ChatGPT

Graph Viewer Integration Pack

Knowledge Coverage Matrix

This matrix maps key product areas from specification PDFs to the current implementation and identifies gaps to address:

- **Financial Intelligence Layer**:
- *Current:* Firestore schema implemented with multi-tenant isolation and a hash-chained audit trail for transactions.
- *Missing:* Real-time streaming of updates (e.g. live query listeners) and WebSocket support for pushing changes.

- **UI/UX (Dashboard, Calendar, Heatmap, Tokens)**:

- *Current:* 13 dashboard sections built with Next.js and Tailwind; uses OKLCH color tokens with APCA contrast validation; includes a Calendar View with user-configurable settings.
- *Missing:* Additional layer toggles (e.g. astronomy overlays, 13-Moon alternative calendar) and an APCA contrast CI gate to enforce color accessibility.

- **Tax Engine (Brazil)**:

- *Current:* Supports Brazil's IRPF progressive tax, Carnê-Leão, MEI DAS, ISS (São Paulo), and PIS/COFINS calculations.
- *Missing:* NFS-e (electronic service invoice) XML parsing, SPED integration (digital accounting records), and multi-jurisdiction tax API hookups for broader coverage.

- **Ingestion Pipeline**:

- *Current:* Worker classes handle file ingestion with a `BrazilianReceiptParser`, including provenance tracking and duplicate detection via `dedupeKey`.
- *Missing:* OCR integration for images/PDFs, use of Firebase Storage for uploads, a Pub/Sub mechanism for processing, and email (.eml/.msg) parsing support.

- **Bank/PSP Connectors**:

- *Current:* Data structures for Itaú, Nubank, C6 Bank, and basic PIX (instant payment) recognition are in place.

- *Missing:* OAuth2 flows for user authorization, live API calls to bank endpoints, and machine-learning for transaction reconciliation (matching transfers between accounts).

- **AI/Insights**:

- *Current:* UI components like an Explanation Panel, an Insight "Spotlight", and a Change-Set ledger for audit are implemented.
- *Missing:* True live agent reasoning in the app (AI-driven analysis in real time), proper formula rendering in explanations, and background “freshness” jobs to update insights/links.

- **Security & LGPD**:

- *Current:* LGPD (Brazilian data protection law) compliance framework is in place, including Portuguese-language UI and BRL currency formatting.
- *Missing:* A comprehensive security rules matrix test harness for Firestore rules, and a “secrets broker” in CI to manage secrets and prevent leaks.

- **Observability**:

- *Current:* Structured log placeholders exist (for consistent logging), but not fully wired.
- *Missing:* A Sentry DSN for error tracking, real metrics dashboards (e.g. performance, ingestion lag), and documented runbooks for on-call troubleshooting.

(Source anchors for this matrix include an Audit Disclosure Letter and the “OKLCH Edit Mode” design HTML for tokens.)

Documentation Structure (Docs-as-Code)

Project documentation is maintained alongside code, following a docs-as-code approach. The repository includes a structured `/docs` directory with modules, architectural decision records (ADRs), and guides:

```
/docs/
  INDEX.json
  /modules/
    1.12-calendar-view.md
    2.9-ingestion-pipeline.md
  /adr/
    ADR-0001-seats.md
    ADR-0002-hitl-change-sets.md
  /guides/
```

```
a11y-oklch-apca.md
br-tax-reference.md
```

- The `INDEX.json` file serves as a registry of all documentation files and their IDs. For example, it maps module IDs to file paths (e.g. module "1.12" -> `modules/1.12-calendar-view.md`). This helps the app or agents quickly lookup documentation content by ID.

- **Module specs** (in `/docs/modules`) correspond to features or sections of the app. Each module doc has a front matter block providing metadata like an ID, title, source PDFs, status, and acceptance criteria, followed by a **Behavior** description. For example, the `2.9-ingestion-pipeline.md` module might start with:

```
---
id: "2.9"
title: "Ingestion Pipeline"
sourcePdfs: ["Truth-Preserving Ingestion"]
status: "⚠ 80% architecture / missing OCR+Storage+PubSub"
acceptance:
  - "Upload→Preview ≤2min with bbox"
  - "Approve → 1 canonical tx"
  - "Re-upload → dedupe"
---
**Behavior**: (Description of how the ingestion pipeline works...)
```

This front matter indicates the spec is mostly complete architecturally but still missing OCR/Storage/PubSub integration (⚠ status). It also lists acceptance criteria such as upload-to-preview time, ensuring only one canonical transaction per receipt, and deduplication on re-upload.

- **Guides** (in `/docs/guides`) provide reference materials or best practices. For example, `a11y-oklch-apca.md` is a guide on **Color Tokens with OKLCH + APCA**. Its content might include guidance like: *"Use OKLCH lightness ramps (L: 0.98→0.12) to ensure APCA contrast ≥ 60 for body text, enforced via a CI gate."* Another guide `br-tax-reference.md` would detail Brazilian tax specifics.

