Tech Stack Recommendations

Last Updated: 10 July 2026, 13:03

Prepared for SkincareLab

Project SkinSync App

Contents

Introduction

SkinSync is a daily-use health product. That shapes every technical decision in this document. The stack needs to support a habit-forming experience that users open twice a day, log personal health data into, and trust to reflect their skin honestly over months. It also needs to stay maintainable as the product grows from a focused MVP into something that may eventually include community features, dermatologist content, and retail integrations. Those are different problems, and the choices made now will either make that growth straightforward or expensive.

Each recommendation here has been evaluated against three things. First, what the product actually requires at launch: offline-capable routine logging, reliable local notifications, photo storage with genuine privacy consideration, and a data model that can surface patterns without overwhelming the user. Second, what a team realistically needs to move quickly and maintain confidently, because a technically elegant choice that the team cannot own is not a good choice. Third, what the architecture will cost to change later, since the decisions with the highest lock-in risk are the ones that deserve the most scrutiny now, not after they are embedded in production.

The philosophy throughout is honest about trade-offs. Every technology recommendation involves giving something up, and this document names those trade-offs directly rather than burying them. A choice that optimises for speed to MVP may create friction in Phase 2. A choice that maximises long-term flexibility may slow initial delivery. Where those tensions exist, they are surfaced so the team can make an informed call rather than discover the cost later. The goal is not a perfect stack. It is a stack the team understands, can build with confidence, and can evolve without regret.

Summary

Summary Decision Table

The table below represents the recommended stack for SkinSync across all major technology layers. Each decision reflects the product's core requirements: offline-first routine logging, reliable notifications, privacy-conscious photo handling, and a data model built to surface patterns over time without overwhelming the user. Later sections in this document elaborate on the reasoning behind each choice and name the trade-offs where they are significant.

Layer	Recommended	Status
Mobile framework	React Native (Expo managed workflow)	Confirmed
iOS native modules	Swift (where Expo limitations require it)	Decision pending
Android native modules	Kotlin (where Expo limitations require it)	Decision pending

Layer	Recommended	Status
Local data / offline storage	WatermelonDB	Confirmed
Push notifications	Expo Notifications + APNs / FCM	Confirmed
Local notifications (reminders)	Expo Notifications (scheduled, device-side)	Confirmed
Backend runtime	Node.js	Confirmed
Backend framework	NestJS	Confirmed
Primary database	PostgreSQL	Confirmed
Caching layer	Redis	Confirmed
Photo storage	AWS S3 (private buckets, per-user scoping)	Confirmed
Photo processing	Sharp (server-side resize and compression before S3 write)	Confirmed
Authentication	Supabase Auth	Confirmed
API layer	REST (GraphQL deferred to Phase 2)	Confirmed
Infrastructure / hosting	AWS (ECS Fargate for backend, RDS for PostgreSQL, ElastiCache for Redis)	Confirmed
CI/CD	GitHub Actions	Confirmed
Error monitoring	Sentry (React Native + Node.js)	Confirmed
Analytics	PostHog (self-hosted)	Confirmed
Ingredient conflict data	Open Food Facts Cosmetics / Skincarisma API	Decision pending
OTA updates	Expo EAS Update	Confirmed
App distribution (internal)	Expo EAS Build	Confirmed
GraphQL layer	Not recommended at this stage	Not recommended

Layer	Recommended	Status
Client-side photo comparison processing	Not recommended	Not recommended

A few decisions warrant brief comment before the later sections expand on them fully.

React Native via Expo is the right call for a team that needs to ship on both platforms without duplicating effort. The managed workflow covers the majority of what SkinSync needs at launch, including notifications, camera access, and file system handling. The risk is that Expo's managed layer occasionally cannot accommodate a specific native requirement cleanly. That is why Swift and Kotlin are listed as decision-pending rather than confirmed: the need for native modules should be deferred until a specific gap is identified, not pre-emptively built in.

WatermelonDB over SQLite directly is a considered choice. It is designed for React Native, handles sync gracefully, and performs well under the kind of repeated small writes that routine logging produces. The trade-off is a steeper initial learning curve than a simpler key-value store, but that cost is paid once and the performance and query flexibility it unlocks matter at scale.

NestJS over a lighter Node.js framework like Express is a deliberate bet on long-term maintainability. The structure it enforces costs a little speed at the start and pays back considerably once the codebase grows and more than one person is working in it. For a product that expects Phase 2 features including community sharing and retailer integration, that structure is worth having from day one.

Supabase Auth is confirmed rather than a custom authentication implementation. Building auth from scratch is one of the most reliably expensive ways to spend early engineering time. Supabase provides solid JWT-based auth, easy social login hooks for later, and a PostgreSQL-native integration that fits cleanly alongside the rest of the stack.

PostHog self-hosted is recommended over a fully managed third-party analytics service. SkinSync handles personal health data, and keeping behavioural analytics within infrastructure the team controls is both the right privacy posture and a sensible baseline for the kind of product this is. It is also the kind of decision that is far easier to make at the start than to reverse later.

The ingredient conflict data source remains decision-pending. Both Open Food Facts Cosmetics and Skincarisma offer usable datasets, but the quality and coverage vary enough that a structured evaluation against SkinSync's actual ingredient list requirements should happen before committing. This is one of the few areas where the right answer depends on testing rather than architecture principles alone.

Mobile

React Native via Expo (Managed Workflow)

Language: JavaScript / TypeScript (TypeScript strongly recommended from day one) **UI framework:** React Native core components, styled via StyleSheet or a lightweight utility library such as Tamagui **Minimum targets:** iOS 16+, Android 10 (API level 29+) **Distribution and build tooling:** Expo EAS Build for both platforms, Expo EAS Update for over-the-air releases

SkinSync is a twice-daily habit product. Users open it in the morning before they have properly woken up and again at night when they are winding down. The mobile layer needs to feel immediate, stay responsive under poor or absent network conditions, and handle camera access, local notifications, and file system writes without drama. React Native via Expo's managed workflow meets all of those requirements, and it does so without requiring the team to maintain two separate native codebases.

The managed workflow is the starting point, not a compromise. Expo's managed layer handles camera permissions, the file system, local and push notification scheduling, and secure storage out of the box. For SkinSync's MVP scope, routine builder, check-off tracking, streak logging, photo capture, and habit reminders, there is nothing in that list that requires dropping out of managed into bare workflow. The instinct to reach for bare workflow early, on the assumption that it offers more control, usually produces more complexity before it produces more capability. The right approach is to stay managed until a specific, well-understood gap forces a different decision.

