ChatGPT

Code Audit Report for PsyRehab

Executive Summary

PsyRehab is a single-page web platform built with React (TypeScript, Vite) for managing patient rehabilitation goals. It targets psychiatric social workers and patients, providing hierarchical goal tracking (6-month, monthly, weekly goals) with AI-driven goal recommendations via an N8N workflow [1] [2]. The codebase is front-end only – relying on Supabase (PostgreSQL, Auth) as the backend – following a recent removal of an Express server to simplify architecture [3].

Overall, the repository demonstrates modern development practices: a clear feature-based structure, strong emphasis on validation and security (extensive docs, use of RLS in DB, etc.), and a comprehensive toolchain (ESLint, Prettier, Vitest). Documentation is thorough, though primarily in Korean, which might hinder non-Korean contributors. Some **areas need improvement**: remnants of the deprecated backend (e.g. server-only code and docs) should be cleaned up, a hard-coded admin credential in source poses a security risk, and automated testing coverage is limited (some tests are outdated/disabled). The CI/CD pipeline builds and deploys on each push to main but doesn't currently enforce tests or lint checks, risking uncaught issues in production. Branch management appears to be trunk-based with deployments from the main branch – a simple approach that could benefit from a more robust PR review process as the team grows.

This report provides detailed findings in each area and outlines an actionable roadmap. Key immediate fixes include purging sensitive credentials, strengthening CI test gates, and updating documentation. Longer-term recommendations focus on expanding testing (including end-to-end tests), formalizing development workflows, and ensuring scalability and compliance for a production healthcare application.

1. Repository Recon

Observations: The project is a **rehabilitation goal management web app** for mental health, enabling goal setting/tracking and AI suggestions. The README (in Korean) highlights core features like a 5-step patient assessment, AI-based goal recommendation via N8N webhooks, hierarchical goal structure (6mo→monthly→weekly), real-time progress tracking, and alerts for at-risk patients [4] [5]. The tech stack is clearly documented: **React 18 + TS + Vite frontend**, Tailwind UI (including Shadcn/Radix UI components), and **Supabase** for auth/database with row-level security [6]. There is **no custom backend API** – the app calls Supabase directly and triggers an external N8N workflow for AI tasks [3]. The source is organized by domain: e.g. `src/components` subdivided into UI, auth, dashboard, goals, patients, etc., plus corresponding pages, hooks, and service modules (AuthService, PatientService, etc.) [7]. Primary feature modules include goal management, patient info, assessments, and admin utilities, each mapped to database tables as documented [8] [9].

Impact: High (for context) – The clear structure and tech choices make it easy for developers to understand the project purpose and setup. However, the README being only in Korean could impede onboarding of non-Korean speakers. All required technologies are mainstream, which is positive for maintainability. No major gaps in initial orientation, though understanding the database schema and AI

integration requires reading supplemental docs (which fortunately are provided). If the platform is intended for a wider audience or open-source, the language barrier in docs is a moderate issue.

Priority: P2 – The repository context is mostly well-documented. Improving accessibility of documentation (e.g. English summary) is a medium priority to broaden collaboration, but not critical for current functionality. The team can address this alongside other improvements.

Specific Recommendations:
- **Add an English project overview:** Provide a summary of the platform's purpose and features in English (in README or Wiki) to support international collaborators.
- **Maintain updated feature list:** Ensure README core features list stays current with the application's functionality (update any changed or new features).
- **Include architecture diagram:** Create a simple diagram of the system architecture (React app → Supabase → N8N) to give new contributors a quick high-level understanding.
- **Highlight primary modules:** In documentation, list key modules (auth, goals, assessments, etc.) and their roles, helping newcomers navigate the codebase structure.