- **Architectural Decision Records (ADRs)** (in `/docs/adr`) capture high-level decisions. For instance:

- `ADR-0001-seats.md` records the decision to **lock initial tech stack "seats"** (database, auth, AI, API, deploy) to specific choices, and notes that CI must verify dependencies and the app must display these choices.
- `ADR-0002-hitl-change-sets.md` proposes that any AI-driven changes to financial data produce a human-in-the-loop change set (with a hash-chained audit log). This is still in "Proposed" status, meaning pending approval.

The `INDEX.json` file ties it all together. For example:

```
{
  "modules": [
    {"id":"1.12","path":"modules/1.12-calendar-view.md"},
    {"id":"2.9","path":"modules/2.9-ingestion-pipeline.md"}
  ],
  "adr": [
    {"id":"ADR-0001","path":"adr/ADR-0001-seats.md"},
    {"id":"ADR-0002","path":"adr/ADR-0002-hitl-change-sets.md"}
  ],
  "guides": [
    {"id":"G-OKLCH-APCA","path":"guides/a11y-oklch-apca.md"},
    {"id":"G-TAX-BR","path":"guides/br-tax-reference.md"}
  ]
}
```

This ensures that agents or the app can programmatically find documentation by category and ID.

RAG Index for Documentation

To support AI agents with Retrieval-Augmented Generation, a **RAG index** of documentation is built at compile time. This index breaks all docs into chunks, making it easier to search relevant pieces of text.

- A `DocChunk` type defines the structure of each chunk in `rag-index.json`. For example:

```
export type DocChunk = {
  id: string;         // e.g. "2.9:modules/2.9-ingestion-pipeline.md:3"
  moduleId: string;   // e.g. "2.9"
  text: string;       // the content snippet
  tags: string[];     // e.g. ["ingestion","OCR","storage"]
  updatedAt: string;  // ISO timestamp of when it was indexed
};
```

- A build script (`scripts/build-rag-index.ts`) walks through the `/docs` directory, reads each Markdown file, and uses front matter plus content to create `entries` for the index. Key steps include:
- Using a library like `gray-matter` to parse the YAML front matter.
- Splitting each doc's content into ~800-character chunks for manageable retrieval pieces.
- Collecting any **tags** in the content (for example, hashtags like `#calendar` or `#APCA` in text become tags).
- Stamping each chunk with an `updatedAt` timestamp and a unique `id` (combining module ID, file path, and chunk index).

- The output is a JSON array of these chunks written to `docs/rag-index.json`. This file can then be loaded by the AI Orchestrator or search functions to quickly find relevant documentation segments when answering questions or making decisions.

By pre-building this index, the system accelerates AI agent access to knowledge and ensures up-to-date context is available for tasks like the Graph Viewer agent.

Tech Stack "Seats" Verification

The project defines fixed technology "seats" for major components, and includes proofs to ensure these choices remain in place:

- **Tech Stack Seats Config:** In `src/config/seats.ts`, the app declares the chosen stack:

```
export const SEATS = {
  database: 'firestore',
  auth: 'firebase',
  ai: 'orchestrator',
  api: 'trpc',
  deploy: 'vercel'
} as const;
```

This means the initial design uses **Firestore** as the database, **Firebase** for authentication, a custom **"Orchestrator"** for AI, **tRPC** for API endpoints, and **Vercel** for deployment.

- **CI Proof Script:** To enforce these choices, a script `scripts/seat-proof.js` checks for required dependencies. For example, it will fail the build if `firebase` is missing (since Firestore/Firebase Auth are expected) or if `@trpc/server` or `zod` (used by tRPC routers) are not in `package.json`. On a CI run, this script prints "SEAT-PROOF: ok" on success or a specific missing component error on failure.

- **Footer Banner:** The app includes a UI element (e.g. `AppSeatBanner.tsx`) that displays the current seats at runtime. It renders a small fixed footer bar showing something like: *"DB: Firestore · Auth: Firebase · AI: Orchestrator · API: tRPC · Deploy: Vercel"* in a subtle style. This is mainly for developers/stakeholders to quickly verify the running configuration.

- **GitHub Action:** A workflow (`.github/workflows/seat-proof.yml`) runs on each push to execute the seat-proof script in CI. This ensures that any divergence from the decided stack (e.g. someone tries to remove Firebase or switch the database without updating the ADR) is caught early in the CI process.

Together, these measures act as guardsrails, keeping the implementation aligned with the architectural decisions (as documented in ADR-0001).

Implementation Stubs and Integrations

Several parts of the system are specified but not fully implemented. Stub interfaces and guidelines are provided for these, so contributors can implement them consistently:

OCR, File Storage, and Pub/Sub Integration

Interfaces are defined to abstract these services:

- **OCR (Optical Character Recognition):** An interface `OCR_GCV` (for Google Cloud Vision) specifies a method to parse image buffers:

```
export interface OCR_GCV {
  parse(buf: Buffer, opts?: { languageHints?: string[] }): Promise<{
    text: string;
    blocks: Array<{ bbox: [number, number, number, number]; text: string; conf:
number }>;
  }>;
}
```

Implementation note: Use Google Cloud Vision API for OCR if possible, and fall back to an open-source OCR (like Tesseract) if GCV is not available. The parser should return full text and also an array of text blocks with their bounding boxes and confidence scores (useful for highlighting text location in images). Document in the code/docs when a fallback is used (for transparency).