TypeScript should be treated as non-negotiable from the first commit. The data model for SkinSync involves routines, products, ingredients, skin condition logs, and photo records with timestamps and metadata. That is a non-trivial graph of related types. Catching type errors at compile time rather than at runtime in a user's morning routine is worth the small overhead of setting TypeScript up correctly at the start.

Why Not Fully Native (Swift + Kotlin)

The case for going fully native is real, and it is worth stating clearly before dismissing it. Native builds give the team direct access to platform APIs without an abstraction layer in the way. Performance ceilings are higher. There is no Expo dependency to manage. For certain categories of app, particularly those with complex animations, heavy graphics processing, or deep platform integrations, native is the correct call.

SkinSync is not that product. Its core interactions are routine logging, photo capture, and notification response. None of those require performance beyond what React Native handles comfortably. The team would be maintaining two codebases, two build pipelines, and two sets of platform-specific

implementations for features that are functionally identical on both platforms. That cost compounds over time, and it compounds fastest during exactly the kind of rapid iteration that an MVP needs.

The table below summarises the key trade-offs across the main options considered.

Approach	Cross-platform code reuse	Time to MVP	Native performance ceiling	Maintenance overhead	Recommended
React Native (Expo managed)	High	Fast	Sufficient for SkinSync	Low to medium	Yes
React Native (bare workflow)	High	Medium	Sufficient for SkinSync	Medium	Only if a specific native gap requires it
Fully native (Swift + Kotlin)	None	Slow	Maximum	High	No
Flutter	High	Medium	Sufficient for SkinSync	Medium	No (see below)

Flutter deserves a brief note. It is a credible cross-platform option and its performance story is genuinely strong. The reason it is not recommended here is ecosystem fit. SkinSync's backend is Node.js and the team's existing experience is likely weighted toward JavaScript. Sharing types, validation logic, and API contracts between a TypeScript frontend and a TypeScript backend is a meaningful practical advantage. Introducing Dart into that context adds a language boundary where none needs to exist.

Native Modules (Swift / Kotlin)

Swift and Kotlin remain decision-pending in the summary table, and that status is intentional. Pre-emptively building native modules before a specific gap is identified adds complexity without adding value. The managed workflow should be given the opportunity to demonstrate its limits before the team invests in bridging past them.

If a native module does become necessary, the most likely candidates are:

- Advanced camera controls, such as custom focus or exposure handling for skin progress photos
- Background processing tasks that exceed Expo's background fetch capabilities
- Deep linking or widget integration that requires platform-specific entitlements not yet supported in managed workflow

In each case, the trigger for writing native code should be a clearly documented Expo limitation against a specific product requirement, not a precautionary assumption. Swift for iOS and Kotlin for Android are the correct languages if and when that point arrives. They are well-supported, have clear bridging documentation for React Native, and are the natural choices for their respective platforms.

Expo EAS and OTA Updates

Expo EAS Build handles both iOS and Android binaries. For internal distribution and TestFlight-equivalent testing, it replaces the overhead of managing certificates, provisioning profiles, and Gradle configurations manually. That is not a trivial saving for a small team.

Expo EAS Update enables over-the-air JavaScript bundle updates for changes that do not require a native binary rebuild. For SkinSync, this is particularly useful for copy changes, insight logic updates, and UI refinements that would otherwise require a full App Store submission cycle. The constraint is that OTA updates cannot modify native code, which is another reason to keep native module usage minimal: the more logic that lives in the JavaScript layer, the faster the team can ship corrections and improvements without waiting on app review.

The practical implication is that SkinSync's release cadence can stay responsive through the critical early weeks, when retention data will be surfacing the adjustments that matter most, without being bottlenecked by a five-to-seven day review cycle every time a small but meaningful change needs to go out.

Backend

Node.js Runtime

Node.js is the confirmed runtime for SkinSync's backend. The decision is not complicated, but the reasoning is worth stating clearly because it shapes everything that follows.

SkinSync's backend workload is predominantly I/O-bound. Routine logs come in, photo metadata gets written, ingredient conflict checks query a dataset, skin pattern calculations read from PostgreSQL and write results back. None of that is CPU-intensive in a way that would stress Node.js's single-threaded event loop. What it requires is efficient handling of many small concurrent requests without the overhead of spinning up threads for each one. That is precisely the workload Node.js was designed for, and it handles it well.

The second reason is less architectural and more practical. The frontend is TypeScript. The backend is TypeScript. That shared language means types, validation schemas, and API contracts can be defined once and used on both sides without translation. For a product where the data model, routines, products, skin condition logs, photo metadata, ingredient records, is reasonably complex and needs to stay consistent across client and server, that is a genuine advantage rather than a cosmetic one.

The one scenario where Node.js would be the wrong call is sustained CPU-heavy computation: image processing at scale, machine learning inference, or complex real-time data crunching running continuously. SkinSync does not have that problem at launch. Photo processing is handled by Sharp before the S3 write, which is a contained, well-bounded operation. Skin pattern analysis, at MVP scale, is a set of queries against a relational database, not a compute problem. If Phase 2 introduces ML-based skin insights or significantly more intensive processing, that work should be isolated into a separate service rather than handled in the main Node.js process. Node.js as the primary application runtime remains the right choice either way.

NestJS Framework

Language: TypeScript (strict mode, non-negotiable) **Node.js version:** 20 LTS minimum

NestJS is the confirmed framework. The honest version of why it was chosen over lighter alternatives is this: SkinSync is not a simple CRUD API, and it is not going to stay small. The MVP alone involves authentication, routine management, photo handling, push notification dispatch, ingredient conflict lookups, and skin pattern logging. Phase 2 adds community features and retailer integration. A framework that imposes no structure is a framework that leaves structure entirely to the team, and that debt accumulates quickly once more than one person is writing backend code.

NestJS enforces a module-based architecture where each domain, routines, users, photos, notifications, ingredients, lives in its own bounded context with its own controllers, services, and data access layer. That is not overhead. It is the structure that makes the codebase navigable six months from now, when the engineer who built the routine module is not the one maintaining it.

The table below compares NestJS against the main alternatives considered.

