ChatGPT

Project Improvement Roadmap (Monorepo Migration & Code Cleanup)

This roadmap outlines a step-by-step plan to reorganize and improve the project's codebase using a **monorepo** structure, enforce consistent code quality (linting/formatting), introduce automated testing, and set up continuous integration. The focus is on **code cleanup and structure** (with testing and CI included), while deferring production deployment steps for later consideration. Following these steps will make the codebase easier to maintain and scale, with unified dependencies and tooling across client, server, and shared modules.

Step 1: Complete Monorepo Migration (Structure Consolidation)

First, finalize the migration from the current `client/server/shared` layout into a proper **monorepo** structure. We will use an `apps/` and `packages/` folder convention:

- **Create Monorepo Folders:** Inside the repository root, create an `apps` directory for applications and a `packages` directory for shared libraries/modules [1]. For example, move the front-end code into `apps/client`, the back-end code into `apps/server`, and any common code into `packages/shared` (or multiple packages as needed). Each of these subfolders will have its own `package.json` (defining name, dependencies, etc.), making them individual workspace packages. This structure allows clear separation of concerns while keeping everything in one repo for easier code sharing and consistency [2].

- **Configure Workspaces in Root:** Update the root `package.json` to declare the workspaces, e.g. add:

```
"workspaces": ["apps/*", "packages/*"]
```

This tells the package manager (npm or Yarn) to treat each folder in `apps` and `packages` as a workspace project [1]. Ensure each workspace has a **unique name** in its `package.json` (for example, `"name": "client"` and `"name": "server"` for the apps, `"name": "shared"` for the library) to avoid conflicts. All dependencies across the repo will now be managed together at the root.

- **Single Top-Level Lockfile:** With npm workspaces, there should be **only one** lockfile in the monorepo (e.g. a single `package-lock.json` at the root) and no individual lockfiles in sub-projects [3]. Remove any `package-lock.json` or `yarn.lock` files from the `client`, `server`, or `shared` subdirectories if they exist. Then run a fresh install (e.g. `npm install` at the root) to generate a new unified lockfile [4]. This ensures all packages resolve their dependencies coherently and avoids mismatches.

- **Adjust Import Paths and References:** After moving files, update import statements throughout the codebase to reflect the new paths. For example, if the client was importing something from `../shared` or a relative path, you may establish a proper import via the package name. A good practice is to give the shared package a scoped name (e.g. `"@yourproj/shared"`) and add it as a dependency to the apps. With workspaces, the client and server can import `@yourproj/shared` as if it were an installed package, but it actually links to your local `packages/shared` code. This might require updating `tsconfig.json` paths if using TypeScript, so that the shared code is included and recognized in both client and server builds. Ensure that each app can compile and run after these path adjustments.

- **Preserve Scripts:** If your client or server had npm scripts (for starting, building, etc.), check that they are now in their respective sub-project `package.json`. You can still run them from the root using workspace commands (more on this in the testing step). For instance, `npm run dev --workspace=client` could start the client dev server (or use the `-w` flag with Yarn/PNPM accordingly).

Ghostwriter Prompt: *"Restructure the repository into an apps/packages monorepo layout. Create an* `apps` *directory with* `client` *and* `server` *subfolders for the front-end and back-end, and a* `packages` *directory (e.g.* `packages/shared`*) for common code. Move all existing client code into* `apps/client` *and server code into* `apps/server`*. Create a* `packages/shared` *for shared modules and move common code there. Update the root* `package.json` *to include* `"workspaces": ["apps/*", "packages/*"]` *and remove any duplicate lockfiles in sub-projects so that only the root has a* `package-lock.json`*. Ensure each sub-project has a unique package name and update all import paths in the code to use the new package structure (for example, import shared code via a package name or correct relative path)."*

> **Rationale:** Adopting a monorepo with defined workspaces makes it *easier to share code and maintain consistency* across client and server. All projects live in one repository, benefiting from **unified dependency management and CI pipelines** [2]. The `apps/*` vs `packages/*` pattern is a common convention to separate runnable applications from libraries [1]. By consolidating to one lockfile and workspace, we avoid dependency divergence and ensure installations are in sync [3].