2. Architectural Review

Observations: The application follows a **feature-oriented architecture** with good separation of concerns. Frontend code is layered into UI components, context providers, custom hooks, and a **service layer** for data operations [10]. This indicates low coupling: for example, components call service classes (which wrap Supabase queries) instead of embedding data access logic, improving maintainability. The directory structure is logically laid out (as per README's project tree) and aligns with domain boundaries (e.g., `components/patients/*` for patient-related UI, `services/goalSetting/*` for goal/assessment logic) [7]. There is evidence of conscious design decisions: use of a centralized event bus (`src/lib/eventBus.ts`) for cross-component communication, and React Context for global state (AuthContext, DashboardContext) to avoid prop drilling. The removal of the Express backend simplified deployment and reduced complexity [11], but some **artifacts of the old server architecture remain** – e.g. a Winston-based security logger module and docs about Express middleware [12] [13] that are no longer used. The **current serverless approach** scales front-end independently and leverages Supabase's real-time and security features. One consideration is that **all business logic runs client-side** (aside from what's enforced via Supabase SQL/RLS), which means things like rate limiting or complex data processing are either delegated to the client or external services. This could become a bottleneck or security concern if not carefully managed (client-side rate limit checks can be bypassed, and heavy computations could strain the browser). However, given the app's reliance on Supabase for critical operations (which can handle scaling) and N8N for AI, the architecture is cloud-native and relatively scalable for a modest user base. No major anti-patterns were observed; the use of static service classes is acceptable here, and the codebase avoids deeply nested shared states or other common pitfalls.

Impact: Medium – The architecture is well-structured, which bodes well for scalability and collaboration. The leftover server-side code/docs are mostly cosmetic issues but could cause confusion (new devs might think an Express server still exists). Client-only logic for things like security checks (e.g. login throttling) is a potential risk area: it doesn't affect normal operation but could be abused, impacting system security if an attacker bypasses the UI. As the application grows (more users or features), having no dedicated backend might limit the ability to implement certain features (e.g. complex scheduled tasks or heavy data aggregation) or integrate advanced security measures (like server-side auditing). These are not immediate problems but should be kept in mind.

Priority: P2 – The architecture is solid, requiring only moderate adjustments. Cleaning up deprecated components is a near-term maintenance task. Over the next few months, it's worth evaluating whether certain server-side capabilities (via cloud functions or Next.js API routes) are needed, but immediate overhaul is not necessary.

Specific Recommendations:
- **Purge deprecated backend code:** Remove or archive server-specific files not used in the new architecture (e.g. `lib/security-logger.js`, Express middleware docs) to avoid confusion [12] [13].
- **Client-security hardening:** For features like rate limiting or admin actions, consider implementing them via Supabase Edge Functions or cloud functions in the future, since client-side enforcement alone is not secure (Medium impact).
- **Leverage Supabase features:** Use Supabase server-side features (RPC/functions, row-level security) to enforce business rules wherever possible, reducing the load on the client and ensuring consistent rules.
- **Event flow documentation:** Document how global events and data flow (e.g. via `eventBus`, context) in an architecture README to help new developers understand the layering.
- **Scalability review:** As user count grows, periodically review if a micro-backend is needed for tasks unsuited to the client (e.g. generating PDF reports, intensive data crunching) and plan accordingly (Long-term).

