

Owner's Manual

Sample · Porchlight

Six plain-language documents about your software, each reviewed and signed off by a Cloudburst engineer, with a recommended path and a firm quote.

This is a sample Owner's Manual. It is based on a real product review, but the product name, table names, and identifying details have been fictionalized, and security findings have been generalized so nothing here describes a live weakness. In the real product, the Manual is delivered in Cloudia, Cloudburst's agentic workspace, where the owner has access to the code and can ask their own questions for three months. This is a static version.

Our guarantee: "If you're not happy with your Owner's Manual, we'll refund your money, and you keep the Manual."

Hosting, databases, and third-party services stay in your name and are billed by those providers.

Overview

What the software does

Porchlight is a local recommendations app. People keep and share lists of service providers they trust: a plumber, an electrician, a landscaper. Partners, professionals who recommend providers to their own clients, share lists and see which providers their clients went on to hire. Members build their own lists over time. Business owners claim their profile with a one-time code and reply to customer messages, with automatic translation when the two speak different languages.

The product has a website and a phone app that share most of their code. There are no ratings, no reviews, no directory, and no lead marketplace. A business exists in Porchlight only because a real person added it.

The five findings that matter most

1. **Every database table enforces its own access rules.** Even a bug in the app that tried to read or change data it shouldn't would be refused by the database. This is the most important protection in the system, and it is applied without exception. (Documents 2, 4)
2. **A confirmed hire, the product's most valuable record, cannot be faked by the partner it credits.** It is written only through one controlled procedure, and a partner is blocked from confirming a hire on their own list. (Documents 2, 4)
3. **The phone app has no automated tests of its own.** It leans on well-tested shared code, but nothing checks its screens or flows. (Document 6)
4. **Sign-up does not verify that a person controls the email address they used.** A deliberate choice to reduce friction during the pilot. It needs reversing before a wider public launch. (Document 4)
5. **Backups have never been tested with a restore.** They exist through the database provider, but nobody has documented bringing one back. (Document 4)

How to use the Manual

Each of the six documents is written in the same shape. The short answer comes first, in a few sentences an owner can read in a minute. Then what we found, with the technical detail a developer will want. Then what it means for the business. Then what to do about it, ranked and tagged Now, Soon, or Later. Each document ends with the name of the engineer who reviewed it. The Manual closes with a recommended path and a firm quote, then the engineer's signed review.

Read the short answers first. Hand the whole Manual to any developer, buyer, or partner who asks a question about the software. The answer is in here.

Document	State
1. What it's built on	Solid
2. User roles and permissions	Solid
3. How data is organized	Solid
4. Security review	Solid, two open items
5. What needs updating	Solid
6. How well it's tested	Good, one gap

