ChatGPT

Proposed Refactor Plan for Expensly Codebase

Core Terminology and Types

- **Unify and Standardize Data Models:** Consolidate all transaction-related types into a single shared module (e.g. a `types.ts` or `models` directory). Currently, the app defines `Transaction` differently in multiple places – for example, one interface uses `payee` with a list of `postings` [1], while another uses `narration` with separate `debitAccounts`/`creditAccounts` lists [2]. These should be merged into one canonical **LedgerTransaction** type that uses consistent Ledger terminology. Use `narration` (or "description") for the transaction's descriptive text (instead of swapping between `payee` vs `narration` in different contexts) [1] [2], and use a **Posting** type for each entry line (each posting having an `account`, `amount`, and optional `currency` [3]). This eliminates confusion and bugs from mismatched types.

- **Use True Ledger Terms:** Enforce strict Ledger vocabulary across the code. For instance, call the transaction description field "**narration**" everywhere (Ledger/PlainTextAccounting often refers to the payee/description as narration) to avoid ambiguity. Likewise, refer to individual debit/credit lines as **postings** rather than generic "accounts." In practice, this means removing the old `debitAccounts` and `creditAccounts` arrays from the data model (which were just lists of account names) in favor of a single `postings: Posting[]` array. If the UI needs to display or filter by "debit accounts" vs "credit accounts," it can derive those on the fly by checking the sign of each posting's amount, rather than storing redundant fields. This keeps terminology precise – e.g. a posting with a negative amount can be considered a credit posting, positive as debit – and prevents mix-ups that arose from the former approach.

- **Consolidate File and Tag Types:** Similarly, standardize the **LedgerFile** type and any other shared entities. Currently `LedgerFile` is duplicated with slight inconsistencies (e.g. defined separately in `Transactions.tsx` and `LedgerManager.tsx` with minor differences in fields) [4] [5]. Create one module export for `LedgerFile` (including all needed fields like `id, name, content, is_primary, created_at, last_updated_at`), and import that wherever needed. This ensures that when the database schema changes or new fields are added, you update in one place. Moreover, include the **tags** field in the unified `LedgerTransaction` type (the code already plans for a `tags: string[]` field [2], though it's not fully used yet). By having tags as a first-class property of transactions, the system is ready for a future tagging feature without requiring a new type definition. In summary: one source of truth for all core types (transactions, postings, files, etc.), using proper Ledger terms, will eliminate the "scattered and broken" type definitions and make the codebase easier to maintain.

- **Fix Generated Types and Mismatches:** Update or replace the auto-generated Supabase types to align with reality. For example, the `Database` type for the `ledger_files` table is outdated – it omits fields like `name` and `is_primary` that do exist and are used in code [6]. To prevent such mismatches, consider manually extending or overriding these types in the shared types module. The unified `LedgerFile` interface can serve as the app's authoritative definition, and the Supabase client can be used with a corrected type. This way, type-checking will catch schema inconsistencies early. Overall, having a coherent types module (possibly grouped by domain: e.g.

`types/ledger.ts` defining `LedgerTransaction`, `Posting`, `LedgerFile`, etc.) will enforce consistency in how data is referenced throughout the app.

Backend Integration Abstraction

- **Decouple Data Access from UI:** Introduce a clear abstraction layer for all backend operations so that the front-end React components are backend-agnostic. Currently, components call the Supabase client directly (e.g. fetching files in the `Transactions` page with `supabase.from("ledger_files").select(*)` inline [7]). This tight coupling makes it hard to switch backends or modify data handling. Instead, implement a **Ledger Service** or repository module (e.g. `ledgerService.ts`) responsible for all data retrieval and persistence. This service would provide methods like `getLedgerFiles()`, `getTransactions(fileId)`, `saveLedgerFile(file)` etc., hiding whether the data comes from Supabase, another API, or local storage. The service can use Supabase under the hood for now, but by funneling all calls through it, you can later swap in a different backend (or an offline mode) by only changing the service implementation, not the entire app.

