**Prompt — Finance Reconciliation Agent (Frontend + Backend)**

Use this prompt to instruct a software-generation platform, an LLM-based code assistant, or a developer team to build a **full deployable solution** for a Company Finance Reconciliation tool. The prompt is written as a single, self-contained specification that describes requirements, inputs/outputs, UX, architecture, modular structure, test data, and deployment. Paste the entire block below to the assistant or platform.

You are an expert full-stack developer. Build a **full, production-ready, deployable** Company Finance Reconciliation application that matches the specification below. Deliver complete code, tests, configuration, and deployment artifacts (Dockerfile, docker-compose, README, and simple CI config). The solution must be modular: each logical function in its own file/module and CSS separated from HTML/JS. Use the **LangChain framework** to clearly separate backend and frontend processing/agents. Use modern, maintainable stack choices described below.

**High-level objective**

A web front-end to ingest **multiple bank accounts** and **multiple transaction CSVs per account (or bank-download by credentials)**, automatically classify transactions as **internal transfers**, **external credit/debit**, or **need verification**, present the combined reconciled table with trace links, allow manual confirmation of doubtful rows, and provide a single downloadable Excel with all banks' transactions and classifications. Provide an Excel download button in the top-right of the results table.

**Technology & architecture requirements**

1. **Frontend**
   * React (single file preview optional) or plain HTML+JS if requested; use a single-file React component as default.
   * Styling with **CSS in a separate file** (no inline styles). Follow a professional, accessible palette. Use Tailwind classes if using React/Tailwind; otherwise create a styles.css.
   * Result table must support sorting, filtering, pagination, and row colour highlight for “need verification”.
   * Show a download Excel button in the top-right of the results area.
   * Use clear icons and colors for status: internal transfer (green), external (blue), need verification (orange/red).
2. **Backend**
   * Python 3.10+.
   * Use **LangChain** to define backend "agents" or pipeline for: parsing CSVs, matching transactions, reconciliation rules, and generating Excel. Frontend/backed separation must be explicit: e.g., a LangChain agent handles reconciliation logic invoked by an API endpoint.
   * Expose a REST API (FastAPI recommended) to handle file uploads, bank credential secure fetch (mocked), reconciliation run, and download.
   * Import and use an **existing authentication module** (the user-provided Python module). Accept path string for the module (example import: from app.auth import auth\_manager) — assume it exposes login(request), current\_user(request), and verify\_token(token) functions. Use that module for all endpoints.
3. **Modularity**
   * Each major function in its own module/file: CSV parsing, normalization, matching engine, reconciliation rules, Excel generation, API routes, frontend build, CSS, and LangChain config.
   * Provide unit tests for reconciliation logic.
4. **Data handling & security**
   * Validate and sanitize CSV contents.
   * If bank-credential downloading is implemented, **mock** it by default: provide a factory function that can be swapped for real bank connectors. Never hardcode credentials. Provide secure storage instructions (env vars/secret store).
   * Use HTTPS-ready settings in config; provide instructions for environment variables (SECRET\_KEY, DB URI if needed).
5. **Deployment**
   * Provide Dockerfile and docker-compose.yml that runs the API and a static/frontend service. Provide a short README.md with deployment steps to Docker and Heroku (or similar).
6. **Deliverable**
   * Fully runnable repository with app/ folder, tests, Dockerfile, docker-compose.yml, and README.md.
   * Sample CSVs and expected output Excel (or generator script).

**Functional requirements (detailed)**

**1) Authentication**

* Use the **existing Python authentication module** provided by the client. Example usage in backend routes:
* from app.auth import auth\_manager
* @router.post("/upload")
* async def upload(files: List[UploadFile], token: str = Depends(auth\_manager.verify\_token)):
* user = auth\_manager.current\_user(token)
* ...
* All endpoints must require authentication.

**2) Upload & Input**

* Frontend must allow:
  + Adding multiple bank accounts (account name, bank name, account number/IBAN, currency).
  + For each account, upload **one or more CSV files** (many banks split downloads) OR choose “Download from bank” (this calls a mocked bank-download endpoint that returns CSV content).
* Accepted CSV fields (support flexible naming; map columns): date, value\_date, description, amount, debit\_credit (or positive/negative), transaction\_id, counterparty\_account, memo. Implement a column-mapper UI if incoming headers differ.