- **Blob Storage (Firebase):** `BlobStore_Firebase` interface defines methods to `put` and `get` files:

```
export interface BlobStore_Firebase {
  put(path: string, data: Uint8Array, meta?: Record<string,string>): Promise<{
url: string, md5: string }>;
  get(path: string): Promise<Uint8Array>;
}
```

Implementation note: Backed by Firebase Storage, `put` should upload a file (returning its public URL and MD5 hash), and `get` should retrieve the file bytes. This will be used in the ingestion pipeline to store receipts/documents and retrieve them for parsing or display.

- **Pub/Sub Messaging:** `Bus_PubSub` provides a simple publish-subscribe event bus API:

```
export type BusEvent = { type: string; payload: unknown; at: string };
export interface Bus_PubSub {
  publish(topic: string, evt: BusEvent): Promise<void>;
  subscribe(topic: string, handler: (evt: BusEvent) => Promise<void>):
```

```
Promise<() => void>;
}
```

Implementation note: For MVP, this can be a lightweight wrapper around a Pub/Sub system (for instance, Firebase Cloud Messaging or even Firestore pubsub via a collection, or Redis if available). The `subscribe` method returns an unsubscribe function. In production, a more robust message broker might replace it, but the interface allows swapping implementations.

These stubs are currently just definitions. The development to-do (see below) includes implementing them and measuring their performance (e.g., recording OCR latency, storage throughput, Pub/Sub reliability).

NFS-e (Service Invoices) and Tax Integration

To extend the tax engine with service invoices and other compliance data:

- **NFS-e Parser Stub:** An interface for parsing Brazilian NFS-e XML is provided:

```
export interface NfseDoc {
  numero: string;
  prestadorCNPJ: string;
  tomadorCNPJ?: string;
  municipio: string;
  itens: Array<{
    desc: string;
    vUnit: number;
    qtd: number;
    trib: { iss?: number; pis?: number; cofins?: number };
  }>;
}
export interface NfseParser {
  parse(xml: string): Promise<NfseDoc>;
}
```

This models a service invoice (NFS-e) with fields for invoice number, provider CNPJ, customer CNPJ, city (municipio), and line items (each with description, unit value, quantity, and a nested object of taxes like ISS, PIS, COFINS if applicable).

Instruction: Start by implementing parsing for one pilot municipality schema (e.g., São Paulo city's NFS-e XML format). Different cities have slightly different XML structures for NFS-e; to avoid being overwhelmed, others can be marked as not implemented. If a user uploads an NFS-e from an unsupported city, the system should throw a clear "not implemented" error indicating that and perhaps log or link to documentation for that city's format. This way, support can be added iteratively, tracking which formats are pending.

- **ISS/SPED:** Similar approach for integrating ISS (service tax) beyond just calculation (like maybe reporting) and SPED (Brazil's digital bookkeeping). Likely use feature flags or stub out integration

points with meaningful error messages if those features are accessed before implementation. Provide links or references where a developer can find more info when implementing these.

OAuth and Connectors Configuration

Bank and PSP connectors will eventually use OAuth2 flows and external APIs. For flexibility, the project introduces *"seats"* for connectors as well, allowing a switch between **mock data** and **real API**:

- **Connectors Config:** In `src/config/connectors.ts`, an object defines which mode to use for bank and PSP connectors:

```
export const CONNECTORS = {
  bank: 'mock' as 'mock'|'open-finance',
  psp:  'mock' as 'mock'|'stripe'|'mp'
};
```

Here `'mock'` indicates using a local stub or fake data (good for development/demo), whereas `'open-finance'` would be a real Open Finance API integration for banks, and options like `'stripe'` or `'mp'` (MercadoPago perhaps) for payment service providers.

- **Toggles UI:** There will be an internal settings UI (for developers or admins) to switch these at runtime. This means a developer can flip a switch to test the real bank APIs (in a sandbox) or revert to mock data easily. This is helpful for testing the live OAuth flows without affecting the primary user experience.

- **OAuth2 Implementation:** For real connectors, implement the OAuth dance: e.g., for `'open-finance'` mode, redirect users to the bank's auth page, obtain tokens, and refresh them as needed. This might require setting up redirect URLs and secret management (which ties into the **secrets broker** mentioned in Security).

