Decentralized Disaster Relief Fund DAO: Technical Documentation

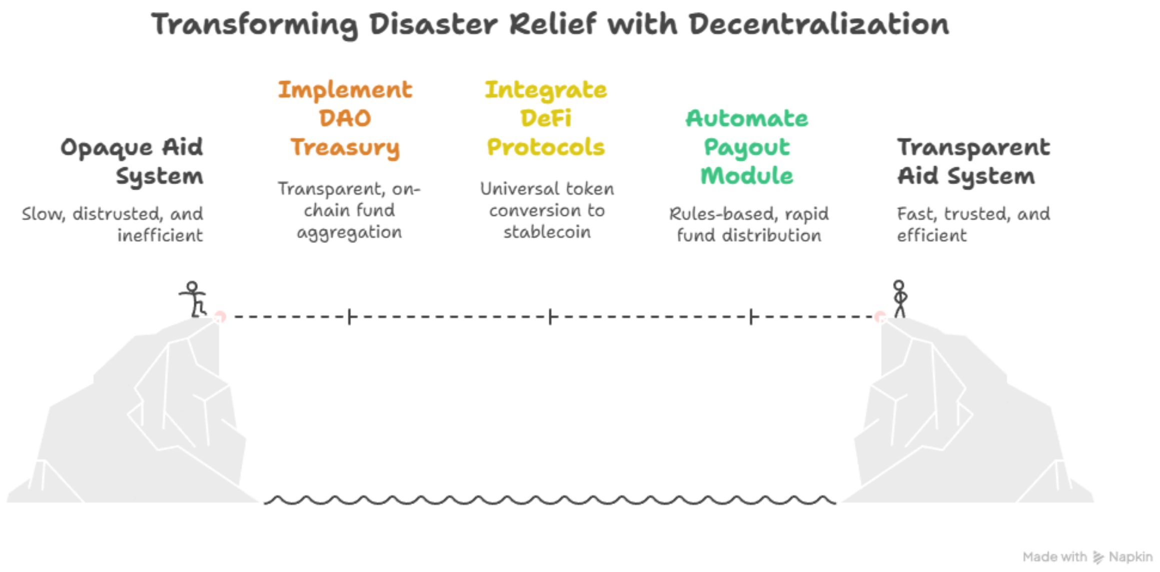
1.0 Introduction

1.1 Project Synopsis

The Decentralized Disaster Relief Fund DAO is a Web3 application designed to address and rectify systemic inefficiencies in traditional humanitarian aid frameworks. Developed as a proof-of-concept, the project demonstrates a new paradigm for collecting and distributing funds in crisis scenarios by applying the principles of decentralization, transparency, and speed. It directly confronts challenges like high administrative overhead and cross-border transfer delays that can slow down critical aid. This effort applies the core tenets of decentralization to a domain where trust, speed, and accountability are of paramount importance.

1.2 Core Proposition

The project's central thesis is the strategic application of blockchain technology's inherent properties—namely transparency, speed, and immutability—to engineer a more efficient, accountable, and globally accessible disaster relief mechanism. The project's tagline, **"Rapid, Transparent Aid via Web3,"** encapsulates the dual objectives of accelerating aid delivery while restoring donor trust through verifiable, on-chain financial flows. This means any donor can trace their contribution from their personal wallet to the final beneficiary using a public block explorer, an unprecedented level of transparency. This system is designed to combat the "slow, bureaucratic processes," like multi-layered approval chains and international banking hurdles, and the "lack of transparency" that plague traditional aid, where funds can be lost to overhead or misappropriation.


Transforming Disaster Relief with Decentralization
Opaque Aid System
Slow, distrusted, and inefficient
Implement DAO Treasury
Transparent, on-chain fund aggregation
Integrate DeFi Protocols
Universal token conversion to stablecoin
Automate Payout Module
Rules-based, rapid fund distribution
Transparent Aid System
Fast, trusted, and efficient
Made with Napkin

2.0 System Architecture and Workflow

The system is engineered as a decentralized application (dApp) on the Solana blockchain. It prioritizes a lightweight, client-centric model that is optimized for rapid development and a clear demonstration of the core concept, while being robust enough to handle real-world financial transactions securely.

