Turbin3 Assignment_2

Part A: User Stories & On-Chain Requirements Document

User Stories

1.USDC-Paid Freelancer:

User story:

As a USDC-paid freelancer, I want to connect my wallet, top up my Vault with a lump-sum paycheck, set a monthly conversion plan, automatically swap slices into SOL, and have that SOL sent back to my wallet—so I never trade manually and can always see what's coming next, what's happened, and the fees I've saved.

1. **Connect Wallet**
   - *Story:* I open the app and click "Connect Wallet" so I can link my USDC account.
   - *Outcome:* My wallet address appears in the app.

2. **Top Up Vault**
   - *Story:* After connecting, I deposit USDC into my Vault so it's ready for conversion.
   - *Outcome:* The app shows my USDC balance in the Vault.

3. **Choose Conversion Schedule**
   - *Story:* I select "On the 1st of each month" so the system knows when to slice and swap.
   - *Outcome:* The schedule "1st of each month" is saved.

4. **Execute Swap**
   - *Story:* On the scheduled date, the system splits my Vault balance into slices and converts one slice into SOL.
   - *Outcome:* I see "Slice 1 of 4: 25 USDC → SOL, fees saved: 0.2 USDC."

5. **Transfer SOL Back**
    - *Story:* Right after each swap, the SOL is sent from the Vault back to my wallet.
    - *Outcome:* My wallet balance increases by the SOL amount.

6. **View Upcoming Conversion**
    - *Story:* I look at the dashboard to see when my next slice will run.
    - *Outcome:* I see "Next swap: August 1, 10:00 AM."

7. **Review Past Conversions**
    - *Story:* I check the history tab to see all slices that have run.
    - *Outcome:* I see dates, USDC amounts, SOL amounts, and fees saved.

2. Retail Crypto Investor ("SOL Stacker")

User story:
As a retail investor, I want to fund my Vault once, pick a weekly amount and slice count, watch each slice convert to SOL and return to my wallet, and then track my total SOL and fees saved—so I can grow my position without logging in.

1. **Connect Wallet**
    - *Story:* I click "Connect Wallet" to link my USDC account.
    - *Outcome:* My wallet address appears.

2. **Fund Vault**
    - *Story:* I deposit USDC into my Vault once so I have funds for ongoing swaps.
    - *Outcome:* Vault balance shows the total USDC I added.

3. **Set Amount & Frequency**
    - *Story:* I enter "$100 weekly in 5 slices" so the system knows how much and how often.
    - *Outcome:* Schedule "$100/week, 5 slices" is saved.

4. **Execute Swap**
    - *Story:* Each week, one slice is swapped into SOL automatically.
    - *Outcome:* I see "Week 1 swap: 20 USDC → SOL, fees saved: 0.1 USDC."

5. **Transfer SOL Back**
    - *Story:* After each swap, the SOL slice is sent back to my wallet.
    - *Outcome:* My SOL balance in wallet increases accordingly.

6. **View Total SOL**
    - *Story:* I go to my dashboard to see the total SOL I've accumulated.
    - *Outcome:* Total SOL amount is displayed.

7. **View Fees Saved**
    - *Story:* I check the savings tab to see cumulative fees slashed.
    - *Outcome:* "Total fees saved: 0.5 USDC" is shown.

3. DAO Treasury Manager

User story:
As a DAO treasury manager, I want our multisig to deposit USDC into a shared Vault, set a daily drip plan, have each drip swapped to SOL and returned to the multisig wallet, and then download a clear report of deposits, swaps, and fees—so our entire team sees full transparency and audit-ready records.

1. Connect Multisig Wallet
    - *Story:* A signer clicks "Connect Multisig" to link our shared USDC account.
    - *Outcome:* DAO's wallet address appears.

2. Deposit DAO Funds
    - *Story:* The multisig deposits USDC into the shared Vault.
    - *Outcome:* Vault balance updates with DAO's USDC.

3. Define Daily Drip Plan
    - *Story:* I enter "$10,000 total, split into 10 daily slices" so the system knows our plan.
    - *Outcome:* "10 slices, one per day" is saved.

