Tech Stack Recommendations

Last Updated: 3 July 2026, 13:41

Prepared for ThreeLochsCommunity

Project Three Lochs App

Contents

Introduction

The recommendations in this document exist to serve one outcome: a product that behaves the way Three Lochs promises to behave. Responsive, personal, proactive, and quiet enough in its operation that residents never have to think about the infrastructure underneath. Every technology decision here has been evaluated against three things. First, whether it can deliver the product requirements as specified, particularly around real-time status updates, push notifications, personalised home screens, and the maintenance request loop that sits at the emotional centre of the resident experience. Second, whether it matches the realistic capabilities of the team building and maintaining it. A technically elegant choice that requires specialist knowledge the team does not have is not a good choice. Third, whether what gets built can be maintained, extended, and handed over without accumulating the kind of technical debt that eventually surfaces as outages, slow releases, and corners being cut in exactly the places residents will notice.

Where a technology has been assessed and found wanting, this document says so plainly. The HOA software market is full of platforms that are comprehensive on paper and limiting in practice, and the worst outcomes in this category tend to happen when a team builds on a foundation that seemed adequate at the start and reveals its constraints at the worst possible moment, usually when the product has residents depending on it. The evaluation process here has been deliberate about identifying those constraints before they become expensive.

The trade-off philosophy throughout is simple. Prefer proven over novel. Prefer maintainable over clever. Prefer a smaller, well-integrated stack over a broader one with more moving parts to manage. Where a significant trade-off exists between two viable options, it is named and the reasoning behind the recommendation is given in full. The goal is not a perfect system in the abstract. It is a system that closes the loop on every maintenance request, sends a notification before a resident has to ask, and does so reliably at 847 homes without requiring heroics from the team keeping it running.

Summary

The table below is the authoritative reference for every technology decision in this document. Later sections elaborate on each choice, name the trade-offs, and explain the implementation detail. Nothing in those sections contradicts what is recorded here.

Layer	Recommended	Status
iOS	Swift + SwiftUI	Confirmed
Android	Kotlin + Jetpack Compose	Confirmed
Cross-platform framework	None. Native per platform.	Confirmed

Layer	Recommended	Status
Backend language + framework	Node.js + NestJS	Confirmed
API pattern	REST with WebSocket support for real-time status updates	Confirmed
Primary database	PostgreSQL	Confirmed
Caching layer	Redis	Confirmed
File storage	AWS S3	Confirmed
Push notifications	Firebase Cloud Messaging (FCM)	Confirmed
Authentication	Auth0	Confirmed
Payment processing	Stripe	Confirmed
Background job processing	BullMQ (backed by Redis)	Confirmed
Search	PostgreSQL full-text search	Confirmed
Hosting + compute	AWS (ECS Fargate for containers)	Confirmed
CI/CD	GitHub Actions	Confirmed
Infrastructure-as-code	Terraform	Confirmed
Monitoring + alerting	Datadog	Confirmed
Error tracking	Sentry	Confirmed
Logging	AWS CloudWatch (forwarded to Datadog)	Confirmed
Email delivery	SendGrid	Confirmed
Mapping + location	Mapbox	Decision pending
Analytics	Mixpanel	Confirmed
Resident-facing web portal	None in phase one	Confirmed
React Native / Flutter	Not recommended	Not recommended

Layer	Recommended	Status
MongoDB	Not recommended	Not recommended
Heroku	Not recommended	Not recommended
Existing HOA platforms (Buildium, AppFolio, etc.)	Not recommended as foundation	Not recommended

A few decisions warrant a brief note here before the sections that follow expand on them fully.

The choice of native mobile development over a cross-platform framework is the most consequential decision in this table. The product experience described across the workshop sessions demands native-quality push notification handling, smooth transitions, and a home screen personalisation pattern that cross-platform tools can approximate but rarely deliver without compromise. SwiftUI and Jetpack Compose are both mature enough now that the productivity gap with React Native has narrowed considerably, and the quality ceiling is meaningfully higher. For a product where the first sixty seconds determine whether a resident trusts it, that ceiling matters.

NestJS on Node.js was chosen over alternatives such as Django or Laravel because the team's existing familiarity sits in the JavaScript ecosystem, and NestJS brings the structure and conventions that a Node project of this complexity needs without requiring a language switch. The modular architecture maps cleanly onto the product's domain boundaries: payments, maintenance, bookings, notifications, and resident management are all discrete enough that NestJS modules keep them separated without ceremony.

PostgreSQL handles everything a community of 847 homes will produce. There is no data volume or query complexity here that requires a more specialised solution, and the relational model fits the domain precisely. Redis serves two roles: caching for home screen personalisation data that needs to return fast, and the queue backing for BullMQ, which handles the background jobs that power the notification logic. Those two roles share an instance in development and separate in production.

Mapbox sits at decision pending because the product requirements include lochside trail maps and estate wayfinding for maintenance reporting, but the scope and fidelity of that feature has not yet been fully defined. If it resolves to simple static maps, a lighter solution may be sufficient. If it includes interactive trail layers and amenity location pins, Mapbox is the right call. That decision should be made before the maintenance reporting module is built.

The not-recommended entries at the bottom of the table are addressed in the dedicated section later in this document. The short version: React Native and Flutter were evaluated and set aside on quality grounds, MongoDB on structural grounds, Heroku on operational grounds, and existing HOA platforms on strategic grounds. None of them can deliver the product as specified without either compromising the resident experience or accumulating constraints that become more expensive with every release.

Mobile

iOS: Swift + SwiftUI

Swift is the recommended language for iOS development. Minimum deployment target is iOS 16, which covers approximately 95% of active iOS devices and unlocks the full SwiftUI feature set without the bridging workarounds that earlier targets require. SwiftUI is the UI framework throughout. It is Apple's current direction and has reached a maturity point where the edge cases that made early adopters nervous are largely resolved. For a product with a heavily personalised home screen, smooth card transitions, and push notification handling that needs to feel immediate, SwiftUI delivers the interaction quality the product requires without custom rendering work.

The home screen personalisation pattern, resident name, neighbourhood, dues status, amenity availability, and the next upcoming event surfaced in a single view, is built most cleanly in SwiftUI using a combination of `@StateObject` for resident session data and `AsyncImage` for estate photography loaded against the loch-blue palette. Skeleton loading states during API calls should be implemented natively rather than relying on a third-party shimmer library, keeping the dependency count low and the loading experience consistent with the platform's own patterns.