**3) CSV parsing and normalization**

* Parse CSVs tolerant to:
  + Different date formats (e.g. DD/MM/YYYY, YYYY-MM-DD).
  + Amounts with commas and currency symbols.
  + Negative amounts and separate debit/credit columns.
* Normalize to canonical model:
* {
* "account\_id": "<internal account id>",
* "bank\_name": "...",
* "account\_number": "...",
* "transaction\_id": "...",
* "date": ISO8601 date,
* "amount": decimal.Decimal,
* "direction": "credit"|"debit",
* "description": "...",
* "counterparty\_account": "...", # if present
* "source\_file": "bankname\_2025-09-01.csv"
* }

**4) Matching algorithm — internal transfers**

* Algorithm requirements:
  + Identify **pairs** (or groups) of transactions across accounts that represent internal transfers: one account has a **debit** and another a **credit** for (nearly) the same amount.
  + Matching rules:
    - **Exact amount match** preferred.
    - Allow **tolerance** on amount (configurable, default 0.50) and **date delta** (configurable, default 3 days). The transfer may appear on different dates (today vs next banking day).
    - Prefer matches where descriptions or counterparty account info matches heuristically.
    - Allow multi-hop matches (e.g., transfer splitted to multiple incoming credits). Score candidate matches based on amount diff, date delta, description similarity. If cumulative credits equal a single debit within tolerance and date window, group as one transfer.
  + Mark matched pairs with a trace\_link referencing the matched transaction IDs (one or more).
  + When matched, classification = internal\_transfer.
* Performance: algorithm should scale to thousands of transactions; implement indexes (in-memory dictionaries keyed by amount/date) to speed matching.

**5) External transactions (incoming/outgoing)**

* Any credit that **cannot** be matched to a debit in another loaded bank account (following the matching rules) is classified as **external\_incoming**.
* Any debit that **cannot** be matched to a credit in another loaded account is **external\_outgoing**.

**6) Doubtful transactions**

* Any match with **low confidence** (score below a threshold, or ambiguous multi-candidate matches) should be classified as **need\_verification** (aka "doubtful").
* Provide a mechanism to **manually confirm** doubtful rows in the UI — a confirm/unconfirm toggle in the table.
* UI should **highlight** doubtful rows with a distinct color and an icon.

**7) Result table layout & UX**

* Display a consolidated, sortable table with columns:
  + Bank (bank name)
  + Account (display last 4 digits)
  + Transaction Date
  + Transaction ID
  + Description
  + Amount (with debit/credit shown)
  + Direction (credit/debit)
  + Classification (internal\_transfer / external\_incoming / external\_outgoing / need\_verification)
  + Trace Links (transaction IDs of matched row(s), clickable to open a modal showing the matched transaction(s))
  + Confirm (checkbox or action button to confirm a previously doubtful classification)
* Top-right: **Download Excel** button that generates a single Excel containing all transactions and classification columns.
* Enable filtering by classification, account, bank, and date-range.
* Clicking a transaction row should open a details modal showing the source CSV line, raw data, and matched transaction(s).

**8) Export**

* Excel export must:
  + Contain all transactions across all accounts in one sheet (or multiple sheets where sheet per account is optional), with columns above plus the raw parsed fields and matching\_score.
  + Be downloadable via a link/button in the top-right of the table.
  + Use a visually clear header style and conditional formatting for need\_verification rows (e.g., orange fill).
  + Provide both .xlsx and .csv export options. Prefer .xlsx for complex formatting.

**9) Colors & Professional UI**

* Use a restrained professional palette. Examples:
  + internal\_transfer: green (for success)
  + external\_\* : blue (for normal)
  + need\_verification: orange/red (attention)
* Use CSS variables or Tailwind tokens so colors are centralized and can be tweaked easily.

**10) LangChain usage & separation**

* Use LangChain agents/pipelines as the backend orchestration layer:
  + **Parser Agent**: reads uploaded CSV bytes, maps headers to canonical fields, and returns normalized transactions.
  + **Matching Agent**: takes normalized transactions, runs matching and classification, returns classification, trace links, and scores.
  + **Export Agent**: builds Excel and returns a download URL or stream.
* The frontend calls the FastAPI endpoints which orchestrate LangChain agents; ensure logic is not implemented in frontend (no classification in frontend).
* Provide a langchain\_config.py that wires the agents and shows how to swap components (e.g., replace mock bank downloader with a real connector).

