ChatGPT

Open Finance Connectors (Brazil) – Architecture & Implementation Report

Introduction

This report presents a comprehensive design for read-only **Open Finance connectors** in Brazil using **Pluggy** and **Belvo**, focusing on ingestion of **accounts, balances, and transactions**. The solution emphasizes robust OAuth2 flows compliant with **FAPI** (Financial-grade API) security profiles and user **consent management** aligned with **LGPD** (Brazil's data protection law). We detail a unified connector abstraction supporting **webhook** and **polling** data ingestion, deterministic **deduplication** across multiple sources (aggregators, CSV files, Gmail receipts) via hashing and provenance tags, and a **reconciliation** engine that matches transactions with receipts/invoices. The design includes a guided **3-step UI flow** (Link → Preview → Confirm) with data diffing, duplicate detection, and source badges for transparency. We provide JSON schemas and TypeScript interfaces for core models (ConsentRecord, UnifiedAccount, UnifiedTransaction, etc.), Next.js API route samples for linking workflows, outlines of testing (including race conditions and golden datasets), and system diagrams illustrating the architecture. Key goals are to ensure **deterministic behavior**, **explainability** (why data was or wasn't merged), full auditability for compliance, and high code quality (≥85% test coverage, no secrets in repo, DSAR support).

Connector Architecture and Ingestion Mechanisms

Connector Abstraction: We define a uniform connector interface to integrate various data sources (Pluggy, Belvo, CSV, Gmail) under a common abstraction. Each connector handles authentication, data retrieval (accounts, balances, transactions), and change notification. This design enables plugging in new providers with minimal changes to the core logic. For example:

```
interface FinancialConnector {
  connect(credentials?: any): Promise<void>;            // initiate OAuth or
link process
  fetchAccounts(): Promise<UnifiedAccount[]>;           // retrieve accounts &
balances
  fetchTransactions(since?: Date): Promise<UnifiedTransaction[]>; // retrieve
new transactions
  handleWebhook(event: any): Promise<void>;             // process incoming
webhook event
}
class PluggyConnector implements FinancialConnector { … }
class BelvoConnector implements FinancialConnector { … }
class GmailConnector implements FinancialConnector { … }
class CsvConnector implements FinancialConnector { … }
```

Connectors share common features like a **token broker** integration for secure API key storage and OAuth tokens, standardized error handling, and scheduling hooks. We will implement a **base class** with shared logic (e.g. standardized error mapping, retry with backoff) and extend it for each provider's API quirks.

Webhook & Polling Ingestion: To maximize reliability and freshness, the system supports both event-driven and polling-based data acquisition. **Aggregators (Pluggy/Belvo)** will primarily use **webhooks** for real-time updates, while also allowing periodic polling as a fallback or for backfilling historical data. For example, Belvo's Open Finance product uses an asynchronous pattern: once a user link is created, Belvo fetches data in the background and notifies via webhooks when data is available [1]. Our service registers a webhook endpoint with each aggregator so we receive events like "historical_update" or "new_transactions_available" as soon as data is loaded. On receiving such an event, a background worker will call the aggregator's API to fetch the new or updated records (e.g. Belvo sends a `historical_update (TRANSACTIONS)` webhook, after which we call `GET /transactions?link={id}` to retrieve details [2]). Similarly, Pluggy can send events like `transactions/created` with a URL to fetch newly added transactions [3] [4].

In parallel, we implement **scheduled polling** jobs as a safety net and for initial bulk loads. For example, upon first linking an account, we may poll the aggregator's `/transactions` endpoint for the last 12 months to ensure we have all historical data (since Belvo automatically loads 12 months on link creation [1] and Pluggy's connectors can retrieve up to 12 months [5]). Polling also acts as a backup in case a webhook is missed or delayed – a periodic job (e.g. every 24 hours) can trigger a sync of all links to catch any discrepancies. We will design the polling vs. webhook interplay carefully to avoid race conditions (discussed in **Testing**). If both a webhook and a scheduled poll retrieve the same new transaction around the same time, our deduplication logic will ensure only one copy is stored (details in **Deduplication**).

Token Management: All sensitive credentials and tokens are managed via a secure token broker. The aggregator API keys (client ID/secret) are stored in a secure vault or server-side config, never exposed on the frontend. When initiating a user connection, our backend either generates a short-lived **widget token** (as in Belvo's Access Token or Pluggy's Connect Token) or directly uses server-side calls. For example, for Belvo we call `POST /api/token/` with our client secrets to get a widget `access_token` (valid ~15 minutes) for that user's CPF/CNPJ [6] [7]. This token is then passed to the front-end widget. For Pluggy, we use their Connect Token API to obtain a token tied to our webhook URL and an optional user ID (so events are traceable) [8] [9]. These tokens obviate the need to embed our master API keys in the front-end, improving security. The connectors will also manage **refresh tokens** or re-auth flows if needed (though in Open Finance, user consent flows replace traditional credential refresh; see **OAuth Flows** below).

Backfill and Scheduling: Upon linking a new data source, an **initial backfill job** is scheduled to ingest historical data. This may involve paging through API results (e.g. fetching transactions in batches of 500 for older dates [10]) or in the case of Gmail, iterating through years of emails. We leverage provider capabilities such as Belvo's automatic historical fetch (they immediately retrieve last 12 months after consent [1]) and Pluggy's support for fetching history on first `update` call. For **Gmail**, which can contain years of receipts, we implement a controlled batch ingestion: first, use the Gmail API to search and pull all messages matching financial keywords (receipts, invoices, etc.) to prioritize relevant emails [11] [12], then eventually consume all emails for completeness. We utilize Gmail **push notifications** (Pub/Sub webhooks) for real-time email ingestion [13], combined with occasional polling as a backup for any missed events [14] – this mirrors our approach with aggregator webhooks plus polling backup.

All connectors integrate with a central scheduling system (could be a cron-based worker or a task queue) to coordinate these jobs. The schedule covers periodic refreshes (e.g. nightly sync for each active link), **consent expiration checks**, and any maintenance tasks (like re-trying failed imports). The architecture thus ensures data is kept up-to-date in near real time, while also handling large historical loads efficiently and without manual intervention.

System Diagram: The high-level data flow is as follows: **User** initiates a connection via our **Front-End** → Our **Next.js API** (backend) interacts with the chosen **Connector** (Pluggy, Belvo, Gmail, CSV) → If OAuth is required, user is redirected to the **Bank/Provider** (through aggregator's widget) for authentication and consent → Once authorized, the aggregator creates a **Link/Item** for the user and our system receives a callback or webhook indicating success → A **background worker** fetches initial data (accounts, balances, transactions) from the aggregator and stores them in our database (in a staging area for preview) → Subsequently, **webhooks** from the aggregator (or scheduled jobs) deliver incremental updates (new transactions, etc.), which our workers process and upsert into the database. Meanwhile, for emails, the Gmail API's Pub/Sub pushes new emails which our **Email Connector** ingests into a raw email store for parsing. All ingested data passes through the **deduplication** and **reconciliation** layers before surfacing in the unified datastore. The **Unified Data Store** holds normalized records (accounts, transactions, consents, receipts) with references back to their sources. Finally, the **Front-End UI** presents a Preview of changes to the user for confirmation, after which the data is committed to the user's ledger.

OAuth2 Flows and Consent Management (FAPI & LGPD)

FAPI-Compliant OAuth2: Security is paramount given the sensitivity of financial data. Our connectors adopt OAuth 2.0 flows that comply with OpenID Foundation's **Financial-grade API (FAPI)** standards [15]. In practice, this means using high-assurance OAuth profiles with PKCE, nonce/state parameters, mTLS or JWS request object validation as required by Open Finance Brasil's guidelines [16] [15]. We leverage the fact that both Pluggy and Belvo provide pre-built **Open Finance-compliant widget flows**, so that users authenticate directly against their bank's portal with no credentials ever touching our system [17] [18]. For example, with **Belvo**, we generate an access token and embed their **Hosted Connect Widget** in our app; the widget orchestrates the entire consent flow: the user selects their institution and is redirected to the bank's Open Finance login page, scans QR or enters OTP as needed, and grants consent for specific data scopes [17] [19]. Upon success, the bank redirects back to Belvo's widget which then signals our application and yields a **Link ID** for the connected account [6] [20]. This flow is fully compliant with Open Banking security (the user authenticates at their bank and authorizes data sharing to our application) and meets the stringent requirements of FAPI (secure redirect, JWT request objects, etc.) – indeed participants in Open Finance Brasil **must certify** compliance with FAPI and OpenID standards [15]. By using the official aggregator SDKs and widgets, we inherit these security measures (the Curity Identity Server notes that supporting FAPI involves advanced OAuth/OIDC server capabilities and consent handling [21]).

For **Google (Gmail)** integration, we likewise use OAuth2: when the user chooses to connect their Gmail, we redirect them to Google's OAuth consent screen to grant read access to their Gmail (scopes `gmail.readonly` and Pub/Sub for push). This is a standard OAuth web flow with our own `redirect_uri` (e.g. `/api/connectors/callback?source=gmail`). The same Next.js callback endpoint can handle multiple providers by inspecting the query parameters (e.g. `source=gmail` or `source=belvo` if that were needed, although Belvo/Pluggy use the widget approach without redirecting to our backend). All OAuth callbacks include verification of the `state` parameter for CSRF protection, and we use PKCE for Google OAuth as required. The retrieved Gmail OAuth tokens (access + refresh token) are securely stored (encrypted at rest) via our token broker.

Consent Capture and Management: Whenever a user links a new financial institution, a **ConsentRecord** is created in our system to log the user's permission details. This record is essential for LGPD compliance (Law No. 13,709/2018) as it documents the legal basis (typically the user's explicit consent) and scope of data usage. It includes fields such as the user's ID, the institution name, which data products were authorized (accounts, transactions, etc.), the timestamp of consent, and expiry or revocation status. Internally, we map aggregator outcomes to our consent model: e.g. Pluggy provides a Consent object tied to the Item with a list of granted permissions (`openFinancePermissionsGranted`) and possibly an expiration [22] [23]. Belvo treats consent as inherent to the Link and manages its state (they also expose a Consents API for querying consents) [24] [25]. We will store the **Consent ID** or Link ID returned by the aggregator, so that we can reference or revoke it later.

Consent Expiry: According to Brazilian Open Finance regulations, a data sharing consent can have a maximum duration of **365 days** [26]. (This is higher than some jurisdictions like the UK's 90-day rule.) In practice, some aggregators default to no automatic expiry and rely on user revocation [27], but from a compliance perspective we expect most consents to be time-bound (e.g. 1 year). Our system will track the `expiresAt` for each consent (if provided by aggregator or set by policy). For example, if a consent was given on 2024-06-11 via Pluggy, the ConsentRecord might show `expiresAt: null` (no fixed expiry) [28], but we may enforce a 365-day limit internally or prompt the user to renew annually to align with regulations. We subscribe to aggregator webhooks for consent events; Belvo will send a webhook when a user's consent expires [29], allowing us to immediately mark that link as expired in our database and notify the user to renew consent. The **runbook** for consent expiry (see Operations) describes how the system handles this: e.g. flagging the connection as "expired" (disabling further data pulls) and sending the user a notification with steps to re-consent.

Consent Revocation: Under Open Finance and LGPD, users can revoke consent at any time. Revocation might happen in three ways: (1) the user explicitly disconnects the account via our app, (2) the user revokes access in their bank's portal (or via the Central Bank's Open Finance dashboard, e.g. **Meu Belvo Portal** which consolidates consents [30] [31]), or (3) our app deletes the link (e.g. during user account deletion). We handle each scenario:

- If the user uses our UI to "disconnect" an institution, we call the aggregator's API to delete the link/item. Belvo notes that deleting a link on our side will automatically revoke the user's consent in their system [32] [33]. We record `revokedAt` in our ConsentRecord and ensure no further data is fetched. All locally stored data remains until the user requests deletion (but marked as inactive).