Security Rules Test Harness

Security is critical, given multi-tenant data. Firestore security rules will be thoroughly tested using a matrix of scenarios:

- A sample test (`tests/security/rules-matrix.test.ts`) uses Firebase's Rules Unit Testing library. For example:

```
const env = await initializeTestEnvironment({ projectId: 'demo' });
const alice = env.authenticatedContext('alice', { tenant: 't1' });
const bob   = env.authenticatedContext('bob', { tenant: 't2' });
await expect(
  bob.firestore().doc('tenants/t1/accounts/a1').get()
).rejects.toThrow();
```

This test initializes a mock Firestore with security rules loaded, then creates two user contexts: Alice in tenant "t1", and Bob in tenant "t2". It then asserts that Bob **cannot read** a document under tenant *t1* (which should be forbidden by rules). Similar tests will cover cross-tenant write attempts, role-based access, and public/own data rules.

- The idea is to create a comprehensive **matrix of access scenarios**, automating checks for each rule in the Security Rules Matrix document. This becomes a harness that can be run in CI. Additionally, any change in Firestore rules should trigger these tests to avoid regressions.

- **CI Secrets Broker:** Another security aspect is ensuring that secrets (API keys, service accounts) are not mistakenly committed. A "secrets broker in CI" refers to a process where CI can fetch needed secrets securely (for testing deployments) without storing them in the repo. It also implies CI should fail if any sensitive keys are found in the code (as a last line of defense).

Observability and Monitoring

The project aims for robust observability (tracking errors and performance):

- **Sentry Integration:** In `src/observability/sentry.ts`, Sentry is initialized at startup:

```
import * as Sentry from '@sentry/nextjs';
Sentry.init({ dsn: process.env.NEXT_PUBLIC_SENTRY_DSN });
export const testError = () =>
  Sentry.captureException(new Error('Observability smoke test'));
```

Here `NEXT_PUBLIC_SENTRY_DSN` is an env variable. The `testError()` function is a small utility to send a dummy exception, used to verify that Sentry is capturing errors (for example, calling it once after deploy as a smoke test).

- **Metrics Dashboards:** The plan is to set up dashboards (likely in a tool like DataDog, CloudWatch, or Grafana) for key metrics. For ingestion, metrics could include OCR queue latency, number of OCR errors, Pub/Sub retry counts, etc. By instrumenting code (or using GCP/Firebase's built-in metrics) and exporting to these dashboards, developers can monitor system health. Specific URLs like `/metrics/ingestion-lag` or `/metrics/ocr-errors` might display or serve these stats.

- **Runbooks:** The documentation will include runbooks for various subsystems. E.g., `docs/runbooks/ingestion.md` might outline what to do if the ingestion pipeline is slow or failing. A snippet might include:

- **Smoke test:** Upload a sample receipt and expect a preview within $\leq$2 minutes.

- **If fail:** Check the OCR queue for backlog, examine storage egress errors, and look at Pub/Sub retry logs.
- **Dashboards:** Refer to the ingestion-lag and ocr-errors metrics dashboards for anomalies.

These runbooks serve as guides for on-call engineers to quickly triage issues in production.

With these observability measures, the system not only logs and tracks issues but also guides developers in diagnosing them effectively.

Graph Viewer Integration – Research & Plan

One major extension in this project is an **interactive Graph Viewer** for financial data relationships. This will allow visualization of entities (accounts, transactions, people, categories) as nodes in a graph, with edges representing value flows (money movement) or relationships. The following sections outline the requirements, evaluate technology options, and propose an implementation plan.

Context and Requirements

The Graph Viewer is intended to feel like a native part of the personal finance app, helping users (especially in Brazil, given the locale) see relationships in their data. Key requirements and context considerations include:

- **Real-Time Updates:** The graph should support real-time updates. As new transactions or insights arrive, the visualization should update live (e.g., via Firestore listeners or WebSockets pushing changes).

- **User Interactivity:** Users must be able to interact with the graph. This means panning, zooming, dragging nodes, clicking for details, and possibly selecting subsets (box or lasso selection). While AI agents will auto-generate and update graphs, users can rearrange or explore them manually too.

- **Node Types and Hierarchies:** Nodes will represent financial entities such as **transactions, accounts, categories, and people**. In essence, nodes are "value containers" – primarily money accounts, but also other assets or entities. We need to handle **fungible vs. non-fungible assets**: e.g., money in a bank account is fungible (divisible and combinable), whereas a car or property is a single asset that isn't divisible for partial payments. The graph design should accommodate grouping accounts under a person or company. For example, if Person A has three bank accounts, the graph might show Person A as a cluster node that can be expanded to reveal the individual accounts. This toggleable detail is a planned feature to let users see summary vs. detailed views.

- **Offline Capability & Embedding:** It's extremely important that graphs can be exported or embedded into knowledge bases like Obsidian or Notion. Users might want to take a snapshot of their financial network graph and include it in a note. Thus, the Graph Viewer should support generating static images (PNG or SVG) and possibly work offline (viewing cached data without an internet connection). This requirement will likely be addressed in later phases (see "seats" below) given its complexity.

- **Progressive Enhancement:** Given the scope, development will be iterative. We will start with an MVP focusing on core functionality, then enhance scalability and advanced features in later "seats" (Scale, Enterprise). This ensures we can deliver value early and incorporate feedback, rather than attempting everything at once.

