ChatGPT

Unified Asset Dashboard – Implementation Roadmap

Overview and Goals

The **Unified Asset Dashboard** is a comprehensive coin-specific dashboard that merges **market data** with **DeFi and on-chain metrics** into a single view. The goal is to provide a "single pane of glass" for each cryptocurrency: users can see price charts and market cap *alongside* fundamentals like Total Value Locked (TVL), active users, liquidity, and upcoming token unlocks. This unified view offers immediate context (e.g. *"Token X has a $100M market cap, $200M TVL, rising user count, but an unlock event in 30 days"*) which empowers better investment decisions. The dashboard will be built into your existing application (using Next.js) and presented in a **pop-up/modal style** (inspired by Banter Bubbles) so users can quickly open detailed coin info without leaving the main screen. All data will be **up-to-date** and aggregated in real-time from reliable sources. By combining these metrics, this feature will rival Messari's asset profiles and **outperform other market trackers** that only show basic price info.

Key Features & Data Sources

The Unified Asset Dashboard will display multiple facets of a token's data, aggregated from various APIs in real-time. Below are the core components and their data sources:

- **Price & Market Data** – Current price, price chart, market capitalization, 24h volume, and price change percentages. Sourced from the CoinGecko API (e.g. the `/coins/{id}` endpoint for detailed data). CoinGecko provides reliable real-time price and market data for thousands of coins [1] [2]. We can also leverage CoinGecko's `/coins/markets` or `/simple/price` endpoints for quick price updates, and `/coins/{id}/market_chart` for historical price charts.
- **Coin Category & Trends** – The coin's category or sector (e.g. *DeFi*, *Gaming*, *Layer-1*, etc.) and whether it's part of any trending lists. CoinGecko's API offers a `/coins/categories` endpoint to get the list of categories and their market stats [3]. We'll use this to label the asset's category and even filter or curate subsets of coins by category (addressing point #1: *"Curated subsets using their categories"*). Additionally, the CoinGecko `/search/trending` endpoint provides trending coins which we can highlight (e.g. indicating if this asset is currently "trending" on CoinGecko).
- **Total Value Locked (TVL)** – The project's TVL in DeFi (how much value is locked in its smart contracts). This will come from DefiLlama. Using DefiLlama's API (likely the Pro API for detailed data), we can fetch the protocol's TVL via an endpoint like `GET /api/tvl/{protocol}`. TVL gives a sense of how much capital the project commands – e.g. a DeFi token with high TVL relative to market cap might be undervalued, and vice versa. If the token doesn't have a DeFi protocol (e.g. it's a pure currency), this section can be omitted or show N/A.
- **Active Users / Addresses** – The number of active users or addresses interacting with the project, indicating adoption and usage. DefiLlama's Pro API provides **active user metrics** for protocols and chains. We can query an endpoint like `/api/userData/activeUsers/{protocolId}` (or a

similar method) to get recent active user counts for the project's protocol. This metric shows user growth or decline. For example, an increase in active addresses over time suggests growing network effect. (DefiLlama Pro includes active user data, which is typically not available in basic trackers [4].)

- **Token Liquidity** – An indicator of how liquid the asset is across exchanges and chains. This could include the total liquidity in top decentralized pools or overall trading liquidity. DefiLlama's **Token Liquidity** data (available via Pro API endpoints like `/api/historicalLiquidity/{token}`) can be used to show how easily the token can be traded (e.g. liquidity on DEXs, or largest pool info). We might display the largest liquidity pool (e.g. "Uniswap ETH-Token pool has $X liquidity") or a total sum of liquidity across major sources. This helps users gauge if the token has deep markets or if price can move easily due to low liquidity.
- **Upcoming Unlocks / Emissions** – Information on any scheduled token unlocks or vesting events. This is crucial for understanding sell-pressure risks. Using DefiLlama's **Unlocks** API, we can fetch unlock schedules. The endpoint `GET /api/emission/{protocol}` returns the vesting schedule for a given token (the token's slug can be determined via the `/api/emissions` list which provides available tokens and their `gecko_id`/slug). We will display the date and amount of the next unlock (and perhaps a small timeline of upcoming unlocks). For example: *"Next unlock: 5M tokens (10% of supply) on 2025-08-01."* This gives users a heads-up on potential dilution events. (Currently, such tokenomics data is hard to find on free platforms; Messari provides it only to Pro users [5].)
- **Additional Fundamentals** – We can include other stats for completeness: circulating supply vs total supply (from CoinGecko's data), fully diluted valuation (FDV), and perhaps protocol revenue or fees if relevant (DefiLlama's free API provides revenue/fee data for protocols). These give further context (e.g. FDV vs TVL ratio). In future iterations, we might also integrate **on-chain growth metrics** like transaction volumes, or **social metrics** (GitHub activity, Twitter followers) but these are optional.

All these data points will be fetched live when the user requests a coin's dashboard, ensuring the **most up-to-date information** is shown. By aggregating data from **CoinGecko** (market data) and **DefiLlama** (DeFi and on-chain data), we leverage reliable sources. Notably, DefiLlama's Pro API is needed for unlocks, active users, and liquidity categories [4] – these are advanced metrics that standard free APIs or competitors don't provide in one place.