Step 2: Set Up Code Quality Tools (ESLint & Prettier)

Next, improve code consistency by introducing **linting** and **formatting** across the monorepo. This will enforce a common code style and catch errors or bad practices automatically.

- **Install ESLint and Prettier:** Use your package manager to install ESLint and Prettier as devDependencies at the root (so they can be used in all workspaces). Also include relevant plugins/presets for your tech stack: for example, `eslint-plugin-react` (for React JSX), `@typescript-eslint/parser` and plugin (for TypeScript support), `eslint-plugin-node` (for Node.js rules) or other needed ones. Similarly, add `eslint-config-prettier` and `eslint-plugin-prettier` to integrate Prettier with ESLint. *(If Ghostwriter cannot run the install itself, run the appropriate* `npm install -D ...` *commands manually, then proceed.)*

- **Create a Shared ESLint Configuration:** To avoid maintaining separate lint rules in each package, define a **central ESLint config** that all parts of the monorepo can use. For example, create a root ESLint config file (e.g. `.eslintrc.json` or `.eslintrc.cjs` in the repo root) that extends recommended rule sets and is applied to all files. In this config, you can use ESLint's `overrides` section to handle differences between front-end and back-end code if necessary (for instance, enabling browser globals for client code vs. Node globals for server code). Ensure it covers:

- **Base Rules:** Start from `eslint:recommended` and the TypeScript recommended rules (`plugin:@typescript-eslint/recommended`). Include React-specific rules (`plugin:react/recommended` and JSX accessibility rules if using React) for the client code, and Node-specific settings for server (you might use `plugin:n/recommended` for Node).
- **Parser Settings:** Use `@typescript-eslint/parser` for TypeScript parsing. Include `sourceType: module`, and set project tsconfig paths if needed for type-aware linting.
- **Prettier Integration:** Extend or include `prettier` at the end of the extends array to turn off formatting rules that might conflict, and add `"plugin:prettier/recommended"` or use `eslint-plugin-prettier` to report formatting issues.
- **Environment Settings:** In the config, specify environment globals appropriately (e.g., `"browser": true` for client files, `"node": true` for server files, via overrides on directories or file patterns).
- **Ignore Patterns:** Add common ignore patterns (like `node_modules/`, `dist/` or build output, perhaps `.turbo/` or similar if using such tools, etc.) so ESLint doesn't waste time on those.

By centralizing the ESLint config, all packages will share the same base rules, ensuring consistency. This approach is better than copying separate config files into each project, which would be harder to maintain [5]. (In larger projects, one can even create a dedicated `packages/eslint-config` and have each workspace extend it [6], but for now a single config file with overrides is sufficient.)

- **Add Prettier Configuration:** Create a `.prettierrc` file at the root (or a `prettier.config.js`) to define formatting options (even if minimal, e.g. 2-space indent, single vs double quotes, etc., or you can largely rely on defaults). This ensures Prettier's rules are explicit. Also consider an `.prettierignore` to exclude certain files (if any, like build output or generated files) from formatting.

- **Update Package Scripts:** In the root `package.json`, add convenient scripts to run linting and formatting. For example:

- `"lint": "eslint . --ext .js,.jsx,.ts,.tsx"` to lint all files in the repo (or you can target specific folders if needed).

- `"format": "prettier --write ."` to format the whole codebase. You might also add these scripts in individual packages if you prefer to lint each separately, but a root-level command that covers all is easiest. Optionally, use the `--workspaces` flag with npm scripts to run lint in each workspace (if you have separate lint scripts per package). However, a single unified lint command at root using a shared config should work and simplify the process.

- **Run Lint and Fix Issues:** After setting up, run `npm run lint` (or the equivalent) to see any lint errors in the codebase. Resolve these errors either manually or by using `eslint --fix` for auto-

fixable issues. Likewise, run Prettier to auto-format the code (`npm run format`). This step may catch unused variables, undefined values, or stylistic issues. Fixing these will immediately **clean up the code** (e.g. removing dead code, standardizing quotes/semi-colons, etc.). The goal is to have zero ESLint errors and have all files formatted consistently.