3. Code Quality Analysis

Observations: Overall code quality is **high**. The code is written in TypeScript with strict typing; we see thoughtful validations and error handling. For example, the AuthService class does extensive input validation (email format, password strength, etc.) and provides user-friendly error messages in Korean [14] [15]. The logic is broken into small, focused methods – e.g. `AuthService.signIn` checks email confirmation, admin approval, profile active status in sequence, improving clarity [16] [17]. Naming conventions are clear and consistent (English identifiers for code, Korean for user-facing text). The DRY principle is mostly followed; utility functions centralize common tasks (e.g. validation helpers in `utils/auth.ts`, permission checkers, etc.). The code organization adheres to SOLID principles, especially single-responsibility: e.g. services handle data/API, components handle presentation, hooks manage stateful logic. **Notable issues:** We discovered a **hard-coded admin credential** in `AssessmentService.loginAsAdmin` – a method that programmatically signs in an admin user with email `admin@psyrehab.dev` and password `admin123!` [18]. This is a serious security concern if that account exists in any environment, as the credentials are exposed in source. It appears intended for dev convenience, but it should not be in production code. Another minor issue is the presence of many `console.log` debug statements (marked with emojis for clarity). While these are dropped in production build by Vite's config (drop_console set to true) [19], relying on logs might hide errors from users – consider using a user notification (toast) for any significant failures, and sending errors to a monitoring service. We also note that the **client-side rate limiting** implementation (using `localStorage` to count attempts) [20] [21] can be bypassed by clearing storage or using a different client, meaning it's not a true security control – this is acceptable as a user experience measure but should not be solely relied on for security (the docs indicate they had a stronger server-side mechanism before). On a positive note, no obvious injection or XSS vulnerabilities were found: all database calls use Supabase's parameterized queries, and UI uses React (which escapes content by default). We did not find any secrets (API keys, etc.) committed in the repo – environment variables are used for those. Type usage is excellent (the project even defines types for DB tables and uses Zod for validation). Unit tests mock the Supabase client to isolate logic [22], showing good testing practices (though tests coverage is limited in scope, as discussed later).

Impact: High – The hard-coded admin credentials are a **High impact security risk** – if left in a production build (even if not obviously invoked, an attacker could exploit it by calling that function via the console or code injection). This must be fixed immediately. The general code quality issues are few, so maintainability remains high. Resolving the identified issues will further strengthen security and reliability. The client-side only enforcement of certain checks is a moderate risk; in a worst-case scenario it could allow brute-force attempts or unauthorized usage if someone bypasses the UI. Addressing that would involve server-side support (either via Supabase or a custom backend). Other code quality aspects (naming, structure) are already strong and pose no negative impact.

Priority: P1 for removing sensitive credentials and any similar security flaws; P2 for other code quality improvements (e.g. removing dead code, enhancing error reporting). The critical fix should be done immediately, whereas refining logs or client-side checks can be scheduled in the normal development cycle.

Specific Recommendations:
- **Remove hard-coded passwords (P1):** Immediately eliminate the `loginAsAdmin` method or at least remove the plaintext credentials [18]. If admin login automation is needed for testing, use environment variables or a separate dev-only config not included in builds.
- **Scrub unused code paths:** Identify and delete any code that is not used in the current architecture (e.g. legacy rate-limit checks or dev test functions) to reduce maintenance overhead and avoid confusion.
- **Enhance error handling UX:** Replace or augment `console.error` logs with user-visible feedback where appropriate. For example, if a data save fails, show a toast notification. Integrate an error tracking service (Sentry or similar) to catch exceptions in production for developers.
- **Client-side security warnings:** Clearly comment or document that certain checks (like `checkRateLimit`) are for UX only. Consider implementing critical security controls on the backend/DB side (e.g., Supabase **failed login throttling** via triggers, if possible) in the longer term.
- **Continue enforcing coding standards:** Keep running lint and format tools; consider adopting a stricter ESLint config (the presence of a `LINT_IMPROVEMENT_PLAN.md` suggests this is in progress). Maintaining this discipline will prevent code quality regressions.