- **Persistent Graph Config:** The configuration/state of a graph (nodes, edges, styling, layout settings) should be stored in the database. This allows AI agents or other services to **edit the graph config in real-time** and have the UI update automatically. In practice, this could mean storing the graph JSON in Firestore and using onSnapshot to push updates to clients. It also means a user's graph (or an agent-generated graph) can be saved and reloaded later, or edited collaboratively between the AI and user.

Choosing a Graph Visualization Library

We evaluated popular web-based graph visualization libraries to find the best fit for our use case (personal finance graphs with potentially thousands of nodes, real-time updates, and rich interactivity):

- **Cytoscape.js:** A well-established library for graph visualization on the web. Cytoscape.js has a rich API, supports interactive features out-of-the-box (panning, zooming, click handlers, even box selection), and is thoroughly documented. It uses Canvas/WebGL under the hood for rendering, but much of its architecture is still tied to the DOM. This means it does not take advantage of web workers for layout or simulation – all computations happen on the main thread. For moderately sized graphs (hundreds of nodes), Cytoscape works well, but for very large graphs (thousands of elements) it can become sluggish unless features like hideEdgesOnViewport are used. It was originally created for biological network analysis, and one drawback is that its **DOM-dependent architecture doesn't support multithreading**, impacting performance on large data sets [1]. However, Cytoscape's ease of use and maturity make it a strong choice for the MVP when graphs are smaller.

- **Sigma.js:** A newer library focused on performance. Sigma.js leverages **WebGL for rendering**, which is significantly faster for large numbers of nodes and edges [2]. It delegates graph data management and algorithms to a companion library called Graphology. This separation means potentially heavy computations (like layout algorithms) can be offloaded or optimized. Sigma.js is highly performant and can handle graphs in the tens of thousands of nodes range with smooth interaction. The trade-off is that Sigma's documentation is less comprehensive, and the integration can be more complex (since you manage Graphology data structures and Sigma rendering separately). For our use case, Sigma would shine in the "Scale" scenario where performance with >10k nodes is needed. We will likely introduce Sigma in a later phase, once basic functionality is proven with Cytoscape. Sigma supports essential interactions and can be extended, but might require more custom code (for example, implementing certain layout or clustering algorithms using Graphology or custom shaders).

- **D3.js (Force-directed Graph):** D3 is the powerhouse of web visualizations and includes force layout capabilities (via d3-force) for network graphs. Its strength is flexibility – one can craft highly customized graphs. But this comes with a **steep learning curve** and a lot of manual work: D3 by itself doesn't automatically render a graph with all interactions; we would need to bind forces, handle tick updates, draw elements (possibly on SVG or Canvas), and implement zoom/drag behaviors. Essentially, using D3 for our graph means writing a custom mini-graph library. While D3 is extremely powerful and could be tuned for our exact needs, it's likely overkill here given the availability of high-level libraries. It's better suited if we had very unique visualization requirements that existing libraries couldn't handle. For standard node-link diagrams, Cytoscape or Sigma will get us further faster.

Additionally, we looked at integration capabilities: Both Cytoscape and Sigma can import/export graph data (Cytoscape has JSON formats, Sigma can interface with Graphology data or GraphJSON). Cytoscape can easily output PNG snapshots via its API (which is useful for the embedding requirement). Sigma being WebGL-based might need an offscreen canvas approach to export images, or use a separate library for that.

Decision: Use **Cytoscape.js for the MVP** implementation to leverage its quick setup and rich features. Plan to integrate **Sigma.js for scaling** up to large graphs in the future. Use D3.js only if needed for specific visualizations or fallback, but not as the primary engine.

Integration Architecture (Frontend, Backend, AI)

The Graph Viewer system will involve both front-end components and back-end/agent logic working in tandem:

- **Frontend (Next.js + TypeScript):** We will create a Graph Viewer React component that can take a graph config (see schema below) as a prop or fetch it from a backend. This component will initialize the chosen graph library (Cytoscape for MVP) and render the nodes/edges with styles from the config. The component handles user interactions:
- Panning/zooming (with sensible limits or inertia).
- Clicking or tapping nodes to perhaps show additional info (could trigger a side panel or tooltip with details, like account balance or person name).
- Dragging nodes: Cytoscape supports repositioning nodes; we can allow the user to adjust layout manually, but might also provide an auto-layout reset option.
- Selecting multiple nodes (Cytoscape supports box selection or we can add lasso selection if needed via extension).
- Toggling grouped nodes: For instance, if a node represents a person with multiple accounts, the UI could allow an expand/collapse action. Cytoscape supports compound nodes (parent-child relationships) which we could leverage for this grouping.

The Graph Viewer component will subscribe to real-time updates. If the graph config is stored in Firestore, we can use a Firestore listener (via a React hook) to get updates whenever an agent or backend modifies the graph. These updates might include adding a node (e.g., a new transaction or entity discovered) or changing an edge weight, etc., to reflect latest data or analysis.