- **Abstract Supabase-specifics:** Move Supabase-specific logic and initialization out of the core app logic into the integration layer. For example, keep using the Supabase JS client (already isolated in `integrations/supabase/client.ts`), but ensure the rest of the app interacts with it through well-defined interfaces. That means pages like `LedgerManager` and `Transactions` should call something like `LedgerService.listFiles()` instead of directly writing queries. The service can return plain JavaScript objects (`LedgerFile` and `LedgerTransaction` types) to the UI. This not only makes the code cleaner, but also eases unit testing (you could mock the service) and future changes. If later the app uses a different database or a custom backend, only `LedgerService` would need updating, keeping the UI layer intact.

- **Prepare for Alternative Backends:** Design the abstraction with future expansion in mind. For example, define an interface for the data store (with methods for reading/writing transactions) and implement one for Supabase. In the future, if moving to a dedicated server or another BaaS, you can create a new implementation of that interface. Also consider using environment configuration to switch backends easily. The goal is that nothing in the components or business logic is tied to Supabase-specific concepts (like `from(...).select()` or realtime subscriptions, etc.); those are confined to the integration layer. By making the backend pluggable, you also facilitate using ledger-cli on the server side: the front end might call a cloud function to get data, and the service can handle that call. In summary, **the UI should only know about a generic data provider**, not the specifics of Supabase – this modularity will pay off if you ever migrate data or support multiple storage options.

- **Centralize Business Logic:** Along with data fetching, move any business rules or transformations into the service layer as well. For instance, if after fetching transactions from the DB we need to massage them (compute derived fields, sort or filter), do that in the service or a utility, not scattered in the UI. The current code does some processing (e.g. sorting transactions by date in the component after parsing [8]). That sorting (and similar logic like identifying the primary file) could be handled in the data layer so that components simply receive ready-to-display data. A centralized backend integration module ensures consistent behavior across different UI parts and reduces duplication. This also means if you modify how transactions are structured or need to enforce some rule (say, all transactions must be balanced), you do it in one place. This abstraction will make the system more robust and easier to extend (for example, if

you add a feature to archive or sync data, it can hook into the service layer without touching UI code).

UI Layer Simplification

- **Use Unified Data Models in UI:** Refactor the frontend components to rely on the new shared types and structures, eliminating the ad-hoc data transformations currently in place. For example, the transactions page now converts parsed data into a separate "table data" format with split debit/credit arrays [9] [10]. This extra step can be removed by having the UI work directly with a `LedgerTransaction` that contains all postings. The UI can decide how to display those postings (e.g. group by positive/negative) without needing to maintain two parallel representations. By passing around a single `LedgerTransaction` shape (with `narration`, `postings`, etc.), we avoid the risk of one view getting out of sync with another. In practice, this means functions like `transformToTableData` can be removed or greatly simplified – the grid component can derive its needed fields from the base data on the fly (as it already started doing with a `splitPostings` helper in the table renderer) [11].

- **Simplify Transaction Display Logic:** Leverage the power of the UI framework (React + grid library) to compute presentation details, rather than storing them in state. For instance, instead of storing separate `debitAccounts` and `creditAccounts` lists in each transaction object, derive them when rendering: e.g. *Debit Accounts* column can filter `transaction.postings` for positive amounts and display those [12], and *Credit Accounts* column does the same for negative amounts. The current approach in `<TransactionsTable>` is half-way there – it uses a `valueGetter` to derive debit accounts from `postings` on the fly [13], but still relies on a pre-filled `creditAccounts` field for credits [14]. We should eliminate this inconsistency by always using `postings` as the source. A single source of truth (the postings array) rendered in different ways means less state to maintain and fewer chances for bugs. For card view vs table view, we can use the same data: the card list can map through `transaction.postings` to show each line, while the table can aggregate them in two columns – both are just different projections of the same `LedgerTransaction` object [15] [16].