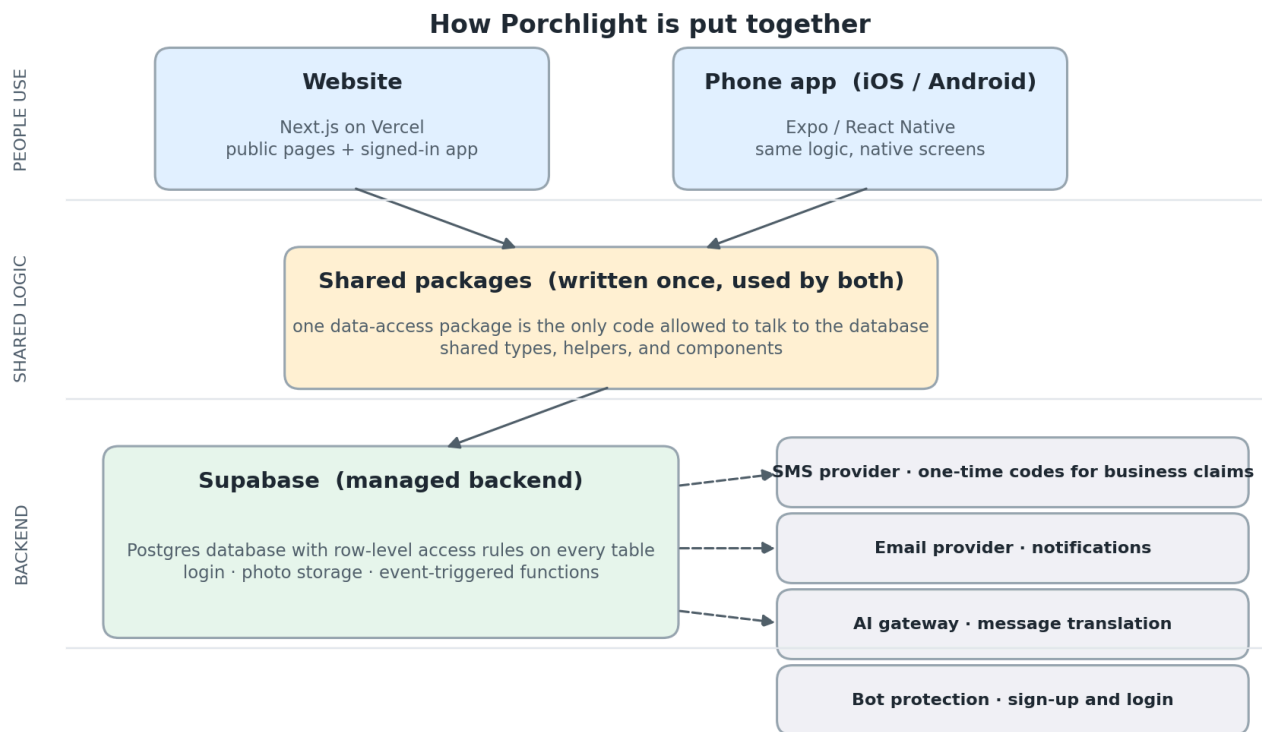
1. What it's built on

The short answer

Porchlight is a web and mobile product built on rented, well-known platforms rather than custom infrastructure. There is no server of your own to patch or keep running. Hosting is Vercel, the database and login system are Supabase, and the application code sits on top. Both the website and the phone app share most of their logic through a set of common packages. These were reasonable choices for a product at this stage, and nothing in the stack would be hard for a new developer to pick up.

What we found

How the pieces fit



Three separate copies: production, staging, and a disposable preview for every proposed change. A change reaches staging before production.

What people use. The website is built with Next.js, a framework that lets public pages (a business's profile) load fast and be read by search engines, while the signed-in parts of the app stay interactive. The phone app is built with Expo, which produces real iOS and Android apps from one React Native codebase.

Shared logic. One data-access package is the only place in the codebase allowed to talk to the database. Neither app writes its own queries. The rules for reading and writing data live in one

auditable place rather than being repeated, and possibly done differently, in two apps.

Backend. Supabase bundles a Postgres database, user login, file storage for photos, and small event-triggered functions as one managed service. A few outside services fill specific gaps: text messages for business verification, email for notifications, an AI gateway for message translation, and bot protection on sign-up and login.

The shape of the codebase

Everything lives in one repository, managed with a build tool (Nx) that understands how the pieces depend on each other and only rebuilds and retests what changed.

```
repo/
├─ apps/
│   ├── web/           the website
│   └─ mobile/         the phone app
├─ packages/
│   ├── types/         shared definitions of what each piece of data looks like
│   ├── data-access/   shared functions for reading and writing data safely
│   ├── ui/            shared visual building blocks (web only)
│   ├── utils/         shared helpers (phone formatting, validation)
│   └─ db-tests/       automated database security tests
├─ supabase/
│   ├── migrations/   every database change ever made, in order
│   └─ functions/     small server-side functions triggered by events
└─ e2e/              automated tests that simulate a real user
```

Outside services, and what breaks without them

Service	Role	If it's unavailable
Supabase	Database, login, file storage	Nothing works
Vercel	Hosts the website; builds and deploys every change	Website unreachable
SMS provider	Sends the one-time code that verifies a business claim	Business claiming only
Email provider	Transactional email (hire notifications, responses)	Email notifications only
AI gateway	Message translation	Translation only
Bot protection	Sign-up, login, and the public intake form	Reduced spam protection

Each service sits behind its own small piece of code. An access-restricted status page shows whether each one is configured correctly in a given environment without revealing secret values, and is checked before every release.

Three separate, isolated environments

Environment	Purpose	Database
Production	The live product	Its own Supabase project
Staging	Rehearsal copy for testing before going live	A separate, non-production project
Preview	Temporary copy created automatically for every proposed change	Shares the staging database

A change always reaches staging before production, and database changes are applied ahead of the code that depends on them: schema first, code second.