- If the user revokes consent directly with the bank or via the Open Finance portal, our next data fetch will fail with an authorization error, or we may receive a webhook if the aggregator monitors revocations. In such cases, we mark the consent as revoked and treat it similarly to expiry. The system audit trail will log the revocation time and reason (e.g. "revoked by user at institution"). We also surface this status to the user ("Connection revoked") and give them the option to reconnect.

- For **Data Subject Access Request (DSAR)** or account deletion: if a user requests removal of their data, we will **delete the Link/Item** via aggregator API (fully revoking consent) and purge all related personal data from our databases (see DSAR in Operations). We maintain only minimal logs (e.g. that a consent existed and was revoked, as allowed for compliance record-keeping).

Audit Trail & LGPD Compliance: Every consent action is logged to an immutable audit log. The ConsentRecord JSON schema (see below) captures all relevant info needed to demonstrate compliance: the **legal basis** (typically "Consent" under LGPD Article 7, item I), the exact **terms** accepted (we store a versioned copy of the consent text or bank-provided permissions list), and timestamps for creation, update, or revocation. In addition, we preserve references to the source of truth for the consent. For Open Finance, the authoritative consent is managed by the bank; however, we may use aggregator APIs to query consent status if available. (Brazil's Open Banking specifies a Consents API for creating, retrieving, and deleting consents in the bank's systems [34]. In our case, aggregators abstract this, but it's important we can retrieve proof of user consent if challenged. We could call Belvo's `GET /api/consents/` if needed to show details [35].)

Our design ensures **transparency** to users about their data. Users can view and manage consents through our app or via the aggregator's portal. Belvo provides the **My Belvo Portal (MBP)** where users can see and revoke consents they've granted [30] [31]. We plan to integrate a link to MBP for Brazilian users as an additional option. In our app's "Connected Accounts" section, each connection shows the data permissions granted (e.g. "Accounts, Transactions, Balance" scopes) and an option to **"Revoke Access"**.

Finally, all data objects in our system carry **provenance metadata** to enable full traceability, which is crucial for LGPD's principles of accountability and transparency. For instance, each transaction knows which consent (and thus which legal basis) allowed its collection, and each parsed email knows which raw email it came from. We ensure every data point can be traced back to its **source document or event**. As an example, our email parsing stores the Gmail message ID and thread ID with each record for de-duplication and linking back to the raw email [36]. Similarly, a transaction fetched from an aggregator carries the aggregator's item/link ID and original transaction ID. This allows us (and the user) to audit where data came from and confirm it was authorized. In the UI or export, we could show something like "Transaction X was obtained via Open Finance consent on 2024-06-11 (Consent ID …) from Bank Y (via Pluggy)." Such traceability meets the **auditability** requirements – *"every parsed data point should be traceable to its source document"* [37] – and builds user trust.

Unified Data Model (JSON Schemas & TS Interfaces)

To harmonize data from aggregators, emails, and other sources, we define **unified models** for core financial entities. This ensures that regardless of source, accounts and transactions are represented consistently in our application. Key models include **ConsentRecord**, **UnifiedAccount**, **UnifiedTransaction**, and related types. Below we provide JSON schema samples and corresponding TypeScript interfaces for these models.

ConsentRecord Schema

This schema captures user consents for data access. It includes the scope of data authorized, timestamps, and status flags needed for compliance.

```
// JSON Schema: ConsentRecord
{
  "$id": "ConsentRecord",
  "type": "object",
  "properties": {
    "id":       { "type": "string", "format": "uuid", "description": "Consent
record unique ID" },
    "userId":   { "type": "string", "description": "Reference to user who gave
consent" },
    "provider": { "type": "string", "description": "Aggregator or source (e.g.
'Pluggy', 'Belvo')" },
    "institution": { "type": "string", "description": "Financial institution
name or code" },
    "products": { "type": "array", "items": { "type": "string" },
"description": "Data products authorized (ACCOUNTS, TRANSACTIONS, etc.)" },
    "permissions": { "type": "array", "items": { "type": "string" },
"description": "Fine-grained Open Finance permissions granted (if available)" },
    "legalBasis": { "type": "string", "description":
"Legal basis for processing (e.g. 'Consent')" },
    "createdAt": { "type": "string", "format": "date-time", "description":
"Timestamp when consent was granted" },
    "expiresAt": { "type": ["string","null"], "format": "date-time",
"description": "Consent expiration timestamp (if any)" },
    "revokedAt": { "type": ["string","null"], "format": "date-time",
"description": "Timestamp if consent was revoked" },
    "consentId": { "type": "string", "description": "ID of consent in provider
system (if provided)" }
  },
  "required": ["id", "userId", "provider", "institution", "products",
"createdAt"]
}
```

```
// TypeScript Interface: ConsentRecord
interface ConsentRecord {
  id: string;
  userId: string;
  provider: 'Pluggy' | 'Belvo' | 'ManualCSV' | 'Gmail';
  institution: string;               // e.g. 'Banco do Brasil' or 'Gmail'
  products: string[];               // e.g. ['ACCOUNTS','TRANSACTIONS']
  permissions?: string[];           // e.g. ['ACCOUNTS_ALL','TRANSACTIONS_ALL']
(for Open Finance connectors) [23]
  legalBasis: string;               // e.g. 'Consent (LGPD Art.7,I)'
  createdAt: string;               // ISO date-time of consent grant
  expiresAt?: string | null;       // ISO date-time if expires (null if none) [28]
  revokedAt?: string | null;       // ISO date-time if revoked
  consentId?: string;              // external reference, e.g. Pluggy consent/
item ID or Belvo link ID
}
```

Explanation: The **ConsentRecord** ties a user to an institution's data access grant. For instance, when a user connects "Bank ABC" via Pluggy's Open Finance connector, we create a ConsentRecord with `provider='Pluggy'`, `institution='Bank ABC'`, `products=['ACCOUNTS','TRANSACTIONS',...]` (based on what the user agreed to share), and timestamps. We log if the consent is later revoked or expires. The optional `consentId` could store the aggregator's identifier (e.g. Pluggy's Item ID, which can be used to retrieve consent info via Pluggy's Consents API [38], or Belvo's Link ID which maps 1:1 to a consent [25]). The presence of `expiresAt` and `revokedAt` helps operationally; e.g. if `expiresAt` is approaching, we know to prompt the user to renew.

UnifiedAccount Schema

This model represents a financial account (e.g. a bank checking account, a credit card, an investment account) in a unified format. It includes identifying details and latest balance info.

```
// JSON Schema: UnifiedAccount
{
  "$id": "UnifiedAccount",
  "type": "object",
  "properties": {
    "id":         { "type": "string", "format": "uuid" },
    "userId":     { "type": "string" },
    "institution":{ "type": "string", "description": "Name/code of the bank or
provider" },
    "accountNumber": { "type": "string", "description": "Masked account
identifier (e.g. IBAN or last4)" },
    "type":       { "type": "string", "description": "Account type (CHECKING,
SAVINGS, CREDIT_CARD, etc.)" },
    "currency":   { "type": "string", "description": "Currency code (ISO 4217,
e.g. 'BRL')" },
    "balance": {
       "type": "object",
       "properties": {
         "current":   { "type": "number", "description": "Current balance or
last statement balance" },
         "available": { "type": "number", "description":
"Available balance (for checking accounts)" },
         "creditLimit":{ "type": "number", "description": "Credit limit (for
credit cards)", "nullable": true }
       },
       "required": ["current"]
    },
    "lastSyncAt": { "type": "string", "format": "date-time", "description":
```

```
"When this account was last updated" },
    "externalIds": {
      "type": "object",
      "description": "External references from providers",
      "properties": {
        "pluggyItemId": { "type": "string" },
        "pluggyAccountId": { "type": "string" },
        "belvoLinkId":   { "type": "string" },
        "belvoAccountId":{ "type": "string" }
      }
    },
    "provenance": {
      "type": "array",
      "items": { "$ref": "DataProvenance" },
      "description": "Sources contributing to this account"
    }
  },
  "required": ["id","userId","institution","type","currency","balance"]
}
```

```
// TypeScript Interface: UnifiedAccount
interface UnifiedAccount {
  id: string;
  userId: string;
  institution: string;          // e.g. 'Itaú', 'Bradesco', 'Nubank', or 'Gmail'
  accountNumber?: string;       // e.g. '****1234' (masked for privacy)
  type: 'CHECKING' | 'SAVINGS' | 'CREDIT_CARD' | 'INVESTMENT' | 'OTHER';
  currency: string;             // 'BRL', 'USD', etc.
  balance: {
    current: number;
    available?: number;
    creditLimit?: number;
  };
  lastSyncAt: string;
  externalIds?: {               // optional mapping of provider IDs
    pluggyItemId?: string;
    pluggyAccountId?: string;
    belvoLinkId?: string;
    belvoAccountId?: string;
    [key: string]: string | undefined;
  };
  provenance: DataProvenance[]; // list of sources that provided data for this
account
}
```

Explanation: **UnifiedAccount** merges account information from any source. Fields like `institution` and `accountNumber` identify the account (we'll use masked numbers or hashes for privacy). The `type` indicates if it's a bank account, credit card, etc. We include a nested **balance** object for the latest known balances – for a checking account, `current` might equal the ledger balance and `available` might factor in holds [39]; for a credit card, `current` could represent the current statement balance and `creditLimit` the card limit. If a provider supplies multiple balance categories (Belvo's API provides available, blocked, and automatic investment amounts [40]), we map them accordingly (we might extend the balance schema to include those if needed). We keep `lastSyncAt` to know when data was last refreshed. The `externalIds` object stores provider-specific identifiers (e.g. Pluggy's `accountId` UUID [41] or Belvo's account resource ID) – this is mainly for internal use if we need to update or reconcile with the provider's data (and for debugging). The `provenance` is an array of DataProvenance entries (defined below) to list the sources. For instance, an account's provenance might include `{source: 'pluggy', id: '03cc0eff-4ec5-495c-... (pluggyAccountId)'}` and perhaps a CSV file name if the user also imported a statement CSV for this account. In many cases, an account will have just one source (e.g. Pluggy OR Belvo). However, if a user connected the same bank via both aggregators (unlikely in normal operation) or switched from one to another, we could merge them into one UnifiedAccount with two provenance entries.

Note: Account deduplication: We will use unique identifiers (account number + institution) to avoid duplicate UnifiedAccount entries. When new accounts are fetched, we compare them against existing ones for the user. If a match is found (same institution and account ID), we update the existing record's provenance to include the new source instead of creating a duplicate. This helps if, for example, the user reconnects a bank via a different aggregator after a consent expires – the system will recognize it's the same account. We may also incorporate the account holder's CPF (if provided by aggregator's identity data) as a secondary check for uniqueness.