- **Consistent Naming and Filtering:** Update the UI labels and filter logic to match the standardized terminology. For example, the table header currently labels the description column "Narration" [17], which should correspond to the `narration` property in our unified type (after we rename from `payee`). Ensure components like the filter builder use the same property names – the filter engine should filter on `transaction.narration` (after we rename it) and on postings/accounts consistently. If we remove `debitAccounts`/`creditAccounts` fields from the data model, the filter UI can be adjusted to allow filtering by account name within postings (e.g. a filter condition "Account contains X" that checks all postings). In the short term, we might keep the existing filter fields but have them operate on `postings` under the hood. In any case, aligning the filter logic with the new types is crucial so we don't have broken filters (right now, there's a potential mismatch where the filter expects `transaction.narration` but parsed data had `payee` [18] [2]). After refactor, everything will uniformly use `narration`, so such issues go away.

- **Clean Up Amount Handling:** Clarify how transaction amounts are determined and presented, and make the UI logic for it straightforward. Currently, the app computes a single `amount` per transaction (used for sorting and summary) by taking the first positive posting's value [19]. This works for simple 2-posting transactions (e.g. expense vs cash) but might be confusing for multi-posting splits. We should define `LedgerTransaction.amount` more explicitly – for example, it

could represent the total outflow of the transaction (a positive number for expense transactions, negative for income transactions). Document this and adjust the UI accordingly. If we continue using the convention that expenses are positive amounts and incomes negative (as seen in sample data where salary payment is stored as -3500 [20]), ensure the formatting and color coding aligns with it. Right now, the table's amount cell renderer just displays the number with currency [21], and the card view prefixes a "+" or "–" sign in front of the formatted amount [22]. We should unify these to avoid user confusion – e.g., decide that *outflows* show with a "–" minus sign (and perhaps red color) since they reduce your balance, whereas *inflows* show "+" and green. If that's the desired convention, we may need to invert how the sign is stored or at least how the color is applied (since currently positive values were being shown in green as "+", which might actually be treating expenses as positive). In short, define the rule clearly (perhaps standard accounting: credits negative, debits positive) and implement one consistent approach in all components. This will make the UI more intuitive and set the stage for future features like budgeting, where the sign matters for calculations.

- **Lean UI Components:** With the heavy lifting moved to the backend and service layer, the UI components can become lean and focused purely on presentation and user interaction. For example, the `Transactions` page component should no longer need to contain parsing logic or complex state management beyond what view to show [23]. It can simply call `LedgerService.getTransactions(selectedFile)` on load or file switch, then render `<TransactionsTable>` or the card list with the data. Consider using React hooks or context for things like the currently selected ledger file or loaded transactions, so that deep components (like a transaction list item) can access them without prop-drilling. We already have contexts for currency and filters; we could add a context for ledger data or simply use the service in components directly. By simplifying components, we reduce the chance of bugs and make the UI code easier to adjust. For instance, adding a **new feature** like tagging UI becomes easier when each transaction already carries a `tags` array – we'd just have to render those tags (which the table and card are already set up to do in a basic way) [24], rather than refactor the data flow. Overall, a thinner UI layer means improved maintainability and flexibility for future changes.

Ledger Import/Export via CLI

- **Replace Custom Parser with Ledger CLI:** Remove the in-browser JavaScript parser (`ledgerParser.js`) and shift to using **Ledger CLI** (or a similar robust parsing engine) for reading and writing ledger data. The custom parser, while a good initial effort, only handles basic ledger syntax and can easily break or fall behind actual ledger features [25] [26]. Instead, leverage the proven Ledger CLI itself to parse the ledger files. For example, set up a backend process (such as a Supabase Edge Function, AWS Lambda, or a small Node/Rust service) that, given the ledger text content, calls Ledger (or an API like Paisa) to parse it into JSON or another structured form. Ledger CLI can output data in various formats; we can use commands like `ledger -f file.journal xml/json` or similar to get machine-readable output, or use a library that interfaces with Ledger's parser. By doing this, we ensure **full compatibility with ledger's format** – all valid ledger entries (commodities, balance assertions, metadata, etc.) will be handled, not just the limited subset our JS regex covers. This reduces tech debt since we're not maintaining a parallel parser that could misinterpret complex cases.

