Self-managed EC2 with Docker Compose	Carries ongoing maintenance burden for node provisioning, OS patching, and manual scaling with no compensating benefit over Fargate. The team's time is better spent on the product.
AWS CloudFormation	Verbose syntax, unhelpful error messages when a stack fails, and a slower feedback loop between a change and a deployed state than Terraform's plan and apply model. The Terraform provider ecosystem also covers Auth0, Datadog, LaunchDarkly, and GitHub, allowing the full infrastructure surface to be managed in one tool. CloudFormation cannot do this without additional scripting.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Technology	Reason
AWS CDK	Generates CloudFormation under the hood, inheriting its slower apply cycle and drift behaviour. The productivity gain over well-written Terraform HCL is modest, and CDK introduces its own construct versioning and compatibility concerns without a proportionate benefit.
Third-party push notification platform (e.g. OneSignal, Courier)	Introduces an additional surface through which notification payloads pass before reaching the device. For Tailwind, those payloads contain the structured journey data needed to assemble the lock screen string. Routing them through a managed third-party platform adds a dependency on that platform's data handling practices and its infrastructure availability, neither of which can be controlled as precisely as an internal service. The lock screen privacy validation step must run under Tailwind's direct control.
Third-party in-app chat SDK (e.g. SendBird, Stream)	The Partner Channel is not a general messaging product. It carries a specific named relationship between one client and their travel partner, with hard requirements on message attribution, voice note handling, and the structural distinction between partner-authored and system-generated content. A third-party chat SDK would impose its own data model on that relationship, create a third-party dependency for content that is among the most sensitive in the product, and add an external surface for message metadata that Tailwind cannot control. The custom implementation on the existing WebSocket layer is the correct choice.
Dedicated search service (e.g. Elasticsearch, Algolia)	PostgreSQL's built-in full-text search using `tsvector` and `tsquery` is adequate for the preference and note search volumes Tailwind will generate at v1 scale. A dedicated search service would add operational surface, a third-party data dependency, and indexing complexity for a query volume that does not require it. The decision should be revisited if the preference corpus grows materially, but introducing Elasticsearch at v1 solves a problem that does not yet exist.
GCP as cloud provider	The confirmed stack is AWS-native across every layer. Splitting providers to capture marginal GCP advantages on analytics or managed Kubernetes would fragment the operational surface without a proportionate benefit. The data residency and GDPR posture is met by AWS EU regions without requiring a separate compliance layer.
Go as primary backend language	Go's goroutine model and compiled performance characteristics are genuine advantages at scale. The reason it was set aside is team fit. A small team maintaining two native mobile codebases and a NestJS backend should not also be context-switching into a third language with materially different idioms. The performance ceiling Node.js reaches before becoming a concern is well above Tailwind's v1 scale. The complexity cost of a polyglot backend is not justified by a headroom argument that may never become relevant.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.

Technology	Reason
Fastify with manual module structure	Its performance advantage over NestJS on Express is real but modest at this scale. The absence of a built-in module system and dependency injection container means the team must impose and maintain their own structure. Teams under pressure let imposed structure drift. NestJS's opinions are a constraint during development and an asset during maintenance, particularly for a codebase that will be handed to engineers who were not in the room when it was built.
Express with custom architecture	No structural enforcement at all. The separation between partner-authored and system-generated content pathways, the household permission enforcement logic, and the confidence level attribution model all depend on disciplined architecture. Express provides none of it. Discipline without enforcement erodes.

© We Are Affective Ltd 2026. All rights reserved. Private and Confidential.