UnifiedTransaction Schema

This is the unified representation of a transaction or movement on an account. It includes core details like date, description, amount, plus metadata for deduplication, provenance, and reconciliation.

```
// JSON Schema: UnifiedTransaction
{
  "$id": "UnifiedTransaction",
  "type": "object",
  "properties": {
    "id":          { "type": "string", "format": "uuid" },
    "accountId":   { "type": "string", "description": "Reference to
UnifiedAccount" },
    "date":        { "type": "string", "format": "date-time", "description":
"Transaction date (posting date in ISO8601)" },
    "amount":      { "type": "number", "description": "Transaction amount
(credits positive, debits negative)" },
    "currency":    { "type": "string", "description": "Currency code, e.g.
BRL" },
    "description":{ "type": "string", "description": "Cleaned description or
merchant name" },
    "rawDescription":{ "type": "string", "description": "Original description
from source (if available)" },
    "type":       { "type": "string", "description": "TRANSACTION type (e.g.
'DEBIT' vs 'CREDIT', or category like 'PAYMENT', 'TRANSFER')" },
    "status":     { "type": "string", "description": "Status (e.g. 'POSTED',
'PENDING')" },
    "category":   { "type": ["string","null"], "description":
"Category label if available (from enrichment or source)" },
    "hash":       { "type": "string", "description": "Deterministic hash for
deduplication" },
    "provenance": {
      "type": "array",
      "items": { "$ref": "DataProvenance" },
      "description": "Sources contributing to this transaction"
    },
    "linkedReceiptId": { "type": ["string","null"], "description":
"ID of linked receipt/invoice record, if reconciled" },
    "createdAt":  { "type": "string", "format": "date-time" },
    "updatedAt":  { "type": "string", "format": "date-time" }
  },
  "required": ["id","accountId","date","amount","currency","description","hash"]
}
```

```
// TypeScript Interface: UnifiedTransaction
interface UnifiedTransaction {
  id: string;
  accountId: string;          // links to UnifiedAccount.id
  date: string;               // ISO8601 date (posting date, assume UTC
normalized)
  amount: number;             // positive for credit, negative for debit
  currency: string;
  description: string;
  rawDescription?: string;
  type?: string;              // e.g. 'DEBIT', 'CREDIT', 'TRANSFER', 'FEE', etc.
  status?: string;            // e.g. 'POSTED' or 'PENDING'
  category?: string | null;   // optional high-level category (if provided by
aggregator's categorization [42] or set by user)
  hash: string;               // deduplication key (hash of normalized fields)
  provenance: DataProvenance[];
  linkedReceiptId?: string | null;  // link to a receipt/invoice (if matched by
reconciliation)
  createdAt: string;
  updatedAt: string;
}
```

```
// Supporting Interface: DataProvenance
interface DataProvenance {
  source: 'pluggy' | 'belvo' | 'gmail' | 'csv';
  sourceId: string;              // e.g. Pluggy Item ID or Belvo Link ID
  recordId?: string;             // e.g. Pluggy transaction.id UUID [43] , or
Gmail message ID, or CSV line ID
  additionalInfo?: string;       // optional context (e.g. CSV filename, email
snippet)
}
```

Explanation: **UnifiedTransaction** represents a financial transaction with all the fields we need for displaying and analysis. We capture the `date` (using the posting date; if only a date without time is provided by the source we store it as ISO date at midnight UTC), the `amount` (we will normalize to a common sign convention – likely **credits as positive, debits as negative**, or vice versa, but we'll be consistent across sources), and a human-readable `description`. Pluggy and Belvo both provide descriptions: Pluggy has `description` and sometimes a `descriptionRaw` [44] [45]; Belvo provides a concept of transaction description and merchant as well. We'll use `rawDescription` to store any unmodified text from the source if we do any cleaning for `description`. For example, an aggregator might give description "VL 1234 AMAZON MKTPLACE" which we might normalize to "Amazon Marketplace" for display, storing the original in rawDescription. The `type` and `status` come from source if available (e.g. Pluggy marks future credit card installments as `PENDING` [46] , and Belvo distinguishes posted vs pending transactions). The `category` is optional – if the aggregator provides a category or we enrich transactions later, it can go here (Pluggy can auto-categorize transactions if enabled [42]).

Deduplication Key: The `hash` field is critical – it's a deterministic hash of key transaction attributes used to identify duplicates across sources. The algorithm for this will combine stable fields like: the **account identifier**, **transaction date**, **amount**, and a normalized merchant name or description. We will likely take the date in a standard format (e.g. `YYYY-MM-DD`), the amount to 2 decimal places (converted to minor units integer to avoid float issues), and a simplified description (e.g. alphanumeric characters uppercased, removing spaces/punctuation). These components can be concatenated and hashed (SHA-256 or similar) to produce a fixed-length key. If two transactions from different sources yield the same hash, we flag them as duplicates (assuming they refer to the same real-world transaction). This approach provides *deterministic and explainable deduplication*: the logic for matching is based on clear fields (date±0, amount, etc.) rather than opaque heuristics, so we can explain to users *why* two entries were merged (same date and amount and account, etc.). We store the hash in the record for quick comparison and to easily enforce uniqueness (we can put a unique index on `(accountId, hash)` in the database to prevent two identical hashes for the same account from being inserted). Note that we include `accountId` in the computation implicitly – transactions on different accounts should not collide even if date/amount are the same.

We are aware of edge cases: for example, if a user has two identical charges on the same day (e.g. two $50 Uber rides on 2025-09-01), the hash might collide. Our dedupe algorithm will incorporate additional factors to minimize false merges (e.g. include a portion of the description or a transaction ID if available). In practice, aggregator-sourced transactions have unique IDs which we use in provenance, so duplicates from the **same source** won't occur (the aggregator would not send the same transaction twice with the same ID unless it's truly the same record). Duplicates arise mainly across different sources, like an email receipt vs. a bank statement. In those cases, having identical date and amount usually implies the same purchase, but it's possible to have coincidence. Our reconciliation logic (below) further disambiguates by checking merchant names etc., but for deduplication we err on the side of not double-counting the same transaction. If our hash is too strict (no allowance for slight differences), we might miss some duplicates – but we incorporate *drift handling* as needed (e.g. if one source reports the transaction on one date and another on the next date due to posting delays, or an amount differs by a few centavos due to rounding, we might still want to treat them as the same; these cases are handled in reconciliation by adjusting matching criteria).

Provenance: The `provenance` array lists where the transaction data came from. Each entry is a **DataProvenance** with a `source` (e.g. 'pluggy', 'gmail') and identifiers to trace back. For an aggregator, we include the aggregator's item/link ID and transaction ID. Example: `{source: 'pluggy', sourceId: 'item_c33872c7-85dc...', recordId: '6ec156fe-e8ac-4d9a-a4b3-7770529ab01c'}` [43] where `recordId` is Pluggy's transaction UUID. For Gmail, a provenance might look like `{source: 'gmail', sourceId: '<Google OAuth user id or email address>', recordId: '<Gmail messageID>', additionalInfo: 'Starbucks receipt email'}`. For CSV, we might use `{source: 'csv', sourceId: 'upload_12345', recordId: 'line42'}`. Provenance entries allow us to maintain a **link to source evidence**. This is extremely useful for audit and user trust: a user can click a transaction and see if it has an attached receipt or which bank it came from. In our database, if a transaction has a linked email receipt, the `linkedReceiptId` will point to the record in our `ParsedReceipt` (or `parsedEntities`) table that contains the details (including possibly an image or PDF of the invoice). Attaching source documents is a key evidence feature – *"attachments... serve as proof of the transaction"* [47] . We plan to store the file path/ID of any receipt in the receipt record and link it here, so later the user or auditor can retrieve the original receipt (for example, viewing a transaction in our UI could show a PDF of the invoice for verification [48]).

Timestamps: `createdAt` and `updatedAt` record when the unified transaction was first created in our system and last modified. If a transaction is initially inserted as pending (from an email) and later updated (filled in with aggregator data), the updatedAt will reflect that.

Additional Models

In addition to the above, we have a model for **Parsed Receipt/Invoice** (from emails). This can be called **ReceiptRecord** or we might integrate it into a broader "parsedEntities" table as per the Gmail parser design [49] [50] . For completeness, a simplified schema for receipts:

```
interface ReceiptRecord {
  id: string;
  userId: string;
  provider: string;              // e.g. 'Starbucks' or 'Amazon' (merchant)
  amount: number;
  currency: string;
  date: string;                  // purchase date from receipt
  invoiceId?: string;            // invoice or order number if parsed
  category?: string;
  emailId: string;               // reference to raw email record
  transactionId?: string;        // if matched, the UnifiedTransaction.id it's
linked to
```

```
  confidence: number;          // parser confidence score (0-1)
  status: 'PARSED' | 'MATCHED' | 'PENDING';  // state in reconciliation pipeline
  parsedAt: string;
}
```

This corresponds to what our email parsing produces – essentially the fields from a receipt needed to match with transactions [51] [52]. We store these separately (with reference to the email message). The `transactionId` will be filled when reconciliation links it to a UnifiedTransaction, and conversely the UnifiedTransaction's `linkedReceiptId` will point back. By storing receipts and transactions separately and linking via IDs, we maintain a **many-to-many** potential (though typically one receipt matches one transaction). This also allows capturing receipts that have no matching transaction (yet or ever) as their own records (they would have status "PENDING" and null transactionId, which could later be updated when a match is found).

Deterministic Deduplication Design

Overview: The system employs deterministic rules to identify and merge duplicate records from different sources in an **explainable** way. Deduplication applies at two levels: **within the same source** (to handle, say, an aggregator accidentally returning a duplicate entry or our email parser double-parsing the same email) and **across sources** (to merge data referring to the same real-world transaction coming from, e.g., a bank feed and an email receipt). Our strategy is to compute a **normalized key** for each transaction and use hashing to compare keys. If keys match, the records are considered duplicates and are merged or handled as one. We favor false negatives (missing a duplicate) over false positives (merging distinct transactions) to avoid data loss/confusion, but we tune the algorithm to minimize both.

Key Normalization: As mentioned in the UnifiedTransaction schema, we construct a key from important attributes: **Account**, **Date**, **Amount**, and **Description**. Specifically, we define:

- **Account component:** we use a stable account identifier (the UnifiedAccount or some combination of institution+account number) to ensure transactions from different accounts never collide. This is critical because two accounts could have transactions with same date/amount (e.g. two different credit cards each had a $100 charge on the same day). Including account scope prevents merging those. In implementation, since we check duplicates per account, the account context is naturally enforced by scoping queries to accountId or by salting the hash with accountId.

- **Date component:** we use the transaction's date. Typically, the posting date is consistent across sources (a bank statement's posting date should match or be very close to the purchase date on a receipt). We will initially use the exact date (to the day) as part of the key. If sources report times, we normalize to date or a standard timezone to avoid time zone mismatches. (For instance, if an email timestamp is 11:00 PM and the bank posts it next day due to UTC vs local time, we might handle that in reconciliation. For deduping, we consider them different dates so they wouldn't hash the same; instead our reconciliation logic will catch it – see below.) We may later extend the deduplication to allow ±1 day matching, but that is tricky to do in a hash – instead we'll rely on reconciliation to handle near-matches.

- **Amount component:** we use the absolute amount (in minor currency units for precision) since sign differences (credit vs debit) should be resolved beforehand (we plan to normalize sign convention from each source). The amount must match exactly to consider records duplicates. The likelihood of two different transactions having exactly the same amount and date is not negligible, but if their descriptions differ significantly, reconciliation would keep them separate. The dedupe hash might not include description to catch cases where description is slightly different but clearly the same transaction – instead we handle description differences by normalizing it or by reconciliation scoring. To be safe, we include a simplified description in the hash for now, but we may adjust this criterion.

- **Description component:** We extract alphabetic characters from the description/merchant name, convert to uppercase, and possibly drop very common words. For example, "Starbucks #1234" and "STARBUCKS 1234" would normalize to "STARBUCKS1234". However, merchant codes or IDs in descriptions could cause two different Starbucks transactions to have same normalized description. Thus, using description in the hash could over-merge. We may decide not to include description in the hash for strict duplicates, and rely on date+amount alone for the initial grouping, then use description in a second validation step. Alternatively, we include a coarse merchant identifier if available (some aggregators provide a merchant category or cleaned name).

Hashing: Once we form the normalized key string (e.g. `ACCOUNT123|2025-09-26|825|STARBUCKS` for a Starbucks $8.25 on 2025-09-26 on account 123), we compute a cryptographic hash (SHA-256, truncated for storage or MD5 if collision risk is minimal) and store it in the `hash` field. This hash can be recomputed consistently, so we can explain duplicates by breaking down the components. In the UI or logs, we can trace that *"Transaction A and B share the same hash because they had the same date (Sep 26, 2025) and amount ($8.25) on the same account."* If we exclude description from the hash, we might then have to explain using additional context: *"and their amounts/dates matched"*. The method is transparent enough to convey reasoning.