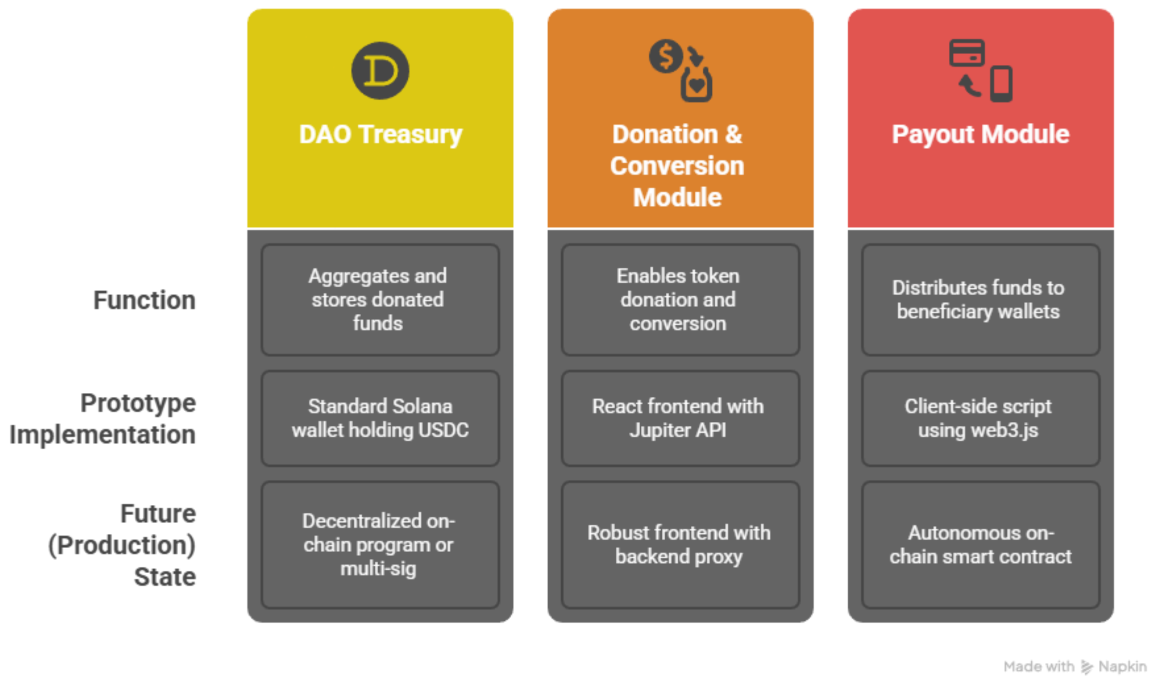
2.1 High-Level Architectural Blueprint

The architecture consists of three primary, interconnected components that create a closed-loop flow defined as **"Donation → Conversion → Storage → Payout."**

1. **Donation & Conversion Module (Frontend):** A React-based web interface where users connect their Solana wallets to donate. Its core technical feature is the deep integration with the Jupiter DEX Aggregator API, which allows users to donate any SPL token without needing to perform a manual swap beforehand, drastically improving the user experience.
2. **DAO Treasury (On-Chain):** A central, on-chain wallet that serves as the transparent repository for all relief funds. For the prototype, this is a standard Solana wallet holding the stablecoin USDC. Using a stablecoin is a deliberate choice to mitigate the price volatility inherent in many cryptocurrencies, ensuring the fund's value remains stable and predictable.
3. **Payout Module (Backend/Script):** An administrative function that executes the

automated, programmatic distribution of funds from the DAO Treasury to pre-configured beneficiary wallets. This component directly addresses the "last mile" delivery challenge, ensuring funds can be moved to on-the-ground relief organizations without intermediary delays.

Comparison of Disaster Relief Fund DAO Prototype Components



	DAO Treasury	Donation & Conversion Module	Payout Module
Function	Aggregates and stores donated funds	Enables token donation and conversion	Distributes funds to beneficiary wallets
Prototype Implementation	Standard Solana wallet holding USDC	React frontend with Jupiter API	Client-side script using web3.js
Future (Production) State	Decentralized on-chain program or multi-sig	Robust frontend with backend proxy	Autonomous on-chain smart contract

2.2 Core Workflows

2.2.1 Donation Workflow

This workflow is designed to be seamless for the user, abstracting away the complexity of token swaps into a single, intuitive action.

