Product Requirements Document (PRD)

Name od the project – HeliSync

1. Overview

1.1 Purpose

The Blockchain Indexing Platform enables developers to index Solana blockchain data (e.g., NFT bids, token prices, lending pool updates) into a Neon Postgres database using Helius webhooks. It provides a streamlined, user-friendly solution that eliminates the need for managing RPC nodes, validators, or custom webhook setups.

1.2 Goals

* Simplify blockchain data indexing for developers.
* Deliver a customizable, real-time data pipeline from Solana to Neon Postgres.
* Provide an intuitive, sleek, and modern web interface built with Next.js for configuration and monitoring.
* Support Solana Devnet for testing and validation with comprehensive test coverage.

1.3 Target Audience

* Blockchain developers building decentralized applications (dApps) on Solana.
* Data analysts tracking NFT markets, token prices, or lending pools.
* Companies integrating real-time blockchain data into relational databases.

2. Key Features

2.1 User Authentication & Credential Management

* Sign-up/Login: Secure authentication using OAuth (Google) or email/password via Firebase.
* Neon Postgres Credential Storage: Allow users to input and securely store Neon Postgres credentials (hostname, port, username, password, database name) encrypted with AWS Secrets Manager.
* Validation: Verify database connectivity upon credential submission with a success/error message in the UI.

2.2 Neon Postgres Database Integration

* Schema Setup: Automatically configure the user’s Neon Postgres database with tables for indexed data (NFT bids, token prices, borrowable tokens).
* Data Retention: Users can configure retention policies (e.g., 30 days, 90 days, or indefinite, subject to Neon Postgres capacity).
* Extensibility: Use JSONB fields for flexible metadata storage.

2.3 Customizable Data Indexing

* Event Selection: Enable users to toggle which blockchain events to index:
  + NFT Bids: Track bidder, bid amount, NFT identifier, and timestamp.
  + Token Prices: Monitor token identifier, platform, price, and timestamp.
  + Borrowable Tokens: Capture token identifier, available amount, lender details, and timestamp.
* User Control: Provide sleek toggle switches or dropdowns in the UI for enabling/disabling event types.

2.4 Automated Blockchain Data Retrieval

* Webhook Integration: Subscribe to Solana blockchain events via Helius webhooks (e.g., NFT sales, token price changes, lending updates).
* Real-Time Processing: Process webhook payloads and insert data into Neon Postgres with <1s latency.
* Error Handling: Retry failed webhook processing 3 times; log errors and notify users via the dashboard.

2.5 Web-Based Dashboard

* Core Functionality:
  + Input and manage Neon Postgres credentials with a clean form layout.
  + Configure indexing preferences using modern toggle switches or dropdowns.
  + Display indexing status and logs in a scrollable, minimalistic panel with timestamps and color-coded success/error indicators.
* UI Design: Built with Next.js, featuring a sleek, modern aesthetic inspired by the attached screenshot (assumed to be minimal, dark/light theme-switchable, with smooth animations and a grid-based layout). Uses Tailwind CSS for styling and shadcn/ui components for consistency.
* Notifications: Users can opt into email alerts for indexing failures or significant events (e.g., 1M transactions indexed).

2.6 Scalable Backend & Infrastructure

* Data Processing: Use a background task system (e.g., Redis with Celery) to handle webhook data efficiently.
* Event Updates: Support real-time dashboard updates via WebSockets for a responsive experience.
* Scalability: Handle 1M+ transactions/day across multiple users with Neon Postgres and cloud hosting (AWS).

2.7 Devnet Support & Testing

* Testing Environment: Enable indexing on Solana Devnet with mock transactions.
* Mock Data: Provide sample events (e.g., 100 NFT bids, 50 token price updates) for users to validate setup.

