I cannot generate a PDF file directly. However, I can provide you with the complete plan in Markdown format. You can copy this text, paste it into a text editor (like VS Code or even Google Docs), and then save or export it as `ShadowSwapPlan.pdf`.

Here is the complete plan based on our conversation and your documents.

ShadowSwap: MVP Build Plan

Project: A privacy-preserving DEX on Solana (SOL/USDC).
Core Tech: Anchor, Arcium, Sanctum (Jito-only), and Next.js.

1\. Core Architecture & Components

This project is built from four distinct components that must be built and deployed separately.

*1. Frontend (Next.js App):** The user-facing website where users connect wallets, encrypt orders, and see their order status.
*2. Anchor Smart Contract (Solana Program):** The on-chain program that manages user escrows (`Escrow` PDAs) and stores the encrypted orders (`EncryptedOrder` PDAs).
*3. Arcis Matching Logic (MPC Program):** The private matching rules (e.g., price-time priority) written in Arcis DSL. This is not a server; it's compiled bytecode that you register with the Arcium network.
*4. Off-Chain Settlement Bot (Node.js Server):** A 24/7 backend service that listens for on-chain match events and privately submits the final settlement transaction to Sanctum.

2\. Recommended Folder Structure (Monorepo)

```
/ShadowSwap_Project
%
% %Ø=ÜÂapps
%  %
%  % %Ø=ÜÂfrontend           # Next.js UI
%  %  % %/%pages
%  %  % %/%components
%  %  % %/%package.json
%  %
%  % %Ø=ÜÂanchor_program      # Anchor Smart Contract
%  %  % %/%programs
%  %  %  % %/%shadow_swap
% % %   % %lib%rs
% % % %/te%sts
% % % %/ar%cis_logic
% % %  % %m%atching_logic.arc  # Your MPC matching logic
% % % %A%nchor.toml
% %
% % %Ø%sËÂettlement_bot    # Off-chain Node.js bot
%    % %in%dex.ts
%    % %pa%ckage.json
%
% %Ø%pËÂackages
% %
% % %Ø%sËÂhared_types      # (Recommended) Shared types for frontend/bot
%    % %in%dex.ts
%    % %pa%ckage.json
%
% %Ø%PËÂroject_Details        # Your original PDFs
% % %Sh%adowSwap.pdf
% % %Sh%adowSwap_MVP.pdf
% % %..%
%
% %pa%ckage.json              # Root package.json for monorepo workspaces
% %R%EADME.md
```

3\. End-to-End Workflow (Data Flow)

This is how all components communicate, from order to settlement.

1. **Frontend !’ Anchor Contract:**

* A user submits an order (e.g., "Buy 10 SOL at 150 USDC").
*The Frontend** uses the Arcium SDK to encrypt this into a `cipher` payload.
*The Frontend sends a transaction to the Anchor Contract** calling the `submit*encrypted*order` instruction with this payload:
* `cipher: Vec<u8>`
* `eph_pub: [u8;32]`
* `nonce: u128`
* `order_id: [u8;32]`
CRITICAL: The Anchor contract never* sees the plaintext price or amount.

2. **Anchor Contract !’ Arcium Cluster:**

*The Anchor Contract** receives the encrypted data.
* It performs a CPI to transfer the user's funds (SOL or USDC) into their `Escrow` PDA.
* It creates the `EncryptedOrder` PDA to store the `cipher`.
* It makes a CPI to Arcium's `queue_computation` function, "pinging" the Arcium network to start matching.

3. **Arcium Cluster (Off-Chain):**

* The Arcium network sees the new task in its on-chain queue.
* The MPC cluster fetches the encrypted orders from your contract's PDAs.
*It runs your registered Arcis Matching Logic** (`matching_logic.arc`) to find a match.

4. **Arcium Cluster !’ Anchor Contract:**

*Upon finding a match, the Arcium network sends a transaction back to your Anchor Contract**, calling the `match_callback` instruction with the result (e.g., buyer, seller, amount).

5. **Anchor Contract !’ Settlement Bot:**

* The `match_callback` instruction verifies the call is from Arcium.
* It updates the `status` of both matched orders to `5` (Matched*Pending*\Exec).
* It emits an on-chain event: `MatchQueued`.

6. **Settlement Bot (Off-Chain):**

*The Settlement Bot**, running on Vercel/Railway, is listening for the `MatchQueued` event.
* It parses the event, fetches the two `Escrow` PDAs, and builds the atomic settlement transaction (e.g., "Swap SOL from Seller's Escrow with USDC from Buyer's Escrow").