Push notification handling uses `UNUserNotificationCenter` with a custom delegate to manage foreground notification presentation. The maintenance status update loop, which is the most trust-sensitive notification in the product, should deep-link directly to the relevant request screen when tapped. That behaviour requires a `UNNotificationResponse` handler wired to the app's navigation stack from day one, not retrofitted later. Retrofitting it later always produces edge cases around cold-start routing that take longer to resolve than doing it correctly at the start.

Background app refresh is enabled selectively. Dues confirmation and maintenance status updates are time-sensitive enough to warrant background delivery via FCM silent notifications rather than relying on scheduled background fetch, which iOS throttles aggressively in practice.

Key trade-off to name: SwiftUI's navigation model changed significantly in iOS 16 with `NavigationStack` replacing `NavigationView`. The team should commit to `NavigationStack` from the outset. Mixed navigation models in a single app produce state management problems that are disproportionately expensive to untangle later.

Android: Kotlin + Jetpack Compose

Kotlin is the recommended language for Android. Minimum SDK target is API level 29 (Android 10), covering approximately 93% of active Android devices. Jetpack Compose is the UI framework, maintained by Google and now the unambiguous primary direction for Android UI development. The same reasoning that applies on iOS applies here: for a product where the first sixty seconds set the resident's expectation for everything that follows, the rendering quality and gesture responsiveness

of a native Compose implementation is measurably better than what a cross-platform bridge can produce under real device conditions.

The home screen is implemented using a `LazyColumn` with a pinned header carrying the resident's personalised details and estate photography. The card components for dues status, amenity availability, and the event RSVP prompt are stateless composables fed from a single `ViewModel` that holds the home screen state as a `StateFlow`. This pattern keeps the UI layer thin and the business logic testable without spinning up the full UI in tests.

Navigation uses Jetpack Navigation Compose with a single `NavHost` at the root of the activity. Deep linking from push notifications follows the same requirement as iOS: the maintenance status notification must route directly to the correct request screen on both cold start and foreground tap. Android's intent handling for deep links needs to be wired correctly in the manifest and tested against FCM data payloads early, not as a late-stage addition.

Background notification delivery on Android requires FCM data messages rather than notification messages when the app needs to process the payload before displaying it. The maintenance loop notifications fall into this category. The `FirebaseMessagingService` implementation should handle foreground, background, and terminated-app states explicitly, with the notification display built in code rather than delegated to the FCM SDK's default handling. Default handling is convenient but produces notifications that cannot be deep-linked reliably.

Key trade-off to name: Jetpack Compose recomposition behaviour requires deliberate state management. Passing unstable lambdas or mutable state objects directly into composables causes unnecessary recomposition that manifests as subtle jank on mid-range devices. The team should run Compose compiler metrics during development, not just in final performance passes.

Why Native Over Cross-Platform

The cross-platform options evaluated were React Native and Flutter. Both were set aside. The reasoning is not primarily about raw performance, though that matters. It is about where the product's quality ceiling sits and what happens when you approach it.

Dimension	React Native	Flutter	Native (SwiftUI / Compose)
Push notification deep linking	Requires third-party libraries; cold-start routing is fragile	Better than React Native; still adds abstraction over native APIs	Full platform API access; no abstraction layer
Home screen personalisation rendering	Bridge overhead visible on animation-heavy screens	Good; Dart rendering engine is independent of platform	Highest ceiling; hardware-accelerated, no bridge

Dimension	React Native	Flutter	Native (SwiftUI / Compose)
Platform feel (scroll physics, transitions)	Approximate; noticeable on iOS particularly	Good but not identical to platform conventions	Exact; inherits platform conventions automatically
Team familiarity	High (JavaScript ecosystem)	Low	Moderate (requires Swift and Kotlin capability)
Long-term maintainability	Dependent on Meta's investment priorities	Dependent on Google's investment priorities	Tied to Apple and Google platform roadmaps directly
Notification customisation	Limited without native modules	Better than React Native; some limits remain	Unrestricted

The team familiarity row is the honest counterargument for React Native, and it deserves a direct response. The productivity advantage of building in a familiar language is real, particularly in the first few months. But the Three Lochs notification experience, specifically the maintenance request loop and the proactive status updates, depends on push notification handling that React Native wraps imperfectly. The third-party libraries that fill the gap have patchy maintenance records and introduce version dependencies that compound with every React Native upgrade cycle. The short-term productivity gain from React Native would be spent managing that complexity within the first year of the product being live.

Flutter avoids the JavaScript bridge problem but introduces a different one: the Dart rendering engine produces UI that is visually close to native but not identical. On iOS in particular, scroll physics and modal transitions are slightly off in ways that iOS users notice without being able to articulate. For a product where residents need to trust the app from the first open, that subtle wrongness is a liability.

The practical implication of choosing native is that the team needs engineers who can work in both Swift and Kotlin, or separate specialists for each platform. That is a real cost and it should be planned for. The shared logic between platforms, API calls, data models, notification payload parsing, business rule validation, is documented and kept thin so that maintaining two codebases does not mean maintaining two entirely separate products. The native choice adds short-term staffing complexity in exchange for a product quality ceiling that the resident experience requires.

Backend

Node.js is the runtime. The team's existing capability sits in the JavaScript ecosystem, and there is no domain-specific requirement in this product that changes that calculus. The argument for

request computation, and a notification dispatch pattern that is inherently asynchronous. The event loop model is a natural fit for a system that spends much of its time waiting on database queries, external API calls to Stripe and FCM, and background job results coming back through Redis.

The one honest trade-off with Node.js is CPU-bound work. If the product ever needs to do heavy in-process computation, image processing at scale or complex PDF generation for board reports, Node is not the right tool for that specific task. For Three Lochs, neither of those requirements is large enough to change the runtime recommendation. Photo uploads from maintenance reports are passed directly to S3; the backend handles the pre-signed URL generation and metadata, not the file itself. That keeps the CPU profile light.

Framework

NestJS is the framework. The alternatives evaluated were Express, Fastify, and Hapi.