**11) Modular file structure (suggestion)**

/app

/api

routes.py

auth\_routes.py

/auth

\_\_init\_\_.py # existing module integrated (example import: app.auth.auth\_manager)

/core

parser.py # CSV parsing + normalization

matcher.py # matching algorithms

exporter.py # Excel generation

bank\_connector.py # mock bank download factory

config.py

/langchain

agents.py # langchain agent definitions

chains.py

/models

schemas.py # Pydantic models for request/response

/frontend

index.html

app.jsx or app.js

styles.css

main.py # FastAPI app

Dockerfile

docker-compose.yml

tests/

test\_matcher.py

test\_parser.py

README.md

sample\_data/

bank1\_2025-08.csv

bank2\_2025-08-part1.csv

**12) Sample CSV and mapping examples (include in repo)**

* Provide at least 3 sample CSVs with different formats demonstrating:
  + Bank A: Date,Description,Amount,Type,TranID
  + Bank B: value\_date,txn\_desc,credit,debit,ref
  + Bank C: combined amount signed with - for debits.
* Provide a small script tools/generate\_test\_cases.py that demonstrates reconciliation and produces expected Excel output.

**13) Tests & QA**

* Unit tests for parser normalization and matching logic, with edge cases:
  + simple 1:1 transfer
  + delayed credit (within date tolerance)
  + split incoming credits equal one debit
  + ambiguous candidate matches (classify as need\_verification)
* Integration test that runs the API end-to-end on sample CSVs and verifies Excel content fields and classifications.

**14) Configuration & environment**

* config.py must allow adjustable values:
  + amount\_tolerance (default 0.50)
  + date\_delta\_days (default 3)
  + matching\_score\_threshold (for confirmation)
  + excel\_output\_path
* Environment variables for secrets and mode:
  + APP\_ENV=dev|prod
  + SECRET\_KEY
  + BANK\_CONNECTOR\_MODE=mock|real
* Document how to replace the mock connector with actual bank connectors.

**15) Logging & error handling**

* Provide structured logging and clear error responses for upload errors and parse failures.
* Return useful HTTP status codes and JSON error messages.

**16) UX details & micro-interactions**

* Use modals for trace inspection.
* Row-level confirmation uses optimistic UI and calls an API to persist the confirmation.
* Provide a summary bar above the table: totals by classification (counts & sum amounts), total unmatched, total internal transfers.

**17) Accessibility & responsiveness**

* Table and buttons must be keyboard accessible.
* Responsive layout for desktop/tablet.

**Acceptance criteria (what you must deliver)**

1. A Git repository (code tree as described) that can be docker-compose up to start the app (backend + frontend).
2. All endpoints protected via the provided auth module.
3. Frontend UI accepts multiple accounts and CSV files, runs reconciliation, shows combined result table, allows confirming doubtful rows, and provides Excel download in top-right.
4. Matching algorithm implements the tolerance and date window rules and returns trace\_links and matching\_score.
5. Unit tests covering parser and matcher; integration test covering end-to-end run.
6. README with run, test, deploy, and configuration instructions plus sample CSV files / sample expected outputs.
7. Provide sample environment file .env.example.

**Example API endpoints (all require auth)**

* POST /api/accounts — add account metadata
* POST /api/accounts/{account\_id}/upload — upload transactions CSV(s)
* POST /api/reconcile — triggers reconciliation across all uploaded accounts (returns reconciliation job id and results or returns results synchronously for smaller datasets)
* GET /api/reconcile/{job\_id}/results — get results
* POST /api/transactions/{transaction\_id}/confirm — confirm a doubtful classification
* GET /api/export?format=xlsx — returns XLSX file or pre-signed download URL
* POST /api/bank/download — (mock) download transactions using bank credentials provided in request (for demo only; must be opt-in)

**Example expected output row (JSON)**

{

"bank": "Bank A",

"account": "\*\*\*\*1234",

"transaction\_id": "TXN-00123",

"date": "2025-09-01",

"description": "Transfer to main",

"amount": -1500.00,

"direction": "debit",

"classification": "internal\_transfer",

"trace\_links": ["TXN-00234"],

"matching\_score": 0.95,

"source\_file": "bank\_a\_sep.csv",

"needs\_verification": false

}

**Extra guidance for implementer (notes to the code generator)**

* Keep reconciliation logic in backend (LangChain matching agent). Frontend only responsible for upload, display, and user interactions.
* Implement a clear, commented codebase with README and docstrings.
* Make configuration easily tunable (tolerances, scoring weights).
* Provide a simple admin or debug view to re-run matching with different thresholds.