1. **Connect Wallet:** A donor visits the web application and connects their Solana wallet (e.g., Phantom, Solflare) using the @solana/wallet-adapter library, which provides a secure, standardized connection method.
2. **Specify Donation:** The donor selects any SPL token they wish to donate from a dropdown list and enters the desired amount.
3. **Get Quote:** In milliseconds, the frontend calls an internal API endpoint (/api/quote) which, in turn, queries the Jupiter API. Jupiter returns the optimal swap route to convert the donor's token into USDC, providing a real-time,

guaranteed exchange rate.

4. **Execute Swap:** The donor reviews the quote and approves the transaction. The frontend calls the internal /api/swap endpoint, which uses the quote data to get a serialized, ready-to-sign swap transaction from the Jupiter API.
5. **Transaction Confirmation:** The donor signs the transaction in their wallet. This cryptographic signature is their authorization. The swap is executed on the Solana blockchain, and the resulting USDC is deposited directly into the DAO Treasury wallet. This event is immutable and publicly recorded.
6. **UI Update:** The application displays a success message with a link to the transaction on a block explorer. The public dashboard immediately updates in real-time to reflect the new treasury balance.

2.2.2 Payout Workflow

This workflow is designed for administrative control and automated distribution based on predefined, transparent rules.

1. **Trigger Payout:** An administrator navigates to a protected, access-controlled section of the dApp.
2. **Define Parameters:** The administrator initiates the payout based on a hardcoded rule (e.g., "Distribute 50% of treasury funds equally to 2 pre-vetted beneficiary wallets"). In a full implementation, these rules would be set by DAO governance.
3. **Construct Transactions:** A client-side script or lightweight backend function (/api/payout/execute) uses @solana/web3.js to construct the necessary SPL token transfer transactions from the treasury to the beneficiaries. This ensures the transactions are standard and verifiable.
4. **Authorize Payout:** The holder of the treasury wallet's private key (the administrator) is prompted to sign and authorize these transactions. This is the primary point of centralization in the prototype, a deliberate trade-off for speed of development.
5. **Distribution:** Once signed, the transactions are broadcast to the Solana network, and the funds are distributed nearly instantaneously. The public payout log is updated with links to the on-chain transaction records for full auditability.

3.0 Folder Structure and Code Explanation

The project is built using Next.js, following a standard structure for a modern, full-stack web application. The logic is intentionally kept "backend-light," with much of the functionality handled by Next.js API Routes that act as a secure proxy to the Jupiter API.

```
reliefDAO/
├── public/
│   └── # Static assets like images, fonts, etc.
├── src/
│   ├── api/
│   │   ├── quote/
│   │   │   └── route.ts       # Handles GET requests to fetch swap quotes from Jupiter.
│   │   ├── swap/
│   │   │   └── route.ts       # Handles POST requests to generate swap transactions
from Jupiter.
│   │   ├── treasury/
│   │   │   └── balance/
│   │   │       └── route.ts   # Retrieves and returns the treasury's SOL and USDC
balance.
│   │   └── payout/
│   │       └── execute/
│   │           └── route.ts   # Constructs the unsigned payout transaction.
│   ├── components/
│   │   ├── ui/              # Reusable UI elements (Buttons, Modals, etc.).
│   │   ├── WalletConnectButton.tsx # Component for connecting a Solana wallet.
│   │   ├── DonationForm.tsx      # The main form for initiating a donation.
│   │   ├── TreasuryDashboard.tsx # Displays the current treasury balance and recent
activity.
│   │   └── PayoutTrigger.tsx     # Admin component to initiate the payout workflow.
│   ├── hooks/
│   │   ├── useJupiterApi.ts      # Custom hook to encapsulate logic for fetching
quotes and swaps.
│   │   └── useTreasury.ts       # Custom hook for fetching and managing treasury
state.
│   ├── lib/
│   │   ├── solana.ts          # Solana connection setup and helper functions.
│   │   └── constants.ts        # Application-wide constants (e.g., Treasury address,
USDC mint).
│   ├── app/
│   │   ├── layout.tsx          # Main application layout.
│   │   ├── page.tsx            # Homepage / Main dashboard.
│   │   ├── donate/
│   │   │   └── page.tsx        # Page containing the DonationForm component.
│   │   └── admin/
│   │       └── trigger/
│   │           └── page.tsx     # Admin page containing the PayoutTrigger component.
├── .env.local               # Environment variables (RPC URL, Treasury address).
├── package.json
└── next.config.js
```