4. Documentation & On-Boarding

Observations: The repository contains extensive documentation, demonstrating a strong commitment to clarity and process. The main README includes setup instructions (in Korean) covering installation, environment variables, dev server usage, and even a summary of available npm scripts [23] [24]. It also provides a detailed project directory breakdown, which is very helpful for orientation [7]. Beyond the README, there are multiple markdown docs: - **DEPLOYMENT.md** – a step-by-step guide for deploying the app (Vercel/Netlify instructions, environment config, production checklist) [25] [26]. - **DBconnect.md** – a comprehensive report of database schema and how features map to tables [8] [9], as well as API architecture and AI workflow descriptions. - **SECURITY_AUDIT.md** – outlines security audit procedures and checklists [27] [28], indicating a forward-looking approach to security compliance. - **RATE_LIMITING.md** – describes how rate limiting and IP blocking was implemented when an Express server was in use [13]. (This is somewhat outdated now that the Express backend is removed.) - **CLAUDE.md** – instructions for an AI coding assistant (Anthropic Claude) on how to handle this repository [29] [30], which, while not relevant to developers, shows the team's experimentation with AI in development. Notably, what's missing is an explicit **CONTRIBUTING.md or developer onboarding guide** that covers coding style, branching model, or how to run tests. Also, because much of the documentation is in Korean, non-Korean developers might struggle despite the thoroughness of content. The API architecture described in DBconnect.md mentions a Next.js App Router backend, which conflicts with the current front-end-only implementation [31] – this suggests documentation hasn't been fully

updated to reflect the new architecture (it may have been written during planning). On-boarding a new developer would involve reading the README (which is well-structured for setup) and potentially translating parts of the docs. The domain-specific information (mental health context, terminology) is partially explained (some Korean terms, e.g. "김사회복지사" in test data, meaning "Kim Social Worker"), but an English glossary might help if the team grows internationally. In summary, documentation is **detailed but inconsistent in language and slightly outdated in places**.

Impact: Medium – The presence of rich documentation is a big positive, ensuring knowledge is shared (high security and DB detail awareness on the team). However, language and outdated info create moderate friction. A non-Korean engineer or external auditor might misinterpret or miss critical information. Outdated docs (like references to a backend that no longer exists) could lead to wasted effort or design confusion. The lack of a contributing guide means new devs might not know the expected workflow or code style expectations, though some of that is inferable from context and existing code. Improving docs will mainly impact developer efficiency and collaboration quality, rather than end-user functionality.

Priority: P2 – While the app runs fine with current docs, it's important to update and internationalize documentation in the mid-term, especially if the project is intended to be long-lived or involve new contributors. Aligning documentation with the current architecture should be done sooner than later to avoid technical debt in knowledge.

Specific Recommendations:
- **Update architecture references:** Revise documentation (DBconnect.md, RATE_LIMITING.md) to remove or clarify sections that assumed an Express/Next backend. Clearly state the current architecture (frontend-only SPA with external services) in an architecture overview section.
- **Contributing guide:** Create a `CONTRIBUTING.md` covering how to run tests, code style (e.g. using Prettier, any naming conventions), and the preferred Git workflow (feature branching, PRs, etc.). This will standardize onboarding for new developers.
- **Bilingual key docs:** Provide an English translation or summary for critical docs like the README and SECURITY_AUDIT.md. This could be as simple as an English section within the same document or a separate README_EN.md. Even if the core team is Korean, English docs will be valuable for future audits, open-source community, or non-Korean stakeholders.
- **Visual aids:** Add an architecture diagram and/or entity-relation diagram of the database to the documentation. The textual DB schema is useful [8], but a visual ERD could accelerate understanding of how data entities relate (patients, goals, assessments, etc.).
- **Ensure docs match code:** For every significant code change (e.g. removing a module or adding a feature), establish a practice of updating relevant documentation. This could be enforced by code review (e.g. "does this PR require a docs update?" checklist). Over time, this prevents divergence like the Next.js mention which is now outdated.