Competitive Advantage – Why This Beats Competitors

This unified dashboard is designed to **outshine existing crypto tracking tools** by offering a broader scope of data in one convenient interface. Here's how it compares to and beats notable competitors:

- **Messari Profiles** – Messari's asset profile pages are rich in data (including token unlocks, financial details, etc.), but much of their advanced content (like detailed token unlock schedules or certain on-chain metrics) is gated behind a **paid subscription (Messari Pro)** [5]. Our Unified Dashboard will provide similar insights **for free and in real-time**, without requiring users to navigate through multiple tabs. Additionally, Messari's UI, while comprehensive, can be overwhelming – our version will distill the key actionable metrics (TVL, users, unlocks) in a straightforward view. Essentially, we're offering **Messari Pro-style data in a simplified, free dashboard**. This lowers the barrier for retail users to access data that was once available only to professionals.
- **Banter Bubbles** – Banter Bubbles presents crypto markets as interactive bubbles, emphasizing visual trends (price movers, market cap sizes, etc.). It's great for spotting momentum, but it doesn't offer deep fundamentals. For example, you might see Token A's price is up 20% (big green bubble), but Banter Bubbles won't tell you Token A's user base or if a token unlock is imminent. Our

dashboard complements that *visual trend* approach with a quick fundamental check: by clicking on a coin (bubble), the user could get a pop-up of our unified dashboard showing the coin's fundamentals (within the same app). This **"popup" style detail panel** (inspired by Banter's quick view) provides context that Banter Bubbles alone lacks. In short, Banter Bubbles shows *what's pumping*, but the Unified Asset Dashboard explains *why it's pumping and the risks involved* – giving users an edge.

- **Velo Data and Other Market Trackers** – Platforms like Velo (and similar market dashboards or charting tools) focus mostly on price charts, technical indicators, and sometimes exchange data. They often lack **integrated on-chain metrics**. Our solution outdoes these by **aggregating cross-domain data**: not just market prices and charts, but also DeFi metrics (TVL) and on-chain usage. For example, where a typical tracker might show price and volume, our dashboard will additionally show that the project's TVL is at an all-time high, or that active addresses have doubled this month – insights a price chart alone can't give. Moreover, most free trackers do not include token unlock info at all. By **combining off-chain market data and on-chain data in one view**, we provide a more *holistic picture* of the asset's health. This holistic view (price + fundamentals + on-chain) is typically only found on expensive platforms or through manual research.
- **CoinGecko/CoinMarketCap** – These popular trackers provide basic data like price, market cap, supply, and maybe some category tags or exploratory info. However, they **do not show on-chain metrics** like protocol TVL or active addresses on the coin page. For instance, CoinGecko might list a token's market cap and a link to its DeFiLlama page, but it doesn't embed the TVL or user stats. CoinMarketCap similarly lacks on-chain stats, and token unlock schedules are usually found in separate articles or not at all. Our unified dashboard **beats these by integrating on-chain context directly**. It turns a coin page from a simple price display into a mini analytical report (almost like a *free mini-Messari profile*). This level of insight is not available on CMC/CG without piecing together information from multiple sites.
- **Other Analytics Tools** – There are specialized tools (Glassnode, Nansen, Token Terminal, etc.) that offer on-chain or financial metrics, but they are either paid services or focus on specific areas (e.g., Nansen for whale wallets, Token Terminal for protocol revenues). None of them serve as a simple unified "coin dashboard" for the general user. Our solution targets the broad user base by being part of your app (easy access), free, and *actionable at a glance*. It's unique in aggregating metrics across price, DeFi, and tokenomics in one UI. **Few (if any) free tools currently aggregate all these data points in real-time in one place**, which gives us a strong competitive edge.

By excelling in these areas, the Unified Asset Dashboard will provide users with **immediate insight** into both the **market performance** and **fundamental health** of a token – something competitors make you hunt across multiple pages or pay subscriptions for. This unique value proposition can increase user engagement on your platform (since they can do more research without leaving) and position your app as a one-stop solution for crypto insights.

Technical Architecture & Approach

The dashboard will be implemented as part of your Next.js application, leveraging Next.js's capabilities for both server-side data fetching and a modern React-based UI. Below is an outline of the architecture and key technical decisions:

- **Next.js Integration**: We will create a new Next.js page (or set of pages/components) for the asset dashboard. Since your app is Next.js, we can either use a dynamic route (e.g. `/dashboard/[coinId].js`) for a coin detail page or a React component that can be rendered in a modal. Given

the requirement for a popup modal (point #2), the likely approach is to have a React component for the dashboard and use it within a modal context on whichever page triggers it. We can manage the modal visibility via React state or a global store (like using React Context or a state management library if the app is large). If using a dynamic route, we can still display it as a modal by layering it on the current page (Next.js can do route-as-modal patterns, or we simply mount a component).

- **Data Fetching Strategy**: To ensure the **"most up to date"** info, we should fetch data at request time. We have a couple of options in Next.js:
- Use `getServerSideProps` on the coin detail page route to fetch data from CoinGecko and DefiLlama APIs on each request. This ensures fresh data on each load and keeps API keys hidden (since the fetching happens server-side). The data is then passed to the page component.
- If using a popup component on an existing page (without navigation), we might fetch data on the client side when the component mounts or when a user selects a coin. In that case, we can use a combination of React hooks (`useEffect`) or a data fetching library like SWR or React Query to call our APIs. We might set up internal API routes (e.g. Next.js API route `/api/coinData/[id].js`) that act as proxies to the external APIs, so that we can safely store API keys on the backend and also potentially combine multiple external calls into one response. For example, an API route could take a coin ID, then internally fetch CoinGecko data and DefiLlama data, merge it, and return a JSON to the frontend. This approach can reduce latency (one client call instead of several) and keeps the client code simpler.
- We may also implement caching for API responses (to avoid hitting rate limits). Next.js API routes or `getServerSideProps` can cache data in memory for short durations or we could use the **Incremental Static Regeneration (ISR)** feature for a hybrid approach (although for truly real-time data, SSR or client fetch is better). Initially, simple SSR or on-demand fetch is fine given moderate traffic.
- **Mapping Coin to Protocol**: One technical aspect is linking a coin (as identified by CoinGecko or its symbol) to the corresponding DefiLlama protocol slug or ID (for TVL, users, unlocks). We will need a **mapping strategy**:
- DefiLlama's `/api/emissions` endpoint gives a list of all tokens with unlock data, including a `gecko_id` for each [6]. We can leverage that: e.g., find the entry where `gecko_id == coin's CoinGecko ID` to get the `protocol` slug for unlocks. Similarly, DefiLlama's general data (TVL, users) often uses the protocol slug or a protocol numeric ID. We can retrieve DefiLlama's master protocol list (their free API has `/v1/protocols` which lists protocols with their ids, names, and CoinGecko IDs). During the build of the dashboard, we might do a one-time fetch of this list and store a mapping (e.g. {gecko_id: protocolId}) in memory. Then for each coin, we know which protocol to query.
- If a coin doesn't have a corresponding DefiLlama entry (like a pure currency or a very new token), certain sections (TVL, unlocks, etc.) will simply be blank or skipped.
- Example: For **Aave**, CoinGecko ID is "aave". Using `/api/emissions` we find an entry with `"gecko_id": "aave"` which might have `"protocol": "aave"` or similar. We then use `/api/tvl/aave`, `/api/emission/aave`, `/api/userData/activeUsers/{protocolId_for_aave}` accordingly. We will handle these lookups under the hood, so the user/dev doesn't have to manually map them.
- **UI Components**: The dashboard UI will be divided into sections:
- **Header**: Coin name, symbol, and category tags. Possibly a small logo/icon (CoinGecko provides coin images URLs in their data).
- **Price & Chart Panel**: Prominently show the current price, 24h change, market cap, and a mini price chart. We can use a chart library (like **Chart.js** with react-chartjs, **Recharts**, or **visx**) to plot the price