Dimension	Express	Fastify	Hapi	NestJS
Structure and conventions	Minimal; structure is entirely team-defined	Minimal; performance-focused with some plugin conventions	Good conventions; less community momentum now	Strong; opinionated module architecture out of the box
TypeScript support	Manual setup; not first-class	Good; TypeScript supported but not the default posture	Adequate	First-class; TypeScript is the default
Dependency injection	None; manual wiring	None natively	Limited	Built-in; central to the architecture
Module separation	By convention only	By convention only	By convention only	Enforced by the framework
WebSocket support	Third-party (socket.io)	Plugin-based	Plugin-based	Built-in gateway abstraction over socket.io or ws
Background job integration	Manual	Manual	Manual	`@nestjs/bull` integrates BullMQ with minimal boilerplate
Testing patterns	Manual setup	Manual setup	Good	First-class; test module factory built in

Dimension	Express	Fastify	Hapi	NestJS
Onboarding a new engineer	High variance; depends entirely on team conventions	High variance	Moderate	Low variance; framework enforces the patterns

Express is the honest baseline. Most Node.js engineers have used it. It is fast to start with and flexible. But that flexibility is the problem. A product with the domain complexity of Three Lochs, separate modules for payments, maintenance, bookings, notifications, resident management, and the board dashboard, will develop its own conventions or it will develop inconsistency. Either outcome requires the team to make and maintain architectural decisions that NestJS makes by default. Express is fine for an API with three routes. It is a liability on a team-built product that needs to be handed over, extended, and debugged by engineers who did not write the original code.

Fastify's performance advantage over NestJS is real in benchmarks and largely irrelevant in practice for this product. The bottleneck for Three Lochs will always be database queries and external API calls, not framework throughput. Choosing Fastify over NestJS for performance reasons here is solving a problem that does not exist while giving up the structural benefits that matter.

Hapi had a strong following several years ago. Its community momentum has declined and the pool of engineers who know it well has shrunk. That is a maintainability concern, not a technical one, but maintainability concerns compound.

NestJS is recommended because the domain structure of this product maps directly onto NestJS modules. Each feature area is a module with its own controller, service, and repository layer. That separation is enforced by the framework rather than agreed by convention, which means it survives team changes and stays consistent under deadline pressure. The built-in WebSocket gateway handles the real-time maintenance status updates without adding a separate dependency. The `@nestjs/bull` integration wires BullMQ into the module system with decorators rather than manual queue configuration. These are not cosmetic conveniences. They are the difference between a backend that is coherent in year two and one that has accumulated workarounds.

Language and Type Safety

TypeScript is non-negotiable. Strict mode is on from day one, with `"strict": true` in the compiler configuration and no exceptions added to loosen it. The codebase does not use `any` as an escape hatch. Where a type is genuinely unknown at compile time, `unknown` is used and narrowed explicitly.

The reasoning is straightforward. The maintenance request loop, the dues payment flow, and the notification dispatch chain all involve data moving between the mobile client, the REST API, the database, and FCM. A type error in the notification payload, a property name mismatch between the API response and the mobile model, or an unhandled nullable in the dues calculation is not a compiler warning in a weakly-typed codebase. It is a silent failure at runtime, discovered by a

resident who submitted a request and heard nothing. TypeScript with strict mode converts that class of error into a build failure before it reaches a device.

DTOs use `class-validator` decorators throughout. Every incoming request is validated at the controller boundary before it reaches the service layer. This is NestJS's recommended pattern and it is applied consistently: no request object is trusted until it has passed through a validation pipe. The `ValidationPipe` is registered globally with `whitelist: true` and `forbidNonWhitelisted: true`, which strips unknown properties from incoming payloads rather than passing them through silently.

Database models are typed via TypeORM entities with strict column type declarations. The relationship between a maintenance request, the resident who submitted it, the property it relates to, and the notifications it generates is expressed in the entity graph at the TypeScript level, not inferred at query time. That explicitness catches schema drift early and keeps the service layer honest about what it can and cannot assume about the data it receives.

One constraint worth naming: strict TypeScript discipline requires the team to be consistent about it. A single engineer who routinely reaches for `as any` to move faster creates type holes that the rest of the codebase cannot protect against. This is a team standard, not a linter rule, and it should be treated as such from the first pull request.

Database

Primary Database: PostgreSQL

The domain of Three Lochs is relational by nature. A maintenance request belongs to a resident, who belongs to a property, which belongs to a neighbourhood, which has amenities, bookings, and associated notifications all connected to it. That web of relationships is not incidental to the product. It is the product. The notification that tells a resident their streetlight has been fixed is only possible because the system can traverse from a resolved maintenance ticket back to the resident who submitted it, find their notification preferences, and dispatch the right message through FCM. A database that cannot express and query those relationships cleanly is a liability in exactly the places the product cannot afford one.

PostgreSQL handles this domain with no friction. The relational model maps directly onto the entities: residents, properties, neighbourhoods, dues records, maintenance requests, amenity bookings, announcements, and the notification log that underpins the proactive communication pattern. Foreign keys enforce integrity. Transactions ensure that a dues payment and its associated confirmation notification are either both committed or both rolled back. Joins between the maintenance request, its status history, and the resident record are straightforward queries, not application-layer assembly problems.

The alternatives considered were MySQL, MongoDB, and PlanetScale.

Dimension	MySQL	MongoDB	PlanetScale	PostgreSQL
Relational model fit	Good; relational with strong foreign key support	Poor; document model requires denormalisation or application-side joins	Good; MySQL-compatible, strong horizontal scaling story	Excellent; full relational model, strong constraint enforcement
JSON support	Limited; JSON column type available but querying is awkward	Native; document model is the design	Limited	Excellent; `jsonb` column type with full indexing and querying
Transaction support	Good; InnoDB engine supports ACID transactions	Multi-document transactions available but complex	Good	Excellent; ACID-compliant, mature transaction handling
Full-text search	Adequate for simple cases	Good natively	Adequate	Good; sufficient for resident directory and announcement search without adding a separate service
TypeORM support	Good	Good	Partial; some TypeORM features unsupported	Excellent; TypeORM's primary target
Community and ecosystem	Large; well-understood	Large	Growing but younger	Very large; extensive tooling, documentation, and operational knowledge
Hosting on AWS RDS	Fully supported	Requires Atlas or self-managed EC2	Not natively on RDS	Fully supported on RDS and Aurora PostgreSQL
Schema migrations	Standard	Schema-less; migrations are an application concern	Requires branching workflow; some friction with TypeORM	Standard; TypeORM migrations work cleanly

metadata, the specific details of an issue that do not fit a fixed schema, architectural approval submissions with variable field sets, resident notification preferences that may evolve as the product grows, all benefit from a column that can hold structured JSON and be queried efficiently against it without a separate document store. MySQL's JSON support is functional but querying against it is noticeably more cumbersome. Second, PostgreSQL's constraint system is more expressive. Partial indexes, check constraints, and exclusion constraints give the data model more precision without pushing validation logic into the application layer. Both matter when the product is being maintained by engineers who did not write the original schema.