- **Use Paisa/Plain Text Accounting Tools:** The refactor should take inspiration from how Paisa (an open-source Ledger GUI) manages data – Paisa essentially uses ledger/hledger under the hood and maintains a database of transactions for fast queries [27]. We can adopt a similar approach: after using ledger CLI to parse the file, **load the transactions into our database** (e.g. into a

`transactions` table, and a related `postings` table). The application can then query this structured data for display and filtering, rather than re-parsing the text each time. This gives us the best of both worlds: the ledger text remains the single source-of-truth that can be imported/exported, but day-to-day operations can use the speed of a database. When it's time to save or export, we can also call out to ledger/hledger libraries to **write back a properly formatted ledger file**. For instance, if a user adds a new transaction via the UI, we would insert it into the `transactions` table and then regenerate the ledger file text (ensuring the formatting matches ledger conventions, like date formats and two-space indentations for postings). This could be done by our own templating or by using ledger CLI's print command to format one transaction and appending it – either way, using ledger's rules guarantees consistency.

- **Parsing as a Service:** Implement a dedicated function or microservice for parsing ledger files. The workflow could be: the client uploads or edits a ledger file's content, then invokes the parse service (perhaps automatically on save). The service runs ledger/hledger to parse the text. We then update the database with the parsed results (within a transaction to keep it consistent). The front-end can be notified (or can poll) to retrieve the structured data. By offloading parsing to the back end, we not only get accuracy but can also handle large files without bogging down the browser. This is important if the user's ledger grows to thousands of transactions – the CLI in a proper environment (or a compiled library) will be faster and more memory-efficient than a browser JS approach. Additionally, the CLI can catch errors in the ledger file format that our current parser might miss (for example, unbalanced transactions, syntax errors) and we can report those back to the user in a meaningful way (today the app might just fail silently or log a console warning [28]). In summary, **treat ledger CLI as the authority** for parsing/validating and integrate it into the app's backend flow.

- **Seamless Export and Sync:** Ensure that any changes in the structured data can be written back to the ledger text file consistently. The `LedgerManager` currently allows editing the raw text in a Monaco editor and saving it [29]. After this refactor, we should augment that flow: when the user hits save on the editor, run the backend parser to update the DB (so the structured data stays in sync with whatever they typed), or conversely, consider disabling direct text editing once a structured approach is in place (to prevent divergence). A better approach might be a hybrid: allow expert users to edit text, but on save, parse it and update the database (and if the text has errors, show an error instead of saving). Likewise, when a user adds or edits transactions via a form UI (a feature we likely want for ease-of-use), we should update the DB and then regenerate the text file from it. We could maintain the ledger file content in the database as well (still in the `ledger_files` table) as the canonical text, but it should always be the output of the last successful parse/generation, not a separately edited thing. Using ledger CLI to format output will help here – for instance, if we store transactions in the DB, we can script an output in ledger format (date, narration, postings, comments) and perhaps even use ledger CLI's own formatting capabilities to ensure it's ledger-compliant. The key is **round-trip fidelity**: a ledger file imported, then exported, should remain semantically the same (maybe with minor formatting differences at most). This approach eliminates the custom parser's maintenance burden and aligns the app with the ledger ecosystem standards.

- **Consider Hledger/Beancount for Flexibility:** (Optional) While ledger-cli is the primary target, note that tools like **hledger** or **beancount** offer similar plaintext accounting with possibly easier libraries for parsing. Paisa, for example, is compatible with ledger, hledger, and beancount [30]. Designing our import/export system with a clear interface (e.g. a function `parseLedgerFile(content) -> LedgerTransaction[]`) means we could swap out the backend engine if needed (say, use a beancount parser library in JS, or a Rust crate for ledger). The refactor should thus focus on *capabilities* (parse and format ledger data) rather than a

specific binary. In practice we might start by calling ledger's binary via a server, but as the project grows, we could move to an embedded library for performance. By keeping this layer modular, we maintain the option to adapt without rewriting the app. This also ties into future features: for example, ledger/hledger can generate balance reports, budget reports, etc. We could call those CLI commands on the backend to implement budgeting features (instead of writing that logic from scratch). So, integrating the CLI not only handles parsing now, but opens up possibilities to reuse powerful accounting queries down the road.