history (e.g. last 7 days price). CoinGecko's `/coins/{id}/market_chart?vs_currency=USD&days=7` can give us data for plotting a 7-day price line. Alternatively, since this is a small chart, we might use a lightweight approach like sparklines or just an SVG path. But an interactive chart library adds polish (users can hover for exact values).

- **Key Metrics Panel**: A grid or list of important figures:
    - Market Cap, 24h Volume, Circulating Supply (from CG data).
    - TVL and maybe a TVL change (24h or 7d change if available – DefiLlama often provides current TVL and we could compute change from historical TVL endpoints).
    - Active Users (maybe show the latest daily active users count, and a 7d average or trend arrow if data available – e.g. "Active Users: 1,200 (+5% this week)").
    - Liquidity info (e.g. "DEX Liquidity: $X across Y pools" or "Largest pool: $Z liquidity on Uniswap").
    - Next Unlock info (e.g. "Next Unlock: Aug 1, 2025 – 5,000,000 tokens (10% of supply)"). Possibly color-code it if the unlock is soon or large.
    - Any other notable metric (like protocol revenue, P/E ratio if we compute it from market cap/revenue). These metrics will be displayed with clear labels and possibly tooltips explaining them. Short explanations can be included for context (for instance, hovering over TVL could show "Total value locked in this project's smart contracts").
- **Charts/Visualizations**: We can include mini charts for TVL and active users as well, if data is available. For example, a small line chart for TVL over the last 30 days (to show trend) and similarly a chart of daily active users over time. DefiLlama's API might allow pulling historical TVL (the free API `/v1/chain/<chain>/protocols/<protocol>` returns timeseries, or Pro API might directly give a timeseries). If not, we can call their standard API for historical data (DefiLlama's free endpoints for TVL timeseries are well-known). These charts would make the dashboard more insightful (spiking TVL or falling users would be immediately visible).
- **Modal/Popup Implementation**: The entire dashboard component can be rendered inside a modal dialog. We'll implement a modal overlay (using a React component). We can either use a ready-made modal from a UI library (like Material-UI's Modal, or a lightweight library like react-modal) or create a simple custom modal with a semi-transparent backdrop and a content card. The modal should be triggered when a user selects a coin (e.g., clicks on a coin name or a "Details" button). It should be dismissible (click outside or an [X] button). This approach is *"similar to Banter Bubbles"* where clicking a bubble shows a quick info panel. In our case, clicking an asset triggers the Next.js data fetch and displays the unified dashboard in a popup.
    - If performance is a concern (fetching data might take a moment), we can show a loading spinner in the modal while data loads. Alternatively, pre-fetch some data for popular coins to make it instant (Next.js can prefetch if using `<Link>` for routes, or we can prefetch via SWR for hovered items).
- **Integration into Current Application**: Since this will live within your existing app, we'll integrate seamlessly:
- If your app already lists coins or has a table of assets, we can add an action (like making the coin name clickable or an info icon) that opens the Unified Dashboard modal for that coin.
- If you have a sidebar or menu, we might add a "Unified Dashboard" entry where users can search for a coin and open its dashboard.
- The design (colors, fonts) will follow your app's styling to appear native. If you use a CSS framework or component library, we will use that for consistency.
- Next.js allows adding this without a full page refresh (if we do it as a component + API calls) so it will feel very fluid in-app.

- **Security & API Keys**: We will keep any sensitive API keys on the server side. CoinGecko's basic API is open (no key needed for most endpoints at a reasonable rate limit), and DefiLlama's Pro API requires an API key (the spec shows a long API URL that includes a key token). We'll store the DefiLlama API key as an environment variable (e.g. `DEFILLAMA_API_KEY`) in Replit's secret configuration or a `.env` file, and ensure it's not exposed in client-side code. All calls to DefiLlama will go through our backend (Next.js server functions) so the key remains safe.
- **Performance Considerations**: Combining multiple data sources means multiple API calls. To keep the dashboard snappy, we'll try to fetch in parallel where possible (e.g. trigger CoinGecko and DefiLlama requests concurrently in `getServerSideProps` or our API route). The expected payload sizes are small (JSON with a few fields, plus maybe arrays for charts), so network time is the main factor. If needed, we might introduce an in-memory cache (like a simple LRU cache) for extremely frequent requests (for example, caching a coin's data for 1 minute could drastically reduce redundant calls if users open the same dashboard often). Given the app context, a short caching is fine as data won't stale much in 1 minute.
- Also, consider rate limits: CoinGecko's free tier allows up to 50-100 calls/minute from an IP. DefiLlama Pro has high limits (1000/min) [7], so that's generous. We should still avoid unnecessary refreshes. If user opens multiple dashboards rapidly, our caching or making sure we don't double-fetch the same coin concurrently will help.
- **Docker & Deployment**: The app will be containerized for consistency between Replit and other environments. We will create a Docker configuration (see **Deployment** section below) that sets up Node and runs the Next.js app. This ensures that if you choose to move off Replit or run on a cloud VM, the setup remains easy. On Replit itself, you might run without Docker (using Replit's built-in environment), but having a Dockerfile will allow using Replit's Nix or Docker mode if needed and will be useful for local development and CI/CD.