Framework	Structure enforced	TypeScript native	Scalability story	Learning curve	Phase 2 readiness	Recommended
NestJS	High (opinionated, modular)	Yes, first-class	Strong	Medium	High	Yes
Express	None	Partial (community types)	Depends entirely on team conventions	Low	Low without significant discipline	No
Fastify	Minimal	Partial	Strong	Low to medium	Medium	No
Hono	Minimal	Yes	Good for edge/serverless	Low	Low	No

Framework	Structure enforced	TypeScript native	Scalability story	Learning curve	Phase 2 readiness	Recommended
Koa	None	Partial	Depends on team conventions	Low	Low	No

Express is the most common counter-argument. It is fast to start, widely understood, and has a vast ecosystem. The problem is that it offers no opinions on how to organise code, handle dependency injection, structure modules, or separate concerns. For a small, simple API that will never grow, that is fine. SkinSync is neither small in scope nor unlikely to grow. Teams building on Express without strong internal conventions tend to produce codebases that work well for the first three months and become progressively harder to extend cleanly after that. The structure NestJS imposes costs a small amount of initial setup time and pays back considerably once the codebase is in active development by more than one person.

Fastify is worth a brief note. Its performance benchmarks are genuinely impressive, and its plugin system is solid. The reason it is not recommended is that performance is not the constraint here. SkinSync's backend is not handling millions of concurrent requests. Choosing Fastify over NestJS for raw throughput at this scale would be optimising for a problem SkinSync does not have, while trading away the structural guarantees that matter for a codebase expected to grow.

TypeScript: Strict Mode is Non-Negotiable

TypeScript strict mode must be enabled from the first commit and must not be disabled to resolve errors more quickly. This is not a style preference.

SkinSync handles personal health data. The data model has relationships that matter: a photo belongs to a user, a skin log entry references a routine, an ingredient conflict check operates on a specific product combination. Loose typing in that context means bugs that are invisible at development time and visible only when a user's data behaves unexpectedly. That is the wrong place to find them.

Strict mode enforces:

- `strictNullChecks`, so null and undefined are handled explicitly rather than silently
- `noImplicitAny`, so every value has a declared type rather than falling back to any
- `strictFunctionTypes`, so function signatures are checked correctly in both directions
- `strictPropertyInitialization`, so class properties are provably initialised before use

The NestJS ecosystem is built around TypeScript and its decorator-based patterns work correctly and safely only when types are properly declared throughout. Turning strict mode off is not a

and the backend agree on the shape of a routine object, at compile time rather than at runtime, is not.

Database

PostgreSQL as Primary Database

PostgreSQL is the confirmed primary database for SkinSync. The reasoning is straightforward, but it is worth making explicit because the data model here is not trivial and the wrong choice at this layer is expensive to undo.

SkinSync's data is relational by nature. A user has routines. Each routine contains ordered product steps. Each product has ingredients. Skin condition logs reference specific routine entries. Progress photos attach to users with timestamps and metadata. Streak records derive from check-off events. That is a graph of related entities with referential integrity requirements, not a collection of loosely structured documents. A relational database is the right tool for that shape of data, and PostgreSQL is the strongest option in that category.

Beyond the relational fit, PostgreSQL brings two capabilities that will matter as SkinSync's insight layer matures. Its JSONB column type handles semi-structured data, such as skin condition metadata or ingredient property objects, without requiring a separate document store. And its window functions and aggregation support make the kind of pattern queries SkinSync needs, correlating skin condition logs with product usage over time, expressible directly in SQL rather than requiring application-layer computation. That keeps the data layer clean and the query performance predictable.

The table below compares PostgreSQL against the main alternatives considered.

Database	Data model fit	Relational integrity	Pattern query support	JSONB / semi-structured support	Hosting on AWS	Phase 2 readiness	Recommended
PostgreSQL	Excellent	Full foreign key and constraint support	Strong (window functions, CTEs, aggregations)	Yes, native JSONB	RDS, well-supported	High	Yes
MySQL / MariaDB	Good	Good	Adequate	Limited	RDS, well-supported	Medium	No

Database	Data model fit	Relational integrity	Pattern query support	JSONB / semi-structured support	Hosting on AWS	Phase 2 readiness	Recommended
MongoDB	Poor for relational data	None by default	Limited without aggregation pipeline	Native document model	Atlas (separate vendor)	Low for relational queries	No
SQLite (server-side)	Good for small datasets	Full	Limited at scale	No	Not applicable for server use	Low	No
DynamoDB	Poor for relational data	None	Very limited without full table scans	Key-value and document	Native AWS	Low	No

MongoDB comes up frequently in early-stage product discussions because its flexibility feels appealing when the data model is still being defined. The problem is that SkinSync's data model is not genuinely uncertain. The core entities are well understood, and their relationships matter. Losing referential integrity and relational query capability in exchange for schema flexibility would be trading away exactly the things PostgreSQL does well, in return for a benefit SkinSync does not need.

DynamoDB is the other option that surfaces in AWS-native conversations. It is fast, it scales without operational overhead, and it fits naturally into AWS infrastructure. It is also the wrong choice for a product whose most valuable queries, show me what this user's skin was doing when they introduced product X, involve joining across multiple entities and aggregating over time. That kind of query is straightforward in PostgreSQL and genuinely painful in DynamoDB.

Hosting: AWS RDS for PostgreSQL is the confirmed hosting approach, consistent with the broader AWS infrastructure decision. RDS handles automated backups, point-in-time recovery, and failover without requiring the team to manage a database instance directly. For a small team shipping an MVP, that operational simplicity matters. The trade-off is cost relative to a self-managed instance, but the engineering time saved is worth more than the hosting delta at this stage.

Schema design principles worth establishing early:

- All primary keys should use UUIDs rather than sequential integers. SkinSync will eventually expose user-facing references to records (photo IDs, routine IDs), and sequential integers leak information about record volume and creation order in ways that UUIDs do not.
- Soft deletes, via a `deleted_at` timestamp column, are preferable to hard deletes for user-generated content. If a user deletes a product from their routine, the historical skin log entries

- Foreign key constraints should be declared and enforced at the database level, not just assumed at the application level. NestJS and TypeORM will not catch a missing cascade rule at compile time.

Redis as Caching Layer

Redis is the confirmed caching layer, hosted via AWS ElastiCache. The decision is not about raw performance at launch scale. It is about having the right tool in place before the cases that require it become urgent.

The clearest use cases for Redis in SkinSync are session and token storage, ingredient conflict lookup caching, and rate limiting. Each deserves a brief explanation.

Session and token storage. Supabase Auth handles JWT issuance, but short-lived session state and token revocation checks benefit from a fast in-memory lookup rather than a round-trip to PostgreSQL on every authenticated request. Redis's TTL-based key expiry maps cleanly onto session lifetimes.