Ghostwriter Prompt: *"Set up ESLint and Prettier for the entire monorepo. Install ESLint and Prettier (with TypeScript, React, and Node plugins/configs) as dev dependencies at the root. Create a root ESLint configuration file that extends recommended rules for React, Node, and TypeScript, and integrates Prettier (use eslint-config-prettier and eslint-plugin-prettier). Make sure to include appropriate environments (browser for client, node for server) and ignore* `node_modules` *and build folders. Also add a Prettier config file with our formatting preferences. Update the package.json scripts to add 'lint' (running eslint on all apps and packages) and 'format' (running prettier). Then fix any linter errors and format the codebase accordingly to enforce a consistent style across client, server, and shared code."*

> **Rationale:** Introducing ESLint and Prettier will enforce a unified coding style and best practices across the monorepo. Rather than maintaining separate configs in each project (which is hard to keep consistent), a shared setup reduces duplication and ensures everyone follows the same rules [5]. This step will catch bugs (like undefined variables, unused code) and apply consistent formatting, significantly improving code quality and readability.

Step 3: Implement Testing Framework (Vitest for Unit Tests)

With the structure and linting in place, the next step is to add **automated tests** to increase confidence in the code. We will set up a unit testing framework across the monorepo. Based on the tech stack, **Vitest** is a great choice (it's fast and works well with Vite + TypeScript + React, and can also test Node code). Alternatively, Jest could be used (especially for the Node server), but to keep things simple and consistent, we can use Vitest for both front-end and back-end tests.

- **Install Testing Libraries:** In the client app, install Vitest and any testing utilities needed (for example, `@testing-library/react` and `jsdom` if you plan to test React components that interact with the DOM). In the server app, you can also install Vitest (and perhaps any libraries for testing server logic, e.g. supertest if testing HTTP endpoints). Because we're in a monorepo, you might choose to install Vitest at the root and use it for all workspaces. However, it's often cleaner to add as a devDependency in each project that has tests (client and server), so that each can run tests independently. (If using npm workspaces, the Vitest binary will be hoisted to the root `node_modules` anyway, so one global install could suffice.)

- **Configure Vitest:** Create a `vitest.config.ts` (or `vite.config.ts` if using Vite which can include test config) in each project if needed. For the client (React) app, you'll likely configure Vitest with `globals: true` and `environment: "jsdom"` [7] so that it can simulate a browser-like environment for component testing. For the server app, you can use `environment: "node"` (the default) for a Node.js environment. If the projects are simple, you might not need a custom config at all, as Vitest can work with zero config by default. Just ensure it can find the test files (by default Vitest looks for files ending in `.test.ts` or `.spec.ts`, etc., in the project). Organize your tests alongside code or in a dedicated `__tests__` folder as you prefer.

- **Write Sample Tests:** Add at least one basic test in each major part of the project to verify the setup. For instance, in the shared package, you could test a utility function. In the client, test a React component (e.g., that a component renders with given props, using React Testing Library). In the server, test a simple module or an API endpoint response (possibly using an HTTP request simulation). These initial tests help confirm that the testing framework is working in both environments. For example, create `packages/shared/math.test.ts` to test a math utility, or `apps/client/src/App.test.tsx` to test a component rendering. Keep them simple (assert true is true) just to get the pipeline green initially.

- **Update Package Scripts for Testing:** In each relevant `package.json` (client, server, shared if applicable), add a script like `"test": "vitest run"` (or simply `"vitest"` which by default runs tests non-interactively). Also, add a **root-level** script to run all tests across workspaces. With npm, you can leverage the workspaces feature to run tests in all projects with one command. For example, adding in root `package.json`:

  ```
  "scripts": {
    "test": "npm run test --workspaces"
  }
  ```

  This will run the `test` script in each workspace sequentially [8]. Ensure the workspace names are unique and that each has a test script, so that this command succeeds for all. Alternatively, use a tool like Turborepo or Nx to run tests in parallel, but initially the npm workspaces approach is sufficient.