**End of prompt**

Now produce:

1. A succinct file-by-file scaffold (tree with brief one-line purpose per file).
2. A minimal example of the matching algorithm in Python pseudocode or real code (50–80 lines) showing how to match debits to credits using tolerance and date window — this code must be copy/paste runnable and include type hints and simple unit-test assertions at the bottom demonstrating 3 cases: exact match, delayed match, ambiguous => need\_verification.
3. A short README snippet (commands) showing how to run the project in Docker and run tests.

Keep the scaffold, code sample, and README concise but complete — this will be the actionable artifact you hand to engineers or paste into a code-generation agent.

**Product Name Suggestion**

Since your company is **Headstart Financial**, I suggest:

**Headstart Ledger**

* "Ledger" emphasizes reconciliation, reporting, and tax compliance.
* Sounds professional, aligns with finance, and broad enough to cover **banking + trading + tax**.

**2. Revised Prompt**

Here’s the new **specification prompt** you can use with your software platform / AI dev tool:

**Prompt — Build Headstart Ledger**

You are an expert full-stack developer. Build a **deployable, production-grade finance reconciliation and trading profit/loss summary system** for **Headstart Financial**.

**Functional Modules**

1. **Bank Reconciliation Module** (existing spec from before):
   * Upload multiple bank accounts, reconcile transactions, classify as internal/external, detect doubtful ones, produce downloadable Excel.
   * Implement matching rules with tolerance on amount and date.
2. **Trading Summary Module** (new):
   * Accept **transaction CSVs** from crypto exchanges, stock brokers, or manual trading logs.
   * Normalize trade records (date, ticker/symbol, buy/sell, qty, price, fees, etc.).
   * Summarize **daily profit/loss** and cumulative **capital gains/losses**.
   * Apply **Australian taxation rules** for capital gains:
     + Short-term (< 12 months) → taxed as ordinary income.
     + Long-term (≥ 12 months) → 50% discount applied for individuals.
     + FIFO matching for cost basis.
   * Output: Summary table with total buys, sells, realized P/L, unrealized P/L, taxable capital gains.
3. **Frontend**
   * Tabbed interface:
     + Tab 1 = **Bank Reconciliation**
     + Tab 2 = **Trading Summary**
   * Each tab has file upload, summary tables, Excel download.
4. **General Requirements**
   * Use **LangChain** to clearly separate frontend/backed processing.
   * Use **FastAPI** backend + React frontend (with CSS separated).
   * Modular file structure (each function in its own file).
   * Provide Excel export with professional formatting.
   * Docker-ready full deployable stack.

**3. Updated File Structure**

headstart-ledger/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py # FastAPI entrypoint

│ │

│ ├── api/ # API routes

│ │ ├── \_\_init\_\_.py

│ │ ├── bank\_routes.py

│ │ ├── trading\_routes.py

│ │ └── auth\_routes.py

│ │

│ ├── auth/ # Authentication module

│ │ ├── \_\_init\_\_.py

│ │ └── auth\_manager.py

│ │

│ ├── core/ # Core business logic

│ │ ├── \_\_init\_\_.py

│ │ ├── parser.py # Bank CSV parsing

│ │ ├── matcher.py # Bank reconciliation

│ │ ├── exporter.py # Excel export

│ │ ├── bank\_connector.py # Bank connectors

│ │ ├── trading\_parser.py # Parse trading CSV

│ │ ├── trading\_summary.py # Calculate P/L & CGT (AU tax logic)

│ │ └── config.py

│ │

│ ├── langchain/

│ │ ├── \_\_init\_\_.py

│ │ ├── bank\_agents.py

│ │ ├── trading\_agents.py

│ │ └── chains.py

│ │

│ ├── models/

│ │ ├── \_\_init\_\_.py

│ │ └── schemas.py # Pydantic models

│ │

│ └── frontend/ # Frontend (tabbed UI)

│ ├── index.html

│ ├── app.jsx # React app with Bank + Trading tabs

│ └── styles.css

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_parser.py

│ ├── test\_matcher.py

│ ├── test\_trading\_parser.py

│ └── test\_trading\_summary.py

│

├── sample\_data/

│ ├── bank1.csv

│ ├── bank2.csv

│ └── trading.csv

│

├── tools/

│ └── generate\_test\_cases.py

│

├── Dockerfile

├── docker-compose.yml

├── requirements.txt

├── README.md

└── .env.example