End to end, in one example

A business owner replies to a customer message on their phone. The mobile app passes the reply to the shared data-access code, which sends it to Supabase. There is no in-between server. Supabase checks, at the database level, that this person is allowed to post as this business before accepting the write. If the customer's preferred language differs, a small function starts a translation in the background, and the customer sees the translated reply within a couple of seconds.

Accounts you should hold

These are billed by their providers and should be in your name, not a developer's.

Account	What it controls
Domain registrar	The address itself
Vercel	Website hosting and deploys
Supabase (production and staging)	Database, user accounts, photos, backups
Source control (GitHub)	The code
SMS, email, bot-protection providers	Verification codes, email, spam protection
Apple Developer and Google Play	The phone app listings

What this means for you

Every layer is either a well-known managed platform or your own application code on top of it. There is no bespoke infrastructure anywhere, and no vendor the product could not be moved off if it came to that. A developer who knows Next.js, React Native, and Supabase will be productive in days. The monthly cost of the stack is a handful of subscriptions, all in your name.

What to do about it

- **Now.** Confirm every account in the table above is in your name and that you hold the login.
- **Later.** Nothing else. The architecture is sound for this stage.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

2. User roles and permissions

The short answer

Porchlight does not assign each person a permission level from a dropdown. What a person can see and do depends on whether they are signed in, what data they own, and what real-world step they have completed, such as verifying a business by phone. Five practical roles fall out of that. Every boundary is enforced twice: once in the app, and again by the database itself. One role (operator) is granted by a configuration setting rather than inside the app, which is worth knowing about.

What we found

The five practical roles

Role	How you get it	What you can do
Anonymous visitor	Open the site	View public profiles and public lists; browse providers by city and category; open a shared list link; respond to a friend's request for a recommendation. Cannot see anything private, add a provider, or message a business.
Member	Sign up	Everything a visitor can do, plus: build lists, add providers, write private notes, message businesses, save items, follow a specific list, confirm a hire. Full control over your own data only.
Partner	Set at sign-up or during onboarding	Everything a member can do, plus: a branded profile, hand-off links for lists, and a dashboard showing hires credited to you. Cannot confirm a hire on your own list.
Business owner	Claim a business by phone verification	A dashboard scoped to the business you claimed: edit the public profile, read messages sent to it, see who has publicly recommended it. No visibility into any other business or any private note a customer wrote.
Operator	Configured, not granted in the app	Internal review tools (setting up a new partner's first list) and the system health page.

Things worth calling out

A partner cannot inflate their own numbers. The block on confirming a hire on your own list is enforced in the database as well as in the interface. This protects the one number partners care about.

Private means private, counts included. Only a list marked discoverable contributes to anything public: browse pages, public counts, anywhere a stranger could see. A private or link-only list

contributes nothing. This reversed an earlier rule and was a deliberate product decision.

Following targets a list, not a person. A member chooses which of someone's lists to follow. A private list cannot be followed at all.

Printed and public contact details are separate from sign-up details. A partner's public phone and email are stored apart from the email they signed up with. Anything printed or shown publicly uses only the public ones. This closed a real gap.

The operator role is a configured allow-list. A short list of email addresses, set in the deployment configuration, gates a few internal pages. If the list is empty, nobody has operator access. There is no default admin account. This is appropriate for a small team and does not scale gracefully; see Document 4.

Claiming a business: the one place access is granted for good

Claiming is the one moment where someone's access changes lastingly, from "anyone" to "this business's owner." Search by phone number or website, find the unclaimed listing, request a one-time code, receive it by text at the business's own number, enter it. A handful of wrong guesses invalidates the code, and the code is stored only in a form that cannot be read back, even with direct database access.

What a partner's clients can see, and vice versa

A hand-off recipient never needs an account to view a list. The partner learns that a specific named person hired someone off their list only if they personally labeled that hand-off link when creating it. Otherwise the dashboard shows "a client hired" with no way to trace it to an individual. This is an intentional privacy boundary.

How it's enforced

Every boundary above is enforced in the application (which buttons and pages appear) and again at the database, using a Postgres feature called Row Level Security. Even a bug in the app that tried to fetch or change data it shouldn't would be refused by the database. Document 4 covers this in full.