Key File Explanations

- **src/api/**/*.ts**: These Next.js API Routes serve as the "minimal backend." They are server-side functions that securely handle communication with external services like the Jupiter API and the Solana RPC node. This proxy layer is crucial because it prevents exposing sensitive logic or API keys to the client and allows for better management of API rate limits and error handling.
- **src/components/DonationForm.tsx**: This is the core user-facing component. It manages the state for the input token and amount, and orchestrates the entire user interaction flow for making a donation. It utilizes the useJupiterApi hook and is responsible for handling loading states and displaying clear success or error messages to the user.
- **src/hooks/useJupiterApi.ts**: This custom hook simplifies interaction with the internal API endpoints for quoting and swapping. It abstracts away the fetch calls and state management related to the Jupiter integration, making the component code in DonationForm.tsx much cleaner, more readable, and focused on presentation.
- **src/lib/solana.ts**: This utility file initializes the Connection object from @solana/web3.js using the RPC URL from the environment variables. It ensures a single, consistent connection instance is used across the application, which is

vital for performance, reliability, and preventing resource leaks.

4.0 API Integrations & Technology Stack

The technology stack was chosen to prioritize development velocity, user experience, and the unique advantages of the Solana ecosystem, such as high speed and low transaction fees.

4.1 Jupiter API Integration

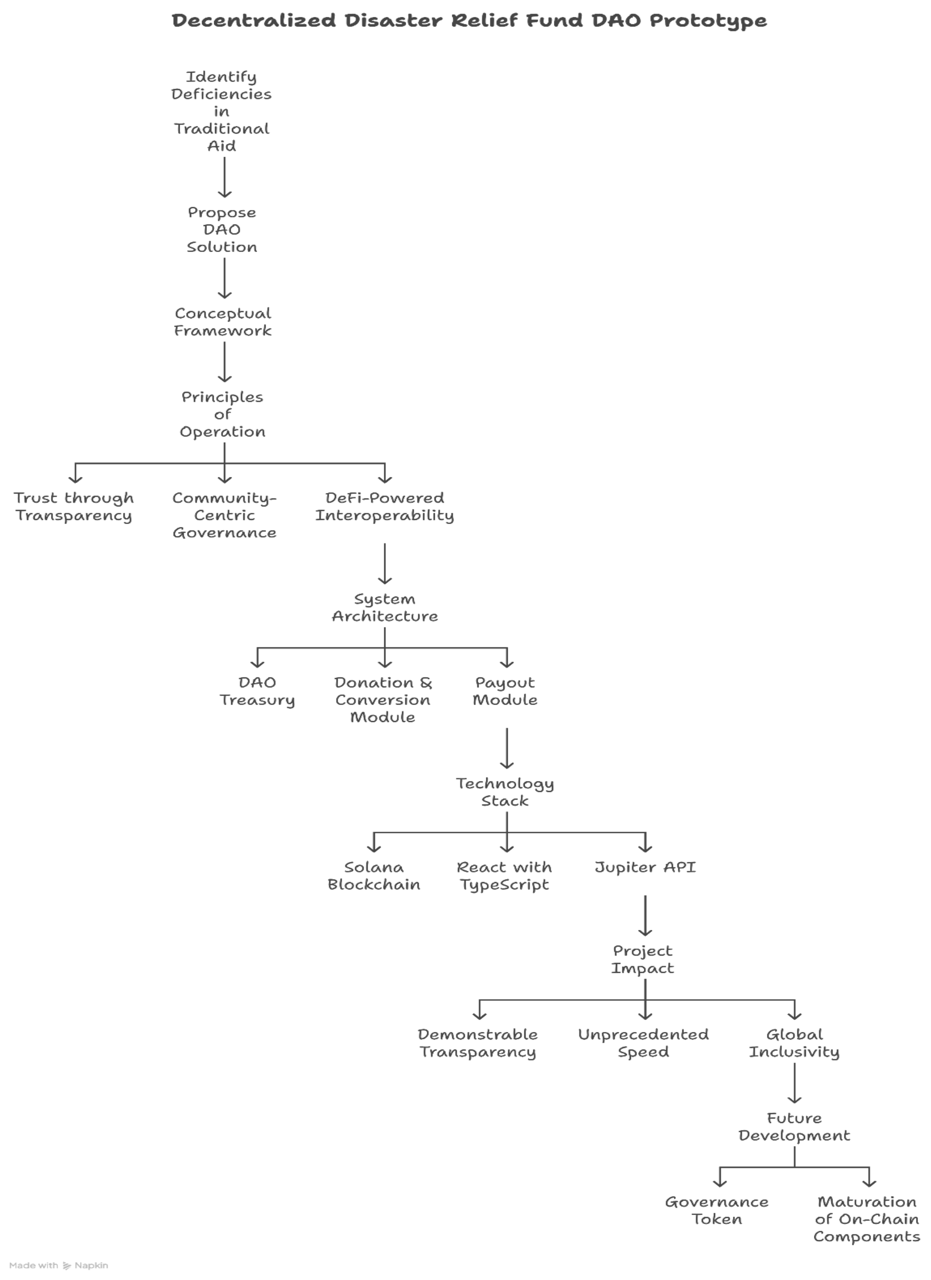
The integration with Jupiter is the technical cornerstone of the project, enabling the "Financial Abstraction" pattern. It allows the DAO to accept any token on Solana without building or managing its own exchange infrastructure.