Ingredient conflict lookup caching. Whether SkinSync sources ingredient conflict data from Open Food Facts Cosmetics, Skincarisma, or an internal dataset, the lookup pattern is the same: a user adds a product, the system checks its ingredients against the rest of their routine. That check involves the same ingredient pairs being queried repeatedly across users. Caching conflict results in Redis with an appropriate TTL means the ingredient dataset is queried once per unique combination rather than once per user per check. At MVP scale this is a convenience. As user numbers grow, it becomes a meaningful reduction in database load.

Rate limiting. Push notification dispatch, particularly for the reminder system, needs rate limiting at the user level to prevent edge cases where a configuration error or a retry loop sends a user multiple notifications in a short window. Redis's atomic increment operations and key TTLs make per-user rate limiting straightforward to implement correctly.

The table below sets out what belongs in Redis versus what stays in PostgreSQL.

Data type	Store in Redis	Store in PostgreSQL	Rationale
Session tokens and short-lived auth state	Yes	No	Fast TTL-based expiry, no persistence needed
Ingredient conflict lookup results	Yes (with TTL)	Source of truth only	Repeated queries across users; caching reduces DB load
User rate limit counters	Yes	No	Atomic increments, TTL-based reset, no audit trail needed

Data type	Store in Redis	Store in PostgreSQL	Rationale
Routine and product records	No	Yes	Persistent, relational, requires referential integrity
Skin condition logs	No	Yes	Historical data, queried in aggregations, must be durable
Progress photo metadata	No	Yes	User-owned records, must survive Redis eviction
Streak and consistency data	No	Yes	Long-term history, referenced in pattern queries
Computed skin insight results	Optionally (short TTL)	Yes (canonical)	Cache the result of an expensive query; PostgreSQL holds the authoritative record

The last row deserves a little more attention. As SkinSync's insight layer develops, some pattern queries will become expensive enough that recomputing them on every request is wasteful. Caching the output of a skin pattern analysis in Redis with a short TTL (one to two hours is usually sufficient) means users get a fast response without the application rerunning the full query on every open. The canonical result always lives in PostgreSQL. Redis holds a temporary computed view of it. That distinction matters: if Redis is cleared or an entry expires, the application falls back to PostgreSQL gracefully rather than losing data.

What Redis should never be used for in this product. It should not be the primary store for anything a user would expect to persist. Redis is an in-memory store, and while ElastiCache supports persistence options, treating Redis as a reliable long-term data store for user-generated content introduces risk that PostgreSQL simply does not carry. Any data that a user created, their routine, their skin logs, their photos, lives in PostgreSQL. Redis holds only what is safe to lose and recompute.

Infrastructure

AWS as Cloud Provider

The summary confirms AWS as the infrastructure platform, with ECS Fargate for the backend, RDS for PostgreSQL, and ElastiCache for Redis. Those choices are not revisited here, but the reasoning behind the broader AWS commitment is worth stating, because infrastructure decisions carry more lock-in than most, and the team should understand what they are getting and what they are giving up.

cross-provider networking and IAM complexity that would otherwise require ongoing management. Second, SkinSync handles personal health data and progress photos. AWS's compliance posture, particularly around data residency, encryption at rest, and audit logging, provides a defensible baseline for GDPR obligations without requiring the team to build those controls from scratch. Third, for a team at MVP stage, AWS's breadth means the infrastructure can scale in place rather than requiring a migration when Phase 2 features arrive.

Region: `eu-west-1` (Ireland) is the recommended primary region. SkinSync's initial user base is UK-based, and eu-west-1 is the closest AWS region with full service availability across the entire confirmed stack. Data residency within the EU also simplifies GDPR compliance documentation, which is relevant for a product handling skin condition logs and personal health data. If international expansion becomes a serious consideration in Phase 2, a secondary region can be added without restructuring the primary deployment.

The one honest trade-off with AWS is cost predictability. Fargate, RDS, and ElastiCache are not the cheapest options for running a small workload, and costs can creep upward if resource sizing is not reviewed regularly. At MVP scale, the team should set AWS Cost Explorer alerts from day one and review instance sizing at the end of the first month of live traffic. A db.t3.medium RDS instance and a cache.t3.micro ElastiCache node are appropriate starting points; neither should be over-provisioned on the assumption that more headroom is always better.

Containerisation: ECS Fargate

Confirmed: ECS Fargate for the NestJS backend

ECS Fargate runs containers without requiring the team to manage EC2 instances directly. There are no AMIs to patch, no autoscaling groups to configure, and no underlying compute to reason about. The team defines the container, sets the CPU and memory allocation, and Fargate handles the rest. For a small team whose primary concern is shipping product rather than managing infrastructure, that operational simplicity is the correct trade-off.

The table below compares the main containerisation and hosting options considered.

Option	Operational overhead	Cost at MVP scale	Scaling story	Team complexity	AWS-native fit	Recommended
ECS Fargate	Low	Medium	Strong (task-level autoscaling)	Low	Excellent	Yes
ECS on EC2	Medium	Lower at scale	Good (requires ASG management)	Medium	Excellent	No

Option	Operational overhead	Cost at MVP scale	Scaling story	Team complexity	AWS-native fit	Recommended
EKS (Kubernetes)	High	High	Excellent	High	Good	No
Elastic Beanstalk	Low	Low to medium	Limited	Low	Good	No
App Runner	Very low	Low	Good for simple workloads	Very low	Good	No
Self-managed EC2	Very high	Low	Manual	Very high	N/A	No

EKS is the option most likely to come up in conversation, particularly if anyone on the team has Kubernetes experience. It is a genuinely powerful platform and the right choice for systems that require fine-grained scheduling, custom operators, or multi-cluster orchestration. SkinSync is not that system. Kubernetes introduces a significant operational surface area, cluster upgrades, node pool management, networking complexity, that a small product team should not be absorbing at MVP stage. The cost in engineering attention is real and it is largely invisible until something goes wrong at two in the morning.

App Runner is worth a brief note. It is simpler than Fargate and cheaper at low traffic volumes. The reason it is not recommended is flexibility. App Runner makes some deployment decisions on the team's behalf that become difficult to override later, particularly around VPC configuration and private service-to-service communication. SkinSync's backend needs to communicate privately with RDS and ElastiCache inside a VPC. Fargate handles that cleanly. App Runner's VPC connector support exists but adds complexity that negates much of its simplicity advantage.

Elastic Beanstalk lands in a similar position: straightforward to start with, but its abstractions become friction rather than help as the deployment requirements grow more specific. It is not recommended for a product expecting meaningful architectural changes in Phase 2.