What this means for you

A buyer or auditor will ask "could the wrong person see or change something they shouldn't?" The answer is: not without defeating two independent layers, and the second one is the database itself. The one thing to know is that operator access is a configuration setting, so whoever manages the deployment controls who holds it. Keep that list short and know who is on it.

What to do about it

- **Now.** Confirm who is on the operator allow-list in production.

- **Later.** If the internal team grows beyond a handful of people, move operator access to a proper role stored in the database, so it can be audited and changed without a deploy.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

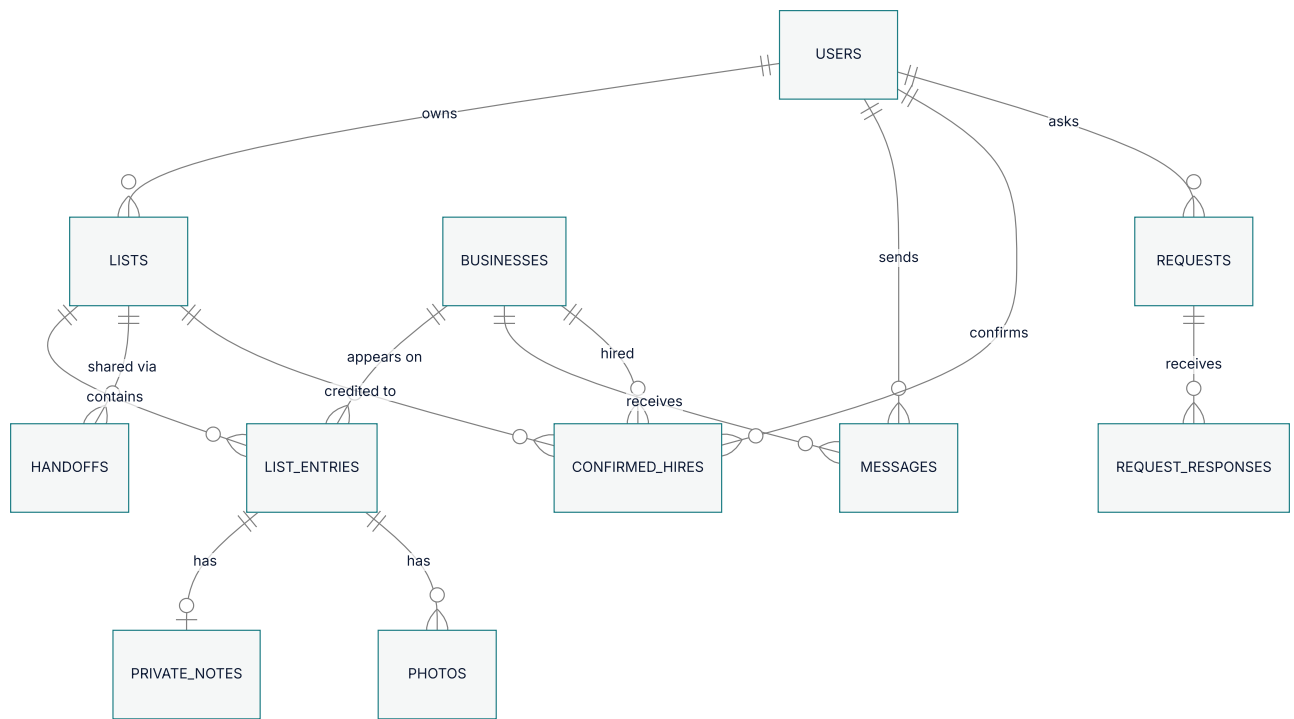
3. How data is organized

The short answer

Porchlight stores its data in a single Postgres database with about twenty tables, built through a few dozen dated, individually recorded changes. There is a complete history of every structural change ever made. The schema is small enough that a new developer can learn all of it in a day, and the reasoning behind the important decisions (why phone number is the key for a business, why hires count distinct people) is written down in the codebase rather than living in one person's memory.