- **GET /api/quote**:
  - **Frontend Action:** User selects a token and enters an amount.
  - **Backend Action:** The Next.js API route calls Jupiter's /v6/quote endpoint.
  - **Parameters:** inputMint, outputMint (USDC), amount, slippage. The slippage parameter is critical for protecting the user from price changes that might occur between getting the quote and confirming the transaction.
  - **Response:** A JSON object containing the optimal route and expected output amount, which is displayed to the user for confirmation.
- **POST /api/swap**:
  - **Frontend Action:** User confirms the donation after reviewing the quote.
  - **Backend Action:** The API route calls Jupiter's /v6/swap endpoint.
  - **Parameters:** The quoteResponse object from the previous step and the userPublicKey.
  - **Response:** A base64-encoded, "serialized transaction." This is a complete, ready-to-sign package of instructions for the Solana blockchain, which the frontend can pass directly to the user's wallet for approval.

4.2 Technology Stack

Technology/Tool	Category	Role in Prototype

Solana	Blockchain Platform	The underlying public ledger for all transactions. Its high throughput and low fees are essential for the use case. Devnet is used for testing.
React (Next.js)	Frontend Framework	Powers the entire user interface and the lightweight API routes, enabling a fast, server-rendered, single-page application experience.
@solana/wallet-adapter-react	Frontend Library	Provides a simple, secure, and standardized modal for connecting to various Solana wallets, abstracting away connection complexity.
Jupiter API	DeFi Aggregator	The engine for token conversion. It finds the most efficient swap paths across the entire Solana ecosystem, maximizing donation value.
Node.js / Next.js API Routes	Backend	Serves as a secure server-side proxy to communicate with external APIs, protecting sensitive operations from the client.
TypeScript	Language	Enforces type safety across the entire stack, significantly reducing runtime errors and improving developer productivity.
@solana/web3.js	Blockchain Library	The core library used in API routes and client-side scripts to construct, serialize, and parse Solana transactions.