MongoDB is not recommended, and this deserves a direct statement rather than a table row. The core argument for MongoDB in any project is schema flexibility. The core argument against it here is that flexibility in the wrong places creates integrity problems that surface late and are expensive to fix. The relationship between a resident, their property, their dues history, and their maintenance submissions is not flexible. It is precise and contractual. A dues record that references a property that no longer exists in the system, or a maintenance request with no resident attached, is not a schema design question. It is a data corruption problem that PostgreSQL's foreign key constraints prevent by default and MongoDB requires the application to prevent explicitly, which means it occasionally does not. For a product handling payment records and maintenance accountability, that is not an acceptable trade-off.

PlanetScale's horizontal scaling story is compelling for products with genuinely large write volumes. Three Lochs will never produce write volumes that PostgreSQL on RDS cannot handle. The community produces a bounded, predictable amount of activity: roughly 847 residents paying dues monthly, a finite number of amenity bookings per day, and maintenance requests that are logged discretely rather than streamed continuously. PlanetScale introduces a branching schema workflow that adds operational complexity without solving a problem this product has.

PostgreSQL Configuration on AWS

The database runs on Amazon RDS for PostgreSQL. Multi-AZ deployment is on from day one, not added later when uptime becomes a concern. The cost difference between single-AZ and Multi-AZ on an instance sized for this product is modest. The cost of an unplanned outage during a period when residents are trying to pay dues or check maintenance status is not.

The instance class recommendation for launch is `db.t3.medium`, which is sufficient for the anticipated load at 847 homes. The scaling path is straightforward: RDS supports vertical scaling with a brief maintenance window, and read replicas can be added if read-heavy dashboard queries from the board interface begin to affect response times for residents. That situation is unlikely at this scale but the architecture supports it without migration work.

Connection pooling uses PgBouncer in transaction mode, deployed as a sidecar to the NestJS containers on ECS. TypeORM's built-in connection pool is not sufficient for a containerised backend where multiple ECS tasks each maintain their own pool. Without PgBouncer, the combined

connection count from multiple container instances can exhaust PostgreSQL's connection limit under moderate load. PgBouncer keeps the effective connection count to the database stable regardless of how many containers are running.

Migrations run via TypeORM's migration tooling as part of the deployment pipeline in GitHub Actions. No migration runs against the production database manually. The migration step runs before the new container version is deployed, and the pipeline is configured to halt if the migration fails. Schema changes that cannot be applied without downtime are handled with backward-compatible multi-step migrations: add the new column, deploy the code that writes to both old and new, backfill, remove the old column in a subsequent release. That pattern avoids maintenance windows and keeps the deployment process predictable.

Caching Layer: Redis

Redis serves two distinct roles in this stack and they are worth separating clearly, because conflating them leads to poor cache key design and eventually to cache invalidation problems that are frustrating to debug.

The first role is application caching. The second is the queue backend for BullMQ, which handles the background jobs described in the backend section. In development, a single Redis instance handles both. In production they are separate instances: one optimised for low-latency reads with persistence configured appropriately for cache data, one optimised for the reliability characteristics that a job queue requires. Running them together in production is a common cost-saving decision that creates an operational risk: a memory pressure event on the cache instance affecting the job queue, or vice versa, is exactly the kind of failure that takes down the notification loop at the worst possible time.

What belongs in cache

The guiding principle is simple. Cache data that is read frequently, changes infrequently, and whose staleness for a short window carries no material consequence. Cache nothing where serving stale data would cause a resident to act on incorrect information.

The home screen personalisation bundle is the primary cache target. When a resident opens the app, the API must return their name, neighbourhood, dues status, amenity availability for today, and the next upcoming community event. That is at minimum five separate queries on a cold read. The acceptable latency for that response is low: the home screen has to appear quickly enough that the app feels immediate rather than loading. Caching the assembled home screen bundle per resident, with a TTL of five minutes, means the common case returns from Redis in single-digit milliseconds rather than triggering a multi-query database round trip.

The invalidation rule for dues status is explicit: whenever a payment is confirmed via the Stripe webhook handler, the cached home screen bundle for that resident is invalidated immediately. A

resident who pays their dues and opens the app thirty seconds later should see their dues marked current, not see a cached state that still shows them as outstanding. That specific scenario is an emotional landmine: the product's most anxiety-reducing feature failing at exactly the moment it needs to work.

Amenity availability data is cached at the amenity level, not per resident, with a TTL of two minutes. Pool availability, gym session slots, and court booking status change when bookings are made or cancelled. Two minutes of potential staleness is acceptable for a resident browsing what is available this afternoon. It is not acceptable for the booking confirmation flow itself, which reads directly from the database.

Community announcements and event listings are cached with a longer TTL, thirty minutes, since they change on board action rather than continuously. The cache is invalidated when the board posts or edits an announcement. The event feed on the home screen reflects this: a new event posted by the board will appear within thirty minutes for a resident who has not refreshed, and immediately for anyone whose cache has been invalidated.

The resident directory is not cached at the query level. It is a less frequent access pattern and the data sensitivity around resident contact information means serving a stale directory entry, one where a resident has updated their preferences or opted out of the directory, is a worse outcome than a slightly slower query.

What does not belong in cache

Maintenance request status is read directly from the database every time. The reason is the same as the dues status rule, inverted. A resident who opens their maintenance request and sees it still marked as received when it was assigned to a named team member an hour ago has just been shown a lie. The entire trust architecture of the product depends on the maintenance loop being accurate in real time. No TTL is short enough to make caching that data safe. The query is a simple primary key lookup on a single row; it does not need caching.