What we found

How the pieces connect



The categories of information

People and businesses

Table	What it holds
users	Every account: name, avatar, city, preferred language, and partner details where applicable. A partner also carries separate public phone and public email fields, the only contact details allowed on anything printed or public

Table	What it holds
businesses	Every service provider added by at least one person. Phone number is the unique fingerprint that prevents the same plumber being stored twice
verifications	Temporary one-time codes used only during claiming

Lists and entries, the core feature

Table	What it holds
lists	Each named, shareable collection a user owns, with its visibility setting (private, link-only, discoverable)
list_entries	The link between a person, a list, and a business: "this business is on this person's list"
private_notes	A note only the list owner can see, stored separately from anything public
photos	Photos attached to an entry

Attribution, the data that proves the product's value

Table	What it holds
confirmed_hires	A confirmed "this member hired this business, off this list." The most protected record in the system
handoffs	Each hand-off link a partner generates, and whether and when it was first opened
link_events	Lighter tracking of a shared link being clicked; an earlier, weaker signal than a confirmed hire

Communication

Table	What it holds
messages	Every message between a member and a business, including original and translated text
requests, request_responses	A private request for a recommendation, and the answers friends submit
pending_recommendations	A responder's recommendation, held until they create an account

Social

Table	What it holds
follows	Who follows which list

Table	What it holds
saves	A person's private bookmarks

Onboarding (internal, not customer-facing)

Table	What it holds
onboarding_requests	Submissions from prospective partners
ai_extractions	Usage tracking for the AI tool that reads a submitted provider list out of free text

Design decisions worth knowing about

- **Phone number anchors every business record.** It is normalized to a standard format before storing, and it is the only thing checked to decide "have we seen this business before." This one rule prevents duplicate records for the same plumber piling up over time.
- **A hire counts distinct people, not entries.** If one partner has the same plumber on three lists, that plumber's public count goes up by one. This was a specific, dated correction in the database history.
- **Only a discoverable list contributes to anything public.** This reversed an earlier rule under which private lists still counted toward public tallies. Private means private, counts included.
- **Nothing is ever bulk-imported.** No code path pulls listings from Google, Yelp, or any outside directory. Every row exists because a real person added it. The dataset stays trustworthy at the cost of never looking bigger than it is.
- **A few tables are one step ahead of the product.** Confirmed hires distinguish a first-time hire from a later reconfirmation, which the product does not heavily surface yet. Captured now so no historical data needs reinterpreting later.

What's easy and what's a rebuild

A report an owner might want, and what it would take:

Question	Effort
Which partners generated the most hires this quarter?	Small. The data is already linked.
Which businesses are on the most lists?	Small. One query.
Which categories have the fewest providers in a given city?	Small.
Which businesses got hired after a message exchange?	Medium. Messages and hires are both linked to a business and a list, but not to each other.

Question	Effort
Revenue per partner	Not possible today. There is no billing data in the system.

What this means for you

This is a compact, well-named schema with a written paper trail. A buyer's diligence team will find it easy to read. Most reports an owner would want are a small change, not a rebuild, because the core relationships (user, list, business, hire) are already connected. The one thing the data cannot answer is money, because the product does not bill through its own database.

What to do about it

- **Soon.** If reporting becomes a priority, start with the "small" questions above. Each is a day's work.
- **Later.** If billing is ever added, decide where payment records live before the first transaction, so attribution and revenue can be joined.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

4. Security review

This is a code-and-configuration review, not a penetration test. It does not substitute for one before a wider public launch.

The short answer

Porchlight's security fundamentals are good for its stage. The database refuses any request a user isn't entitled to, even if the app has a bug. No secrets are stored in the code. Verification codes are stored in a form that cannot be read back. The product's most valuable record, a confirmed hire, cannot be faked by the partner it credits. The two open items are decisions rather than code defects: sign-up doesn't yet verify email addresses, and nobody has tested restoring a backup.

What we found

Findings are ranked by how much they matter.

1. Open: sign-up skips email verification

By deliberate product decision, to reduce friction during the pilot, an account can be created without proving control of the email address. The consequence is that notifications can go to an address the person does not own, and account recovery is weaker than it should be. Turning verification on requires first setting up a dedicated email-sending configuration, which the team has noted as a prerequisite.

2. Open: no restore has been tested

Backups come with the database plan, but no one has documented restoring one. Until that happens, nobody knows how long a recovery would take or whether it works. See "Backups and what happens when someone leaves" below.

3. Watch: operator access is a configured allow-list

Internal tools are gated by a short list of email addresses in the deployment configuration. If the list is empty, nobody has access; there is no default admin account. This fails safe and is fine for a handful of people, but a plain list in configuration is harder to audit and rotate as the team grows.