4. Execute Swap
    - *Story:* Each day, one slice is swapped into SOL automatically.
    - *Outcome:* I see "Day 1 swap: 1,000 USDC → SOL, fees saved: 1 USDC."

5. Transfer SOL Back
    - *Story:* After each swap, the SOL returns to our multisig wallet.
    - *Outcome:* Multisig's SOL balance increases each day.

6. View Conversion Summary
    - *Story:* I check the dashboard to see how much USDC has been swapped and how much SOL we hold.
    - *Outcome:* Totals for USDC converted, SOL acquired, and fees saved are displayed.

7. Download Audit Report
    - *Story:* I click "Export Report" to get a CSV of every deposit, swap, and fee.
    - *Outcome:* A CSV file with dates, amounts, and fees is downloaded.

Potential On-Chain Requirements

User Story: Connect Wallet

Potential On-Chain Requirements:

- *None required on-chain.*
    - Wallet linking is handled entirely in the browser via standard wallet extensions(Solana Wallet Adapters) supporting the most wallets.(eg: Phantom,Solflare)

User Story: Deposit Funds

Potential On-Chain Requirements:

- Vault PDA creation
    - On the user's first deposit, the program must create a unique "Vault" account on-chain. This special account holds only that user's USDC.

- `deposit()` instruction
    - A smart-contract function that takes USDC tokens from the user's wallet and moves them into the Vault.
    - It checks that the person signing the transaction actually owns the Vault before accepting funds.

- USDC balance update
    - After the transfer, the Vault account's `usdc_balance` field must be incremented by the deposited amount, so the on-chain state always matches reality.

User Story: Define Conversion Plan

Potential On-Chain Requirements:

- `configureDCA()` instruction
    - A function that records the user's plan directly in the Vault account, including:
        - Total amount of USDC they wish to convert.
        - Number of slices to break that amount into.
        - Timing interval, such as a UNIX timestamp or a monthly marker (e.g. "day 1 of each month").

- Schedule fields in Vault
    - The Vault account must have dedicated fields—for example `total_amount`, `slice_count`, and `interval_seconds`—so the plan lives on-chain and can't be tampered with.

User Story: Execute Swap

Potential On-Chain Requirements:

- `executeSwap()` instruction
    - Reads the next slice amount from the Vault's stored plan.
    - Withdraws exactly that USDC from the Vault.
    - Sends it through a DEX aggregator program (Jupiter) to swap into SOL in a single on-chain call.

- State updates
    - After the swap, the Vault's `slices_completed` counter increments by one.
    - The Vault's `next_swap_time` field updates to the following scheduled date.

- Slippage & fee tracking
    - The instruction logs how much was spent versus how much SOL was received, storing the difference so you can report savings later.

User Story: Transfer SOL Back

Potential On-Chain Requirements:

- `transferSOL()` instruction
    - Pulls the newly acquired SOL from the Vault's SOL balance.
    - Sends it directly to the user's personal SOL token account tied to their wallet.
    - Verifies the recipient address matches the original owner of the Vault.

- SOL balance update
    - The Vault account's SOL-held field decreases by the exact amount sent, keeping on-chain accounting precise.

User Story: View Dashboard Summary

Potential On-Chain Requirements:

- Read access to Vault fields
    - No new instructions—front-end simply reads on-chain data:
        - `usdc_balance` for current USDC waiting to convert
        - `next_swap_time` for upcoming swap dates
        - `total_sol_received` for cumulative SOL outflows
        - `total_fees_saved` for the user's cost-saving tally

- Summary Fields
    - These summary fields are stored in the Vault PDA so the dashboard can pull them directly without extra computations.