The front-end is also responsible for **export features**. For MVP, we can use Cytoscape's built-in image export to let users snapshot the graph as PNG. We will also provide a way to export the graph data as a DOT file (Graphviz format) for import into other tools. This can be done either by constructing DOT from the JSON or using a library.

- **Backend / AI Orchestrator:** The heavy lifting of determining what nodes and edges appear on the graph can be done by an AI agent (or a set of cloud functions). The Orchestrator (our AI backend) will have access to user data and can generate relationships – for example, identify that "Account A" and "Account B" have transfers between them totaling X amount, or that "Person P" is the owner of those accounts. The orchestrator could run analyses (like fraud detection patterns, spending network, etc.) and then output a graph spec. We envision a flow like:
- Orchestrator writes/updates a **graph config JSON** in a Firestore document (or it could return it via an API call, but Firestore sync is simpler for real-time).

- The front-end Graph Viewer listens on that document. When it updates, the new config is loaded into the visualization.
- Alternatively, for user-requested updates (like the user toggles to expand a person node), the front-end could either handle it locally (by already having the data) or send a message (perhaps via Pub/Sub or a direct function call) to an agent to fetch more detail and update the config.

In terms of **backend roles**, we might not need a heavy dedicated backend for graph drawing (since we use front-end libs), but for the **Enterprise** case, if using Graphviz for layout, we could have a backend service (or WASM in front-end) generate layouts. Also, exporting to formats like GEXF (for Gephi) might be done in backend if it needs heavy computation.

- **AI Control Interface:** We will define how the AI agent can control the graph. Given the JSON schema (next section), the agent can decide to add a node by pushing a new JSON entry in the nodes array and similarly for edges. It might also set certain styling or layout options. We should ensure this interface is safe (the agent shouldn't break the UI by sending bad data). Using TypeScript types and JSON schema validation will help. Perhaps the orchestrator uses our `DocChunk` index to find relevant docs (like explaining evidence for a link) and attaches those as evidence links in the graph.

- **Real-Time Data Flow:** To reiterate, real-time updates likely rely on Firestore's ability to stream updates to clients. If needed (for non-Firestore data sources or more complex triggers), we might incorporate WebSockets or an events system (e.g., if a big change is processed in a backend job, it could notify the client to refresh). Given the rest of the app already uses Firestore for live updates, sticking to that for graph state is consistent.

- **Security & Privacy:** Since graphs can combine sensitive financial info from multiple sources, we must enforce that the data displayed respects the same multi-tenant isolation and privacy rules. If the graph agent tries to pull in data from another tenant or an unauthorized source, the backend should prevent it. Essentially, the graph should be just another view of the user's own data and insights derived from it, nothing extraneous.

- **Performance Considerations:** For MVP, performing layout and rendering on the client for maybe up to a few hundred nodes should be fine. As the data grows, we will need to be mindful of not overloading the client. Techniques include:

- Only load subgraphs relevant to the current view (if the overall graph is huge, an agent could partition it).
- Use clustering (group many small nodes into an "aggregate" node until user drills down).
- Switch to WebGL rendering (Sigma) when needed, as discussed.

- Possibly use Web Workers for physics/layout calculations if we stick with Cytoscape (it doesn't natively do that, but we could pre-calculate a layout on the backend or a worker).

- **Export/Embed Implementation:** In MVP, the user can manually click an "Export" button to get a PNG image or DOT file of the current graph. For Obsidian/Notion embedding, one idea is to allow the graph to be identified by an ID, and we generate a public URL to an image snapshot (maybe via a Cloud Function that renders the graph headlessly using the saved config). For offline, if Firestore

caching is enabled, the graph data might be available offline, and we ensure the visualization can still render the last known state without network.

Graph Configuration Schema

To enable a clear contract between the AI/back-end and the front-end Graph Viewer, we use a JSON schema for graph configs. A draft example (v1) is as follows:

```
{
  "$schema": "https://schema.lech.app/graph-config.v1.json",
  "meta": {
    "version": "1",
    "generatedBy": "agent://orchestrator",
    "at": "2025-09-27T00:00:00Z"
  },
  "nodes": [
    {
      "id": "acct:itau:123",
      "label": "Itaú Conta",
      "type": "account",
      "style": {
        "oklch": "oklch(70% 0.05 210)",
        "size": 8
      },
      "data": { "balance": 12345.67, "currency": "BRL" },
      "evidence": [
        { "href": "/tx/abc", "label": "ledger link" }
      ]
    }
  ],
  "edges": [
    {
      "id": "e1",
      "source": "acct:itau:123",
      "target": "cat:groceries",
      "type": "flow",
      "weight": 532.10,
      "style": {
        "width": 1.5,
        "oklch": "oklch(65% 0.07 150)"
      },
      "evidence": []
    }
  ],
  "layout": {
    "engine": "auto",
    "options": { "gravity": 0.6, "collision": true }
```

```
  },
  "interactions": {
    "pan": true,
    "zoom": true,
    "dragNodes": true,
    "select": "box"
  },
  "themes": {
    "light": { "bg": "oklch(98% 0 0)" },
    "dark":  { "bg": "oklch(18% 0.02 240)" }
  },
  "a11y": { "apcaTarget": 60 },
  "export": {
    "dot": true,
    "gexf": true,
    "snapshot": { "format": "png" }
  }
}
```