5. Testing & CI/CD

Observations: Automated testing is in place but could be more robust. The project uses Vitest for unit testing and React Testing Library for components [32]. We found some unit tests (e.g. for assessment history logic) that mock external modules and test service functions in isolation [22] [33] – a good practice. However, test coverage appears limited: many test files are located in an `archive_tests/` directory or marked with `.skip`, suggesting they were retired or not kept up-to-date. In fact, the package scripts indicate that end-to-end tests are **"not configured yet"** [34]. There is also an `e2e/assessment-system.spec.ts` file present, implying an intent to use Playwright or Cypress, but no integration is active. The current CI/CD pipeline (GitHub Actions) is straightforward: on push to main, it installs

dependencies, runs the build (`npm run ci`) and then deploys to Vercel [35] [36]. Crucially, `npm run ci` in package.json is mapped to just building the app (no tests) [37]. This means **CI does not fail on failing tests or lint** – in fact, tests and type checks aren't invoked in the deploy workflow, which is a gap. There is a `ci:full` script defined (running type-check, lint, test, build) but it isn't used in the pipeline. Thus, a broken unit test or a TypeScript error could slip into main unnoticed. We did note a custom script for security auditing (`security:audit`) which generates reports on dependencies and code issues [38], but it's likely run manually rather than in CI. The **continuous integration** lacks other enhancements like coverage reporting or artifact generation. On the plus side, the project uses Prettier and ESLint – although it's unclear if there's an automated check, developers can run `npm run lint` and `npm run type-check` locally or in pre-commit. **Continuous delivery** to Vercel is set up and working, deploying main to a live environment (with environment secrets configured on Vercel). No staging deployment is mentioned – presumably all merges to main are considered production-ready. In summary, while the foundation is there (Vitest, scripts, a CI pipeline), the testing practice is not fully realized (some tests outdated, no integration tests) and CI could be stricter.

Impact: Medium – The lack of enforced tests in CI means regressions or bugs might only be caught in runtime or by manual QA, which is risky especially as the app grows. For a healthcare-related app, correctness is paramount. Insufficient test coverage (especially no end-to-end tests for user flows like goal creation or assessment) leaves room for undiscovered bugs when changes are made. That said, the existing tests and type system mitigate some risk by catching obvious errors during development. The current CI/CD process could allow a faulty build to deploy if, for instance, a feature works in dev but fails in a production build scenario or if linting uncovers a security no-no. Implementing tests in CI will improve confidence in releases and prevent breaking changes from reaching users. The impact is moderate now, but will become high as the user base and contributor count increases.

Priority: P1 – Improving the CI pipeline to run tests and critical checks is a high priority quick win (it's a small config change with big benefits). Increasing test coverage is P2 since it requires more effort, but should be started soon given the domain's need for reliability. In the next two weeks, enabling existing tests to run and preventing deployment on failures is critical.

Specific Recommendations:
- **Enable tests in CI (P1):** Modify the GitHub Actions workflow to run `npm run ci:full` (or explicitly run `type-check`, `lint`, and `test`) before the build/deploy step [35]. This will fail the pipeline if any test or lint error occurs, acting as a safety net.
- **Fix/Remove broken tests:** Address the tests currently in `archive_tests` – either update them to match current code or remove them from the repository to avoid confusion. Aim to have all tests in the codebase passing.
- **Increase coverage (P2):** Gradually add unit tests for critical functionalities: e.g. goal completion logic, AI recommendation workflow (perhaps by mocking the webhook response), and role-based access behaviors. Even a basic smoke test for each page (ensuring it renders given a certain state) can catch runtime errors.
- **Implement E2E testing:** Introduce an end-to-end testing framework (Cypress or Playwright) for key user journeys (e.g. admin approving a signup, social worker creating a goal for a patient, patient marking a goal complete). This can be run in CI on pull requests. Start with a small happy-path test and expand coverage over time.
- **Continuous monitoring:** Consider adding a step in CI for accessibility tests (`npm run test:accessibility`) and security audit (`npm run security:audit`). These can run on a schedule or on new dependencies. At minimum, ensure `npm audit` runs periodically and address any new vulnerabilities.
- **CI artifacts & coverage:** Configure CI to upload test coverage reports or at least output coverage to the

console (`vitest --coverage`). This will highlight untested parts of the code. Setting a goal to raise coverage by X% over the next quarter can focus the team's testing efforts.