Container configuration principles worth establishing early:

- Each environment (development, staging, production) should use a separate ECS service and task definition. Sharing infrastructure across environments for cost savings introduces change management risk that is not worth it.
- Task CPU and memory should be set conservatively at launch and reviewed against CloudWatch metrics after the first two weeks of live traffic. Over-provisioning is wasteful; under-

Infrastructure as Code

Recommended: AWS CDK (TypeScript)

The infrastructure should be defined in code from the start, not configured through the AWS console and documented after the fact. Console-configured infrastructure is difficult to reproduce, impossible to audit properly, and tends to diverge from documentation over time in ways that only become apparent during an incident.

AWS CDK in TypeScript is the recommended choice, consistent with the rest of the stack. The team is already writing TypeScript for the backend and the mobile client. Using the same language for infrastructure means the cognitive overhead of switching contexts is lower, and the same type safety that applies to application code applies to infrastructure definitions. A misconfigured S3 bucket policy or an incorrectly scoped IAM role is easier to catch in a typed CDK construct than in a raw CloudFormation template.

The alternative worth naming is Terraform. It is the most widely used IaC tool and has a large ecosystem of community modules. The reason CDK is preferred here is not that Terraform is wrong, it is that CDK integrates more naturally with an AWS-only stack and TypeScript-native team. Terraform's HCL adds another language to the repository. CDK keeps the infrastructure code in the same language as everything else, which makes it more likely to be maintained and reviewed as a genuine part of the codebase rather than treated as a separate operational artefact.

What should be defined in CDK from day one:

- VPC configuration, including private subnets for RDS and ElastiCache, public subnets for the load balancer, and NAT gateway for outbound traffic from private subnets
- ECS cluster, task definitions, and service configuration for the NestJS backend
- RDS instance configuration, including subnet group, parameter group, and automated backup retention
- ElastiCache cluster configuration and subnet group
- S3 bucket definitions, including private ACL, per-user prefix structure, and lifecycle policies for photo storage
- IAM roles and policies scoped to least privilege for each service
- Application Load Balancer and target group configuration
- Secrets Manager entries for all credentials (the CDK construct can reference these without exposing the values)

The CDK stack should be broken into logical constructs, one for networking, one for compute, one for data stores, rather than defined in a single monolithic stack. That separation makes it possible to update the compute layer without touching the data layer, which matters when deploying changes in production.

CI/CD: GitHub Actions

Confirmed: GitHub Actions

GitHub Actions is the confirmed CI/CD platform. For a team already using GitHub for source control, it removes the overhead of maintaining a separate CI service and its integration with pull requests, branch protections, and deployment approvals is clean and well-understood.

The pipeline structure below covers both the mobile and backend workstreams. They share a repository trigger model but operate independently, since a backend change should not block a mobile release and vice versa.

Stage	Trigger	Steps	Notes
Backend: pull request checks	PR opened or updated against `main`	Lint, type check (tsc), unit tests, integration tests against a test database	Must pass before merge is permitted; no exceptions
Backend: staging deployment	Merge to `main`	Build Docker image, push to ECR, update ECS task definition, deploy to staging environment, run smoke tests	Staging should mirror production configuration as closely as possible
Backend: production deployment	Manual approval after staging passes	Re-use staging image (same digest, not a rebuild), update ECS task definition, deploy to production, run post-deploy health check	Never rebuild the image for production; deploy the same artefact that passed staging
Mobile: pull request checks	PR opened or updated against `main`	Lint, type check, unit tests, Expo export dry run	Expo export dry run catches bundler errors before they reach EAS
Mobile: internal build	Merge to `main`	EAS Build (development profile), distribute to internal testers via EAS	Used for QA and stakeholder review
Mobile: OTA update	Merge to `main` (JS-only changes)	EAS Update to staging channel	Faster path for copy changes and logic updates that do not require a binary rebuild
Mobile: production release	Manual approval	EAS Build (production profile), submit to App Store and Play Store via EAS Submit	App Store review adds latency; OTA updates handle urgent fixes between binary releases

Stage	Trigger	Steps	Notes
Infrastructure: CDK deploy	Changes to `infra/` directory merged to `main`	CDK diff (previewed on PR), CDK deploy to staging, manual approval, CDK deploy to production	Infrastructure changes should always be previewed with `cdk diff` before applying

A few pipeline decisions are worth calling out explicitly.

The production backend deployment re-uses the same Docker image that was deployed to staging, identified by its ECR digest, rather than rebuilding from source. Rebuilding introduces a small but non-zero risk that the production image differs from the staging image due to a dependency resolution difference or a timing issue with an upstream package. Deploying the same artefact eliminates that risk entirely. If staging passed, what goes to production is exactly what was tested.

Manual approval gates before production deployments are not a bureaucratic obstacle. They are the mechanism that prevents an automated pipeline from pushing a breaking change to production at an inconvenient time. For a twice-daily habit product, a broken production deployment at 7am when users are doing their morning routine is a meaningful trust event, not just a technical incident. The thirty seconds it takes to review and approve a staging-passed build is worth it.

Branch protection on `main` should require at least one passing CI run and one code review approval before a merge is permitted. This should be enforced at the repository level, not left to team convention. Conventions erode under pressure. Repository settings do not.

Environment secrets (AWS credentials, EAS tokens, Sentry DSN, Supabase keys) must be stored in GitHub Actions encrypted secrets, scoped to the appropriate environment, and never logged or echoed in pipeline output. A secret that appears in a CI log is effectively a rotated secret, and rotating secrets under time pressure is an avoidable operational problem.

Third-Party Services

Error Monitoring: Sentry

Confirmed: Sentry (React Native SDK + Node.js SDK)

Sentry is the confirmed error monitoring solution across both the mobile client and the NestJS backend. The decision is straightforward: Sentry has mature, well-maintained SDKs for both environments, its React Native integration handles source map uploads through EAS Build without significant configuration overhead, and its grouping and fingerprinting logic is good enough that the team will spend time fixing real problems rather than deduplicating noise.

The practical setup covers two distinct surfaces.

(a user's check-off not persisting), photo capture errors, and notification scheduling failures. These are the moments where a silent error translates directly into a user losing confidence in the app. Sentry should be configured with a `beforeSend` hook that scrubs any personally identifiable information from error payloads before they leave the device. Skin condition data, routine contents, and photo metadata must never appear in error reports.