Payment history is never cached. It is financial data and it reads from the database. The performance overhead of that decision is negligible.

Dues balances are not cached independently. They are included in the home screen bundle, which has its own invalidation rule. An isolated dues balance cache would create a second invalidation path that is easily forgotten when the dues calculation logic changes.

Redis configuration on AWS

Redis runs on Amazon ElastiCache. The recommended instance class for launch is `cache.t3.small` for the application cache instance. The BullMQ queue instance is discussed in the background jobs section. ElastiCache handles patching, failover, and replication without manual

management, which matters for a team that does not need to be operating Redis infrastructure in addition to everything else.

Persistence for the application cache instance is configured with RDB snapshots only, not AOF. The cache is reconstructable from the database if the instance is lost; persistence is a convenience to avoid a cold cache on restart rather than a durability requirement. The queue instance has different persistence requirements and is configured accordingly.

Key naming follows a consistent namespace pattern: `threelocks:<entity>:<id>:<qualifier>`. For example, `threelocks:homescreen:resident:abc123` or `threelocks:amenity:pool:availability:2024-11-14`. Consistent namespacing makes cache inspection during debugging straightforward and prevents key collisions as the product grows.

Infrastructure

Cloud Provider: AWS, us-east-1

AWS is the cloud provider. The summary table confirms this and the reasoning does not require extensive defence. AWS is where RDS for PostgreSQL, ElastiCache, ECS Fargate, S3, and CloudWatch all live natively, and the integration between those services is tighter and better documented than the equivalent on any other provider. Running the database, cache, containers, file storage, and logging on the same provider eliminates a category of cross-provider networking and credential management complexity that adds cost and operational burden without adding capability.

The region is us-east-1 (Northern Virginia). Three Lochs is based in central Florida. US-East-1 is the geographically closest AWS region with full service availability for every component in this stack, including RDS Multi-AZ, ElastiCache, and ECS Fargate. Latency from central Florida to us-east-1 is consistently low, typically under 20ms, which is well within acceptable bounds for the API response profile this product requires. There is no use case here that justifies the additional operational complexity of a multi-region deployment at launch.

One thing worth stating plainly: us-east-1 has a higher historical rate of AWS service incidents than some other regions, simply because it is AWS's largest and oldest region and carries a disproportionate share of their global traffic. The mitigation is not a different region. It is the Multi-AZ configuration already specified for RDS and ElastiCache, which provides availability guarantees within the region that are sufficient for a community of 847 homes.

Containerisation

The NestJS backend runs in containers on ECS Fargate. The decision between containerisation options was between ECS Fargate, EKS (Kubernetes), and EC2 instances managed directly.

Dimension	EC2 (direct)	EKS (Kubernetes)	ECS Fargate
Operational overhead	High; patching, scaling, and instance management are manual	High; Kubernetes control plane and cluster management require specialist knowledge	Low; AWS manages the underlying infrastructure entirely
Team Kubernetes familiarity required	None	Yes; meaningful learning curve and ongoing operational skill required	None
Scaling behaviour	Manual or via Auto Scaling Groups; configuration-heavy	Excellent; Kubernetes autoscaling is mature	Good; service autoscaling via Application Auto Scaling, simpler than Kubernetes
Cost model	Pay for instance uptime regardless of utilisation	Control plane cost plus node costs; more expensive at this scale	Pay per vCPU and memory used by running tasks; cost-efficient at predictable load
Deployment integration with GitHub Actions	Requires custom scripting	Requires kubectl configuration and cluster credentials management	Native AWS CLI commands; well-supported in Actions
Service discovery	Manual or via Route 53	Built-in via Kubernetes service mesh	Via AWS Cloud Map or Application Load Balancer; sufficient for this architecture
Suitability for this scale	Adequate; overengineered operationally	Overengineered significantly; Kubernetes complexity is not justified at 847 homes	Right-sized; handles the load, scales cleanly, low operational burden
Container image management	Manual	ECR or external registry	ECR natively; first-class integration

EC2 directly managed is not recommended. It requires the team to own instance patching, security group configuration, and scaling policy management in ways that have no product value. The time spent keeping EC2 instances healthy is time not spent on the resident experience.

Kubernetes via EKS is also not recommended. It is a powerful platform for organisations running dozens of services with complex traffic routing requirements and dedicated platform engineering resource. Three Lochs has one backend service, a PostgreSQL database, two Redis instances, and a straightforward scaling requirement that peaks modestly around the first of each month when dues

payments process. Kubernetes introduces a control plane, node pools, YAML manifests, and operational knowledge requirements that are disproportionate to the problem. Teams that adopt Kubernetes before they need it consistently spend the first year managing Kubernetes rather than building product.

ECS Fargate is recommended. The NestJS backend runs as an ECS service with a target tracking scaling policy tied to average CPU utilisation. The initial task definition specifies 0.5 vCPU and 1GB memory per task, with a minimum of two tasks running at all times for availability. That configuration handles the expected load comfortably and scales horizontally if needed without manual intervention.

Container images are built in GitHub Actions, tagged with the Git commit SHA, pushed to Amazon ECR, and referenced in the ECS task definition by that specific tag. No deployment ever uses a `latest` tag in production. Every production deployment is traceable to a specific commit, which matters when diagnosing an incident and needing to know exactly what code was running at the time.

The Application Load Balancer sits in front of the ECS service and handles HTTPS termination. SSL certificates are managed via AWS Certificate Manager with automatic renewal. The ALB also handles the WebSocket connections for real-time maintenance status updates: WebSocket upgrade requests are routed to the same ECS tasks via sticky sessions on the target group, which keeps the WebSocket connection stable across the task's lifetime without requiring a separate connection management service.

PgBouncer runs as a sidecar container within each ECS task, sharing the task's network namespace. This is the same configuration described in the database section. It is worth noting here because the sidecar pattern is idiomatic on ECS and does not require any additional infrastructure: the PgBouncer container is defined in the same task definition as the NestJS container, starts before it, and the NestJS process connects to `localhost:5432` which PgBouncer intercepts before proxying to RDS.