6. Dependency & Environment Management

Observations: The project's dependencies are clearly listed in **package.json**, and they align with the stated tech stack. Notably, it uses React `^19.1.0` and React DOM `^19.1.0` (which suggests the project stays up-to-date with React's latest version) [39] . The use of many @radix-ui libraries and other UI packages (lucide icons, Tailwind merge) indicates a modern UI approach. State management is handled via @tanstack/react-query (v5) and React Context, avoiding heavier solutions like Redux. The Supabase JS client is at v2.49, which is recent. Dev dependencies (Vite 6, TypeScript ~5.8, ESLint 9, Vitest 3, etc.) are also quite current, implying active maintenance of the toolchain. The team has a script for checking dependency vulnerabilities (`npm audit` and a custom `security:audit` script) – the latest security report shows **1 low-severity vulnerability** in an indirect package (a regex DOS in brace-expansion) and no high/critical issues [38] . This proactive auditing is commendable. **Environment management:** The app requires a `.env` file with Supabase URL/anon key and N8N webhook URL; these are clearly documented [24] . Because there's no server component, environment setup is straightforward (no DB migrations in the repo, since Supabase is managed elsewhere). The Vercel deployment uses environment variables for secrets, which is standard [26] . We did not find a Dockerfile – the app likely doesn't need one for deployment (since Vercel or Netlify handle it) and for local development, running `npm dev` suffices. Node version isn't pinned in package.json (no `engines` field) or .nvmrc, but the CI uses Node 18 which is appropriate [40] . One observation: a `VITE_API_URL` env variable is mentioned in README but currently the app doesn't use a custom API (it proxies `/api` to localhost:3001 in dev just in case) [41] . This could be a leftover from when an Express API was optional. It might confuse new devs who wonder if they need to run an API server. Additionally, having 750+ total dependencies (incl. dev) [42] is normal for a React project of this scope, but it warrants attention to keep them updated. There is no mention of a lockfile (`package-lock.json`) handling or any issues there, presumably it's committed and used to ensure consistency.

Impact: Low/Medium – At present there are no glaring dependency issues. A low-severity vulnerability exists, which should be patched but poses minimal immediate risk. Outdated dependencies can become a problem if left too long (security and compatibility), but the team seems to be monitoring them. The environment setup is simple, which lowers onboarding time. The absence of containerization is fine given the target deployment is static hosting; if a dev environment consistency issue arises, Docker could be considered, but it's not urgent. The only risk is if someone misses setting env vars – but the docs cover that well. If the project scales, they should continue to monitor Supabase library updates (for example, ensuring compatibility if Supabase updates their API) and any changes needed for React (React 19 usage suggests maybe using an experimental version – ensure that's stable or ready for production).

Priority: P3 – Generally maintain the status quo with regular maintenance. Address the known minor vulnerability via `npm audit fix` or library upgrade (could likely be resolved by updating ESlint or related packages). It's good to periodically check for major version updates (for example, React Query v5 is used; if v6 comes out, plan to evaluate it). These tasks are ongoing but not urgent.

Specific Recommendations:
- **Patch vulnerabilities:** Run `npm audit fix` to resolve the **brace-expansion** low-risk vulnerability [38] if possible (likely by bumping a transitive dependency). Keep running these audits monthly.
- **Maintain updates:** At least every minor release cycle, update core dependencies (React, Vite, Supabase, Tailwind etc.) after testing. Don't fall too many versions behind, especially for security patches on any server-facing library (though most dependencies here are front-end, thus lower security impact).