On the backend side, the NestJS SDK integrates via an exception filter that catches both handled and unhandled errors at the controller layer. Every error that reaches Sentry should include the request context (route, method, status code) but must not include request bodies that contain user health data. The NestJS integration supports a `requestFilter` configuration for exactly this purpose; it should be set up before the first staging deployment, not added later as an afterthought.

Data handling constraints for Sentry:

- Sentry is a third-party service hosted outside AWS infrastructure. Personal health data must never be included in error payloads.
- PII scrubbing must be configured explicitly. Sentry's default data scrubbing catches obvious fields like email and credit card numbers, but it will not recognise "skin_condition" or "routine_steps" as sensitive without explicit configuration.
- Source maps should be uploaded to Sentry as part of the EAS Build pipeline and deleted from the Sentry project after the associated release is superseded. Retaining source maps indefinitely is unnecessary and slightly increases the risk surface of a compromised Sentry account.
- The Sentry DSN should be treated as a secret and injected via environment variable, not hardcoded in the application bundle.

The table below compares Sentry against the main alternatives considered.

Tool	React Native SDK quality	Node.js SDK quality	Self-hosting option	PII scrubbing controls	Cost at MVP scale	Recommended
Sentry	Excellent	Excellent	Yes (Sentry self-hosted)	Configurable, explicit	Low (free tier sufficient at launch)	Yes
Bugsnag	Good	Good	No	Limited	Low	No
Datadog APM	Good	Excellent	No	Limited for mobile	High	No

Tool	React Native SDK quality	Node.js SDK quality	Self-hosting option	PII scrubbing controls	Cost at MVP scale	Recommended
Firebase Crashlytics	Good (mobile only)	Not applicable	No	Limited	Free	No
Rollbar	Good	Good	No	Configurable	Low	No

Firebase Crashlytics is free and handles mobile crash reporting adequately, but it covers only the mobile layer. Running two separate error monitoring tools, one for mobile and one for the backend, doubles the operational overhead and removes the ability to correlate a mobile error with a backend failure in the same session. Sentry's unified approach across both surfaces is the more practical choice.

Sentry self-hosted is worth a brief note. It is technically available and would keep error data within AWS infrastructure, which aligns with the privacy posture elsewhere in the stack. The reason it is not recommended is operational overhead. Sentry self-hosted requires a non-trivial deployment (PostgreSQL, Redis, Kafka, and several Sentry-specific services), and maintaining it reliably is a distraction from building product. For a small team at MVP stage, Sentry's managed service with proper PII scrubbing configured is the right balance between privacy and operational simplicity.

Application Performance Monitoring

Recommended: AWS CloudWatch (infrastructure and backend) with Sentry performance tracing (application-level)

SkinSync does not require a dedicated APM platform at launch. The combination of CloudWatch for infrastructure-level metrics and Sentry's built-in performance tracing for application-level spans covers the monitoring needs of an MVP without adding another paid service to manage.

CloudWatch is already in the stack by virtue of running on AWS. ECS task metrics (CPU, memory, task health), RDS performance insights (query latency, connection counts), and ElastiCache metrics (cache hit rate, eviction counts) are all available without additional configuration. CloudWatch alarms should be set from day one on the metrics that matter most: ECS task failure rate, RDS connection exhaustion, and ElastiCache eviction rate spiking unexpectedly. These are the signals that indicate something structural is wrong, as opposed to an individual error caught by Sentry.

Sentry's performance tracing, enabled via the same SDK already in use for error monitoring, adds transaction-level visibility into the NestJS backend. Individual API routes can be traced end-to-end, showing time spent in database queries, Redis lookups, and external API calls (ingredient conflict

particularly community sharing and retailer integration, introduce significantly more complex backend interactions, a dedicated APM tool should be reconsidered at that point.

Analytics: PostHog (Self-Hosted)

Confirmed: PostHog (self-hosted on AWS infrastructure)

PostHog is the confirmed analytics platform, self-hosted rather than using PostHog Cloud. This decision deserves more explanation than most, because the self-hosting choice has real operational implications and should be made consciously rather than by default.

SkinSync logs personal skin condition data, progress photos, and routine behaviour that is directly tied to users' appearance and health. Sending behavioural analytics to a third-party managed service means that user interaction patterns, which screens they visit after a skin flag, how long they spend on the progress photo comparison, whether they return after a missed streak day, flow through infrastructure the team does not control. Even when analytics events are anonymised or pseudonymised, the behavioural patterns of a health-adjacent product carry a different sensitivity than, say, an e-commerce app. Self-hosting PostHog on AWS means that data stays within the same infrastructure boundary as the rest of SkinSync's user data, under the same access controls, and subject to the same data retention policies.

The operational cost is real. Self-hosted PostHog requires its own deployment (it runs on ClickHouse and a small set of supporting services), and it needs to be maintained, backed up, and kept up to date. For a small team, this is not a trivial overhead. The reason it is still recommended over PostHog Cloud is that the alternative, migrating analytics infrastructure after launch once the privacy implications are properly understood, is more disruptive than setting it up correctly from the start. The self-hosting decision is far easier to make at the beginning than to reverse later.

What PostHog should track for SkinSync:

The analytics setup should be designed around the design principles established for this product. The goal is understanding retention patterns and identifying where the experience breaks down, not maximising event coverage.

- Routine completion rate (AM and PM separately)
- Session frequency and return rate at days 3, 7, 14, and 21
- Drop-off point after receiving a skin insight or conflict warning
- Photo comparison engagement (opened, dismissed, time spent)
- Streak gap behaviour: do users return within 48 hours of a missed day
- Notification open rate by time of day and reminder type
- Onboarding completion rate and drop-off point

What PostHog should not track:

- The content of skin condition logs or routine entries
- Photo metadata beyond the fact that a comparison was viewed
- Any event that could be combined with other signals to reconstruct a user's skin history
- Anything that would require consent beyond what the standard privacy policy covers

Data handling constraints for PostHog:

- PostHog should be configured to use anonymised or pseudonymised user IDs, never raw user IDs that could be joined back to the PostgreSQL user table without an additional lookup.
- Event properties must be reviewed before instrumentation is added. The principle is that an analytics event should describe a behaviour, not contain a data value. "User viewed skin insight" is acceptable. "User viewed skin insight showing dryness score of 7" is not.
- PostHog's session recording feature must be disabled. Session recordings in a health-adjacent product capture far more than intended, and the consent and data handling implications are not worth the diagnostic benefit at this stage.
- Retention policies should be configured from the start. Raw event data older than twelve months should be archived or deleted. PostHog self-hosted supports this via its data management settings.