Key elements of the schema:

- **Nodes:** Each node has an `id` (unique), a display `label`, a `type` (e.g. "account", "transaction", "category", "person", etc.), and optional styling and data.
- *Styling:* We use OKLCH color values for consistency with our design tokens (e.g., `"oklch(70% 0.05 210)"` which encodes lightness, chroma, hue). The style can also include size, shapes, icons, etc. The use of OKLCH ensures any text on nodes meets our APCA contrast goals. The schema is designed such that our CI can verify color contrast if needed.
- *Data:* A free-form object for any extra data (like balance, currency in the example) – this can be used for tooltips or for the AI agent to compute on.

- *Evidence:* An array of references that explain or back up the node. For example, a node representing an "Insight" might have evidence links to transactions or documents that led to that insight. In the example, an account node links to a ledger page.

- **Edges:** Each edge has an `id`, a `source` and `target` (which match node IDs), a `type` (like "flow" for money transfer, "ownership", "relation", etc.), and possibly a `weight` or value (for flows, the amount of money that moved). Edges also have `style` (e.g., width/thickness and color, again using OKLCH). Edge `evidence` could link to something like a specific transaction that this edge summarizes, but it's optional. If an edge's evidence array is empty, it might just be a derived relationship.

- **Layout:** We specify the layout engine. `"engine": "auto"` means the front-end can choose the best engine (Cytoscape vs Sigma vs others) based on graph size or context. Other values could be `"sigma"`, `"cytoscape"`, `"graphviz"`, etc. The `options` can include physics parameters like gravity, whether to avoid node overlap (collision), spacing, etc. In Cytoscape, this might map to a specific layout algorithm config (like CoSE, or spread), whereas for Sigma (Graphology) it might tweak the force simulation.

- **Interactions:** A section to toggle which interactions are enabled. Here, panning and zooming are true, node dragging is true, and selection is set to "box" (meaning rectangular marquee selection; could also allow "lasso" or both). This helps the front-end know which UI controls to allow or expose.

- **Themes:** Defines differences for light vs dark mode. At minimum we set background color (in OKLCH) for each. The node/edge colors themselves can be chosen to work on either background (our OKLCH values should be chosen with APCA >= 60 contrast on both backgrounds if possible). Additional theming could include switching label colors (white text on dark, black on light, etc.), but that can also be derived from background automatically.

- **Accessibility (a11y):** We include an APCA contrast target. For example, `"apcaTarget": 60` indicates all text should have at least a 60 contrast score. This parameter could be used by a CI or by the agent to adjust colors if needed. It's a reminder that accessibility is built-in to design choices.

- **Export Options:** Indicates which exports are available: here DOT and GEXF (Graph Exchange XML Format) are true, meaning the user/agent can export to those formats. Also `snapshot: { format: "png" }` indicates the ability to get a PNG image snapshot. If more formats are added in future (SVG, PDF), they could appear here.

The schema is versioned so that if we make changes (like adding new fields), agents and front-end can validate and adjust accordingly. We will maintain this JSON schema (possibly publishing it at the given URL) so that external tools or even the Obsidian plugin can validate graph config files.

Notes: The style fields using OKLCH tie into our design system (we have an OKLCH + APCA guide as noted). By enforcing design tokens in the graph, we keep the graph looking consistent with the app (no wild colors off-brand). The evidence links are crucial for explainability – if an AI agent suggests a connection ("these two people are linked via a shared account"), the user should be able to click and see why the agent believes that (e.g., "shared account 1234 at Bank X"). This fosters trust in the AI-driven insights.

Implementation Plan: MVP, Scale, Enterprise Seats

Taking inspiration from our tech stack "seats", we outline a progressive roadmap for the Graph Viewer feature:

- **MVP Seat:** Implement the Graph Viewer using **Cytoscape.js** as the rendering engine. Focus on a smooth integration into our app's UI (matching Tailwind styles, dark/light themes, etc.). Support core interactions (pan, zoom, click, basic drag) and basic real-time update via Firestore. Provide an **export to DOT** (Graphviz) for basic interoperability. This MVP will likely handle on the order of up to ~500 nodes comfortably. It will allow us to gather user feedback on the usefulness of the visualization and UI/UX adjustments needed. At MVP, some advanced features like hierarchical toggling or fancy layouts might be simplified (e.g., use a basic force-directed layout initially).