- **Remove unused env vars:** If `VITE_API_URL` is no longer used, remove it from documentation and config to avoid confusion. Similarly, purge any config or code paths expecting a backend URL.
- **Pin Node version:** Consider adding an `"engines": { "node": "18.x" }` in package.json or a .nvmrc file. This ensures all developers and CI use a consistent Node version, reducing "it works on my machine" issues.
- **Consider Docker for dev (optional):** For an easier onboarding, you could provide a Dockerfile that builds and serves the app (using `npm run preview`) for those who prefer containerized dev environments. This is low priority as the setup is currently straightforward.
- **Supabase environment sync:** Document or script how to initialize a development Supabase instance (SQL schema, seed data). This could be in the Wiki or a SQL dump in the repo if appropriate (with dummy data). It will help new devs or testers quickly get a working backend to fully run the app locally.

7. Commit & Branch Hygiene

Observations: The project appears to follow a **trunk-based development** model with the `main` branch as the deployment branch. Every push to main triggers a production deployment on Vercel [36]. This implies that feature work may either be done on short-lived feature branches that get merged into main via PR, or possibly commits are made directly to main for smaller changes. We did not find a documented branching strategy or evidence of multiple persistent branches (no dev/staging branch references). The commit history itself was not directly accessible via this audit, but secondary indicators suggest that commit messages might not follow a strict convention (since no mention of conventional commits or changelog generation was found). The presence of security audit and lint plans suggests an intention to maintain quality, which often correlates with meaningful commit messages (e.g. referencing issues or features). However, without an explicit guideline, commit messages could be inconsistent (e.g. mix of Korean and English, or varying levels of detail). The PR process isn't documented; it's unclear if code reviews are enforced. Given that CI doesn't run tests on PRs (only on push to main), it's likely the team is small and collaborating informally. This can risk changes being merged without thorough peer review or passing tests, especially if pushing directly. On the positive side, keeping everything in main means no long-lived branches diverging – the codebase stays unified and avoids complex merges. But as the project grows or if multiple developers work concurrently, lack of a structured workflow could cause issues (e.g. incomplete features accidentally deployed, difficulty tracking what's deployed).

Impact: Medium – Currently, with presumably a small team, the simplicity works, but it doesn't scale well. Poor commit messages make it harder to generate release notes or trace when a bug was introduced. Lack of a PR review process could allow mistakes to slip in unnoticed. If an urgent fix is needed, having all changes on main can complicate rolling back individual features (as opposed to feature flags or separate release branches). Adopting some best practices in Git hygiene will improve collaboration, code quality (through reviews), and traceability. This area doesn't directly impact end-users, but it greatly affects developer efficiency and project transparency.

Priority: P2 – It's important to introduce some process before the team or complexity grows, but the project can function in the short term as is. Improvements here should be part of the next sprint or two.

Specific Recommendations:
- **Adopt a commit message convention:** Encourage using Conventional Commits (e.g. `feat: description`, `fix: description`, etc.) or at least consistently structured messages. This will make it easier to auto-generate changelogs and understand history at a glance.
- **Enforce pull requests for main:** Turn on branch protection for `main` – require pull request merges and at least one code review approval before changes go live. This ensures every change is reviewed by another developer, catching issues early.

- **Use feature branches:** For substantial features or fixes, create a branch off main, work there, and open a PR. Merge into main only when the feature is tested and ready. This dovetails with CI improvements – tests can run on PRs to verify the branch before merge.
- **Tag releases or use releases page:** Consider tagging versions (even if just v1.0, v1.1, etc.) when deploying major changes. This helps track what version is in production and facilitates rollback if needed via git tags.
- **Improve commit frequency and scope:** Each commit should ideally address one issue or feature. Avoid large "batch" commits that mix unrelated changes. This will make debugging via git bisect easier. If not already practiced, try to commit more atomically with clear rationale in the message.
- **Housekeeping of branches:** Regularly delete merged branches on GitHub to keep the repository tidy. Also, document the branching strategy in the contributing guide (e.g. "All work happens on main and feature branches; main is always deployable").