Infrastructure as Code: Terraform

All AWS infrastructure is defined in Terraform. Nothing is provisioned through the AWS console. That rule applies from the first day of the project, not from some future point when the infrastructure becomes complex enough to justify it. Infrastructure that starts as manual console configuration tends to stay that way, because the team learns the shape of things through the console and then finds it easier to change things there too. The result is infrastructure that cannot be reproduced, audited, or safely modified by anyone who was not present when it was originally set up.

The Terraform state file is stored in an S3 bucket with versioning enabled and DynamoDB state locking. Remote state with locking is non-negotiable for any team with more than one engineer

touching infrastructure. State file conflicts from concurrent applies produce corrupted infrastructure state that is difficult to recover from cleanly.

The repository structure separates Terraform configuration by environment. A `terraform/` directory at the root contains subdirectories for `staging` and `production`, each with their own state files and variable definitions. Shared modules for the ECS service definition, RDS configuration, and ElastiCache setup live in a `modules/` directory and are referenced by both environments. This means staging and production are structurally identical and differences are expressed only in variable values, instance sizes, and replica counts, not in divergent configurations that accumulate over time.

Every infrastructure change goes through a pull request with a `terraform plan` output attached. No `terraform apply` runs against production without a reviewed plan. The GitHub Actions pipeline generates the plan automatically when a pull request is opened against the infrastructure branch, posts it as a pull request comment, and requires approval before the apply step runs. That pattern means infrastructure changes have the same review process as code changes, and the person approving can see exactly what will change before approving it.

CI/CD: GitHub Actions

GitHub Actions handles the full build, test, and deployment pipeline. The decision was straightforward: the codebase lives in GitHub, the team is familiar with GitHub, and GitHub Actions has native integration with ECR, ECS, and the AWS CLI that covers everything this pipeline needs without additional tooling.

The pipeline structure covers four distinct triggers with different step sets.

Trigger	Steps	Target
Pull request opened or updated	Lint, TypeScript compiler check, unit tests, integration tests against a test database, `terraform plan` for any infrastructure changes	No deployment; results posted to pull request
Merge to `main`	All PR steps, Docker image build, push to ECR tagged with commit SHA, deploy to staging via ECS task definition update, run database migrations against staging, smoke tests against staging API	Staging environment
Release tag pushed (e.g. `v1.2.3`)	All `main` steps, deploy to production via ECS task definition update, run database migrations against production, smoke tests against production API, Sentry release created and source maps uploaded	Production environment

Trigger	Steps	Target
Scheduled nightly (02:00 UTC)	Dependency vulnerability scan, stale branch cleanup notification	No deployment

A few specifics worth naming. The migration step runs before the new container version is active. The pipeline updates the task definition to point at the new image, but the ECS deployment is configured with a minimum healthy percentage of 100%, meaning new tasks start before old tasks are stopped. The migration step runs as a one-off ECS task against the new image before the service deployment begins, using the same image that will be deployed. If the migration task fails, the pipeline stops and the current service version continues running unaffected.

Smoke tests after each deployment are not comprehensive end-to-end tests. They are a small set of health-check requests that confirm the API is responding, authentication is working, and the database connection is live. They complete in under thirty seconds. Their purpose is to catch a broken deployment before a resident does, not to replicate the full test suite against production.

Secrets are stored in AWS Secrets Manager and injected into ECS task definitions as environment variables at runtime. No secrets appear in GitHub Actions environment variables, in the Terraform state file, or in the repository at any point. The GitHub Actions role uses OIDC federation to assume an AWS IAM role with the minimum permissions needed for each pipeline step. Static AWS access keys are not used.

The Sentry release step at the end of a production deployment uploads source maps and creates a release marker that ties any subsequent error events in Sentry to the specific commit deployed. That traceability matters when an error report comes in from a resident and the team needs to identify exactly which code change introduced it.

Third-Party Services

Error Tracking: Sentry

Sentry is the error tracking tool. It is integrated into both the NestJS backend and the native mobile clients, iOS and Android, from day one.

On the backend, the `@sentry/node` SDK is initialised in the NestJS bootstrap function before the application starts listening. A global exception filter catches unhandled exceptions and forwards them to Sentry with the full request context attached: the resident's ID (not their name or contact details), the request path, the HTTP method, and the correlation ID that links the backend error to the specific notification or API call that triggered it. That correlation ID is set on every incoming request by middleware and propagated through the call chain, which means a Sentry error event can be traced back to the specific FCM message or Stripe webhook that initiated it.

product: the home screen animation and card transition quality is load-bearing for the first impression, and a Sentry performance trace that shows a 400ms frame drop on the home screen load is more useful than a crash report because it catches the problem before it becomes a resident complaint.

Source maps for the backend TypeScript compilation are uploaded to Sentry as part of the production deployment step in GitHub Actions, as noted in the infrastructure section. Every error event in production shows the original TypeScript source line, not the compiled JavaScript. That distinction matters when the team is debugging a notification dispatch failure at 11pm and needs to move quickly.

The release marker created at deployment time means Sentry can show exactly which release introduced an error. Combined with the commit SHA tagging on ECS task definitions, the team can identify the change that caused a problem and assess whether a rollback is needed, without reconstructing that timeline manually.

One data handling constraint applies here. Sentry must not receive personally identifiable information about residents. The default Sentry SDK behaviour is to capture request bodies, which in this product can include names, addresses, and payment-adjacent data. The `RequestDataIntegration` is configured with `allowedHeaders` and the request body is explicitly excluded from all Sentry payloads. Resident IDs are included for traceability; everything else that could identify an individual is stripped before the event leaves the server. This is configured in code, not relied upon in the Sentry dashboard, so it cannot be accidentally reversed by a settings change.

Application Monitoring and Alerting: Datadog

Datadog handles application performance monitoring, infrastructure metrics, and alerting. CloudWatch collects logs from ECS tasks and the RDS instance natively; those logs are forwarded to Datadog via a Firehose stream rather than queried directly in CloudWatch. The practical reason is simple: Datadog's query interface and dashboard tooling are considerably better for the kind of investigation the team will actually do, which is correlating an API latency spike with a database query pattern or a Redis cache miss rate. CloudWatch remains the source of truth and the audit log. Datadog is the lens the team uses to look at it.