User Story: Review Historical Records

Potential On-Chain Requirements:

- Event logs on each action
    - Every time `deposit()`, `executeSwap()`, or `transferSOL()` runs, emit a descriptive log event containing:
        - Timestamp of the action
        - Amounts of USDC and SOL moved
        - Fees or slippage saved

Part B: Process Appendix:

Prompt1:
my capstone project is to buid a dca vault non custodial and scheduled transfers and swaps. My project's value proposition is Pasted below. Here is a brainstormed list of all potential user types: pasted below. Based on the value proposition, which 2-5 of these user types are the most critical to focus on for an initial Proof-of-Concept? For each user you recommend, provide a brief rationale explaining why they are essential for proving the project's core value." Core value proposition and product market fit. The non-custodial DCA Vault empowers freelancers, retail investors, and small DAOs to automate recurring conversions from USDC into SOL, significantly reducing slippage, transaction fees, and timing anxiety compared to lump-sum swaps. By slicing purchases into scheduled micro-swaps, the platform doesn't just cut costs, it also cushions users against market price uncertainty, smoothing out volatility and averaging entry prices over the long term. With transparent, trustless smart contracts and an intuitive UX, users retain full custody of their assets, eliminating centralized risks. This "set-and-forget" automation simplifies SOL accumulation, enabling disciplined dollar-cost averaging without manual intervention or complex configurations, delivering both tangible cost savings and confidence through every market cycle. Users:
USDC-Paid Freelancers & Gig Workers Retail Crypto Investors ("SOL Stackers")
Small-to-Mid-Size DAOs and Treasury Managers crypto newbie. Stakers someone who doesnt want the hustle to manually swap and invest in sol each month or regualarly and dont want to get in bw of price uncertainity and averaging out is better for him.?

AI output:
USDC-Paid Freelancers & Gig Workers

- High frequency of USDC inflows: They're receiving stablecoin payments regularly (weekly or bi-weekly), so they have both the need and the volume to appreciate micro-swaps.
- Immediate pain point: Manual swapping eats time, incurs unpredictable fees/slippage, and adds cognitive load. Automating this flow will showcase your "set-and-forget" value in a real-world cash-flow scenario.
- Feedback loop: You can instrument and measure before-vs-after slippage/fee savings directly on live incomes.

Retail Crypto Investors ("SOL Stackers")

- Large addressable market: Everyday HODLers who want disciplined, emotion-free accumulation of SOL.
- Clear ROI story: Even modest monthly buy-ins (e.g. USDC $500–$1,000) can yield noticeable slippage reduction over time, making a compelling marketing case.
- Brand evangelism potential: Crypto communities love tooling that “buys the dip automatically” and share it in Telegram/Discord channels.

Small-to-Mid-Size DAOs & Treasury Managers

- Bigger ticket sizes: DAOs often manage five- to six-figure USDC treasuries—so the dollar savings on slippage (and the auditability of on-chain receipts) become very tangible.
- Governance buy-in: A successful pilot here can unlock word-of-mouth referrals across multiple DAOs looking to optimize treasury operations.
- Strategic partnerships: DAO tooling integrations (Snapshot, Discord, Discourse) give you a faster path to onboarding.

Prompt2: For a project with this value proposition and focusing on these prioritized user types defined above help map out the key functions or interactions each user would need to perform." Output: A list of key functions and interactions mapped to your prioritized user types.

AI output:1. USDC-Paid Freelancers & Gig Workers

- USDC Deposit & Auto-Detection
    - “Deposit USDC” or connect wallet to auto-pull incoming payments
- DCA Schedule Setup
    - Choose trigger (“on each paycheck” or cron-style)
    - Define allocation amount (fixed USDC or % of each deposit)
- Micro-Swap Parameters
    - Set frequency (e.g. every 1 minute/hour/day)
    - Split into N orders (e.g. over 2 orders)
    - Optional price-range guardrails (min/max price)
- Live Summary & Controls
    - Recurring summary (total USDC, per-order USDC, estimated end date, platform fee)
    - One-click Pause/Resume or Edit schedule

- Notifications & Export
    - Real-time swap alerts (slippage saved, gas used)
    - Download audit-ready CSV/PDF of transactions

2. Retail Crypto Investors (“SOL Stackers”)

- Onboarding & USDC Top-Up
    - “Deposit USDC” or fund via wallet
- Quick-Start DCA Wizard
    - Select recurring time period (daily/weekly/monthly)
    - Enter total allocation (e.g. $120 USDC) and number of micro-swaps
- Advanced Swap Settings
    - Frequency picker (every X minutes/hours/days)
    - Price-range optional filter
- Performance Dashboard
    - Historical swaps, average entry vs. VWAP, total SOL acquired
    - “Cost-Savings” KPI (fees/slippage avoided)
- Alerts & Emergency Stop
    - Price-threshold notifications
    - One-click stop or withdraw all USDC