Folder Structure and Modularity

- **Organize by Feature/Domain:** Refactor the project's folders to group related code into coherent modules, improving clarity and maintainability. For instance, separate the **ledger file management** functionality from the **transactions display** functionality. We might create a structure like: `src/domain/ledger/` for core ledger logic (parsing, types, service) and `src/features/transactions/` for UI components and hooks related to transaction list/filtering, etc. Components that belong together (like `TransactionsTable`, `FilterBuilder`, and `Transactions` page) should reside in a common folder or hierarchy. This makes it easier for new contributors (or your future self) to find all parts of a feature in one place. Right now, some logic is in `src/lib/`, some in `src/pages/`, some in `src/components/...`, which can be fine, but we can improve it by clearly delineating what is UI, what is data logic, and what is shared utilities. For example, after refactor, the `src/lib/ledgerParser.js` file will be removed (since we'll use the CLI), and any remaining utility like `splitPostings` can move into a more appropriately named module (perhaps `src/domain/ledger/utils.ts` or integrated into the service if only used there). Strive for a clean **folder structure** such as:

```
src/
  types/        (all shared type definitions)
  services/     (data access and backend integration code)
  components/   (UI components, possibly subdivided by feature)
  pages/        (Next.js or routing pages that compose components)
```

This modular separation means each concern can be developed and tested in isolation.

- **Shared Types and Utilities Module:** Given the importance of the unified types, consider creating a dedicated directory or file (e.g. `src/types/ledger.ts`) that exports the core interfaces (Transaction, Posting, etc.) and perhaps some constants or helper functions related to ledger data. This could even be a small NPM package or a separate module if you plan to share types between front-end and back-end code (for example, if you write cloud functions in TypeScript, they can import this shared types module to ensure they produce the same shape of data that the front-end expects). By centralizing domain knowledge (like what a transaction consists of) in one place, you reduce the likelihood of divergent logic. We saw the consequences of divergence in the current code: e.g., the ledger file schema in the Supabase types vs the UI interface got out of sync [6] [5]. A single source for types prevents that. Similarly, if there are formatting utilities (like date formatting for ledger, or currency symbols), those can live in a shared util file (some already exist like `CurrencyContext.formatCurrency` and `utils.ts` for classnames).

- **Isolate Integration Code:** Keep all external integration code (Supabase client setup, API calls, etc.) in its own area (the project already has `src/integrations/supabase/` – that pattern

can be expanded). If we add, say, a `ledger-cli` integration, we might have a module like `src/integrations/ledgerCli.ts` which knows how to call the CLI or API. The service layer would call that integration. By isolating these, if the external interfaces change (Supabase SDK updates, or we swap to a different ledger parsing method), we only touch the code in one place. This also aligns with the abstraction principle described earlier. Essentially, the **folder layout should reflect the separation of concerns**: UI vs domain vs external. Each layer gets its own place in the tree.

- **Remove Dead or Redundant Code:** As part of the cleanup, identify and eliminate any obsolete files. For example, once the new parser is in use, `ledgerParser.js` can be deleted. The same goes for any placeholder or sample data that is no longer needed (the `sampleTransactions.ts` used for prototyping could be removed or moved to a tests folder if still useful for testing) [31]. Also, any duplicate type/interface definitions scattered in components should be removed in favor of importing from the shared types. After refactoring, a search for keywords like "interface Transaction" in the repo should ideally show just one definition. Reducing clutter will make the repository cleaner. It might also be a good time to update documentation (the README or a docs folder) to explain the new architecture (e.g. "Expensly now uses a server-side ledger parser and a normalized database for transactions…"), so that the rationale and structure are clear for collaborators.