In summary, the architecture leverages Next.js for server-rendering and API routes, uses external data APIs (CoinGecko, DefiLlama) for the various metrics, and displays everything in a user-friendly modal UI within the existing app. This approach ensures **real-time data**, good performance, and a smooth user experience.

Step-by-Step Implementation Guide

Here is a detailed roadmap of the steps to build and deploy the Unified Asset Dashboard. Each step breaks down what needs to be done in sequence:

1. Set Up the Development Environment (Next.js on Replit)
- *Initialize or use existing Next.js project*: If you already have a Next.js app running on Replit, proceed to step 2. Otherwise, you can create a new Next.js app. On Replit, you can either use the **Next.js Template** (Replit offers ready-to-go templates) or run the initialization manually. For example, open Replit's Shell and run:

```
npx create-next-app@latest unified-dashboard --typescript
```

This will create a new Next.js project (with TypeScript by default; you can omit `--typescript` if you prefer JS). If using the template, Replit will handle installing dependencies.
- *Configure Replit run settings*: Ensure that Replit is set to start the Next.js development server. By default, if it recognizes a Next.js project, it might run `npm run dev`. If not, open the `.replit` file and set the run command:

```
run = "npm install && npm run dev"
```

This will install packages and start the dev server on Replit. Next.js typically runs on port 3000, which Replit's web view will show.
- *Version control*: Optionally, set up a Git repo for your project to track changes. This is good practice as you implement the features step by step.

2. Install Needed Dependencies
Next.js comes with React and basic support for fetching. We may need to install a few additional libraries to build the dashboard:
- **Data fetching**: We can use the built-in `fetch` API (Node 18+ has global fetch). If you prefer, install `axios` for convenience:

```
npm install axios
```

(This is optional; `fetch` works fine and avoids an extra dependency.)
- **Chart library**: To display charts (price and possibly TVL/users), choose a library: for example, install **Recharts**:

```
npm install recharts
```

Recharts is easy for simple line charts. Alternatively, `react-chartjs-2` (which would also require `chart.js`) or `visx` can be used. Recharts will suffice for basic visualizations.
- **UI components**: If your app doesn't already use a UI framework, you might install one for modals and styling. Popular choices:
- **Material-UI (MUI)**:

```
npm install @mui/material @mui/lab @emotion/react @emotion/styled
```

MUI has a Modal component and lots of pre-styled components.
- **Chakra UI** or **Ant Design** or **Bootstrap** are other options.
- If you prefer minimal, you can use pure CSS or Tailwind CSS. Tailwind can be set up in Next.js easily if you want utility classes:

```
npm install tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Then configure Tailwind in `tailwind.config.js` and include the Tailwind directives in `styles/globals.css`. This might be useful for quick styling (e.g., classes for flex, spacing, modal backdrop, etc.).
- **Environment variable library**: Next.js can read from a `.env.local` file for environment variables. We don't necessarily need a library, but ensure to add `DEFILLAMA_API_KEY` in your Replit Secrets (which populates the env vars) or in `.env.local` (which you should not commit). No extra installation needed;

Next automatically loads `.env.local` if present.
- After installing any dependencies, Replit should auto-restart the dev server. If not, run `npm run dev` again.

3. Obtain API Access and Keys
- **CoinGecko API**: No API key is required for CoinGecko's free API. However, they ask for a polite usage (no more than 10-50 calls per second and to include a `User-Agent` or contact email in headers). We will use their public endpoints. Simply note the base URL: `https://api.coingecko.com/api/v3`. (For example, getting coin data: `/coins/{id}`, categories: `/coins/categories`, trending: `/search/trending` etc.) We should also fetch in a way that handles errors or rate-limit responses gracefully (CoinGecko may return HTTP 429 if overused). In a production scenario, consider upgrading to CoinGecko Pro if needed, but initially the free tier is fine.
- **DefiLlama API**: Sign up for DefiLlama's Pro API if you haven't already (since we need unlocks, active users, etc., which are pro features). The user prompt suggests you might already have an API key (the spec JSON was provided). Assuming you have a key, it typically comes as a part of the URL (like `https://pro-api.llama.fi/<YOUR_KEY>`). For security: - In Replit, go to the Secrets (Environment) section and add a new secret: Key = `DEFILLAMA_API_KEY`, Value = `<your_key_string>`. Replit will expose this in the environment for your Node app. - In Next.js, create a file `.env.local` (which is gitignored by default) with:

```
DEFILLAMA_API_KEY=<your_key_string>
```

Then, in your code, construct DefiLlama API calls using this. For example:

```
const DL_BASE = `https://pro-api.llama.fi/${process.env.DEFILLAMA_API_KEY}`;
const tvlUrl = `${DL_BASE}/api/tvl/${protocolSlug}`;
```

We may also store the base URL plus key as a single env var for convenience. - If for some reason you don't have Pro API access immediately, note that some data can be fetched from DefiLlama's **free** API or other sources: - TVL: Free API `https://api.llama.fi/protocol/<protocolName>` returns TVL and other info. - Token Unlocks: DefiLlama also has a free **CSV or JSON** on their site for unlocks (but it may not be official). However, since we aim for the best solution, the Pro API is preferred. - Active users: This is harder to get free. One might use an alternative like Dune Analytics queries or other blockchain scanners, but that's complex. DefiLlama Pro is the straightforward way. - For the scope of this project, we proceed with the assumption of Pro API usage. - **Other APIs** (if any): If we decide to incorporate additional data (like social metrics from an API, or alternate price sources), gather keys/access as needed. But the two above (CG & DL) cover our needs.

4. Implement Data Fetching (Backend logic)
Now, start coding the logic to retrieve data for a given coin. We will likely create a server-side function that collects all required data and returns it in one structured format. There are two main ways to do this in Next.js: - **Option A: Next.js API Route** – Create a file at `pages/api/asset/[id].ts` (or .js if not using TS). This will define an API endpoint like `/api/asset/{id}` where `{id}` could be the CoinGecko coin ID (e.g. "bitcoin" or "aave"). When this API route is called (via `fetch` from the client), it will: 1. Take the coin id from the request query. 2. Look up the corresponding DefiLlama protocol slug/ID (possibly by checking a

cached mapping or calling an initial mapping function – see mapping strategy below). 3. Perform external API calls: - Fetch CoinGecko data: e.g. `GET https://api.coingecko.com/api/v3/coins/{id}?localization=false&community_data=false&developer_data=false` (we can exclude some fields to reduce payload). This will give us price, market cap, supply, etc. Alternatively, use `/coins/{id}/market_chart` for price history if needed, or `/coins/markets` if we just want current data quickly (though the detailed endpoint gives more info in one call). - Fetch DefiLlama TVL: e.g. `GET https://pro-api.llama.fi/<KEY>/api/tvl/{protocolSlug}`. This might return the current TVL (and possibly a timeseries or breakdown). If only current TVL is returned, we might also call the historical endpoint if we want a chart. - Fetch Active Users: If DefiLlama has `GET /api/userData/activeUsers/{protocolId}`, call that. If not directly available, maybe `GET /api/activeUsers` returns a list of all protocols with user stats; we could filter that by our protocol. (We'll need to consult the spec or test a bit. Possibly, the `activeUsers` tag in the API might require calling something like `/api/userData/activeUsers/123` where 123 is the protocol's internal ID. The internal ID could come from the `/protocols` list or might be the index in DefiLlama's DB. We can retrieve the protocol list from `https://api.llama.fi/protocols` and find our protocol to get its ID field. This list can be cached in memory on first use.) - Fetch Unlocks: `GET https://pro-api.llama.fi/<KEY>/api/emission/{protocolSlug}` for the token's unlock schedule. This likely returns a list of upcoming unlock events or emission rates. We'll parse out the next upcoming event (the one with a future timestamp closest to today) to display. - Fetch Liquidity: `GET https://pro-api.llama.fi/<KEY>/api/historicalLiquidity/{token}`. The `token` might be something like the symbol or gecko_id (the spec example was "usdt" [8]). We need to confirm what identifier it expects (perhaps the symbol in lowercase or a specific slug). We might need to test this. This could return historical liquidity data. If so, we can take the latest data point as "current liquidity". It might also break down by chain or DEX; if it's complex, we can simplify our display to either total or just highlight a major liquidity source. 4. Wait for all promises to resolve (we can use `Promise.all` to fetch in parallel). 5. Combine the relevant pieces of data into a JSON object with a structure like:

```
{
  "coinId": "aave",
  "name": "Aave",
  "symbol": "AAVE",
  "price": 80.25,
  "priceChange24h": -2.1,
  "marketCap": 1200000000,
  "volume24h": 300000000,
  "category": "DeFi Lending",
  "tvl": 5000000000,
  "tvlChange24h": 1.5,
  "activeUsers24h": 1200,
  "activeUsersChange7d": 5.2,
  "liquidity": 25000000,
  "nextUnlock": {
    "date": "2025-08-01",
    "amount": 5000000,
    "percentSupply": 0.10
  },
  "priceHistory7d": [ ... ],
```

```
  "tvlHistory30d": [ ... ],
  "usersHistory30d": [ ... ]
}
```

(This is an example structure; we can adjust field names as needed in code.) 6. Return this JSON as the API response. - **Option B: getServerSideProps** – Instead of an API route, you can do the above data fetching inside `getServerSideProps(context)` of a page component (e.g. `pages/[coinId].js`). This would pre-render the page with data. It's suitable if we navigate to a dedicated coin page. But since we want a modal overlay and possibly an SPA feel, using an API route (Option A) might be cleaner: it separates data fetching from presentation and avoids full page reloads. - For a **modal approach**: likely we will go with an API route and client-side fetch. So we'll implement Option A. That means also writing the client-side code to call `/api/asset/[id]` when needed (we'll handle that in the next step with the UI). - **Implement the mapping logic**: Create a utility function (e.g. in `lib/defillama.js`) that can retrieve the protocol slug/ID for a given CoinGecko coin ID. Possible implementation:

```
import cache from 'node-cache'; // or use a simple global object for caching
let protocolMap = {};  // mapping from gecko_id -> { slug, id }

async function getProtocolMapping() {
  if (Object.keys(protocolMap).length === 0) {
    const res = await fetch('https://api.llama.fi/protocols');
    const data = await res.json();
    // data is an array of protocols with fields like name, slug, gecko_id, id,
etc.
    protocolMap = {};
    data.forEach(proto => {
      if (proto.gecko_id) {
        protocolMap[proto.gecko_id] = { slug: proto.slug, id: proto.id };
      }
    });
  }
  return protocolMap;
}

export async function mapCoinToProtocol(geckoId) {
  const map = await getProtocolMapping();
  return map[geckoId] || null;
}
```

This function fetches and caches the mapping on first call. Then, in our API handler, we can do:

```
const protocolInfo = await mapCoinToProtocol(coinId);
if (protocolInfo) {
  const { slug, id } = protocolInfo;
```

```
  // use slug for tvl/unlocks, id for activeUsers
}
```

If no mapping is found, it means the coin might not have a DefiLlama entry (then we skip those fetches). - **Handle API errors**: Wrap fetch calls in try/catch. If one source fails (e.g., DefiLlama down or no data), the API route can still return what it has (or an error message). It's better to return partial data than nothing: e.g., if unlocks data isn't available, we can still show price and TVL. We might include a flag in JSON like `"error": {"section": "unlocks", "message": "Data not available"}` if needed, or simply omit that field. - Test the API route locally by visiting a URL like `http://localhost:3000/api/asset/bitcoin` to see if it returns data (Replit also allows opening the web URL). Use the Replit console to see any `console.log` or error outputs for debugging.

5. Build the Dashboard UI Component
Next, create the React component that will display the unified dashboard using the data fetched.

- **Component Setup**: Create a file `components/AssetDashboard.jsx` (or `.tsx` if using TypeScript). This component will accept a prop like `data` (the JSON from the API route). It could also accept a prop for visibility (if we manage the modal state outside) or we can integrate it into a modal directly.
- **Layout and Styling**: Use a combination of divs, grids, and your chosen styling method to organize the sections. For example:

```
function AssetDashboard({ data, onClose }) {
  if (!data) return null; // or a loading state
  return (
    <div className="asset-dashboard-modal">
      <div className="modal-backdrop" onClick={onClose} />
      <div className="modal-content">
        {/* Header */}
        <div className="asset-header">
          <img src={data.imageUrl} alt={data.name} className="asset-icon" /
>
          <h2>{data.name} ({data.symbol.toUpperCase()})</h2>
          {data.category && <span className="asset-tag">{data.category}</
span>}
        </div>
        {/* Price and Chart */}
        <div className="price-section">
          <div className="price-info">
            <div className="price">${data.price.toFixed(2)}</div>
            <div className={`price-change ${data.priceChange24h >= 0 ?
'up' : 'down'}`}>
              {data.priceChange24h.toFixed(2)}%
            </div>
            <div>Market Cap: ${formatNumber(data.marketCap)}</div>
            <div>24h Volume: ${formatNumber(data.volume24h)}</div>
```

```
          </div>
          <div className="price-chart">
            <PriceChart data={data.priceHistory7d} />
          </div>
        </div>
        {/* Key Metrics */}
        <div className="metrics-section">
          <div className="metric">
            <span>TVL:</span> ${formatNumber(data.tvl)}
            {data.tvlChange24h && <span
className={data.tvlChange24h>=0?'up':'down'}>
              ({data.tvlChange24h.toFixed(1)}%)
            </span>}
          </div>
          <div className="metric"><span>Active Users (24h):</span>
{formatNumber(data.activeUsers24h)}</div>
          <div className="metric"><span>Liquidity:</span> $
{formatNumber(data.liquidity)}</div>
          {data.nextUnlock && (
            <div className="metric highlight">
              <span>Next Unlock:</span> {data.nextUnlock.date} –
{formatNumber(data.nextUnlock.amount)} tokens
              ({(data.nextUnlock.percentSupply*100).toFixed(1)}% of supply)
            </div>
          )}
          {/* ... other metrics as needed */}
        </div>
        {/* Additional Charts */}
        <div className="additional-charts">
          {data.tvlHistory30d && <TVLChart data={data.tvlHistory30d} />}
          {data.usersHistory30d && <UsersChart
data={data.usersHistory30d} />}
        </div>
        {/* Close button */}
        <button onClick={onClose} className="close-btn">Close</button>
      </div>
    </div>
  );
}
```

The above is a rough JSX structure demonstrating sections:

- `.modal-backdrop` covers screen (semi-transparent background).
- `.modal-content` is the centered panel with the data.
- We have a header (icon, name, category tag).
- A price section with price and a small chart.
- A metrics section listing TVL, users, liquidity, unlock etc. We might highlight important things (like unlocks) with a special style (e.g. red text if imminent).

    - Additional charts for TVL and users trends can be shown below if available (these could be small and not overwhelm the UI).
    - A close button to exit the modal. We will need to create `PriceChart`, `TVLChart`, `UsersChart` as small components using Recharts or whichever library. For example, `PriceChart` might render a simple line chart for the 7-day price points.
- **Styling**: Write CSS for the classes used:
    - `.asset-dashboard-modal` could be a fixed position container for the modal.
    - `.modal-backdrop` a full-screen overlay (e.g. `position: fixed; top:0; left:0; right:0; bottom:0; background: rgba(0,0,0,0.5)`).
    - `.modal-content` a centered box (e.g. `position: fixed; top:50%; left:50%; transform: translate(-50%, -50%); width: 80%; max-width: 600px; background: #fff; padding: 20px; border-radius: 8px;`)
    - `.asset-header` could be flex with icon and title.
    - `.price-section` flex container: `.price-info` (with price and changes) and `.price-chart` (width maybe 60% for chart).
    - `.metrics-section` a grid or flex-wrap of metrics. Possibly use a 2-column layout for metrics.
    - `.metric` style for label and value.
    - `.highlight` class on metric to draw attention (e.g. for unlock).
    - Up/Down classes for percentage changes (green/red).
    - `.close-btn` style a simple button or use an icon (like ×). If using Tailwind, these would be replaced by Tailwind utility classes directly in JSX instead of custom class names.
- **Responsive design**: Ensure that on smaller screens the modal still looks good (the flex layouts might wrap to column). Possibly use media queries or flex wrap.
- **Testing the component**: You can test the component by hardcoding some dummy `data` prop initially. Import and render `AssetDashboard` in a page to see if the layout is correct. Once styling and layout are satisfactory, you'll integrate it with real data and triggers.

6. Integrate the Dashboard with Application (Pop-up trigger & state)
Now that the backend API and frontend component are ready, connect them: - Choose where in your app the dashboard should be triggered. For instance, if you have a list of coins (maybe a table or trending list), wrap each coin name (or a "details" button) with an onClick that calls a function to open the dashboard. - Implement state in the parent component/page. For example, in Next.js you might have a page `pages/index.jsx` showing a list of assets. In that page component, add:

```
const [selectedAsset, setSelectedAsset] = useState(null);
const [assetData, setAssetData] = useState(null);
const openDashboard = async (coinId) => {
  setSelectedAsset(coinId);
  // fetch data from API route
  try {
    const res = await fetch(`/api/asset/${coinId}`);
    const data = await res.json();
    setAssetData(data);
  } catch(err) {
    console.error("Failed to fetch asset data", err);
```