3. Small-to-Mid-Size DAOs & Treasury Managers

- Multi-Sig Wallet Integration
    - Connect DAO treasury via Gnosis-style signer flow
- Bulk Vault Creation & Scheduling
    - “Create Vault” template: deposit USDC, define time period, split into orders, optional price range
    - Clone/edit multiple schedules in batch
- Governance Proposal Flow
    - Draft → circulate for sign-off → execute DCA schedule on-chain
- Treasury Dashboard & Reporting
    - Aggregate USDC under management, SOL acquired, slippage saved
    - Detailed logs per vault: signer approvals, tx receipts
- Webhook & Export
    - Push notifications to Discord/governance channels on each swap
    - CSV/JSON export for audit and financial reporting

Prompt3: "Based on these top 1-2 critical user interactions derived above , what are the key technical requirements needed to build a proof-of-concept?"
AI output:1. Wallet Connection & USDC Deposit Detection

> *User clicks "Connect Wallet" → deposits USDC (or we auto-detect incoming payments) → funds their Vault*

Technical requirements:

- Frontend
  - Solana Wallet Adapter (Phantom, Solflare, etc.) to connect user wallets.
  - React UI components for "Connect" button and token-balance display.
- On-chain
  - SPL-Token vault contract (Anchor) that:
    - Defines a Vault PDA to hold user USDC.
    - Implements a `deposit()` instruction to transfer USDC from user → Vault.
- Off-chain
  - WebSocket subscription (via @solana/web3.js) or periodic RPC polling to watch for new USDC deposits into each Vault.
  - Lightweight datastore (e.g. in-memory or a simple JSON/Redis) to index active Vaults and their deposit status.

2. Scheduled Micro-Swaps (DCA Engine)

> *User defines "$X total → N orders every T interval (with optional price guardrails)" → system executes those swaps on schedule*

Technical requirements:

- On-chain
  - Anchor DCA contract with a `configureDCA()` instruction that:
    1. Stores parameters in a Vault state account (total amount, interval, split count, min/max price).
  - `executeSwap()` instruction that:
    1. Pulls the next tranche of USDC from the Vault.
    2. Calls into a DEX aggregator (e.g. Jupiter or Serum) to swap USDC→SOL.
    3. Credits SOL back to user's associated token account.

- Off-chain
    - Scheduler/Crank service (Node.js cron or serverless function) that, every T interval:
        1. Queries all Vault PDAs for due swaps.
        2. Generates signed `executeSwap()` transactions (via Anchor or @solana/web3.js).
        3. Submits them to the network and watches for confirmation.
- DEX Integration
    - Use Jupiter Route API or Serum order instructions to build the best-price swap on-chain.
    - Optionally fetch on-chain price oracles (Pyth/Chainlink) to enforce guardrails before dispatch.

Minimal Supporting Services

- Persistence: simple database (Postgres/Redis) to track Vault IDs and schedule state.
- Notifications: webhook or email hook in your scheduler to surface “swap executed” events.
- Security & UX: display gas-fee estimates, transaction statuses, and audit-style receipts in the frontend.

With just these building blocks—wallet onboarding → deposit → on-chain schedule storage → off-chain cron → on-chain swap execution—you’ll have a living PoC that proves your non-custodial, scheduled DCA vault in action.

Prompt4:From the function map you just created, identify the top 1-2 most critical user stories or interaction paths essential for a POC.
AI output:1. Wallet Onboarding → USDC Deposit → Vault Creation

User story:

> As a USDC-paid freelancer, I want to connect my Solana wallet and deposit USDC into a new Vault so that I can fund an automated DCA plan without manual swaps.