- **Modular Design for Future Extensions:** The new structure should make it easy to add features like **tag management** or **budgeting** without entangling with existing code. For example, you might create `src/features/tags/` for tag-specific UI or logic (though tagging might be simple enough to not need its own module, depending on complexity). For budgeting, you could have `src/features/budget/` with components and services to handle budgets. Because the core ledger data will be well-structured (transactions, postings, etc.), these new modules can hook in by querying that data (e.g. summing expenses per tag or account). The folder structure should thus allow new feature folders to plug in rather than forcing everything into one large folder. Planning this now means the project can grow in a clean, maintainable way, with each feature's code largely self-contained and using shared services/types.

Migration Steps from Old Parser

1. **Unify Models and Terminology:** Begin by refactoring the code to use the new consolidated types and naming, before changing how data is processed. Introduce the shared `LedgerTransaction` and related interfaces, and replace all legacy interface usages with these. For example, update the `Transactions` page and others to use `transaction.narration` instead of `transaction.payee` in their code and JSX [1] [32], and use `transaction.postings` wherever postings were assumed. This step also involves renaming state variables, props, and context values to the consistent terms (e.g., if a state was `transactions: Transaction[]` with the old type, ensure it now refers to the new unified type). Because this is a big cross-cutting change, it's wise to do it incrementally: you might introduce the new types alongside the old for a short period, run tests or verify that the app still compiles and works, then remove the old definitions. After this step, the code should be internally consistent in what we call things (narration, postings, etc.), reducing confusion as we proceed.

2. **Implement Database Schema for Structured Data:** Set up new tables in the database to hold structured transactions and postings (if not already present). For example, create a `transactions` table with columns like `id, file_id (references ledger_files),`

`date, narration, amount, tags…` and a `postings` table with `id, transaction_id, account, amount, currency`. If using Postgres (Supabase), consider using array or JSON types for postings/tags as an alternative, but separate tables with relations are more normalized and flexible for querying. Once the schema is ready, perform a one-time migration of existing data: for each ledger file in `ledger_files`, run a complete parse (using either the old parser for initial migration, or ideally the new CLI parser if ready) to extract all transactions and insert them into the new tables. This migration can be done with a script or a backend function – essentially reading `ledger_files.content` and producing structured rows. Verify after this step that the number of transactions in the table matches expectations and spot-check a few to ensure dates, amounts, and accounts were captured correctly. We will keep the original `ledger_files` table for the actual text, but now we have an authoritative list of transactions in the DB as well.

3. **Develop the Ledger CLI Parsing Service:** Create the backend mechanism to call ledger CLI or equivalent. In a Supabase environment, this could be an Edge Function written in TypeScript or Python that uses a ledger parser library. Alternatively, set up a small Node.js (or Deno) service that the app can call via HTTP. The implementation details might involve installing the `ledger` binary or using a library – for now, get a basic version working (for example, an endpoint that accepts ledger file text and returns a JSON of transactions). Test this service with known ledger samples to ensure it correctly handles things (you can compare its JSON output to what the current JS parser outputs for simple cases, and also test cases the JS parser couldn't handle, like complex postings or comments). Once confident, integrate this service into the app's data layer. That means adjusting the `LedgerService`: instead of using `parseMultipleEntries` in the client [8], call the backend parse function. For instance, `LedgerService.getTransactions(fileId)` might now fetch the file content from DB (or the file path), send it to the parse service, and get back structured data (which we can then also store in the `transactions` table). Initially, you might run the new parser in parallel with the old one in development to cross-verify results before fully switching over.

4. **Switch Frontend to Use Structured Data:** Modify the frontend logic to retrieve transactions from the new structured source. After parsing and storing transactions in the DB (step 3), the front end no longer needs to do heavy parsing or even have the full ledger text in memory. Update `LedgerService.getTransactions(fileId)` to simply query the `transactions` table (and maybe join or nested-select `postings`) for that file's data. Supabase can return JSON of postings easily if using a foreign table or an array aggregate. Thus, when the `Transactions` page loads or a file is selected, the sequence will be: ensure the parse service has run (perhaps on file upload or save), then do a cheap query like `SELECT * FROM transactions WHERE file_id = X` (with postings). This will return already parsed transactions which you set in state. At this point, you should remove calls to the old `parseMultipleEntries` entirely from the front end [23]. The UI should treat the data as coming from the service/DB, not by interpreting raw text. This is a big switchover, so do thorough testing: confirm that the list of transactions shown in the UI matches what was shown before for a given ledger file. Because our new parser is more accurate, you might find some subtle differences (which is good to catch now). From here on, the source of truth for transaction data in the app is the database tables.