Source-specific Uniqueness: Within a single source, we also ensure no duplicates: for example, **Gmail ingestion** uses the Gmail message-ID as a unique key to avoid ingesting the same email twice [53]. We index emails by message-ID so that push and poll cannot insert duplicates even if an event is processed twice. Similarly for **CSV** imports, we can generate a unique ID per file+line, and if the same file is imported again or the same line appears, we avoid duplication. For **aggregators**, each transaction from the API has a unique ID (e.g. Pluggy's UUID or Belvo's transaction ID). We keep a mapping of those IDs we've seen (in externalIds or provenance) and will not insert the same one twice. So the primary deduplication challenge is **across different sources**.

Deterministic Merging Process: When new data arrives (say a batch of transactions from Pluggy or a parsed set of receipts from Gmail), we go through the following process:

1. **Compute hash for each new record** using the algorithm above.

2. **Lookup existing records** in our database with the same hash (and same account). We can do a quick index query on the hash. If none found, this is a brand new transaction – we mark it as new. If a match is found, we retrieve the existing UnifiedTransaction.

3. **Merge or Update:** If a duplicate is found, we do not insert a new record. Instead, we update the existing record's provenance to include the new source. For example, if the existing record came

from an email receipt (provenance was Gmail), and now an aggregator provides a matching transaction, we add the aggregator info to provenance and update fields if needed. Typically, aggregator data might be more authoritative for certain fields (like final posted date or confirmed amount if any discrepancy). We will have a strategy for field precedence: e.g. trust the bank's data for amount and posting date, but we might keep the richer description from the email if it's more user-friendly. Any differences can be recorded in `rawDescription` or a separate field. We then mark that transaction as confirmed (if it was previously a "pending" entry created from a receipt). For instance, an email-derived pending transaction might not have an exact posting date (maybe just purchase date) – once the bank data comes, we update the date to the bank's posting date (which could be +1 day) and mark status posted.

4. **Collision Handling:** In the rare case that the hash matches but the records are actually different (hash collision scenario, e.g. two distinct transactions with same date/amount on same account), our system will catch it by comparing additional fields. If the descriptions are entirely different or aggregator IDs differ, we suspect it's not a true duplicate. Our dedupe logic can include a secondary check: if two records hash equal but, say, their merchant names share no similarity, we might treat them as separate despite the hash. This is a safety check. Alternatively, we generate a *different hash* for those by incorporating description. The end result is that we will **not merge transactions that appear to be different**. At most, this means two duplicates might slip through as separate records (which is preferable to merging separate events incorrectly). Such edge cases can later be manually merged by an admin if needed (see Dedupe Overrides in Operations).

5. **Marking Duplicates:** When a duplicate is detected and merged, we log an event (for audit/testing) and mark the record appropriately. For example, the webhook event might trigger a log "Merged transaction from Gmail with existing Pluggy transaction ID X". We also ensure that any references (like in ReceiptRecord) are updated to link to the unified record.

The **explainability** of this dedupe approach comes from its rule-based nature: date, amount, account must match. There's no black-box ML deciding what's a duplicate – it's deterministic string matching and hashing. This aligns with design guidance for explainability: *"we prefer rule-based logic for specific cases…and it's explainable (we know exactly how the matching works)"* [54] [55].

Provenance Tagging: Once duplicates are merged, the record's provenance list grows to reflect all sources. Each provenance entry includes enough info to fetch source details, which is used by the UI to display **provenance badges**. For example, a transaction might show a **bank icon** and a **email icon**, indicating it was verified by bank data and has a receipt attached. Users can then drill down to see the receipt or the bank statement reference. By tagging merged records with all applicable sources, we maintain full context. Even after merging, the system can produce reports attributing data to sources (useful for debugging if one source was erroneous, etc.). The provenance also aids **DSAR exports**: if a user wants all their data, we can include where each piece came from, fulfilling LGPD's requirement for transparency.

To further strengthen duplicate detection, we consider using the bank's unique IDs where possible. For instance, if a user uploads a CSV from the same bank that we also connected via API, that CSV might have transaction IDs or reference numbers. If we can parse those and match them to aggregator IDs, that's a direct deterministically unique link. In absence of that, content matching suffices.

We also handle **pending vs posted** duplicates: sometimes the bank may show a pending transaction which later disappears and is replaced by a posted transaction (possibly with a slightly different description or date). Pluggy's webhooks actually notify of `transactions/deleted` in such cases (with IDs of deleted pending transactions after merge on update) [56]. Our connector will use that info – if a transaction is deleted by the aggregator (meaning it was likely merged into another), we will reflect that by marking the pending one as removed or merging it with the new one. If the aggregator doesn't provide explicit info, our own dedupe might catch it: a pending and a posted transaction with same amount but one day apart could hash differently (because of date), so we might not auto-merge them, but the aggregator's deletion webhook will prompt us to delete the pending record. In case we missed it, a background "reconciliation" job could identify orphan pending transactions with no final counterpart after a certain time and flag them.

In summary, our deduplication ensures **no duplicate entries** make it to the final ledger. We validate this with CI tests (inserting sample data twice and asserting the count remains the same, etc.). This is crucial for finance accuracy – we even set a CI gate for "0 duplicates allowed" on test datasets (ensuring the pipeline logic does not produce duplicate transactions in output).

Reconciliation of Transactions with Receipts/Invoices

A major feature is reconciling financial transactions with supporting documents (email receipts, invoices). This adds a layer of verification and enrichment to the data, providing **coverage KPIs** (what percentage of transactions have receipts) and surfacing **evidence for mismatches** (transactions without proof, or receipts without corresponding transactions). We implement a reconciliation engine that matches **UnifiedTransactions** to **ReceiptRecords** (parsed from Gmail or other sources) using deterministic and fuzzy matching.

Matching Logic: Our approach to matching receipts to transactions is rule-based with a scoring mechanism, inspired by best practices in expense matching. The primary keys for matching are **amount** and **date**, as these are the most reliable indicators [57] [58]. We perform the following steps for each new receipt (or each new transaction, depending on which side we iterate over, but essentially we try to pair them):

1. **Candidate Search by Amount and Date:** Given a receipt's amount and date, find transactions on the same account (or possibly any account if the receipt doesn't specify which card – but in email receipts we often have the last4 of the card, which we can use to narrow to an account). We look for transactions with the **exact same amount** on the **same date**. This is usually done with a query `WHERE amount = :amount AND DATE(date) BETWEEN :date-1 AND :date+1` to allow a ±1 day window [57]. We start with same day; if none, we expand to ±1 day to account for posting delays or timezone differences (receipt might show purchase on late night, transaction posted next day) [57]. We do **not** initially allow an amount mismatch, except in rare cases (some receipts might not include sales tax or might be in a different currency – those are advanced scenarios we handle separately or manually).

2. The importance of exact amount match is high: *"a transaction for $55.23 should match an email for $55.23"* [59]. We treat this as a hard requirement in automatic matching. Only if a domain-specific rule tells us otherwise (e.g. some airlines email receipts show ticket price in USD but card charged in BRL – out of scope unless we detect currency difference). In general, if no transaction has the exact

amount on the date, we consider the receipt unmatched (or we create a pending transaction placeholder as described later).

3. If multiple transactions on that date have the same amount (possible but uncommon, e.g. identical purchases), we gather all as candidates.

4. If none on the exact date, we check the day before and day after (since credit card transactions often post 1 day after purchase). For those, we require a very strong match on amount and ideally some clue from description to ensure it's right.

5. We do **not** immediately widen to a week or approximate amount for automatic matching; these can be user-reviewed because that gets into risky territory. (The document suggests maybe considering approximate amount if minor discrepancy [60], but also acknowledges usually amounts match exactly, which we will assume.)

6. **Scoring Candidates:** If we have one or more candidate transactions for a receipt, we compute a **similarity score** for each, based on multiple criteria: amount, date, merchant/reference, etc. The receipt and transaction are compared on:

7. **Amount match:** If amounts match exactly, score it highest (e.g. +5 points) [61]. If we were considering approximate matches (like within a few cents, which could occur due to currency or tip in some foreign currency scenarios), we could score slightly less, but in our baseline we likely avoid approximate auto-matches.
8. **Date match:** If the transaction date is the same as receipt date, add points (e.g. +3) [61]. If off by one day (transaction the next day), give a slightly lower score (e.g. +2) [62]. Two days difference maybe +1 if necessary. More than that typically would not be considered unless evidence suggests (like a flight booked earlier but charged later – edge case).
9. **Merchant name match:** We compare the receipt's merchant/vendor name to the transaction description. Often, email receipts have clear merchant names ("Starbucks") while bank statements might have shorthand ("STARBUCKS 1234"). We perform a fuzzy string comparison (e.g. Jaro-Winkler or Levenshtein) after normalizing (case-insensitive, remove special chars) [63] [64]. If similarity is high (say > 0.8), add points (e.g. +2). If it's an exact or very close match, maybe +2 or +3. If completely different (or receipt vendor unknown in transaction text), 0 points for this criterion [65] [66]. For example, receipt says "Apple Store" and transaction says "APL*APPLE ONLINE" – we'd detect "Apple" in both and consider that a match with high confidence [63].
10. **Reference/Invoice ID match:** Some emails include an order number or invoice number. Some banks include references or authorization codes in transaction detail (rarely exposed to user, but maybe in extended data). If our system captures reference IDs from receipts [67] [68] and if we had any matching field on the transaction (perhaps not usually, but if linking to an installment or Pix transaction it could appear), that would be a strong indicator. If found, that gets a high weight (almost a guaranteed match) [69] [70]. In most cases this won't apply because banks don't show e-commerce order IDs.
11. **Category/type match:** This is minor and often unnecessary – if the email obviously is a purchase (expense) and the transaction is an expense, that's expected. If one was categorized as income, that would be weird and would likely fail other criteria anyway. We might not use this in scoring unless needed [71].

We sum these weighted criteria to get a confidence score [72]. For example, a perfect match might score: Amount (5) + Date (3) + Merchant (2) = 10. If another candidate had same amount/date but merchant name not matching, it might score 8. We choose the highest score. If that highest score is above a threshold (say ≥8) and sufficiently higher than the next candidate, we confirm the match [73]. If two candidates are close (like two transactions on same day, same amount), we mark it as ambiguous and do not auto-link – instead, we flag for manual review (the user might be asked which transaction the receipt corresponds to, if any) [74]. This ensures we *“minimize false positives by requiring strong agreement on key fields”* [75].

1. **Linking:** For each confident match, we update both records: the UnifiedTransaction gets a `linkedReceiptId` pointing to the ReceiptRecord, and the ReceiptRecord gets the `transactionId` filled with the transaction’s ID [76] [77]. We also might update the transaction’s `description` or `category` if the receipt provides a clearer one and if our policy is to enrich data (e.g. if bank description was generic but email had itemized info, we could add notes or improve the description). However, we’d do so carefully and perhaps keep original in `rawDescription`. We definitely attach the receipt image or PDF as an **attachment reference** to the transaction for evidence [47]. In practice, this could mean storing a URL to the PDF or the email in the transaction’s record or a join table. The goal is that from the ledger side, one can retrieve the source document easily [48]. This mirrors accounting software behavior where you can click a transaction to see the original receipt – *“the ledger transaction record gets an attachment/reference to the email, so one can see the source document”* [76].

2. **Coverage Metrics:** We compute **coverage KPIs** such as the percentage of transaction count and value that have receipts. For example, if in a given month 90 out of 100 expense transactions have a matching receipt, we have 90% coverage. We can also compute coverage by value (e.g. 95% of total spend is accounted for by receipts). We maintain counters for matched vs. total transactions (for those types where receipts are expected – e.g. we might exclude transfers or bank fees from this metric as they don’t have receipts). This can be output as a KPI in reports or dashboard. Additionally, we track how many receipts remain unmatched (potentially indicating cash transactions or missing data).