Quick-Win Roadmap (Next 2 Weeks)

1. **Remove Hard-Coded Credentials (P1):** Delete or secure the admin auto-login code in `AssessmentService` [18] and rotate any exposed test password. This closes an immediate security hole.
2. **Enforce Tests in CI (P1):** Update GitHub Actions to run lint and Vitest on each push/PR, preventing deployments with breaking issues. Ensure the pipeline uses the full CI script instead of just build [35].
3. **Clean Up Legacy Code/Docs (P2):** Purge unused backend-related files and update docs to reflect the frontend-only architecture (remove outdated Express references [13]). This reduces confusion going forward.
4. **Patch Dependencies (P2):** Run `npm audit fix` to resolve the low-severity vuln in **brace-expansion** and commit the updated lockfile [38]. Verify that all packages (especially Radix UI and React 19) are stable, updating any minor versions as needed.
5. **Add Contributing Guidelines (P2):** Create a basic `CONTRIBUTING.md` covering commit conventions, branching, coding style, and how to run tests. This quick document will align new contributions with project standards.

Long-Term Roadmap (3-6 Months)

- **Expand Test Coverage:** Aim for >80% unit test coverage. Add tests for critical flows (goal completion logic, permission enforcement, etc.). Implement end-to-end tests (Cypress/Playwright) for multi-step user interactions across roles.
- **Security & Compliance Enhancements:** Implement multi-factor authentication or audit logging if required by patient data regulations. Perform a full security audit (as per SECURITY_AUDIT.md checklist) and address any gaps (e.g. enforce stronger password policies, add automatic session timeout).
- **Performance and UX Improvements:** Monitor and optimize performance – e.g. use Lighthouse to ensure the app is fast. Consider adding caching or lazy-loading for large data sets. Continually refine the AI integration for responsiveness (maybe using background processing or user notifications while AI generates recommendations).
- **Architecture Scaling Decisions:** Evaluate if certain features would benefit from an API server or serverless functions (for example, complex report generation, or integrating third-party services beyond Supabase/n8n). If yes, design a lightweight backend component or use Next.js API routes and adjust architecture accordingly.

- **Documentation & Localization:** Keep all documentation updated and consider translating the user-facing UI to English (or other languages) if the platform will be used in broader contexts. Likewise, an English version of key developer docs should be maintained.
- **CI/CD Enhancements:** Automate more in the CI pipeline – e.g. scheduled security scans, daily dependency update checks, and automated deployment to a **staging** environment for testing. Incorporate code quality tools (ESLint rules, Prettier check) into the CI to maintain code consistency.
- **Community and Workflow:** If the project becomes open-source or gains more contributors, adopt community best practices – issue templates, PR templates, a code of conduct, and maybe switch to a GitFlow or GitHub Flow model with feature branches and review workflows to manage the increased collaboration.

---

1 8 9 31 DBconnect.md
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/DBconnect.md

2 4 5 6 7 23 24 README.md
https://github.com/Cooledricesh/Psyrehab/blob/main/README.md

3 11 25 26 35 36 40 DEPLOYMENT.md
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/DEPLOYMENT.md

10 29 30 CLAUDE.md
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/CLAUDE.md

12 security-logger.js
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/lib/security-logger.js

13 RATE_LIMITING.md
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/docs/RATE_LIMITING.md

14 16 17 auth.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/src/services/auth.ts

15 20 21 auth.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/src/utils/auth.ts

18 assessmentService.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/src/services/goalSetting/assessmentService.ts

19 41 vite.config.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/vite.config.ts

22 33 assessment-history.test.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/__tests__/assessment-history.test.ts

27 28 SECURITY_AUDIT.md
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/docs/SECURITY_AUDIT.md

32 vitest.config.ts
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/vitest.config.ts

34 37 39 package.json
https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/package.json

38 42 security-audit-2025-07-06.json

https://github.com/Cooledricesh/Psyrehab/blob/b1a6144118e0c3b2e954dfeebdb0238a5d2a911b/security-reports/security-audit-2025-07-06.json