The metrics that matter most for this product are specific and worth naming directly., API response time for the home screen personalisation endpoint, broken down by resident and by neighbourhood, with an alert threshold at 800ms p95, Maintenance request status update latency from submission to first status change, with an alert if the median exceeds one hour, FCM delivery success rate, tracked via the backend notification dispatch service, with an alert if the delivery failure rate exceeds 2% over a five-minute window, Stripe webhook processing time and failure rate, with an immediate alert on any unprocessed webhook older than ten minutes, BullMQ queue depth for the notification job queue, with an alert if jobs are backing up beyond a defined threshold, Database connection count

via PgBouncer, to catch connection pool exhaustion before it affects residents, Redis memory utilisation on both instances, with headroom alerts before eviction begins

The dashboard structure separates the resident experience view from the infrastructure view. The resident experience dashboard shows the metrics that directly affect what a resident sees and feels: home screen load times, notification delivery rates, maintenance loop latency, dues confirmation times. The infrastructure dashboard shows what is happening underneath: ECS task CPU and memory, RDS query performance, Redis hit rates, queue depths. Both matter, but the resident experience dashboard is the one the team checks first when something feels wrong, because it answers the question that actually matters: are residents being affected right now.

Alerting routes to a dedicated Slack channel for warning-level alerts and to PagerDuty for critical alerts. Critical is defined narrowly: the notification dispatch service failing, the dues payment webhook processing stopping, or the API becoming unavailable. Everything else is a warning. Alert fatigue is a real operational risk and a tightly defined critical threshold keeps the team responsive rather than habituated to noise.

Push Notifications: Firebase Cloud Messaging

FCM is confirmed as the push notification provider. The reasoning is covered in the summary, but the implementation specifics are worth stating here.

FCM is integrated into the NestJS notification module via the Firebase Admin SDK. The notification service is a dedicated NestJS module with a single responsibility: accept a notification job from BullMQ, resolve the resident's FCM token from the database, construct the appropriate platform-specific payload, dispatch it, and log the result. That separation keeps the notification logic out of the maintenance, dues, and booking modules, which should be unaware of how notifications are delivered.

The payload structure for maintenance status notifications is a data message, not a notification message, on both platforms. As described in the mobile section, data messages give the native client full control over how the notification is displayed, which is required for the deep-link behaviour that routes a tap directly to the correct maintenance request screen. The payload includes the request ID, the new status, the assigned team member's name, and the expected completion date where applicable. The mobile client assembles the display notification from those fields, which means the notification text is generated on the device using the same copy standards applied throughout the app.

FCM token management requires a small but important implementation detail. Tokens expire and change when a resident reinstalls the app or clears app data. The mobile client sends the current FCM token to the backend on every app launch, and the backend upserts it against the resident record. Notifications sent to a stale token fail silently in FCM unless the backend checks the response for `UNREGISTERED` error codes and removes the token immediately. That check is

implemented in the notification service's dispatch handler. Accumulating stale tokens is not just a minor inefficiency; it masks the true notification delivery rate in monitoring and can cause maintenance status updates to disappear without the team knowing.

Topic-based messaging is used for community-wide announcements. When the board posts an announcement, the notification module publishes to a neighbourhood-specific FCM topic rather than enumerating every resident token individually. Residents are subscribed to their neighbourhood topic at registration and unsubscribed if they opt out of community announcements. This approach scales cleanly and does not require the backend to manage a list of tokens for bulk sends. Per-resident notifications, dues confirmations, maintenance updates, booking reminders, use individual token dispatch.

The data handling constraint for FCM is straightforward. Notification payloads must not contain personally identifiable information beyond what is needed to display the notification. A maintenance status update payload includes the request ID and status text. It does not include the resident's full name, address details, or any payment information. The mobile client retrieves the display name from local session state, not from the notification payload.

Analytics: Mixpanel

Mixpanel is the product analytics tool. The implementation approach on this product requires more deliberate thought than analytics typically gets, because the resident data involved is sensitive and the community is small enough that careless instrumentation could inadvertently expose individual behaviour patterns.

The core principle for instrumentation is simple. Track actions and flows, not identities. Mixpanel events use a pseudonymous resident ID, never a name, email address, or property identifier that could be reverse-engineered to an individual in a community of 847 people. The Mixpanel user profile attached to that ID contains neighbourhood and tenure data but nothing more specific.

The events worth tracking are directly tied to the design principles established through the workshops., Home screen load and time-to-meaningful-content, the point at which dues status, amenity availability, and the next event are all visible, Notification open rate by notification type, which is the signal identified as the measure of whether the concierge tone is earning attention, Task completion funnels for the three primary resident actions: dues payment, amenity booking, and maintenance request submission, Session one action completion, the measure of whether the first session ends with something real, Maintenance request submission to resolution viewed, tracking whether residents actually return to close the loop themselves or whether the proactive notification makes that unnecessary, App opens that originate from a push notification tap versus organic opens, Board announcement read rates, broken down by announcement type

What is explicitly not tracked: which specific resident submitted which request, granular location data within the estate, or any behavioural pattern that could identify a resident's daily routine.

Mixpanel's data residency is configured for US servers, which is consistent with the product's Florida jurisdiction.

The Mixpanel integration sits in a thin analytics module on the backend and in a lightweight wrapper on the mobile clients. Calls to Mixpanel are fire-and-forget and never block the main execution path. An analytics failure must never affect the resident's experience of completing a task. The wrapper handles this by dispatching events asynchronously and swallowing errors silently in production while logging them in development.

The board dashboard does not surface Mixpanel data directly. The board sees operational metrics: dues collected, open tickets, upcoming bookings. Resident behaviour data stays within the product team. That boundary is worth maintaining clearly, both for resident trust and to prevent the board from drawing conclusions about individual residents from aggregate usage patterns.

Email Delivery: SendGrid

SendGrid handles transactional email. The use cases in this product are bounded and deliberate: dues receipts, maintenance request confirmation emails as a fallback when push notifications are disabled, password reset flows managed via Auth0, and board-originated announcements for residents who have opted into email delivery.

The SendGrid integration in NestJS uses the official `@sendgrid/mail` client, called from within the notification module. Email is always a secondary channel, not the primary one. The notification service dispatches push via FCM first. Email is sent concurrently, not as a fallback after FCM failure, because the use cases differ: push notifications are for time-sensitive status updates, email is for receipts and records that residents may want to refer back to later. A dues payment generates both: an FCM notification that says "all sorted for June," and a SendGrid receipt email with the payment details and amount for the resident's records.