3. **Handling Unmatched Receipts (Mismatches):** If a receipt cannot be matched to any transaction (score below threshold or candidate set empty) [78], this is a discrepancy. There are a few possibilities: (a) The transaction hasn’t posted yet (timing issue), (b) The transaction was cash (so it will never appear in bank feed), or (c) Data gap (perhaps our aggregator missed a transaction, or user paid with a different account not connected). We handle this by **creating a pending transaction** in our system for tracking purposes [79] [80]. Essentially, we generate a UnifiedTransaction with the info from the receipt (amount, date, merchant) but mark it in a special status (e.g. `status: 'PENDING'` or a boolean `unconfirmed=true`). This record serves as a placeholder indicating “User spent BRL X at Merchant Y on Date D according to a receipt, but no official transaction is recorded yet.” We link the receipt to this placeholder (so `linkedReceiptId` and `transactionId` connect them). If later a matching real transaction comes in (say, a day or two later via aggregator), our deduplication will catch it: the pending transaction and the new one will have matching amount/date and thus hash. Rather than inserting a duplicate, the system will merge them – effectively fulfilling the match. The pending record will get updated (status set to posted/ confirmed, date adjusted if needed, etc.) and the provenance will now include the aggregator source. We then consider that receipt reconciled. This approach ensures that receipts aren’t forgotten; they either match or become their own “transaction” which we look to reconcile later [80]

[81]. We also expose these pending entries in the UI (perhaps in the Preview step, or in a special “Unmatched receipts” list). The user could manually associate an unmatched receipt with an existing transaction if we failed to automatically (some manual override UI). If after a certain period (e.g. 7 days) a pending receipt remains unmatched and the user doesn’t provide more info, we treat it as a **cash transaction** or missing bank data. It will remain in the ledger as an “unconfirmed expense” which the user might classify as cash or investigate.

If a UnifiedTransaction has no receipt match (which is the inverse case – a transaction with no receipt), it’s also noted as a coverage gap. We can mark those in the UI (e.g. a small icon indicating “no receipt available”). This could prompt users to attach one if they have it (maybe they have a paper receipt they could scan, though that’s beyond our scope). The KPI “coverage” essentially is (# transactions matched with receipts / total transactions that should have receipts). Our output can highlight **evidence for mismatches**: for each transaction without receipt, we can list it (with date/amount/merchant) as lacking evidence; for each receipt without transaction, we list that as well. These lists are part of the reconciliation report to identify issues. For instance, if we output a **Coverage Report**, it might say: *95% of expenses have receipts. Unmatched transactions: [list of dates/merchants]. Unmatched receipts: [list of receipts].* Each entry would provide evidence details – e.g. a link to the email for an unmatched receipt, or transaction details that might help locate a lost invoice.

1. **Evidence Attachment:** For each successful match, as mentioned, we attach the receipt image or PDF to the transaction record as evidence. In our system, this could mean uploading the file to cloud storage and saving a URL, or storing the binary in a blob column for quick access. Many modern systems do this for one-click reconciliation – e.g. Receipt-AI syncing the receipt image to QuickBooks for easy auditing [82] [83]. We follow that pattern: once a link is made, the **receipt becomes part of the transaction’s record** (as an attachment or link). This not only provides evidence for that transaction, but also gives **end-to-end traceability** for audits – from transaction entry in the ledger all the way to the source document [76] [77].

The reconciliation logic will run at appropriate times: after each batch of new transactions or receipts is ingested, we attempt to match them. Likely we’ll integrate it such that when new transactions come from an aggregator, we check if any pending receipts can be matched now, and when new receipts come from Gmail, we immediately look for matching transactions. We might also run a scheduled reconciliation sweep daily to catch any that might have been missed or to retry matches for pending receipts (like “try to match any receipts that are 1-3 days old with any new transactions” as the Gmail guide suggests) [81].

Example: As a concrete illustration, suppose a user’s Gmail yields a receipt: *Starbucks, Sep 26 2025, Total R$8.25*. Our parser creates a ReceiptRecord (provider “Starbucks”, amount 8.25, date 2025-09-26, maybe referenceId null, etc.) [51] [52]. Our aggregator (Pluggy) later returns a credit card transaction on Sep 27 2025 for R$8.25 with description “STARBUCKS 1234”. The reconciliation engine will find that transaction for the receipt (date one day off, amount exact) and score it: amount match +5, date within 1 day +2, merchant fuzzy match “STARBUCKS” +2 -> total ~9, which likely is above threshold. It links them: the UnifiedTransaction (which has id say TXN-100245) gets linkedReceiptId pointing to the receipt, and the receipt record gets transactionId “TXN-100245” [84]. The example in the documents shows exactly this scenario, where the output JSON had `"transactionId": "TXN-100245"` for the Starbucks receipt after linking [84]. Our system would mirror that behavior. The transaction now has an attached receipt image (maybe the email body or PDF if the email had one) accessible via the UI.

KPIs & Mismatch Evidence: After reconciliation, we compute metrics like: "Out of 50 transactions this month, 45 have receipts (90%). 5 transactions missing receipts: [list]. 2 receipts had no matching transaction (likely cash purchases or pending): [list]." We might expose this on an admin dashboard or provide it to the user as insights (some apps show "You have X unverified transactions – attach receipts or mark them as verified"). This fulfills the requirement of outputting coverage KPIs and evidence for mismatches – essentially giving visibility into where the data is complete and where it isn't.

To ensure quality, we will maintain a **golden test set** of transactions and receipts (see Testing) to verify that our reconciliation logic correctly matches them and flags the intended mismatches. We will also log whenever a match is made or not made, to continuously evaluate if our thresholds are performing well in the wild.

User Interface Design – 3-Step Flow and Data Review

The frontend will guide the user through a **three-step process** when linking a new data source: **Link → Preview → Confirm**. The design focuses on clarity, giving users insight into what data will be imported, highlighting deduplications and differences, and ensuring explicit user consent at the final step before data is saved.

Step 1: Link Account – The user begins by choosing a provider (e.g. "Connect with Belvo" or "Connect with Pluggy", or "Upload CSV" or "Connect Gmail"). This triggers the appropriate **start endpoint** on our side (see API Routes below). For Open Finance (Pluggy/Belvo), we embed the provider's **connect widget** in a modal. For example, the user will be prompted to enter their CPF/CNPJ (per Open Finance spec) [85], select their bank, and then complete the bank's OAuth login in a popup [17]. This is handled by the aggregator's widget – e.g., Pluggy Connect or Belvo Widget – which then closes and returns control to us once the bank consent is done [18]. We detect that the item/link was created (via a webhook or a callback event from the widget SDK) and proceed. If it's a **Google** or other OAuth that redirects, after the redirect the user returns to our app. If CSV, the user will have uploaded a file in this step. Essentially, Step 1 ends with our system having the data source credentials/tokens and typically having fetched the data (or at least kicked off fetching).

- In the UI, once the connection is authorized, we display a loading indicator like "Importing data..." while our back-end fetches accounts and transactions. We subscribe to webhooks or poll as needed to get that initial batch. For Belvo, since they notify via webhook when historical data is ready [86], we might wait for those events. For Pluggy, we might call `POST /items/{id}/update` and wait for `item/created`/`item/login_succeeded` events [87] [88]. During this time, the UI periodically checks (or the backend pushes via WebSockets) to see if data is available.

Step 2: Preview Data – Once data is fetched (or partially fetched), we present a **preview screen** to the user. This is a crucial step where the user can review what will be added or changed before finalizing. The preview includes:

- A list of **accounts** that were retrieved. For each account, we show the institution name, account type, and maybe the current balance. If an account is new to our system, we might label it "New account detected". If it appears to duplicate an existing account the user already had, we indicate that ("Already linked account – will merge data"). In case of a conflict (say the user connected the same bank via two aggregators unintentionally), we'd highlight it here.

- A summary of **transactions** to be imported. Likely this is grouped by account. For each account, we show recent transactions or a count of new transactions found. We specifically call out how many of these are **new vs. duplicates vs. updates**:

    - **New Transactions**: those that our system did not have before (brand new entries that will be added). We can list them perhaps with a green "+" icon.
    - **Duplicates**: transactions that match ones already in our system (so they won't be added as new, but perhaps their record will be enriched). We might not list every duplicate unless the user wants details, but we should indicate "X transactions were recognized as already existing and will not be duplicated". If we allow, the user could expand to see which those are, with perhaps a diff of any updated fields.
    - **Updates**: in some cases, a duplicate might lead to an update (e.g. we had a pending placeholder from an email, and now we have final data). We can show these as "Matched with existing – will update details". A "diff view" would highlight changes like date changed by +1 day, description changed from "Pending Payment" to "STARBUCKS 1234", status from Pending to Posted, etc. We can present this in a side-by-side or inline diff format. For example, "Transaction on 09/26 R$8.25 – now confirmed with bank data" could be the message, with ability to view details.

The preview essentially gives the user a **before/after glance** at their ledger: new entries vs already present ones. This is important in case the user imported a CSV they had partially manually entered data for – they can see if it's going to create duplicates or not.

- **Dedupe State Indicators:** Each listed transaction in the preview can have an icon or badge indicating its dedupe status. For example, a duplicate/match might have a link icon or an = sign showing it ties to an existing one, and new ones a + sign. We can also use color coding (grey for duplicates, normal for new). We will also show **provenance badges** next to each transaction: e.g. a bank icon meaning this data comes from the bank feed, an email icon if it's backed by a receipt. In preview, initially, transactions coming from aggregator will have a bank/aggregator badge. If any of them match with an email we already had, we could even show both badges at this point (meaning "we found it matches an email receipt you had"). Conversely, if we just imported receipts, those may show an email icon, and if matched to a transaction already in system, also a bank icon. This gives the user immediate visual confirmation of reconciliation happening.

- For CSV uploads, the preview would show how many records will be added. If the CSV included some rows that were duplicates of existing ones, we'd show them similarly – maybe grouped under "Already in system (to be skipped)".

- We also include any **error or attention messages**. For example, if some accounts failed to import (maybe one account in the bank had no consent to share transactions), we'd warn here. Or if the aggregator couldn't fetch some older transactions due to API limits, we note that as partial data. Additionally, if any ambiguous duplicates or matches required user input, we would prompt here: e.g. "We found two transactions that could match the receipt from Amazon on 2025-09-01. Please select the correct one:" with a small UI to pick. This manual resolution step can be part of the preview. The user's choice (or if they skip, we keep it pending) then influences the final outcome.

Essentially, the Preview is the **diff view** of changes. We could present it somewhat like a **git diff** or review screen: - New Accounts: [list] - New Transactions: [count per account, option to expand list] - Matched Transactions: [list of those matched, possibly with minor updates] - Duplicates Skipped: [count, with option to view] - Possibly a summary: “Total new transactions that will be added: X”.

We also show provenance here – e.g. if a transaction is matched between email and bank, perhaps label it “ Matched existing email receipt”. If a new transaction has no receipt, maybe an icon “ No receipt found” to encourage user to provide one later (just informational).

Step 3: Confirm – After reviewing, the user clicks “Confirm Import” (or “Save”) to finalize. Only at this point do we commit the new data to the main database (if we had been staging it separately). Upon confirmation, the system will: - Insert new records into the production tables (or mark staged ones as active). - Merge duplicates as already determined (those merges ideally already happened in staging). - Execute any post-insert logic like recalculating account balances (though in read-only context, balances came from source). - Begin normal monitoring (webhooks) for these new connections going forward.

After confirmation, the UI might show a success message like “Bank ABC connected! 24 new transactions imported. 3 receipts matched.” Possibly we present the coverage KPI: “You have receipts for 22 of 24 transactions” for instance, highlighting any unmatched that user might want to manually handle. The accounts and transactions then become visible in the regular dashboard of the app.