- **Run Tests:** Execute the test script (e.g. `npm run test` at root). All tests in all packages should run. Initially, some may fail (especially the sample tests if they were just placeholders). Make sure to see the test runner output to confirm that it's picking up tests from both `apps/client` and `apps/server`. Once confirmed, you can replace or expand the sample tests with real tests for your core functionality. The objective here isn't to write a full test suite immediately, but to **integrate the testing framework** so that any new bugs can be caught by adding tests going forward.

Ghostwriter Prompt: *"Integrate a unit testing framework (Vitest) into the monorepo for both client and server. Install Vitest as a devDependency in both* `apps/client` *and* `apps/server` *(include* `@testing-library/react` *and* `jsdom` *for client tests). Configure Vitest: for the client, use a jsdom environment (simulate browser for React), and for the server, use the node environment. Add a basic* `vitest.config.ts` *to each if needed (or a single root config with* `defineWorkspace` *if using Vitest's workspace mode). Create sample test files in each project (e.g., a React component test in the client and a utility or API test in the server) to verify the setup. Update the package.json scripts: add "test" scripts in client and server that run Vitest. Also, add a root "test" script that runs all workspace tests (e.g. using npm workspaces or a tool like Turborepo). Ensure that running* `npm run test` *from the root executes tests in all packages. Finally, run the tests to confirm everything passes."*

> **Rationale:** Setting up automated tests is crucial for catching regressions. Using **Vitest** provides a unified testing solution that works for both front-end and back-end in a monorepo. We leverage npm workspaces to run tests across all projects with a single command [8], which simplifies continuous integration. With the testing framework in place,

developers can write new tests when fixing bugs or adding features, leading to more robust code over time. (Vitest is chosen for its speed and Vite integration, but the setup would be similar if opting for Jest – the key is that both client and server are covered by tests.)

Step 4: Configure Continuous Integration (CI Pipeline)

Finally, establish a basic **CI pipeline** so that all these quality checks (linting and tests) run automatically on each push or pull request. This will ensure that the codebase remains healthy and that no broken code gets merged without detection.

- **Choose a CI Service:** If your project is hosted on GitHub, **GitHub Actions** is a convenient choice. For GitLab, GitLab CI would be analogous, or CircleCI/Travis, etc., depending on your repository. Here, we'll assume GitHub for an example. Replit itself doesn't run CI for you, but you can integrate with GitHub to trigger Actions when you push changes.

- **Set Up Node.js Workflow:** Create a workflow file (e.g. `.github/workflows/ci.yml` in the repository) that runs on your main branch pushes and pull requests. Use a Node.js setup since this is a JavaScript/TypeScript project. A typical GitHub Actions workflow will:

- Use the latest stable Node (e.g. `node-version: 18` or whichever version your project requires).
- Checkout the repository code.
- Install dependencies (run `npm install` at the root – this will install all workspace deps).
- Run the lint script (`npm run lint`) and run the test script (`npm run test`). You can have separate steps for these, so it's clear if linting fails or tests fail.
- (Optional) If you have build scripts or type-check scripts, you can run those too in CI to ensure everything compiles.
- You might also enable caching for `~/.npm` to speed up installs in CI (GitHub Actions has actions/setup-node that support dependency caching by default).

Make sure the workflow is triggered on the appropriate events (`on: [push, pull_request]` typically). This way, every time code is pushed or a PR is opened, the CI will automatically verify code quality and tests.

- **Verify CI Locally (if possible):** Before committing the workflow, you can do a dry run by running the same steps locally: i.e., do a fresh clone, run install, then `npm run lint && npm run test` to see that everything passes. This is essentially what the CI will do. Resolve any issues now (for example, if tests assume a local dev database, you might need to mock those or skip in CI, etc.). The goal is that the project should pass all checks on a clean environment.

- **Monitor the Pipeline:** After pushing the CI config, check the first run in the CI provider's interface. Ensure that the jobs for linting and testing are succeeding. If something fails (maybe a test is flaky or an ESLint rule is too strict and was missed locally), fix the issue and push an update. Once it's green, you have a working safety net: any future code changes that break lint or tests will cause the CI to fail, alerting you to address the problems before merging.