The table below compares the main analytics options considered.

Tool	Self-hosting option	Privacy posture	Feature depth	Cost at MVP scale	Phase 2 readiness	Recommended
PostHog (self-hosted)	Yes, recommended	High (data stays in AWS)	Strong (funnels, cohorts, feature flags)	Infrastructure cost only	High (feature flags useful for Phase 2 rollouts)	Yes
PostHog Cloud	No	Medium (third-party)	Same as self-hosted	Low (free tier generous)	High	No
Mixpanel	No	Low (third-party)	Strong	Medium	High	No
Amplitude	No	Low (third-party)	Strong	Medium to high	High	No
Firebase Analytics	No	Low (Google infrastructure)	Adequate	Free	Medium	No

Tool	Self-hosting option	Privacy posture	Feature depth	Cost at MVP scale	Phase 2 readiness	Recommended
Segment	No	Low (third-party, data broker model)	Strong (routing layer)	High	High	No

One PostHog capability worth flagging for Phase 2 is feature flags. PostHog's feature flag implementation is solid, and for a product that will be rolling out dermatologist-reviewed content templates, community features, and retailer integrations incrementally, having a feature flag system already in the stack means those rollouts can be controlled without requiring separate tooling. It is not a reason to choose PostHog on its own, but it is a genuine Phase 2 benefit of a decision that was made for privacy reasons first.

Push Notifications: Expo Notifications with APNs and FCM

Confirmed: Expo Notifications (local, device-side scheduling) + Expo Notifications with APNs / FCM (remote push)

The notification architecture for SkinSync splits cleanly into two distinct cases, and it is worth being precise about which is which because they have different implementation paths, different failure modes, and different implications for user trust.

Local notifications are scheduled on-device by the Expo Notifications API without any server involvement. These cover SkinSync's routine reminders: AM and PM nudges at user-configured times. Because they are scheduled locally, they fire reliably even when the device has no network connection, and they do not require a server round-trip or a push notification token to function. For a habit-forming product where the reminder is the mechanism that keeps the routine alive through the unglamorous early weeks, local notification reliability is not a nice-to-have. It is central to the product working at all.

Remote push notifications are server-initiated and routed through APNs (Apple Push Notification service) for iOS and FCM (Firebase Cloud Messaging) for Android. Expo's push notification service acts as a unified abstraction over both, accepting a push token and a payload and handling the routing to the correct platform. Remote notifications are the right mechanism for event-triggered messages: a weekly skin summary becoming available, a streak milestone being reached, or (in Phase 2) a dermatologist template being updated.

The distinction matters for a specific reason. Teams that route all notifications through the server, including routine reminders, introduce a dependency on network connectivity and backend availability for something that should work regardless. If the backend is slow or unavailable at 7am,

- Expo's push notification service processes token and payload data in transit. Notification payloads must not contain health data. A payload should contain enough information to route the user to the correct screen on open ("open routine" or "view summary"), not to reconstruct what that screen contains.
- Remote push should be rate-limited at the user level via Redis (as described in the database section) to prevent accidental notification storms during backend errors or retry loops.
- Users must be able to disable all remote push notifications independently of local reminders. These are different consent decisions and should be presented as such in settings.

The Expo push notification service introduces a dependency on Expo's infrastructure for remote push delivery. For a team using the Expo managed workflow, this is already an accepted dependency. If the team moves to the bare workflow in future, direct APNs and FCM integration via `@react-native-firebase/messaging` or the `node-apn` library is a straightforward migration path. It is not something to build pre-emptively, but it is worth knowing the exit route exists.

CMS

No CMS is recommended for the MVP. The content surface area at launch does not justify the overhead of a separate content management system, and introducing one pre-emptively adds a system to maintain without a clear return.

The MVP content that might appear to require a CMS, ingredient conflict descriptions, skin insight copy, onboarding text, and reminder messages, is better managed as structured data in PostgreSQL or as localisation strings in the application codebase at this stage. The volume is small, the update frequency is low, and the content is closely coupled to application logic (an ingredient conflict description needs to match the conflict data model precisely). Managing it outside the codebase and the database creates a synchronisation problem without solving a real content operations problem.

Phase 2 changes this calculation. Dermatologist-reviewed templates represent content that will be authored by people who are not engineers, updated on a schedule that is independent of the release cycle, and potentially localised for different markets. That is the point at which a headless CMS becomes the right tool. The options worth evaluating at that stage are Sanity and Contentful. Both have strong structured content models, good API access, and TypeScript SDK support. Sanity's real-time collaborative editing and its ability to define custom document schemas in TypeScript make it a natural fit for a team already working in that language. Contentful's content delivery network and CDN caching are more mature at scale. Neither decision needs to be made now.

The one preparation worth making at MVP stage is to ensure that any hardcoded copy in the application is stored in a way that makes extraction straightforward later. String constants in a dedicated localisation file, rather than scattered inline through component code, means the migration to a CMS-driven content model is a refactor rather than an archaeology exercise.

What Is Not Recommended

The table below consolidates every technology that was explicitly considered and rejected during this engagement. Each entry corresponds to a decision made in one of the earlier sections. Nothing here is included for completeness; each rejection has a specific reason that matters for SkinSync in particular.

Technology	Reason not recommended
Fully native Swift + Kotlin build	Requires two separate codebases, two build pipelines, and duplicated implementations of features that are functionally identical on both platforms. The performance headroom it provides is not needed for routine logging, photo capture, and notification handling. The maintenance cost compounds exactly when MVP iteration speed matters most.
Flutter	Credible cross-platform option, but introduces Dart into a codebase where every other layer is TypeScript. The shared-language advantage between mobile client and backend is a genuine practical benefit that Flutter removes without offering a compensating one specific to SkinSync's requirements.
React Native bare workflow (pre-emptive)	Not ruled out permanently, but adopting it before a specific Expo managed limitation is identified adds complexity without adding capability. The trigger for moving to bare workflow should be a documented gap, not a precautionary assumption.
Express	No structural opinions on code organisation, dependency injection, or module separation. Works well for simple APIs that will not grow. SkinSync is neither simple in scope nor unlikely to grow, and teams building on Express without strong internal conventions tend to produce codebases that become progressively harder to extend. The structure NestJS enforces is worth the initial setup cost.
Fastify	Strong performance benchmarks and a solid plugin system, but raw throughput is not the constraint for SkinSync's backend. Choosing Fastify over NestJS would optimise for a problem this product does not have while trading away the structural guarantees that matter for a growing codebase.
Hono	Well-suited to edge and serverless deployments. SkinSync's backend runs on ECS Fargate and has Phase 2 complexity that benefits from NestJS's module system. Hono's minimal structure is an asset in a different context, not this one.
Koa	No structural enforcement, similar objection to Express. Requires the team to impose its own conventions consistently across a codebase that will have multiple contributors over time.