7. **Settlement Bot !’ Sanctum Gateway !’ Solana:**

*The Bot** sends this transaction to the Sanctum Gateway API, using its private `GATEWAY*API* KEY` and specifying the `strategy: 'private_only'`.
*Sanctum** routes the transaction directly to a Jito validator, bypassing the public mempool, ensuring MEV-resistance. The trade is settled on-chain.

4\. Phase-by-Phase Build Plan (MVP)

Phase 1: Environment & Foundations (Day 1)

*Dev A (Anchor):**

* Set up Rust, Solana CLI, Anchor CLI.
* Run `anchor init anchor_program`.
* Define the account structs in `lib.rs`: `OrderBook`, `EncryptedOrder`, and `Escrow` based on the LLD.

*Dev B (Frontend/Arcium):**
* Set up Node.js, Next.js.
* Create the `frontend` app with wallet connection.
* Use `arc-cli` to register your MPC cluster and get your `ARCIUM*CLUSTER*ID` and MXE keys.

Phase 2: On-Chain Order Logic (Day 2)

*Dev A (Anchor):**
* Implement the `submit*encrypted*order` instruction. This must:
1. Receive the encrypted payload (`cipher`, `eph_pub`, etc.).
2. Perform the CPI to transfer tokens into the `Escrow` PDA.
3. Initialize the `EncryptedOrder` PDA with `status = 0` (Active).
4. Make the CPI to Arcium's `queue_computation`.

*Dev B (MPC Logic):**
* Write the price-time matching logic in `matching_logic.arc`.
* Compile it to `.arc` bytecode and register it with Arcium.

Phase 3: Frontend Encryption & Submission (Day 3)

*Dev B (Frontend):**
* Build the SOL/USDC order form.
* Integrate the Arcium SDK to encrypt the form data into the payload.
* Write the client-side code to send the transaction to the `submit*encrypted*order` instruction.

*Dev A (Anchor):**
* Implement the `match_callback` instruction. This must:
1. Verify the caller is Arcium.
2. Parse the `MatchResult`.
3. Update the status of both `EncryptedOrder` accounts to `5` (Matched*Pending*\Exec).
4. Emit the `MatchQueued` event.

Phase 4: Off-Chain Settlement (Day 4)

*Dev B (Bot):**
* Create the `settlement_bot` Node.js project.
* Add code to connect to the Solana WSS endpoint and listen for the `MatchQueled` event.
* Write the logic to build the atomic settlement transaction.

*Dev A (Bot/Sanctum):**
* Write the `fetch` logic for the bot to send the built transaction to the Sanctum Gateway API.

* Ensure the request uses the `GATEWAY*API*KEY` and `strategy: 'private_only'`.
* Implement the 3x retry logic as specified in the LLD.

Phase 5: Full UI & Cancellation (Day 5)

*Dev B (Frontend):**
* Add WebSocket/polling logic to the frontend to listen for order status changes (e.g., `MatchQueued`, `SettlementSucceeded`) and update the UI.
*Dev A (Anchor):**
* Implement the `cancel_order` instruction. This must:
1. Verify the signer is the order `owner`.
2. Verify order `status == 0` (Active).
3. Perform a CPI to transfer funds from the `Escrow` PDA back to the user.
4. Update order `status = 2` (Cancelled).

Phase 6 & 7: Testing, Demo, & Post-MVP

*Both:** Test all edge cases (insufficient funds, price mismatch, cancel a matched order).
*Both:** Record a full end-to-end demo.
*Post-MVP:** Begin work on Phase 2 features, starting with a dynamic order struct for multi-token support, as outlined in the original `ShadowSwap_MVP.pdf`.

5\. Environment Variables

`apps/frontend/.env.local`

```

# Public keys

NEXT*PUBLIC*SOLANA*RPC*HOST="https://api.devnet.solana.com"
NEXT*PUBLIC*ANCHOR*PROGRAM*ID="<Your*Anchor*Program_ID>"
NEXT*PUBLIC*ARCIUM*CLUSTER*ID="<Your*Arcium*Cluster_ID>"
NEXT*PUBLIC*USDC_MINT="EPjFW..." # Devnet USDC
NEXT*PUBLIC*WSOL_MINT="So111..."
```

`apps/settlement_bot/.env`

```

# Secret keys

SOLANA*RPC*HOST="https://api.devnet.solana.com"
SOLANA
```