- **Scale Seat:** As usage grows or for more complex graphs, integrate **Sigma.js** as an alternative engine. In this phase, we implement the logic for `layout.engine: "auto"` – the app can decide, for example, if the graph has more than X nodes or edges, to initialize a Sigma renderer instead of Cytoscape. Sigma will leverage WebGL to maintain performance with large datasets. We'll also refine real-time updates for scale (ensuring that adding a node doesn't re-simulate the entire layout

unnecessarily, etc.). Possibly introduce background workers for layout computation on big graphs (Graphology can run in a web worker). The UI might allow switching between an overview (Sigma for big picture) and a detailed view (Cytoscape for a specific subgraph) if needed. In this stage we also add more export options, e.g., enable GEXF export for power users who want to import data into Gephi or other tools.

- **Enterprise Seat:** Offer advanced, high-precision graph capabilities. This includes using **Graphviz** for layout on demand – for instance, for printing or embedding, a perfectly arranged directed graph might be needed (Graphviz excels at clear layouts for smaller static graphs, like decision trees or organizational charts). We can use a WebAssembly version of Graphviz in the browser or call a cloud function that returns an SVG layout for a given graph. Enterprise users might also want a full **Gephi integration**: since we plan GEXF export, they could export the graph and load into Gephi for deep analysis. In this phase, we could also add features like **timeline playback** (if the graph has a temporal aspect, show changes over time) or integration with our Insights engine (annotate graph with AI insights). We ensure that large organizations with many entities can still visualize their data by perhaps offering filtering tools (to slice the graph) and by ensuring front-end performance (maybe even cluster nodes that are off-screen). This seat solidifies the Graph Viewer as an analysis tool fit for heavy-duty use, beyond the basics.

Each of these phases (seats) builds on the previous. By MVP, we'll already have the JSON schema and basic agent integration; Scale and Enterprise mostly swap out or extend components behind the scenes while keeping the agent interface (the JSON config) consistent. This way, the AI agent doesn't need to know which library is rendering – it just produces the config and sets the `layout.engine` appropriately if it has a preference.

Additional Project Artifacts

Finally, a few additional pieces round out this integration pack:

- **ADR Examples:** We maintain Architecture Decision Records for transparency. For example, ADR-0001 ("seats") captured the decision to fix our stack as Firestore/Firebase/etc early on, which directly influenced the Graph Viewer approach (we'll store graphs in Firestore, use Firebase Auth to ensure security on who can see what graph). ADR-0002 on Human-in-the-Loop Change Sets, while not directly about the graph, will influence how any AI-suggested graph additions are approved (perhaps important for enterprise governance).

- **Developer TODO (Gap Tracking):** The team keeps a gap-focused TODO list, which aligns with this pack's content:

- **OCR/Storage/PubSub:** Implement the `OCR_GCV` class with Google Vision API, integrate Firebase Storage for uploads, and stand up a basic Pub/Sub (possibly using Firestore or Cloud PubSub) to connect ingestion steps. After implementation, measure and log latencies for each step for observability.
- **NFS-e/ISS/SPED:** Complete the NFS-e parser for São Paulo and define schemas for others, marking them unimplemented. Integrate ISS tax calculations and lay groundwork for SPED data export. Possibly use feature flags to enable these as they are finished.

- **OAuth/Connectors:** Build out the OAuth2 flow for bank connectors (initially test with one bank's sandbox). Use the `CONNECTORS` config to switch between mock data and real API calls. Eventually integrate with Brazil's Open Finance standards for broader support. Ensure secure handling of tokens (tie in with the secrets management).
- **Security:** Finalize the security rules test harness by covering all CRUD operations and roles. Integrate a secrets broker in the CI pipeline to fail builds if forbidden patterns (API keys, etc.) are found. Also, consider adding automated penetration testing or at least static code analysis for security.

- **Observability:** Plug in the Sentry DSN (once obtained) and test error reporting. Set up the metrics dashboards and alarms for critical issues (e.g., if OCR queue time exceeds X minutes, trigger an alert). Flesh out runbooks for all major subsystems (ingestion, connectors, AI, etc.), so on-call engineers have guidance.

- **Agent Handoff Checklist:** Before we consider this Graph Viewer feature (and associated agents) ready, we'll verify:

- Documentation is up-to-date: the `/docs/INDEX.json` and `rag-index.json` are generated and include the new graph docs or any changes.
- The Seats banner in the app reflects all expected components (including if we consider the graph engine a "seat", though currently it's more of a sub-component).
- CI is passing the seat-proof and security tests (green across the board).
- All stubbed interfaces compile without errors; unimplemented features gracefully throw a clear error with a message or link (rather than just crashing).
- The next-agent (Graph Viewer agent) brief is attached for the team, including the JSON schema and guidance on how the agent should reason about graphs.

With this integration pack, the team and any AI agents have a comprehensive view of what's in place and what needs building for the Graph Viewer and related subsystems. It sets the stage for development to proceed confidently, ensuring that design, documentation, and implementation are all aligned.

---

[1] [2] You Want a Fast, Easy-To-Use, and Popular Graph Visualization Tool? Pick Two!
https://memgraph.com/blog/you-want-a-fast-easy-to-use-and-popular-graph-visualization-tool