5. **Sync Text Editing with DB:** With the front end now showing DB-backed data, we need to handle the case where the user edits the ledger file text manually in the editor (if that feature remains). Implement a hook in the save flow: when the user clicks "Save" in `LedgerManager` [29], after updating the `ledger_files.content` in Supabase, immediately invoke the ledger parse function for that file and update the structured tables. This could be done via a database trigger

or just by calling the parse service from the front-end right after the save succeeds. The result would update the `transactions` table with any new or changed transactions. Then, have the front-end refresh the displayed transactions list (the `Transactions` page) – if using a context or global store for transactions, update it; if not, maybe the easier route is to listen for changes via Supabase's subscription or simply refetch when the file's `last_updated_at` changes. The goal is real-time consistency between the text and what's shown in the UI. Conversely, if you introduce a form-based way to add or edit transactions (which would be a great improvement for usability), do the reverse: when the user adds a transaction in the UI, create a new record in the `transactions/postings` tables, then regenerate the ledger text for that file (either immediately or marking that the file needs regeneration). That text regen can use a template or even a simple approach of inserting the new transaction line(s) into the right spot in the file content. Ensure to update the `ledger_files.content` and perhaps mark `last_updated_at`. By the end of this step, whether changes come from text input or UI input, the two representations (text and DB) remain in sync. This effectively replaces the old approach where the front-end parser tried to keep up – now the **source of truth is the combination of the DB and ledger file, with the CLI ensuring they stay consistent**.

6. **Refactor Filter and UI Logic:** Now that the data model is consistent and coming from a single source, adjust any remaining front-end logic that was built around the old parser's outputs. For instance, the filter functionality (in `filterEngine.ts`) can be refactored to work directly on `LedgerTransaction` objects (with postings and tags). We might enhance it to filter by postings: e.g. a condition for "account contains X" would check `transaction.postings` array for any posting whose account name matches. In the short term, you can simplify filters by only supporting text search on the combined narration or account fields if implementing full account-based filtering is complex. The important part is to ensure that `filterTransactions()` uses the correct fields (after the rename to narration, etc.) so it doesn't silently fail. Also update the UI components like `<TransactionsTable>` to rely solely on the unified data. For example, remove the now-unneeded `transformToTableData` function and instead pass the raw transactions to the table component. Inside the table, use the `splitPostings` helper (or inline logic) to display debit/credit breakdown [13] [12]. Confirm that sorting by date or amount in the table still works – it will, if we pass those fields properly. Since we have a centralized `formatAmount` and color logic, ensure the table's custom cell renderers use them consistently (you might move those renderers or the formatting logic into a shared util so the card view and table use the exact same formatting code). After this, the UI should be cleaner: essentially just mapping over transactions from state and rendering them, with all heavy parsing/filtering done either in the DB or via simple array methods.

7. **Test Thoroughly:** At this stage, conduct comprehensive testing with real ledger data. Use a variety of ledger file scenarios: multiple transactions, multi-posting transactions (splits), transactions with comments and metadata, different currencies, etc. Verify that importing a file yields the correct number of transactions displayed and that all postings line up. Test editing the text in the editor – introduce an intentional format error to see if the backend parser catches it and that the UI displays an error (you might need to add error handling: e.g., if the parse service returns an error, show a toast or message in LedgerManager). Test adding a transaction via the UI (if implemented) and see that it appears in the text file and vice versa. Also test the filter functionality to make sure it still filters the displayed list correctly using the new data shape. Performance-wise, ensure that for larger files the experience is smooth (the initial parse might be a bit slower if calling an external service, but it should be one-time per load). Any discrepancies found during testing should be addressed now – for example, if you notice that the "amount" column is showing positive for what should be income, you might tweak the logic

to flip the sign or adjust color coding. The aim is that the refactored app behaves equal or better compared to the pre-refactor version, with no regression in features.