3. Technical Requirements

3.1 Tech Stack

* Frontend: Next.js (with server-side rendering) for a fast, modern UI; Tailwind CSS + shadcn/ui for styling.
* Backend: Node.js with Express.js for API and webhook handling; Redis + Celery for background tasks.
* Database: Neon Postgres (serverless, scalable) for user data storage.
* Webhook Service: Helius API for Solana event streaming.
* Infrastructure: AWS (ECS for containers, Lambda for optional webhook processing).

3.2 Database Structure

* NFT Bids: Fields: NFT identifier, bidder, bid amount, currency, timestamp, metadata (JSONB).
* Token Prices: Fields: token identifier, platform, price, timestamp, metadata (JSONB).
* Borrowable Tokens: Fields: token identifier, available amount, lender, timestamp, metadata (JSONB).
* Performance: Indexes on identifiers and timestamps for fast queries.

3.3 Webhook Integration

* Subscribe to Helius events (e.g., NFT sales, token transfers).
* Parse payloads and map to Neon Postgres tables.
* Retry failed inserts 3 times; log errors for dashboard display.

3.4 Security

* Encrypt Neon Postgres credentials with AWS Secrets Manager.
* Use HTTPS for all communications.
* Rate limit API endpoints (e.g., 100 requests/minute per user).

3.5 Testing

* Unit Tests:
  + Authentication: Verify OAuth login and credential validation.
  + Webhook Processing: Test payload parsing and database insertion.
  + Error Handling: Simulate webhook failures and confirm retries.
* Integration Tests:
  + End-to-end flow: Sign up, configure indexing, process mock Devnet events, view logs.
  + Neon Postgres: Validate schema creation and data retention settings.
* Performance Tests: Simulate 1M transactions/day to ensure <1s latency.
* Tools: Jest for unit tests, Cypress for UI testing.

4. User Flow

1. User signs up/logs in via a sleek Next.js login page.
2. User enters Neon Postgres credentials in a minimal form; system validates and encrypts them.
3. User toggles indexing options (e.g., NFT bids) in a modern dashboard layout.
4. Backend subscribes to Helius webhooks based on selections.
5. Webhook events are processed and stored in Neon Postgres; real-time updates appear in the dashboard log panel.
6. User receives email notifications for errors or milestones (optional).

5. Deliverables

* Web Platform: Next.js-based frontend and Express.js backend for indexing configuration.
* Neon Postgres Integration: Automated schema setup and data ingestion.
* Webhook Processing: Real-time Helius event handling with error retries.
* Documentation:
  + User Guide: Step-by-step setup instructions (e.g., entering credentials, selecting events).
  + API Docs: REST endpoint details (e.g., /configure-indexing, /status) with examples in Postman format.
  + Developer Guide: Notes on extending the platform (e.g., adding new event types).
* Test Suite: Comprehensive Jest/Cypress tests covering unit, integration, and performance scenarios.

6. Success Metrics

* Adoption: 100 active users within 3 months post-launch.
* Performance: Process 1M+ transactions/day with <1s latency.
* Reliability: 99.9% uptime for webhook processing and dashboard.
* User Feedback: 90%+ satisfaction for UI sleekness and ease of use.

7. Open Questions

1. Custom Queries: Should users define custom indexing rules in a future version?
2. Access Control: Is multi-user role support needed (e.g., admin vs. viewer)?
3. Hosting: Confirm AWS as the preferred provider?

8. Next Steps

1. Design UI mockups based on the screenshot’s sleek, modern style.
2. Build a prototype (Next.js frontend, Express.js backend, Neon Postgres integration).
3. Run Devnet tests with mock data and validate with Jest/Cypress.
4. Draft and review documentation.
5. Iterate based on user feedback.

Notes on UI Design

Since I can’t see your screenshot, I’ve assumed it’s a clean, modern design (e.g., dark/light theme, minimal typography, grid layout, smooth transitions). The Next.js frontend with Tailwind CSS and shadcn/ui will deliver:

* Sleekness: Thin borders, subtle shadows, hover effects.
* Modernity: Theme toggle, responsive grid, animated toggles.
* Consistency: Unified color palette (e.g., blue accents, neutral backgrounds).