Key steps:

1. User clicks “Connect Wallet” → approves via Phantom/Solflare.

2. UI shows their USDC balance.

3. User deposits (or auto-detects) USDC into the Vault PDA via an on-chain `deposit()` call.

4. Vault state (balance + owner) is recorded and displayed in the dashboard.

2. DCA Schedule Setup → First Micro-Swap Execution → Feedback

User story:

> As a retail SOL stacker, I want to define "$X total → N micro-swaps every T interval" and see my first swap execute (with slippage report) so I know the system works end-to-end.

Key steps:

1. User opens "Recurring" tab, enters total USDC, number of slices, and interval.
2. Frontend calls on-chain `configureDCA()` to store parameters.
3. Off-chain scheduler/crank picks up Vault, submits `executeSwap()` for tranche #1 via Jupiter
4. UI receives confirmation and shows "Swap 1 complete: $Y USDC → Z SOL, slippage saved: A%."

Prompt:Review my core user functions/stories and requirements . Considering my project's refined value proposition , do these stories truly hit the mark? Are the requirements granular enough to map to specific technical components (e.g., database schemas, API endpoints, specific blockchain programs)? What's missing or unclear?"

AI output:
Blockchain & Smart Contract Specificity

Your stories are currently vague on precisely how on-chain accounts and interactions will be structured and executed.

- Missing:
    - Specific Anchor program instructions (`deposit`, `configureDCA`, `executeSwap`) parameters and account structure.
    - Clear PDA definitions (seeds, bump, state stored per vault).

Off-chain Scheduler & Worker

Your "scheduler" is crucial—but right now, it's only broadly defined. There's no clear specification on:

- Missing:
    - Exact off-chain logic for polling blockchain state.
    - Detailed transaction-building and submission flow.

DEX Integration Clarity

Currently, you assume Jupiter integration, but specifics are lacking:

- Missing:
    - Jupiter API endpoints or program instructions explicitly defined.
    - Handling of slippage and fallback mechanisms.
- Recommendation:
  Clearly document interaction pattern:
    - Jupiter API endpoint: `https://quote-api.jup.ag/v6/quote`
    - Define slippage thresholds in swaps clearly in contract state or off-chain fallback logic.

UX and Notifications

The stories indicate UX (notifications, alerts), but specifics aren’t well-defined yet.

- Missing:
    - Clear API/Webhook definitions for notifying front-end.
    - Types of notifications and status updates.

Critical Questions to Clarify Immediately:

- Who pays gas fees for the recurring swaps?
    - Vault owners, treasury, or a fee-payer account?
- How do you securely manage private keys or signing authority for scheduled swaps?
- What happens if a swap transaction fails?
    - Retry logic? Error alerting?

Rationale for Changes From all parts:

After reviewing the AI's outputs and the user stories it provided, I realized they lacked an in-depth explanation of the code flow for each story. We therefore brainstormed the on-chain functions needed—such as `deposit()`, `configureDCA()`, `executeSwap()`, and `transferSOL()`—and integrated them into the original specification. We also defined helper integrations like the Jupiter swap API and dashboard components, detailing what each view will display, and added dedicated user stories for the on-chain requirements.

1. From Broad Functions to Atomic User Stories

- Before: Had large, overlapping stories ("Connect wallet, top up, set schedule, swap, dashboard…" all in one).
- Change: Split each into single-action stories (e.g. "Connect Wallet," "Top Up Vault," "Choose Conversion Schedule," "Execute Swap," "Transfer SOL Back," etc.).

2. De-Jargoning for Non-Technical Stakeholders

- Before: Many blockchain terms crept in ("PDA," "CPI," "RPC polling," "PDAs," "CPI," "keeper programs").
- Change: Rewrote every story in plain English (e.g. "I deposit USDC," "I pick the 1st of each month," "I see my next swap date," "SOL arrives in my wallet").

3. Helper Services & Dashboard Integration

- Before: Glossed over how the front-end would show history, upcoming swaps, or fee savings.
- Change: Spelled out dashboard requirements under each story. Also emitted logs of transactions for better

4. Incorporating Jupiter (DEX)

- Before: The DEX integration and crank service were mentioned but loosely.
- Change:Integrated "calls into Jupiter's Route API" as part of `executeSwap()`.