8. **Cleanup and Remove Legacy Code:** Once everything is working with the new system, safely remove the old code that is no longer needed. This includes the custom `ledgerParser.js` module entirely (we won't use it anymore), any references to `parseEntry` / `parseMultipleEntries` in the codebase, the `sampleTransactions.ts` (if it was only for demo and no longer used), and duplicated type/interface definitions that we unified. Also remove any temporary logic that was used to bridge between old and new (for instance, if at one point you kept both `payee` and `narration`, now you can drop `payee` completely). Essentially, **eliminate all the tech debt we identified** – the code should no longer contain misleading terms or structures. After deletion, refactor the project structure if not done already (organize files into the new folders as discussed). Finally, update documentation: the README or an architecture doc should describe that the app uses a backend ledger parser and a database of transactions. This will help future maintenance. With the legacy cruft gone, the codebase will be cleaner, more logical, and far easier to extend.

By following this refactor plan, Expensly will have a clean, maintainable foundation. The core data structures will be consistent and true to accounting terminology, preventing bugs caused by misnamed fields or duplicate definitions. The backend abstraction will allow the project to scale or switch technologies with minimal pain. The UI will be leaner and focused on displaying data, not fixing it up, which makes adding features like **tag filtering or budgeting** much simpler (since the data is readily available in structured form). And crucially, leveraging ledger's own CLI for parsing and output means the app stays aligned with the broader plain-text accounting ecosystem – ensuring reliability and easing future expansions (like generating reports, handling multiple currencies, or sharing data with other ledger tools). This comprehensive cleanup sets the stage for implementing tagging, budgeting, and other advanced features in a straightforward way, without accumulating further technical debt. The end result will be a more professional, robust Expensly app that is easier to maintain and extend.

Sources:

- Code for custom Ledger parser and its output structure [33] [26]
- Examples of inconsistent type definitions for Transaction and LedgerFile [34] [2] [5]
- UI logic splitting postings into debit/credit accounts (to be refactored) [9] [12]
- Direct Supabase usage in components (to be abstracted) [7]

---

[1] [4] [7] [8] [9] [10] [15] [18] [19] [22] [23] [32] [34] Transactions.tsx
https://github.com/arpit-982/expensly-app/blob/3902f158c566cae4a61b7819740450fc1ca5e3f9/src/pages/Transactions.tsx

[2] filterEngine.ts
https://github.com/arpit-982/expensly-app/blob/3902f158c566cae4a61b7819740450fc1ca5e3f9/src/lib/filterEngine.ts

[3] [25] [26] [28] [33] ledgerParser.js
https://github.com/arpit-982/expensly-app/blob/3902f158c566cae4a61b7819740450fc1ca5e3f9/src/lib/ledgerParser.js

[5] [29] LedgerManager.tsx
https://github.com/arpit-982/expensly-app/blob/3902f158c566cae4a61b7819740450fc1ca5e3f9/src/pages/LedgerManager.tsx

[6] types.ts
https://github.com/arpit-982/expensly-app/blob/607551ffaa99f03dfedbb50a03ec203d5c736e2c/src/lib/types.ts

11 12 utils.ts
https://github.com/arpit-982/expensly-app/blob/607551ffaa99f03dfedbb50a03ec203d5c736e2c/src/lib/utils.ts

13 14 16 17 21 24 TransactionsTable.tsx
https://github.com/arpit-982/expensly-app/blob/607551ffaa99f03dfedbb50a03ec203d5c736e2c/src/components/transactions/TransactionsTable.tsx

20 31 sampleTransactions.ts
https://github.com/arpit-982/expensly-app/blob/3902f158c566cae4a61b7819740450fc1ca5e3f9/src/data/sampleTransactions.ts

27 Packages - ledger - NixOS Search
https://search.nixos.org/packages?channel=unstable&show=ledger-live-desktop&size=50&sort=relevance&type=packages&query=ledger

30 Ledger CLI - beancount - Paisa.fyi
https://paisa.fyi/reference/ledger-cli/