If the user cancels at the preview step (or closes the flow), we will **discard the fetched data** (or keep it staged but inactive, to be safe, then purge later). We do not write anything to the main ledger without confirmation to respect user control. The consent with the aggregator would still exist (unless they cancel earlier in flow). If they cancel, we might offer to delete the link or leave it (some apps, once you connect, just don’t import anything if you cancel, but the connection remains authorized. We likely keep the connection and allow user to retry import, as the consent is already given).

UI Considerations: - We ensure the UI is responsive and clear, using concise labels: e.g. a transaction entry might look like:
`[Bank Icon] 2025-09-26 – R$8.25 – Starbucks – *New*` or
`[Link Icon] 2025-09-26 – R$8.25 – Starbucks – Matched to existing (Receipt ✓)`
with perhaps a hover or expand to see details.

- **Diff view for updates:** If an existing record is updated (like pending -> posted), we could show old vs new values. Possibly a small tooltip or modal when clicking the item: “Date was 2025-09-25 (from email) now updated to 2025-09-26 (posted). Description was ‘Pending Starbucks’ updated to ‘STARBUCKS #1234’.” These changes are likely minimal but important to convey.

- **Provenance badges:** Small icons or labels next to each transaction indicating sources: e.g. “Pluggy” or the bank’s logo, “Belvo”, “CSV”, “Email”. We might use logos for banks or generic icons (like a cloud for aggregator, file for CSV, envelope for email). Since the user may not know what Pluggy/Belvo are (they just know they connected their bank), showing the bank name/logo is more user-friendly. So for aggregator provenance we can use the institution name (which we have). For an email-sourced transaction, maybe an email icon and text “added from receipt”. We ensure not to clutter, but give an ability to see provenance. Perhaps clicking on the badges could show the actual sources (like “Data from Nubank via Pluggy API. Receipt from Gmail message ‘Your Nubank receipt…’”).

- We also integrate **consent information** into the UI. For example, on the accounts list, it might show "Connected via Open Finance – consent valid until 2024-06-11" if we know expiry [26]. And maybe a "Revoke" button.

- Throughout, we abide by a simple and reassuring UX: The user at confirm knows exactly what's happening – no duplicates, no surprises. This addresses trust issues by being transparent.

Edge case UI flows: If linking fails (e.g. user's bank credentials fail or aggregator returns an error), we handle that gracefully (error message and option to retry). If the data fetched is huge (say 1000+ transactions), we won't list all individually in preview (we might summarize or paginate). The user can still confirm without scrolling through all, trusting our dedupe. But we ensure counts are correct and perhaps allow downloading a summary report if needed.

Implementation Plan (TypeScript & Next.js)

The application is structured as a monorepo with a dedicated **connectors package** (`/packages/connectors`) implementing the logic described. We also have Next.js API routes under `/pages/api/connectors/*` that handle HTTP requests for the linking workflow and incoming webhooks. The core data models and services (deduplication, reconciliation) are implemented in a **unified module** (e.g. `/packages/core` for models and `/packages/services` for logic).

Connectors Package

In `/packages/connectors`, we have modules for each provider: `pluggy.ts`, `belvo.ts`, `gmail.ts`, `csv.ts`. Each exports functions or classes to manage that provider. We also export a factory or registry so the API routes can get the correct connector instance by name. For example:

```
// pseudo-code for connectors index
const connectors = {
  pluggy: new PluggyConnector(config.pluggyApiKey, config.pluggySecret),
  belvo: new BelvoConnector(config.belvoKey, config.belvoSecret),
  gmail: new GmailConnector(config.googleOAuthClient),
  csv: new CsvConnector()
};
export function getConnector(name: string): FinancialConnector {
  return connectors[name];
}
```

PluggyConnector: uses Pluggy's Node/TS SDK if available, or direct REST calls. It handles creating connect tokens (`createConnectToken()`), retrieving items (`getItemStatus()`), updating items (`updateItem()`), and fetching data (`fetchAccounts()`, `fetchTransactions()`). It will also implement `handleWebhook(event)` to parse incoming webhook JSON from Pluggy (which might be of type `item/created`, `item/error`, `transactions/created`, etc.) [89] [90]. For example, if `event.event === 'transactions/created'`, we will call `fetchTransactions(createdAtFrom = event.transactionsCreatedAtFrom)` using the provided URL [4]. We will likely use the `createdTransactionsLink` URL given by Pluggy webhook to fetch all new transactions in one go [3]. The connector then returns those raw transactions. The raw data is converted to our unified models (using mappers that map Pluggy's schema to our UnifiedTransaction schema). Similarly for accounts. We also record the Pluggy Item ID in ConsentRecord and externalIds.

BelvoConnector: uses Belvo's SDK or REST API. It will create a widget token (`createAccessToken(cpf, scopes)`) and perhaps handle the OAuth callback if Belvo needed one (Belvo's widget actually does the redirect internally and then calls our webhook with link creation events, so we might not have a direct callback). We handle Belvo webhooks: for instance, a `historical_update (TRANSACTIONS)` event indicates initial data loaded [86], and a `new_transactions_available` indicates incremental update [91]. In each case, we call the Belvo API to fetch the relevant data (Belvo might not include data in webhook, just the link id and which resource updated). BelvoConnector's `handleWebhook` will accordingly call `GET /transactions?link={linkId}` (with filters if needed, maybe Belvo only gives new ones if we ask for recent). Belvo's API also provides `GET /accounts?link={id}`, etc., which we use on initial load. The connector then maps Belvo's data (which is likely already in a normalized form since Belvo returns JSON with fields similar to ours – owner info, account balances, transaction fields like date, amount, description, category etc. per their docs). We ensure to capture the Belvo `linkId` and `id` for each transaction (if provided; Belvo likely has an internal transaction ID or we can composite linkId+index).

GmailConnector: uses Google's Node API or REST (we might have a GmailService in user's Google account). After OAuth, we subscribe to push notifications via Pub/Sub (or use Gmail's `watch` API). The connector will handle incoming Pub/Sub messages (delivered to our webhook endpoint likely) and then fetch new emails via Gmail API (as outlined in Gmail Hub doc). We then parse emails (which may involve our parsing rules/ML). Parsed receipts become ReceiptRecords. The GmailConnector also can perform an initial historical sync: it will use Gmail API to list messages (possibly filtering by queries like `label:inbox subject:receipt OR invoice` as suggested [11]) and batch retrieve them. We'll have to be mindful of API quotas and maybe spread the load. Parsed results are then forwarded to the reconciliation engine.

CsvConnector: fairly straightforward – it takes an uploaded CSV (provided via the API route, likely parsed with a library like PapaParse or similar on backend), maps columns to our schema (we might support specific formats or a user mapping step if needed), and outputs a list of transactions. If the CSV includes account info, we'll either expect one file per account or have an account column to group by. We may need the user's help to identify which account the CSV belongs to if not obvious (maybe file name or the user selects during upload). For now, we assume either separate file per account or that the CSV has an account name we can match to an existing account or create new.

Unified Model Mappers: For each connector, mapping functions like `toUnifiedAccount(rawAccount)` and `toUnifiedTransaction(rawTxn, accountId)` will produce our unified objects. They also compute the `hash` for transactions (taking into account if a receipt placeholder exists to possibly align date – but that is advanced; we likely compute straightforwardly). They attach initial provenance (for aggregator, just one source entry; for Gmail receipts, source entry with email details).

Dedupe & Recon Integration: Connectors will not themselves decide deduplication beyond not duplicating known same IDs; instead, after connector returns data, we pass that data to a **Dedupe Service** in our core logic. For example, `PluggyConnector.handleWebhook` might fetch new transactions and then call `Deduplicator.mergeTransactions(accountId, transactions[])` to handle merging into DB. That service will use our DB to find existing hashes and either insert or merge accordingly. Similarly, after GmailConnector parses receipts, it would call `Reconciler.matchReceipts(receipts[])` to attempt matching with existing transactions. Or we could unify both via a pipeline: all new records (accounts, transactions, receipts) funnel through a **Staging Pipeline** that does dedupe and recon, producing a final list of changes to apply. Given the complexity, we might stage data first, then run a separate process to dedupe & recon and produce a diff for the preview.

Next.js API Routes (Open Finance linking workflow)

We implement several API endpoints to handle the user-driven linking process and incoming callbacks:

- `POST /api/connectors/start` – Initiates the connection flow for a given provider. Expected input: `{ provider: string, institutionId?: string, ...}`. For **Pluggy** Open Finance, for example, the client might provide the connector/institution id (if user selected a specific bank). We then use `connectors.pluggy.createConnectToken({institutionId, webhookUrl, clientUserId})` which returns a token and a URL (or just token). We respond with `{ connectToken: "...", aggregator: "pluggy"}`. The front-end will then use Pluggy's SDK: e.g. `PluggyConnect.open(token)` to launch the widget [92]. Alternatively, if using redirect (not widget), we might return a URL for front-end to redirect the user to. Pluggy's newer OAuth flow provides a URL that we can have the user open [93], but given they have a widget, we'll likely stick to that. For **Belvo**, we call `connectors.belvo.createAccessToken(cpf, scopes)` and get an access token, respond with it or embed it in a short HTML that auto-initializes the widget. Belvo's front-end integration is similar – include their script, call `BelvoSDK.createWidget(accessToken)` which opens the UI. (We might also need to pass a `redirectUrl` for mobile as per Belvo docs, but essentially it's handled in widget config). For **Gmail**, calling start might not be necessary if we can directly redirect. Possibly we don't use an API route but just do `window.location = googleAuthUrl` from the client. But we can have `GET /api/connectors/start?provider=gmail` that sets up the OAuth client and returns a `url` to Google's consent page. We'd then do `res.redirect(url)` so the user's browser goes to Google. For **CSV**, there is no external OAuth; the user will upload a file via a form to e.g. `POST /api/connectors/start?provider=csv` (or perhaps a dedicated `/api/connectors/csv-upload`). That route will accept the file (Next.js can parse multipart or we use an API route with `FormData`). We then parse the CSV server-side, store the parsed data in a temporary store (session or DB table), and return an acknowledgement. Then the front-end can proceed to preview (maybe we redirect user to a preview page for CSV). In simpler terms, CSV flow might bypass a distinct preview fetch if we can attach it to the user session or temp DB.

- `GET /api/connectors/callback` – Handles OAuth callbacks from providers (like Google, potentially not needed for Pluggy/Belvo since they use widget). For Google, it will receive `code` and `state`. We verify state (stored in session or a nonce store), then call Google token endpoint to exchange code for tokens. On success, we store tokens (perhaps in a `Credentials` table or secure store keyed by user and source). Then we initiate the Gmail ingestion: we could either directly start a background job to fetch emails, or simply mark the connection as established and let the background worker or webhook handle it. Possibly we trigger an initial pull here (to have something to show in preview). We then redirect the user to the next step – maybe a Next.js page like `/connectors/preview?provider=gmail`. Alternatively, our callback could render a small HTML that posts a message to the opening window to close it (if we popped out) or just closes itself and signals the main app. But since we can integrate Google in the same tab with redirect, we may prefer to redirect user back to a dedicated page.

For aggregator widgets, typically after consent the widget closes itself (Pluggy's returns control, Belvo's might require an intermediate but presumably similar). We might not get an HTTP hit on our side except webhooks. However, Pluggy's OAuth v2 flow (non-widget mode) would involve a redirect from bank back to a Pluggy URL which then passes a code to our system via the Item, but since we are using the widget, it's abstracted.

If we did use a direct OAuth (imagine our own aggregator-less approach), then callback would handle it similarly: get auth code from bank, exchange for token, etc. But that's out of scope because we rely on aggregator.