Ghostwriter Prompt: *"Create a GitHub Actions workflow file to run our linting and tests on every push. In a new file* `.github/workflows/ci.yml`*, define a workflow that triggers on push and pull_request events. Use the*

latest Node.js (for example, Node 18) on an Ubuntu runner. Steps should include: checkout the code, set up Node (with cache enabled for npm), then run `npm install` *at the repository root (to install all workspace dependencies). Next, add a step to run* `npm run lint` *and another to run* `npm run test`*. Ensure the workflow fails if either lint or tests fail. This will automate our code quality checks on each commit."*

> **Rationale:** A Continuous Integration pipeline is essential for a professional development workflow. By running linting and tests on every push, we prevent bad code from slipping in. With a monorepo, this is especially useful because the CI can test all packages together, ensuring that changes in one part do not break others [2]. GitHub Actions provides a simple way to set this up with our Node.js project. After this step, our project will have automated enforcement of code quality — any lint errors or failing tests will be caught early by the CI.

Future Consideration: Deployment Workflow (Post-Cleanup)

(The focus for now is on code cleanup, so the deployment setup is optional at this stage. Replit's environment might not directly support custom Docker deployments, but once the code is clean and tested, we can plan for deployment.)

After the above steps, the codebase will be well-structured and stable. The next potential improvement would be to set up a **production deployment pipeline**. This could include:

- **Containerization:** Writing a `Dockerfile` for the server (and possibly one for the client if using a static build) to containerize the application for deployment. This would let you run the app in any environment (e.g., using Docker or deploying to services like AWS, GCP, etc.). If using a PaaS like Railway, you'd configure it to run your server start command. *Note:* Replit itself doesn't require Docker – it runs projects in its own containers – and it may not support building custom Docker images to deploy. So this step is more relevant if moving to a different host (or using Replit's native deployment features).

- **Environment Configuration:** Preparing configuration for production (like `.env` files or secrets for API keys in CI/deployment settings), ensuring the build process can produce optimized output (e.g., using `npm run build` for the client to produce static files to serve, etc.).

- **Automated Deployment:** Extending CI to CD (Continuous Deployment), such as adding a workflow job to deploy to a service when changes are merged into a main branch. For example, deploying the client to a static hosting (Netlify/Vercel) and the server to a cloud host or container registry. This can be integrated later once the application is ready to go live.

We will **defer these deployment steps** for now, as the immediate goal is to refactor and improve the code quality. Once the monorepo refactor, linting, and tests are in place (and bugs are resolved via those tests), we can confidently proceed to set up a production-ready deployment pipeline.

By following this roadmap step-by-step, we will transition the project to a cleaner, more maintainable state: - A unified monorepo structure for all components of the project (making inter-module collaboration easier and more coherent), - Automated linting/formatting to enforce code standards and simplify reviews, - A

testing framework in place to catch issues early and facilitate bug fixing with confidence, - And a CI system to continuously guard quality.

Each step is designed to be executed with the assistance of Replit's Ghostwriter (or any AI pair-programming tool) by providing the prompts above. This will incrementally apply the improvements. After these enhancements, the project will be in a strong position to tackle specific bugs or new features, with a robust foundation in place.

Sources:

1. Pixelmatters – *Monorepo Setup with apps/ and packages/* [1] [3]
2. Gregory Gerard – *ESLint in a Monorepo (shared config benefits)* [5]
3. Pixelmatters – *Running tests in multiple workspaces (npm --workspaces)* [8]
4. Gregory Gerard – *Monorepo advantages (code sharing, unified CI/CD)* [2]

---

[1] [3] [4] [6] [8] How to manage multiple Front-End projects with a monorepo
https://www.pixelmatters.com/blog/how-to-manage-multiple-front-end-projects-with-a-monorepo

[2] [5] ESLint in a Monorepo
https://gregory-gerard.dev/articles/eslint-in-a-monorepo

[7] Vitest Monorepo Setup
https://www.thecandidstartup.org/2024/08/19/vitest-monorepo-setup.html