Technology	Reason not recommended
MongoDB	Frequently proposed for early-stage products because schema flexibility feels useful when the data model is uncertain. SkinSync's data model is not genuinely uncertain. Its core entities and their relationships are well understood, and losing referential integrity and relational query capability in exchange for document flexibility trades away exactly what PostgreSQL does well for a benefit this product does not need.
MySQL / MariaDB	Adequate for relational data, but PostgreSQL's JSONB support, window functions, and aggregation depth make it the stronger choice for SkinSync's pattern query requirements. MySQL's JSONB support is more limited and its query planner is less capable for the kind of cross-entity aggregations that the insight layer will require.
SQLite (server-side)	Appropriate for embedded or edge use cases. Not suited to a multi-user backend with concurrent writes and long-term pattern queries across a growing dataset.
DynamoDB	Fast and operationally simple on AWS, but its query model is fundamentally misaligned with SkinSync's most valuable data access patterns. Correlating skin condition logs with product usage over time involves joins and aggregations across multiple entities. That is straightforward in PostgreSQL and genuinely painful in DynamoDB.
GraphQL (at this stage)	Not ruled out for Phase 2, where community features and retailer integration may introduce the kind of flexible, nested query requirements that GraphQL handles well. At MVP, the API surface is well-defined and REST covers it cleanly without the added complexity of a schema, resolver layer, and client-side query management. Introducing GraphQL pre-emptively adds overhead that does not pay back until the query flexibility is actually needed.
Firebase (full platform)	Firebase Crashlytics covers only mobile crash reporting, which leaves the backend unmonitored or requiring a separate tool. Firebase Analytics sits on Google infrastructure with a lower privacy posture than is appropriate for a health-adjacent product handling personal skin data. Firebase as a full platform would introduce significant vendor dependency across multiple layers (auth, database, analytics, notifications) in a product where data control and infrastructure ownership are deliberate design decisions.
Client-side photo comparison processing	Running image comparison or processing on-device introduces inconsistent behaviour across device generations, risks performance degradation on older hardware, and makes it difficult to apply consistent compression and privacy controls before a photo is stored. Server-side processing via Sharp before the S3 write gives the team a single, controllable point where resize, compression, and metadata stripping happen reliably.
ECS on EC2	Lower cost at scale than Fargate, but requires managing autoscaling groups, AMI patching, and instance-level configuration. For a small team at MVP stage, the operational overhead is not worth the cost saving. Fargate handles the same workload without requiring the team to reason about the underlying compute.

Technology	Reason not recommended
EKS (Kubernetes)	The right platform for systems requiring fine-grained scheduling, custom operators, or multi-cluster orchestration. SkinSync is not that system. Kubernetes introduces cluster upgrades, node pool management, and networking complexity that a small product team should not be absorbing at this stage. The operational cost is real and tends to surface at inconvenient times.
Elastic Beanstalk	Straightforward to start with, but its abstractions become friction as deployment requirements become more specific. VPC configuration, private service-to-service communication, and environment parity between staging and production are all harder to control cleanly on Beanstalk than on Fargate.
App Runner	Simpler than Fargate and cheaper at low traffic, but its VPC connector support for private database communication adds complexity that negates much of the simplicity advantage. SkinSync's backend needs clean private connectivity to RDS and ElastiCache, and Fargate handles that more predictably.
Self-managed EC2	Maximum control and lowest hosting cost, but requires the team to manage patching, instance health, and manual scaling. The engineering time cost is far higher than the hosting saving at this stage.
Terraform	A credible IaC choice, particularly for multi-cloud environments. The reason AWS CDK is preferred is that CDK keeps infrastructure code in the same TypeScript the team is already writing for the backend and mobile client. Terraform's HCL adds a language boundary that makes infrastructure changes feel like a separate operational concern rather than a first-class part of the codebase.
Bugsnag	Covers both mobile and backend, but its PII scrubbing controls are less configurable than Sentry's, and that matters for a product handling personal health data. Sentry's explicit `beforeSend` and `requestFilter` configuration gives the team precise control over what leaves the device and the server.
Datadog APM	Strong backend observability, but expensive at MVP scale and heavier than SkinSync needs at launch. The combination of CloudWatch and Sentry performance tracing covers the same ground without the cost or the operational overhead of a full APM platform. Worth reconsidering if Phase 2 significantly increases backend complexity.
Rollbar	Adequate error monitoring, but offers no meaningful advantage over Sentry for this stack and lacks Sentry's depth of React Native integration.
PostHog Cloud	The managed version of the confirmed analytics tool. Rejected on privacy grounds rather than capability. Behavioural analytics for a health-adjacent product should stay within infrastructure the team controls, not flow through a third-party managed service. The self-hosted version provides identical functionality with a higher privacy posture.

Technology	Reason not recommended
Mixpanel	Strong analytics feature set, but hosted on third-party infrastructure with limited self-hosting options. The privacy posture is not appropriate for a product handling personal skin and health data.
Amplitude	Same objection as Mixpanel. Capable tool, wrong privacy posture for this product category.
Firebase Analytics	Free, but sits on Google infrastructure and offers limited control over data residency and retention. Not appropriate for a health-adjacent product where behavioural data carries meaningful sensitivity.
Segment	A routing and data pipeline layer rather than an analytics destination. Adds a third-party data broker into the flow, which directly conflicts with the privacy posture that informed the PostHog self-hosting decision. The added cost and vendor dependency are not justified at MVP scale.
CMS (at this stage)	The MVP content surface does not warrant a separate content management system. Ingredient conflict descriptions, insight copy, and onboarding text are small in volume, low in update frequency, and tightly coupled to application logic. Managing them outside the codebase creates a synchronisation problem without solving a real content operations problem. The decision should be revisited in Phase 2 when dermatologist-reviewed templates introduce genuinely non-engineer-authored content.
Sentry (self-hosted)	Technically possible, but requires a non-trivial deployment running PostgreSQL, Redis, Kafka, and several Sentry-specific services. Maintaining that reliably is a meaningful operational distraction for a small team at MVP stage. Sentry's managed service with proper PII scrubbing configured is the right balance between privacy and operational simplicity at this point.