Dynamic templates in SendGrid hold the email copy. This matters for the same reason consistent tone matters everywhere else in the product. The person who writes and maintains the email templates should be working from the same voice standards as the rest of the product, and template management in the SendGrid dashboard is more accessible to a non-engineer than editing code. The template IDs are stored in environment variables and referenced by the notification service, so updating a template does not require a deployment.

The unsubscribe and preference management for email is handled explicitly in the resident profile, not delegated to SendGrid's list management. A resident who opts out of email notifications has that preference stored in the database and the notification service respects it before making the SendGrid API call. SendGrid's own suppression list handles hard bounces and spam complaints, and the backend processes SendGrid webhooks to sync those suppressions back to the resident record. A notification sent to a suppressed email address does not generate an error in SendGrid; it

silently drops. Without the webhook sync, the backend would continue attempting email delivery to an address that will never receive it.

One sending domain point worth stating: all email from the product sends from a Three Lochs subdomain, such as `notify.threelochs.com`, with SPF, DKIM, and DMARC records correctly configured. SendGrid's domain authentication handles this. Email that arrives from a generic SendGrid domain rather than a recognisable Three Lochs address undermines the same trust the notification copy is trying to build. A resident who receives a dues receipt from `mail.sendgrid.net` is right to be suspicious of it.

What Is Not Recommended

Technology	Reason
React Native	Push notification deep linking is the core reliability requirement of this product. React Native wraps the native notification APIs imperfectly, and the third-party libraries that fill the gap have inconsistent maintenance records. Cold-start routing from a notification tap produces edge cases that are disproportionately expensive to resolve. The short-term productivity gain from building in a familiar JavaScript environment would be spent managing that fragility within the first year of the product being live.
Flutter	Avoids the JavaScript bridge problem but introduces a different one. Flutter's Dart rendering engine produces UI that is visually close to native but not identical, particularly on iOS where scroll physics and modal transitions are slightly off in ways that residents notice without being able to name. For a product where the first sixty seconds set the resident's expectation for everything that follows, that subtle wrongness is a liability the product cannot absorb.
Firebase Realtime Database	A document-oriented, eventually consistent store built for high-frequency collaborative data. The Three Lochs domain is relational, transactional, and integrity-dependent. Dues records, maintenance requests, and payment history need foreign key constraints and ACID transactions, not a real-time sync model optimised for chat applications. Firebase Realtime Database would require the application to enforce data integrity that the database should enforce by default, and in practice that enforcement is never complete.
MongoDB	The schema flexibility argument for MongoDB is real in domains where the data model is genuinely unpredictable. This is not one of them. The relationship between a resident, their property, their dues history, and their maintenance submissions is precise and contractual. A dues record that references a property that no longer exists in the system is not a schema design question; it is a data corruption problem. PostgreSQL prevents it by default. MongoDB requires the application to prevent it explicitly, which means it occasionally does not. For a product handling payment records and accountability to 847 homeowners, that trade-off is not acceptable.

Technology	Reason
MySQL	A legitimate relational database that would work here. Set aside in favour of PostgreSQL for two specific reasons: PostgreSQL's `jsonb` column type handles maintenance request metadata and variable-schema approval submissions more cleanly than MySQL's JSON column support, and PostgreSQL's constraint system is more expressive, allowing partial indexes and check constraints that push validation into the database rather than the application layer. The gap between them is not large, but PostgreSQL is the better fit for this domain and is equally well supported on AWS RDS.
PlanetScale	Its horizontal scaling story is compelling for products with genuinely large write volumes. Three Lochs will never produce write volumes that PostgreSQL on RDS cannot handle. 847 homes paying dues monthly, a finite number of daily bookings, and discretely logged maintenance requests do not constitute a scaling problem. PlanetScale's branching schema workflow adds operational complexity and introduces friction with TypeORM migrations without solving any problem this product actually has.
Heroku	A reasonable choice for early-stage prototypes where operational simplicity is the primary constraint. For a production product handling payment records and maintenance accountability for a premium residential community, Heroku's limited control over database configuration, network topology, and deployment behaviour creates constraints that compound as the product matures. The gap between what Heroku exposes and what AWS provides natively for RDS, ElastiCache, and ECS is too wide to ignore once the product has residents depending on it.
EKS (Kubernetes)	Powerful at scale, disproportionate here. Three Lochs has one backend service, one database, two Redis instances, and a scaling requirement that peaks modestly around the first of each month. Kubernetes introduces a control plane, node pools, YAML manifest management, and specialist operational knowledge that serves organisations running dozens of services with dedicated platform engineering resource. Teams that adopt it before they need it consistently spend the first year managing Kubernetes rather than building product.
EC2 (directly managed)	Requires the team to own instance patching, security group configuration, and scaling policy management. None of that work produces any resident-facing value. ECS Fargate handles the same compute requirements with a fraction of the operational overhead, leaving the team's attention on the product rather than the infrastructure underneath it.

Technology	Reason
Resident-facing web portal (phase one)	Not a technology limitation, a scope decision. The resident experience is a mobile-first product. A web portal in phase one would split design, development, and testing effort across two surfaces without meaningfully increasing resident reach. The board dashboard is the only browser-based interface required at launch. A resident web portal can be assessed in a later phase once mobile usage patterns are established and the case for a parallel surface is grounded in actual resident behaviour rather than assumption.
Django / Laravel	Neither was seriously evaluated because the team's existing capability sits in the JavaScript ecosystem. A runtime switch to Python or PHP would require a language transition across the whole team for no domain-specific gain. Node.js handles the I/O profile of this product well, and NestJS provides the structure that a Node project of this complexity requires. The argument for switching would need to be compelling enough to justify that overhead. It is not.
Express / Fastify / Hapi	All three were evaluated as alternatives to NestJS. Express is fast to start with but its flexibility becomes a liability on a product with this many domain boundaries; the team ends up maintaining architectural conventions that NestJS enforces by default. Fastify's performance advantage is real in benchmarks and irrelevant here, where the bottleneck will always be database queries and external API calls. Hapi had genuine strengths but its community momentum has declined and the pool of engineers who know it well has shrunk, which is a maintainability concern that compounds over time.