4. Watch: no penetration test or dependency vulnerability scan has been run

This review is a structural read of access rules, secret handling, and known code paths. Document 5 covers what is outdated at the package level, which is a related but separate risk.

What's solid

Every table enforces access at the database level. Row Level Security is on every table, no exceptions found. The database itself refuses a request for data a user shouldn't see, even if a future bug in the app tried to allow it.

No secrets in the codebase. A direct search for API keys, private keys, or live credentials came back clean. Every environment file in the repository is a template with placeholders. Real secrets are supplied at deploy time through the hosting providers' own secret storage.

Verification codes are never stored readably. The one-time claim code is stored only as a one-way hash. Even direct database access could not read out a valid code. A handful of wrong guesses invalidates it.

A hire is well protected against being faked. It can only be written through one tightly controlled server-side procedure; a partner is blocked from confirming a hire on their own list; and if a hand-off link's tracking token does not cleanly match, the system records no attribution rather than guessing.

Public endpoints have basic abuse protection. The partner intake form and the public response form record a hashed visitor identifier for rate limiting, and bot protection sits in front of sign-up, login, and the intake form.

Passwords and sessions are handled by Supabase. No password or token handling code exists in the app.

Track record: past issues, caught and closed

The database history shows several past instances where a rule was more permissive than intended. The most serious: a read rule on a core table was broader than it should have been, which could have let a visitor enumerate records that were only supposed to appear on private lists. The fix narrowed the rule so a record is readable only when it is claimed, on a discoverable list, or on one of the reader's own lists. The same update fixed a printed-handout flow that could fall back to a partner's private sign-up email when no public contact was set, and consolidated all public-contact rendering into one place so that fallback cannot be reintroduced.

This is not unusual for a product with database-level rules this granular. What matters is the response: a dedicated, automated test suite now exists specifically to catch an anonymous or wrong-role client attempting something it shouldn't, and it blocks every proposed change that fails. Each time this class of bug has appeared, the team fixed the hole and added a permanent check against it recurring. Document 6 covers the suite.

Backups and what happens when someone leaves

Backups are provided by the Supabase plan. The owner should confirm which plan is in place, what the retention is, and whether point-in-time recovery is enabled. No restore has been documented.

When a developer leaves: secrets live in the hosting providers' secret storage, not on anyone's laptop, so the exposure is their login to Vercel, Supabase, and source control. Removing those three logins and rotating the service keys closes it.

What this means for you

Nothing here suggests careless engineering. The code is careful, and the database is the second line of defense a buyer's security questionnaire will ask about. The two open items are both decisions rather than bugs: turn email verification on before you open the doors wider, and prove a backup can be restored.

What to do about it

- **Now.** Confirm the Supabase backup plan and retention, and document one restore.
- **Soon.** Turn on email verification at sign-up, after setting up a dedicated email sender. Timed to the wider launch.
- **Soon.** Schedule a penetration test and a dependency vulnerability scan before a public launch.
- **Later.** If the internal team grows, move operator access from a configured list to a database role.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

5. What needs updating

The short answer

Nothing in the core stack is dangerously out of date. The web framework, the UI library, the language, and the runtime are all on current or near-current versions, and an automated tool opens a pull request every week when a package has an update. Two versions are pinned on purpose, with the reason written down. There are two small housekeeping items and one process gap: no vulnerability scan runs automatically.

What we found

"Dependencies" means every third-party package, framework, and tool the product relies on but didn't write. Each is a small ongoing obligation: it can receive security patches, breaking changes, or go unmaintained.

Why this matters when the software "works fine"

Software that works today stops working for one of three reasons that have nothing to do with your code. A provider retires an old version of an API. A security patch is published for a package you use, and until you take it, you are the one running the known hole. Or a package you depend on stops being maintained, and the next time you need to change anything nearby, you find the ground has moved. None of these announce themselves. They show up on an ordinary Tuesday as a build that will not complete or a feature that quietly stopped. Keeping dependencies current is how you make that Tuesday boring.