- `GET /api/connectors/preview` – Fetches the staged data to show in preview. This might query a staging table or in-memory cache where we stored the results of initial fetch. Implementation depends: we might store the fetched accounts/transactions in a temp DB table keyed by a "session id" or user id + connection id. Or even keep them in the user's session (but that might be too heavy if thousands of transactions). Likely better to store in a temporary SQL table or as JSON in Redis with an expiry. This endpoint would then retrieve that data and send to client as JSON for rendering the preview. Alternatively, we might pre-render a Next page with the preview (SSR). But doing it via an API gives flexibility for the frontend (especially if using a React component). The response could be structured like: `{ accounts: [...], newTransactions: [...], duplicateTransactions: [...], updatedTransactions: [...] }`. Or we let the front-end calculate categories like new/duplicate if we simply label each transaction object with a flag. Possibly easier: each UnifiedTransaction might carry a flag `status: 'new'|'duplicate'|'updated'`. Then UI groups them. The preview API could also deliver counts and highlights.

If the user triggered linking multiple sources at once, we might allow previewing combined changes, but usually one at a time.

- `POST /api/connectors/confirm` – Commits the changes. The request would include identifying which staged import to confirm (e.g. `provider: belvo, linkId: X` or some import session ID). The server then moves data from staging to permanent. If our design uses the same database tables for staging (with a flag) and final, this might just be an update query (e.g. set `confirmed=true` on all staged records for that session). If we had them separate, we'd do inserts into main tables. Also, at confirm, we may finalize consent records (mark them active and tie to user account fully). We may also trigger initial background sync scheduling (like schedule daily updates for that link, subscribe to aggregator webhooks if not done). After processing, respond with success or any post-import info. The front-end can then show final status or redirect to main dashboard.

- `POST /api/connectors/webhook` – This would be the endpoint (or multiple endpoints) that aggregators call to deliver events (and Gmail's Pub/Sub would call a separate endpoint likely for email). We might set up routes like `/api/webhook/pluggy` and `/api/webhook/belvo` to segregate them, or a single endpoint that inspects a header/secret to determine origin. Pluggy allows setting a webhook URL per item or globally, and includes an `X-Webhook-Secret` if configured. Belvo similarly will sign their webhooks. We will verify these to ensure authenticity. The webhook handler will parse the JSON payload. For Pluggy, it may contain `itemId`, `event` etc. We hand this to the Connectors service: e.g. `connectors.pluggy.handleWebhook(payload)`. That function (as described earlier) fetches data if needed and then calls dedupe & upsert. We should ensure webhooks respond quickly (within say < 10 seconds) to avoid aggregator timeouts. So heavy lifting (fetching large data sets) might be offloaded to a background job: the webhook could just enqueue a job (e.g. via Bull or a simple in-memory task queue) and respond 200 immediately. Our background worker (which could be another Next.js API or a separate process subscribed to the queue) then does the actual `fetchTransactions` call and DB writes. This avoids timing issues and allows scaling (webhooks can come concurrently for many events, better processed in workers). We will set appropriate concurrency and perhaps locking to avoid race (like if a polling job and webhook job try to update same account simultaneously – covered in Testing).

Additionally, Google's Pub/Sub for Gmail doesn't directly hit our Next API; it goes through a Google Pub/Sub subscription. We might have to use a push subscription which does send an HTTP request. If so, we'll have an endpoint for that (with their JWT verification). On receiving a Gmail `historyId`, we call Gmail to get new messages and process them similarly, then ack the message. If using pull, we might run a separate worker to poll Pub/Sub. For design, we can assume push, which is handled similar to aggregator webhooks.

Database Design and Upserts

We likely use a SQL database (PostgreSQL). Key tables: `consent_records`, `accounts`, `transactions`, `receipts`. `transactions` table stores unified transactions. We add unique constraints or indexes to enforce no duplicates (e.g. unique (account_id, hash) on transactions, unique (gmail_message_id) on raw emails, etc.). The **deduplication and insertion logic** is concentrated in a service that uses those constraints to merge records.

For example, to insert transactions, we might attempt a bulk insert; duplicates (by unique key) will violate constraint, so we catch that and instead perform an update. Alternatively, we do an `UPSERT (ON CONFLICT ... DO UPDATE)` with logic to update provenance and any null fields. If using Postgres, we could store provenance as JSONB array or a separate table `transaction_provenance(tx_id, source, source_ref)`. We might maintain receipts linking in a `receipt_id` column or separate join.

The **dedupe algorithm** can also run in application code: for each new transaction, check DB for existing hash. If found, merge in memory and update DB accordingly (which might be simpler for merging provenance arrays etc. than writing one monstrous SQL UPSERT). Given performance (not too high volume, even 10k transactions is fine to loop), we can do it in code with transactions.

Provenance storage: Could be a JSON field listing objects, or a separate relational structure. For queryability, separate table might be better (to query e.g. how many transactions came from Gmail vs aggregator). But JSON is simpler initially. We will ensure that provenance includes reference IDs in a human-readable form for audit (like storing Gmail message-id strings, etc.), as well as cross-refs like aggregator transaction IDs. This ensures if an aggregator re-sends something, we can detect it by checking those IDs too (though hash covers content, provenance covers actual source IDs).

Background Workers and Task Coordination

We implement background processing either via Next.js API routes triggered by events (some could be long running if using serverless, but maybe we deploy Next.js in a Node server mode, or use a separate worker service for heavy tasks). Possibly we use **BullMQ** (Redis-based queue) to enqueue tasks from the webhooks and have a worker consume them. The tasks include things like "fetch transactions for link X from provider Y" or "parse emails from historyId X". These workers execute the connector calls and call dedupe/upsert as described.

We also schedule recurring jobs for periodic sync. In a Next.js environment, we might not have built-in scheduler unless using something like Vercel cron or a separate cron job hitting an API endpoint. Alternatively, incorporate something like **Temporal** or simply rely on aggregator's periodic updates (Belvo auto daily updates with webhooks [94]). For safety, a daily cron that triggers `connectors.pluggy.updateAll()` (iterating all active items and calling update) can be set. We must be careful to not overlap with webhooks to avoid races.

Concurrency & Race Conditions: If a webhook for new transactions arrives exactly when our cron is also pulling transactions for the same account, we could double-fetch. Our dedupe (unique hash) ensures no duplicate final records, but it's inefficient. To mitigate, we can design a **locking mechanism** per account or link: e.g. mark account "sync in progress" in DB while syncing, or use a distributed lock in Redis keyed by link. If a webhook triggers, it tries to acquire lock – if already locked by poll, it can either skip or wait. Or simpler, we can decide to disable polling for providers that have reliable webhooks (like Belvo's recurring link webhooks) to reduce chance. But given not 100% guarantee, we code idempotently.

Testing Strategy

We will implement comprehensive tests to ensure all components work together and edge cases are handled:

Provider Stubs: We will create stub classes or use Nock to simulate responses from Pluggy and Belvo, so we can test our connector logic without hitting real APIs. For example, we'll stub `PluggyConnector.updateItem` to return a fixed set of transactions from a fixture, and simulate webhooks by calling our webhook handler with sample payloads. We'll have JSON samples akin to actual responses (perhaps derived from docs or sandbox). This allows testing flows fully offline.

Webhook vs Poll Race: We will write a test that simulates a scenario where a poll job and a webhook event for the same data occur at nearly the same time. For instance, in a test, spawn two asynchronous calls: one calls `fetchTransactions` (poll) and the other calls `handleWebhook` with the same new transaction. Both will attempt dedupe/upsert. We expect no duplicates and no conflicts. Our unique index on (account, hash) ensures if both try to insert the same, one will error – we catch and handle it by ignoring the second insert and maybe merging provenance. We can simulate the second one seeing the record already exists after the first commits, and ensure it then merges. The test will assert only one transaction ends up in the DB and it has both provenance entries (maybe indicating it came via both routes, though realistically if it's the same aggregator, we might not double-mark it – but if poll and webhook did, we could just note it happened but not duplicate the source entry).

Deduplication Collisions: We will craft a test dataset where two distinct transactions result in the same hash, to test our collision handling. E.g., two receipts: both for R$50 on 2025-09-10 on the same account (say one is Starbucks, one is Uber) – these would hash same if we only used date/account/amount. Our algorithm might incorrectly merge them if description not considered. In the test, we see if both come in as separate receipts, does our dedupe incorrectly drop one? Ideally, our code should treat them as separate because their descriptions differ significantly. If our current hash logic would merge them, that's a bug – the test helps refine the logic (maybe by including a portion of merchant in hash or by adding logic "if two receipts hash same but merchant similarity < 0.5, don't merge"). We ensure through tests that such scenarios result in two distinct transactions. Similarly test that identical data from different sources *does* merge. For example, feed a transaction via CSV and via aggregator with exact same date/amount/description – after processing both, database should have one record with provenance showing both 'csv' and 'pluggy'.

Reconciliation Golden Set: We prepare a controlled set of transactions and receipts and expected matches to verify the scoring and matching. For instance: - Transaction: 2025-09-01, $100.00, "Amazon Marketplace". - Receipt: 2025-09-01, $100.00, "Amazon.com Order #123". We expect a match. - Transaction: 2025-09-02, $50.00, "Uber BV". - Receipt: 2025-09-02, $50.00, "Uber trip". We expect match (fuzzy "Uber" ~ "Uber"). - Two transactions 2025-09-05, $20.00, "McDonald's" and "Shell Gas", one receipt 2025-09-05 $20 "Shell". Should match to Shell, and not to McD. - Receipt for $30 on 2025-09-10 (Starbucks), but transaction posted 2025-09-11 $30 "STARBUCKS 123". Expect a match (date off by one allowed). - Receipt for $15 on 2025-09-12, no corresponding transaction at all. We expect it to create a pending transaction and remain unmatched. - Transaction $200 on 2025-09-15 with no receipt. Expect it to remain without match.

We run our reconciliation on this dataset and assert: - The matched ones get linked (both sides have each other's IDs set). - Confidence scores are above threshold for those matched, and below for mismatched. Possibly test the actual score values if we expose them or log them. - The unmatched receipt resulted in a pending UnifiedTransaction with proper fields and a flag or status indicating unconfirmed. - The unmatched transaction remains without any linkedReceiptId. - KPIs computed (maybe run a function to compute coverage on this set: expecting X of Y matched = rate, etc.) meet expected values.

Unit tests for smaller functions: e.g., hash generation function given certain input strings, fuzzy match function on typical names ("Apple" vs "APL*Apple" -> high similarity, etc.), ensuring our normalization (removing accents, punctuation) works on Brazilian store names (maybe test "Café São Paulo" normalization).

Integration tests covering full flows: Simulate a user linking an account and an email and see end-to-end. For example: - Use stub aggregator that returns one account and two transactions. - Use stub email provider that returns one matching receipt for one of those transactions. - Run through link (call start which would generate a token, skip widget by directly calling what widget would do – perhaps directly call our webhook to simulate item created, then simulate calling our preview). - Or simply call the internal functions: fetch aggregator data, parse email, then run dedupe+recon. - Verify final state: 2 transactions in DB, one of them has receipt linked, one has no receipt, etc., and no duplicate.

Coverage & Quality Gates: We will measure our test coverage using a tool like Istanbul/NYC. Our CI pipeline will enforce ≥85% coverage across the connectors and reconciliation modules. We will also incorporate **gitleaks** or similar in CI to ensure no secrets (API keys, etc.) are committed – for example, we make sure aggregator keys are provided via environment variables (and we test that our config loader picks from env and that no test prints them). We include a test specifically to ensure DSAR export completeness: call an export function for a test user who has some consents, accounts, transactions, receipts, and verify the JSON (or zip of JSONs) contains all expected data (consent records with correct dates, all transactions, etc.) and no unexpected data from other users. This ensures our DSAR implementation is correct and that if user requests their data, we include provenance and source references for full transparency.

All tests will be run in CI on every push. The “0 duplicates” gate means we likely run a specific integration test with a known scenario of potential duplicates and assert the outcome has no duplicates. If duplicates were present, the test would fail. In practice, this is covered by our dedupe tests as described. We could formalize it by, for instance, after an import, querying the DB for any account where count of transactions vs count of distinct hashes differ – that would indicate duplicates. We might write a quick DB query in a test that asserts `COUNT(*) == COUNT(DISTINCT hash)` per account for test data. In production, we might also have a periodic sanity check on data to alert if duplicates ever slip in.