```
    setAssetData({ error: "Failed to load data" });
  }
};
const closeDashboard = () => {
  setSelectedAsset(null);
  setAssetData(null);
};
```

- Here, when `openDashboard` is called (with a coin's ID), it sets the selected coin, then fetches the data from our API route and stores it in state (`assetData`). We separate `selectedAsset` and `assetData` to manage the rendering: we might show a loading state if `selectedAsset` is set but `assetData` is not yet available. - `closeDashboard` resets the state. - Render logic: In the JSX of that page:

```
{selectedAsset && (
  <AssetDashboard data={assetData} onClose={closeDashboard} />
)}
<ul>
  {coins.map(coin => (
    <li key={coin.id}>
      {coin.name}
      <button onClick={() => openDashboard(coin.id)}>View</button>
    </li>
  ))}
</ul>
```

This is a simple example where `coins` is an array of coin objects (with `id` and `name`). In practice, you might already have this list from an earlier fetch (maybe you fetched top 100 coins or trending coins for the homepage). If not, you can use CoinGecko's `/search/trending` or `/coins/markets` to populate a list of coins to display. For curated categories (point #1), you might list some category names and allow the user to drill down first. - If you want a category view: you could create a page `/categories` that lists categories from CG (via `getStaticProps` or SSR) and then, when a category is selected, show the coins in that category (CoinGecko's categories endpoint returns an array of coins in each category, with basic data). You can then trigger the dashboard for any coin in that list. This would satisfy the curated subset requirement by allowing users to browse via categories. - **Pop-up behavior**: The `<AssetDashboard>` is rendered conditionally when `selectedAsset` is not null. The backdrop click or close button triggers `onClose` which clears it. This means the modal will unmount and disappear. - **Loading state**: We should handle the scenario of slow API fetch: - When `openDashboard` is called, we set `selectedAsset` immediately (so modal opens with maybe a spinner) but `assetData` might be null for a brief time. In `AssetDashboard`, we can check `if (!data) { return <div className="loading">Loading...</div>; }` or similar to show a loading indicator inside the modal content box. Alternatively, set a separate `loading` state and use that to show a spinner overlay. - Also handle `assetData.error`: if an error occurred, display a friendly message instead of data. - **Testing the flow**: Try clicking on a coin's button in the UI; the modal should appear and populate with real data after a moment. Check that closing works. Adjust any styling or state logic if needed (for example, prevent scrolling the background when modal is open by adding `overflow: hidden` to body, etc.)

7. Implement Category Filtering and Curated Subsets (Optional Enhancement)
To address the idea of **curated subsets using categories** (point #1), consider adding a section to showcase certain groups of tokens. This could be: - A **Categories page or section**: which lists high-level categories (like Smart Contract Platforms, DeFi, NFTs, Layer 2s, etc.). You can fetch all categories from `GET /coins/categories/list` (CoinGecko) or the detailed one `GET /coins/categories` which also provides aggregated market caps. If the list is large, maybe focus on top N categories by market cap or manually pick some that are relevant. - When a category is selected, display the list of coins in that category (CoinGecko's `/coins/markets` endpoint has a parameter `category`). For example, `GET /coins/markets?vs_currency=usd&category=decentralized-finance&order=market_cap_desc` would give the top coins in DeFi. Then the user can click on a coin to open the dashboard modal. - Alternatively, a **Trending coins section**: use `/search/trending` to get currently trending coins on CG, and allow quick selection of those for the dashboard. - A **Watchlist or curated list**: if you as the project owner have a specific subset (like "Our Picks" or "Analyst Favorites"), you can hard-code or configure a list of coin IDs to feature. Then present them as buttons to open the unified dashboard. This can draw users' attention to interesting assets. - These curated subsets improve user experience by not overwhelming them with thousands of coins. Instead, they can browse by themes or popularity, then dive in with the unified dashboard for details.

8. Testing and Refinement
With the core functionality in place, thoroughly test the dashboard: - Check multiple coins, especially ones from different sectors: - **Large DeFi project** (with TVL, e.g., Aave, Uniswap): see that TVL and user metrics appear logically. - **Token with upcoming unlock** (if any known; perhaps look at DefiLlama's unlocks list to find one). - **Simple coin (no DeFi)** like Bitcoin or Dogecoin: these should show price but likely no TVL or unlocks. The dashboard should handle missing data gracefully (e.g., hide the TVL section or say "N/A"). - **Stablecoin**: maybe show supply instead of TVL (since stablecoins have different metrics). If such nuance is needed, consider customizing per category (not critical for initial version, but be aware). - Check the UI on different screen sizes (Replit has a web view; you can also open in a new tab and use browser dev tools for responsive design). Make adjustments for mobile if needed (stack sections vertically, make font sizes readable). - Ensure that the modal backdrop closes on click outside and that the close button works. Verify that after closing, the state resets (open it again fresh). - Test performance: The first time calling the API route might be a bit slow (especially if it has to fetch the protocol list for mapping). Subsequent calls should be faster due to caching. If performance is an issue, consider moving the mapping pre-fetch to initialization (e.g., fetch protocol list at server start time) or refine caching. Also consider using Next.js middleware or API route caching headers if needed. - Handle any errors: simulate what happens if CoinGecko API fails (maybe temporarily change the URL to wrong one to see error handling) – ensure the user sees either a message or partial data rather than a broken modal. - **Rate limit testing**: If you quickly open many dashboards, ensure that we're not spamming too many requests. The caching of the protocol list helps reduce calls to that endpoint. We might also cache the `/api/emissions` list similar to protocols to avoid repeated calls for each coin unlock lookup. - Logging: In a dev environment, log the combined data to console to verify correctness (are we showing the right TVL? Does it match what's on defillama.com for that protocol? etc.). Fine-tune any calculations (like percent of supply for unlock – ensure you have total supply from CG to compute that). - Once satisfied, clean up debug logs and ensure code clarity. Write comments in code for maintainability, especially in the data fetching logic.

9. Prepare for Deployment (Docker configuration)
With everything working in the dev environment, set up Docker for deployment (and for reproducibility): - Create a `Dockerfile` at the root of your project with the following content:

```
# Use an official Node.js runtime as a parent image
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Build the Next.js application
RUN npm run build

# Expose the port (Next.js defaults to 3000)
EXPOSE 3000

# Set environment variables for production
ENV NODE_ENV=production

# Start the Next.js server
CMD ["npm", "run", "start"]
```

This Dockerfile uses a lightweight Node 18 image, installs dependencies, builds the Next.js app (for production), and then starts it. The final container will run `npm run start` which launches the Next.js server in production mode. - Since we're likely using environment variables (like `DEFILLAMA_API_KEY`), ensure that in a Docker deployment scenario, you provide those env vars. If running the container locally, you can use `docker run -e DEFILLAMA_API_KEY=yourkey -p 3000:3000 imagename`. In a hosting environment, you'd add those to the config. - (If you want to test the Docker image locally: build it with `docker build -t unified-dashboard .` then run it with `docker run -p 3000:3000 unified-dashboard` and access http://localhost:3000. On Replit, you might not need to do this, but it's good to have for portability.)

Replit Note: Replit supports Docker-based repls or Nix, but it's often easier to just run Node directly. You might not actually run the Docker container on Replit; instead, Replit will use the package.json start script. The Dockerfile is more for deployment on other platforms (or if you use Replit's deployment feature which might accept a Dockerfile). However, including the Dockerfile in the repo's root is useful for future use and signals how to run the app anywhere.

10. Deploy and Monitor

- **On Replit**: If you've been developing on Replit, your application is essentially already "deployed" on the Replit URL. Make sure the Replit is set to **Always On** if you want it continuously accessible (this might

require Replit's paid plan). Otherwise, the web service might shut down when idle. Replit might also have a Deploy feature to create a static link. - If you want to deploy elsewhere (for example, Vercel is a natural choice for Next.js apps, or a VPS using Docker): - **Vercel**: You can push your code to a GitHub repo and connect it to Vercel. Vercel will detect Next.js and deploy automatically. You'd need to add environment variables (DEFILLAMA_API_KEY) in Vercel's dashboard. Vercel gives you a domain and handles scaling. This is a good approach if you expect many users, as Replit might not handle heavy traffic as well. - **Docker on a server**: Use the Dockerfile to run on any cloud provider (AWS, DigitalOcean, etc.). Ensure you configure the env vars and appropriate domain/port. - **Post-deployment testing**: After deployment, test the live site. Try out a few dashboards, ensure data shows up. Monitor the logs (Replit console or Vercel logs) for any errors like failed API calls or rate limit warnings. - **Analytics & Feedback**: It might be useful to integrate something like simple analytics or even console logs to see usage frequency. At minimum, keep an eye on DefiLlama/CoinGecko usage quotas (CoinGecko might have an hourly limit of ~100 calls/minute from one IP; with a modest user base this is fine, but if scaling up, consider caching or requesting higher tier). DefiLlama Pro has high limits but if your usage grows, ensure it stays within bounds. - **Documentation**: Write a short guide (perhaps in your project's README) for future maintainers or yourself about how the data is fetched and how to update the API keys, etc. This is partly done by this roadmap itself. Also, add notes on how to add a new metric if needed, or how to update the mapping if DefiLlama changes their API.

11. Future Improvements (Continuous Development)
Although the initial deployment will already offer a powerful tool, you can plan some future enhancements: - **Refine UI/UX**: Add search functionality to quickly find a coin by name/ticker and open the dashboard. Enhance the design (maybe allow the modal to be dragged or resized, etc., if using a UI library). - **Notifications/Alerts**: Perhaps let users set an alert if an unlock is approaching or if TVL drops X% etc. This would differentiate even further from static trackers. - **More Data Integrations**: Include **protocol revenue** and P/E ratios (market cap / annualized revenue) if relevant, using DefiLlama's revenue API. Add **developer activity** (GitHub commit stats via an API like Santiment or CryptoMiso) to gauge development. Or **social sentiment** from an API. Each metric should only be added if it provides clear value. - **Performance Optimizations**: If the user base grows, implement server-side caching (e.g. cache coin data for 1 minute in an in-memory store or use Next.js revalidation for static props if acceptable) to reduce API calls. Also consider using WebSockets or CoinGecko's WebSocket (if available) for real-time price updates in the dashboard without refresh. - **Mobile App**: If your application extends to mobile, consider how to incorporate this dashboard in a mobile-friendly way (React Native or a responsive webview). - **User Customization**: Allow users to customize which metrics they see or to hide sections. For instance, some might not care about active users or might want extra metrics – a settings panel could toggle them. - **Comparison Feature**: Perhaps allow comparing two assets side-by-side (this might be beyond original scope but could be interesting – e.g. see Asset A vs Asset B metrics in one view).

These improvements can be scheduled on the roadmap after the core is stable. The key is that the framework we built (modular data fetching and component-based UI) will make it easier to extend with new data or features.

Conclusion

By following this roadmap, you will have created a **Unified Asset Dashboard** integrated into your Next.js application on Replit, with a clear step-by-step development process and a robust architecture. The end result is a **comprehensive coin detail modal** that provides users with price, on-chain fundamentals, and tokenomics at a glance. This not only enriches your application's functionality but also provides a

competitive edge over other platforms. Users will appreciate the ability to get all relevant info in one place – something even top-tier platforms often lack in a free, consolidated format.

With deployment complete, you can now direct your community or users to this feature, highlighting how it aggregates data from multiple sources in real time. The groundwork laid out here ensures the feature is maintainable and extensible for future needs. Enjoy the process of building it, and soon you'll have a powerful tool that truly **unifies asset data** for your users!

---

1 2 3 Most Comprehensive Cryptocurrency Price & Market Data API | CoinGecko API
https://www.coingecko.com/en/api

4 7 Pricing | DeFi Llama
https://docs.llama.fi/pro-api

5 Token Unlocks
https://docs.messari.io/docs/token-unlocks

6 8 defillama-api-spec.json
file://file-6CiSDcBmz8BsLDZo8HT4N8