The core stack, as of this snapshot

Component	Version	Notes
Web framework (Next.js)	Current major	
UI library, web (React)	Current major	
UI library, mobile (React)	One version behind web	Set by the mobile framework; see below
Mobile framework (Expo)	Pinned to one SDK line	
Language (TypeScript)	Current, pinned identically across every package	Good practice
Runtime (Node.js)	Current long-term-support line	

Component	Version	Notes
Database client (Supabase)	Two different minimum versions declared	See below
Styling (Tailwind CSS)	One major behind	Low risk
Test runners	Current	

Two pins that are deliberate, not neglect

- **The database command-line tool is locked to a specific version.** The team's notes explain that newer releases changed how test credentials are issued and how a fresh local database is set up, in ways that silently broke the test suite. A considered tradeoff, but someone needs to re-evaluate it periodically. A tool frozen indefinitely eventually falls too far behind to update easily.
- **Six package versions are force-pinned** through the package manager's override mechanism. These read as security-advisory fixes: a small indirect dependency had a known vulnerability, and the team forced the whole project onto a patched version rather than waiting. A healthy, proactive pattern.

The mobile and web split

Web runs the current major version of React; mobile runs one behind, because that is what comes with the mobile framework's current release. The shared UI package supports both. This is not a mistake, but it does mean upgrading the mobile framework is a coordinated multi-package effort, not a one-line bump. The automated update tool is configured to bundle the mobile packages together for this reason.

How updates are managed

An automated dependency bot is active, with two non-default choices:

1. **Mobile packages are grouped.** They move as a set or the build breaks. The team added this grouping after six separate, un-mergeable update requests arrived at once from an earlier setup.
2. **Major-version upgrades for the load-bearing frameworks are excluded from the automatic weekly cadence.** Minor and patch updates flow automatically; a major bump is a scheduled, deliberate decision.

Every update, automated or manual, goes through staging first.

Which updates carry risk

Update	Risk	Why
Weekly minor and patch updates	Low	Automated, staged, tested
Database client version alignment	Low	Housekeeping

Update	Risk	Why
Database command-line tool	Medium	Pinned for a reason; test suite may need adjusting
Mobile framework major upgrade	Medium	Multi-package, coordinated
Web framework major upgrade	Medium	Deliberate, scheduled

What this means for you

The update machinery is in place and someone has been using it. The only real risk is the one that applies to every project: someone has to merge the weekly pull requests, and someone has to decide, about once a quarter, whether the deliberate pins are still necessary. Without a vulnerability scan, you are relying on the bot's alerts alone to learn about a known hole.

What to do about it

- **Soon.** Align the two declared versions of the database client to one. Housekeeping; do it the next time either file is touched.
- **Soon.** Add an automated vulnerability scan to the build so a known hole is reported, not discovered.
- **Later.** Re-evaluate the database command-line tool pin on a quarterly reminder.
- **Later.** Schedule the mobile framework major upgrade as its own piece of work.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

6. How well it's tested

The short answer

Porchlight has three distinct layers of automated testing, all required before any change can be merged. This is more disciplined than many products at this stage, and it directly reduces the odds of a change silently breaking something an owner would only hear about from an angry customer. The main gap is the phone app, which has no automated tests of its own. A change to the website is safe; a change to the mobile app is a gamble until that gap is closed.

What we found

Layer 1: unit tests. Does each piece of logic do what it claims?

About twenty unit test files cover the shared logic layer and specific parts of the website: phone formatting and validation; the core data-access functions for business lookups, hire confirmation, photos, and list operations; and website pieces that are easy to get subtly wrong, such as search inside a message thread, image resizing, the health-check page, and search-engine metadata. A broken unit test blocks a merge.

Layer 2: database security tests. Do the access rules hold?

A dedicated package of eight test suites runs against a fresh copy of the database, not the app. They simulate different users (an anonymous visitor, one signed-in user, a different signed-in user) attempting actions and check that each is allowed or refused exactly as intended. Suites cover list visibility, the anti-fraud rules on confirmed hires, how endorsements are counted, hand-off links, partner profile fields, business read visibility, and following a list.

This layer exists because permission mistakes have happened in this codebase's history (Document 4). It is the team's structural response, and it runs as a required, blocking check.

Layer 3: end-to-end tests. Does a real user's journey work?

About twenty browser tests simulate a person using the website: signing up, adding a provider from a photo, messaging a business, following a list, claiming a business, browsing by city and category, printing a list, going through the full partner setup flow, and more. They run against a freshly built copy of the site and a throwaway database, the closest thing to a human clicking through before every release.