Operational Considerations & Runbooks

To maintain the system, we prepare runbooks for common scenarios:

- **Consent Expiry/Renewal:** We monitor consent expiration dates (if provided) and receive aggregator webhooks on expiry [29]. Our system logs an event like “Consent for Bank ABC expired on 2025-07-01”. The runbook for support: when a consent is near expiry (e.g. 30 days prior), the system emails the user a reminder to renew. If expired, the user sees in the UI that the connection is inactive (and lastSyncAt will be old). The user can re-initiate linking (via the same flow, which likely creates a new consent). The runbook steps:
- If user contacts support that data isn’t updating, check if consent expired (in ConsentRecord or via Belvo’s status).
- If yes, instruct user to reconnect via app (or if using Belvo’s renewal mode via MBP, let them do that).
- Ensure old consent is marked expired. If user renews, a new consent is created (with new ID and new expiration). Optionally, after renewal, merge the new link’s accounts with old as described under reconnect.

Because Belvo can handle renewal in the widget (it might prompt the user to renew an expired consent rather than creating a whole new link), our system should handle both cases (new link vs extended existing link). Pluggy’s docs indicate “by default no expiration, user can revoke anytime” [27], but if they did implement expiration or user sets one, treat similar to above. We also automate deletion of expired consents if appropriate: If aggregator recommends deleting expired links, we could do that after some time. Belvo says if user doesn’t renew, the consent remains but worthless; it relies on user to revoke or renew via portal, but they do notify us. We likely keep the data but mark it stale.

- **Consent Revocation:** If the user revokes consent via bank or portal, our next data fetch will fail with an error like “USER_AUTHORIZATION_NOT_GRANTED” (Pluggy) or Belvo will send a link error. The runbook:
- Mark the connection as revoked in our DB (`ConsentRecord.revokedAt = now`).
- Stop any scheduled jobs for that link.
- Communicate to user (e.g. show in app “Access revoked by user/bank, please reconnect if you wish to continue syncing”).

- Ensure no further processing for that link. If revocation was user-initiated in our app (via a "disconnect" button), then we actively call aggregator to delete link. That triggers aggregator to revoke consent and wipe their stored data for that user [32]. We then similarly update our records. The runbook from support perspective: If a user says "I removed the app's access in my bank," we verify the link's status (maybe try a small fetch or use aggregator's consents API). We then instruct them to reconnect if they want, or if they intended to revoke permanently, we could also remove the local data if they request (or just leave it read-only). We maintain audit logs of revocation events.

- **Reconnect Flow:** When a consent is expired or revoked, or an aggregator link breaks (maybe password change in screen-scraping context, though in Open Finance OAuth there is no password to update; but e.g. for Pluggy direct connectors outside OF, the user might need to re-auth – not our current focus, but mention for completeness). The user can use our UI to reconnect. We might allow them to click "Reconnect" next to a broken connection, which essentially initiates a new Connect flow for that institution. In many cases, the aggregator will create a new link with a new ID. We should detect that it corresponds to the same bank account(s). As discussed, we'll dedupe accounts by accountNumber. The runbook for an engineer: if we see duplicate accounts after a user reconnected (maybe our logic didn't merge them automatically because account numbers weren't exactly the same format), we might manually merge in DB (assign transactions from one to the other and delete duplicate account). But ideally automated. We will also consider that the new link might retrieve past transactions again. This could introduce duplicates if we aren't careful, but our dedupe by hash will catch transactions that overlap with already stored ones. E.g., user had data until July, consent expired for August, now they reconnect in September, aggregator might give last 3 months (including August which we missed). The transactions from August will be new to us (because we didn't have them), no duplicates. If aggregator repeats older transactions we already have (they often return last N months on initial sync), those from before July will hash-match existing and thus be skipped as dupes. So user's ledger won't double count. Our preview would show something like "We found X new transactions since your last sync (Aug-Sep) and skipped Y duplicates from earlier data." So reconnect flow results in partial import which our system handles normally.

- **Deduplication Overrides:** In case our deduplication erroneously merges two different transactions, or keeps two separate when they are same, we need a manual remedy. This likely requires an admin interface or at least DB operations:

- If two transactions were merged but shouldn't be (false positive duplicate): We would need to **split** them. Because we stored one unified entry with two provenance, to split, we'd create two entries (one per source). We might require manual DB editing to duplicate the row and adjust one's provenance to remove the wrong source. This is complex to automate. Instead, to reduce need, our matching criteria is strict. But if it happens, our support team would use internal tools to fix it.
- If two transactions remained separate but are actually the same (false negative duplicate): This is easier to fix – we can merge them by deleting one and adding its provenance to the other. Or mark one as duplicate of the other. If detected by user (they see two entries that look the same), they can contact support or we could build a UI feature "Merge these transactions" that essentially does what dedupe would have (assuming they confirm they are duplicates). This user-initiated merge could then update the records and maybe retrain our system logic if needed.

The runbook for support: if user says "I see duplicate transaction X", verify if amounts/dates same, likely cause. If obvious duplicate, we execute a merge (possibly via an admin API or direct SQL). And investigate why dedupe missed it to improve logic. Conversely, if user says a transaction is wrongly merged (e.g. "I had two different purchases of $50 on same day but app combined them"), we might split as described. This should be extremely rare if our logic accounts for description differences.

- **Provider Outages and Errors:** If Pluggy or Belvo have an outage (their status pages or our webhook events like `connector/status_updated` to OFFLINE [95]), our system might start failing to fetch data or webhooks might not come. The runbook:
- We have monitoring to detect if scheduled jobs consistently fail for a provider or if we receive a `connector/status_updated` event marking many connectors offline [95].
- We check aggregator status page (Pluggy Status, Belvo Status) to confirm the outage.
- If confirmed, we might temporarily pause polling (to not spam failing requests) and possibly surface a message in-app: e.g. "Data sync is temporarily unavailable due to an outage at Pluggy. We will retry automatically."
- We might queue a retry once the service is back.
- If only specific banks are down (maybe in Open Finance, a particular bank's APIs are down – aggregator might mark connector offline), we could inform affected users that their institution is currently offline. Pluggy's webhook tells us when connector goes OFFLINE or UNSTABLE [95], we can use that for targeted messaging (like show a warning icon next to that connection).

For Gmail, outages are rare; if Google API quota is exceeded or down, we log and show error to user if they attempt manual refresh. Similarly for CSV (no outage issue, but if a file is badly formatted, we show error feedback).

The runbook also covers how to catch up after an outage: e.g. aggregator was down for 1 day, so no webhooks came. When back up, likely our next scheduled poll or user manual refresh will pull the missing transactions. Because consents are still valid, data should still be accessible. We might implement a recovery job that runs after downtime to sync any gaps.

- **Data Subject Requests & Exports:** For DSAR (Data Subject Access Request), we have an admin tool or user self-service to export all their data. The system can compile a ZIP or JSON containing:
- ConsentRecords (with all fields) for that user,
- All accounts and transactions for that user,
- All receipts,
- Possibly raw emails if requested (though raw email content might be personal data too; LGPD would include that if considered user data, we should be able to provide copies of any stored emails or attachments).

- We also include provenance details so the user knows where each item came from (this is part of being transparent). This export should be easy since our data model keeps everything linked by userId. The runbook instructs: if a DSAR is received, run the export script (or press button in admin UI) and deliver the package to user securely within the legal timeframe. Our tests ensure export completeness (no missing linked records). We also have a deletion runbook (if user invokes right to erasure): that would involve deleting all data for user – accounts, transactions, receipts, consents (and optionally telling aggregator to delete links which revokes consents as done when user disconnects). We'll use a cascading delete in DB or explicit deletions. Ensure backups and logs are handled as legally required.

- **Scaling & Maintenance:** The connectors architecture allows adding new banks or switching aggregators if needed. For example, if we want to support another aggregator, we implement the

FinancialConnector for it and plug it in. Provider outages and new banks (the ecosystem might add new institutions) are handled mostly by aggregator. E.g., if a new bank is added, we don't change code, aggregator's `/connectors` list just grows. If aggregator changes API version, we update the SDK and our mappings (we monitor their changelogs). These maintenance tasks would be documented.

- **Security & Secrets:** We will use environment variables for aggregator API keys and OAuth client secrets. The runbook includes rotating these if needed (we could have them in a secrets manager, and our app picks up new ones on restart). The gitleaks CI ensures we don't accidentally commit them. If a secret leak is suspected, rotate keys on aggregator dashboard immediately and update env. Also, we handle user tokens carefully – e.g., Gmail refresh tokens are encrypted in DB. Also consider encryption at rest for any raw email content (since emails are sensitive). We already store raw emails as bytes; the Gmail doc suggests storing as BYTEA for fidelity [96]. We ensure that DB access is secure and audit-logged (as mentioned with Firestore rules in that doc, but for SQL we use role-based access or at least limit who can query production data).

- **Logging & Monitoring:** We will log key events: link created, data fetched, number of new transactions, any errors. We especially log deduplication actions (like "Merged transaction from CSV with existing ID 1234") so we can trace if something went wrong. Monitoring alerts on any unexpected conditions (like duplicate hash collision should ideally never happen unless manually forced, but if DB uniqueness constraint is violated not caught, that's a bug – we'd have an error log). We also want to monitor the reconciliation coverage over time: maybe produce a metric "current receipts coverage % for user's data" – if it drops or is low, maybe user might want to know or we consider adding other sources.

In conclusion, this architecture and plan provide a robust open-finance integration with full **OAuth security** (leveraging FAPI-compliant flows via aggregators) and **LGPD-compliant consent and data handling**, combined with advanced **deduplication** and **reconciliation** features to ensure data quality and traceability. Users can link their accounts easily and trust that the system will intelligently merge data from multiple sources, avoid duplicates, and attach supporting documents to their transactions for a complete financial picture. All major components are covered by thorough tests and monitoring, and operational guidelines are in place for maintenance and support.

Sources:

- Pluggy Open Finance integration and webhooks [56] [92]
- Belvo Open Finance data aggregation and consent management [24] [19]
- Brazilian Open Finance OAuth and consent regulations [26] [15]
- Gmail parsing and reconciliation strategies [57] [76] [47]
- Email ingestion and de-duplication (Gmail IDs for dedupe) [36] [53]

---

[1] [2] [19] [24] [25] [29] [30] [31] [32] [33] [39] [40] [91] Banking Aggregation Overview (Brazil)
https://developers.belvo.com/products/aggregation_brazil/aggregation-brazil-introduction

[3] [8] [9] [56] [87] [88] [89] [90] [95] Webhook
https://docs.pluggy.ai/docs/webhooks

4 5 10 41 42 43 44 45 46 Transaction
https://docs.pluggy.ai/docs/transactions

6 7 20 86 Extract Banking Data in Brazil (Widget)
https://developers.belvo.com/products/aggregation_brazil/aggregation-brazil-integration-widget

11 12 13 14 36 37 47 48 49 50 51 52 53 54 55 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 96 12-Gmail Hub_ Financial Email Intelligence Service – System Specification & Implementation Guide.pdf
file://file-RcdqU9pZbnLjajZ9Jk5uvL

15 16 26 34 Open Finance Brasil - Ozone
https://ozoneapi.com/the-global-open-data-tracker/library/open-finance-brasil/

17 18 85 92 93 Creating an Item
https://docs.pluggy.ai/docs/creating-an-item

21 Financial-grade APIs for Open Banking Brazil | Curity Identity Server
https://curity.io/open-banking-brazil/

22 23 27 28 38 Consents and expiration
https://docs.pluggy.ai/docs/consents

35 Belvo API Docs
https://developers.belvo.com/apis/belvoopenapispec

94 Introduction - Belvo Developer Portal
https://developers.belvo.com/developer_resources/resources-asynchronous-workflows