5.0 Usage Instructions

5.1 Making a Donation (Public User)

1. **Navigate & Connect:** Open the web application and click "Connect Wallet" to

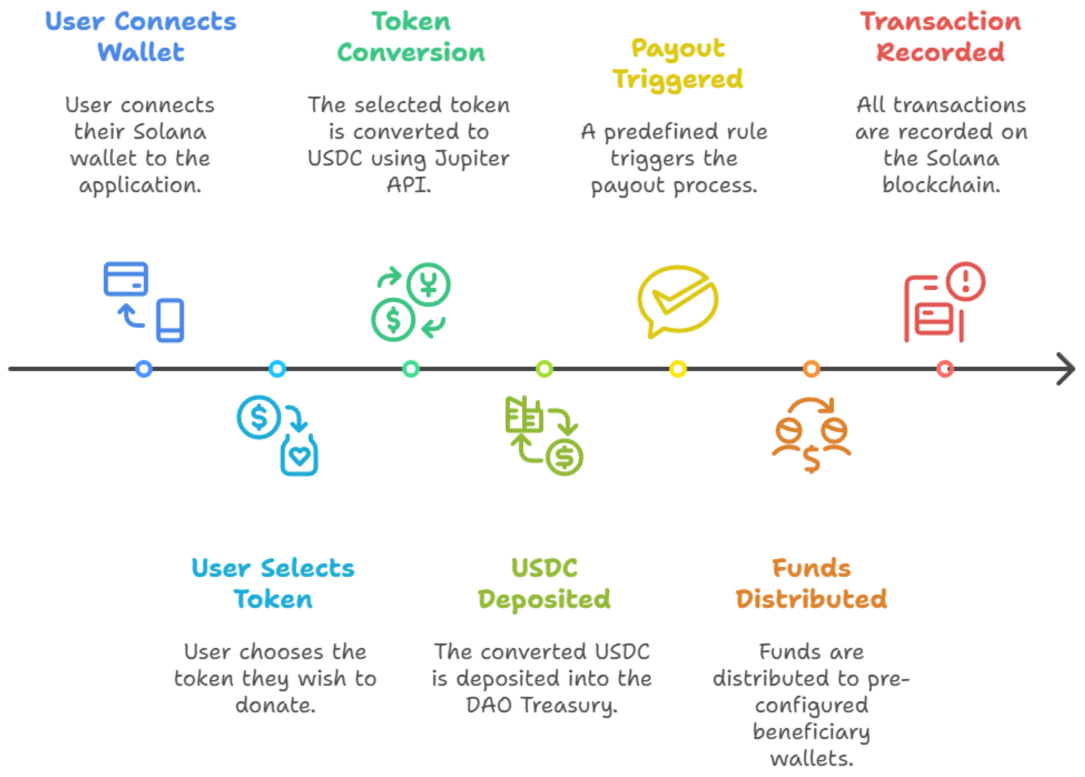
connect your Solana wallet. The process is designed to be familiar to any Web3 user.

2. **Go to Donate Page:** Navigate to the "Donate" page, which is the main hub for contributions.
3. **Select Token & Amount:** Choose the token you wish to donate and enter the amount. The UI provides instant feedback, displaying the estimated USDC value you will be contributing based on a real-time quote.
4. **Donate & Approve:** Click "Donate Now." A transaction approval request will appear in your connected wallet, summarizing the transaction for your final review.
5. **Confirmation:** Once you approve and the transaction is confirmed on the Solana blockchain (typically in seconds), a success notification will appear. The main dashboard will automatically reflect the updated treasury total.

5.2 Triggering a Payout (Administrator)

1. **Navigate to Admin Page:** Go to the restricted "Trigger" page. In a production environment, this route would be protected by robust authentication and authorization checks.
2. **Review Payout:** The page will display a clear summary of the payout action that will be taken based on the system's hardcoded rules, including recipient addresses and amounts.
3. **Trigger & Approve:** Click "Trigger Payouts." The application will construct the necessary transactions. As the treasury holder, you will be prompted to sign these transactions in your connected wallet, providing the final, secure authorization.
4. **Confirmation:** Once approved, the funds are distributed. The "Payout Log," visible to the public, will update with links to the transactions on a block explorer, ensuring end-to-end transparency.

Decentralized Disaster Relief Fund DAO Process



User Connects Wallet

User connects their Solana wallet to the application.

User Selects Token

User chooses the token they wish to donate.

Token Conversion

The selected token is converted to USDC using Jupiter API.

USDC Deposited

The converted USDC is deposited into the DAO Treasury.

Payout Triggered

A predefined rule triggers the payout process.

Funds Distributed

Funds are distributed to pre-configured beneficiary wallets.

Transaction Recorded

All transactions are recorded on the Solana blockchain.