How they're enforced

All three layers run as required, blocking checks on every proposed change. None can be skipped. This was not always the case: the history shows the end-to-end suite was previously allowed to fail without blocking a merge, a gap the team identified and closed.

Where a change is safe, and where it's a gamble

Area	Covered by	Confidence
Shared data-access logic	Unit + database tests	High
Database access rules	Database tests	High
Website user flows	End-to-end tests	High
Website admin flows	End-to-end tests	High
Phone app screens and flows	Nothing direct	Low
Shared UI components	Indirectly, via website tests	Medium
Performance under load	Nothing	Unknown

What's not covered

- **The phone app has no automated tests of its own.** Its logic is partly covered because it shares the well-tested data-access and helper packages, but nothing exercises its screens or flows directly. One end-to-end test checks basic mobile-web compatibility, which is not the same thing.
- **The shared UI package has no dedicated tests.** Exercised indirectly through the website's end-to-end tests.
- **No code-coverage measurement is configured.** This review is based on what the suites demonstrably check, not a percentage.
- **No load or performance testing.** All testing is functional.

What this means for you

For a pre-launch product of this size, this is a strong testing posture, particularly the database-security layer, which many teams skip until an incident forces it. In practice: a developer can change the website or the shared logic with confidence that the tests will catch a regression. A developer changing the phone app should expect to test by hand until the gap is closed, and should budget for that.

What to do about it

- **Soon.** Add automated tests for the phone app's own screens and flows, matching the discipline already applied to the website. One to two weeks.
- **Soon.** Add a code-coverage report so "how well is it tested" becomes a number the team can watch over time.
- **Later.** Add dedicated tests for the shared UI package.

- **Later.** Run a basic load test before a public launch.

Reviewed and signed off: Jordan Ellis, Lead Engineer, Cloudburst Studio · October 7, 2026

Recommended path and quote

The short answer

We recommend going straight to **Grow, at \$4,000 a month**. Porchlight is in good shape, so it doesn't need an Adoption Sprint first. What it needs is steady hands: someone merging the weekly updates, closing the two open security items, and working down tech debt, starting with the phone app, which has no automated tests of its own today.

Why Grow

The findings point to the same three needs:

- **Every change checked before you see it.** At every tier, Cloudia's automated tests check each change and we review it; you check the preview and approve. The phone app is the gap (Document 6), and Grow's steady tech debt lane is where its own tests get built.
- **Updates that get merged, not just opened.** The update bot already opens a pull request every week (Document 5). At Grow, we review and apply them, and keep security current.
- **Open items closed in order.** Email verification, a tested restore, and the vulnerability scan (Documents 4 and 5) go into the first month.

Run would work if someone on your team wants to drive every change and keep tech debt on their own list. Scale makes sense later, once several features need to move at once.

Your first 90 days at Grow

When	What happens
Month 1	Confirm account ownership and the operator list. Document and test one backup restore. Add the automated vulnerability scan. Set up a dedicated email sender and turn on email verification.
Month 2	Automated tests for the phone app's core screens and flows. Align the database client versions.
Month 3	A code-coverage report, so test coverage becomes a number you can watch.

Your own feature requests run alongside this from day one. Grow keeps two features in progress.

The three paths, compared

Path	What it takes	Price
Run it yourself	A developer who knows Next.js, React Native, and Supabase, about a day a week to merge updates and close the open items, plus manual testing of the phone app	Your developer's time
Adoption Sprint	Not needed. The software already deploys through a tested pipeline with staging.	From \$3,500 (not recommended here)
Grow (recommended)	You write tickets. Cloudia tests each change and we review it, keep it updated, and work a steady tech debt lane. Email, next business day, with a monthly check-in	\$4,000 a month, three-month minimum

The quote covers Cloudburst. Hosting, databases, and third-party services (Vercel, Supabase, and the SMS, email, and bot-protection providers) stay in your name and are billed by those providers.

Reviewed and signed off

The six documents and the recommendation above are findings. This page is what makes them a Manual: an engineer read each one against the codebase and stands behind it.

Engineer	Jordan Ellis, Lead Engineer, Cloudburst Studio
Date	October 7, 2026
Checked	14 claims against the codebase. 2 corrected before publication.
Open	1 question for the owner: backup plan and retention (Document 4)

J. Ellis

Signature on file. Review recorded in the project's findings log.