## Prompt: Create a Production-Ready, Technically Feasible Decentralized Coinsensus dApp

**Objective:** Develop a fully decentralized, mobile-first application named "Coinsensus" that enables secure, private, and transparent social consensus and voting. The dApp must transition from a Firebase-based simulation to a blockchain-native architecture, incorporating advanced decentralized identity (DID) and verifiable credential (VC) management, Zero-Knowledge Proofs (ZKPs) for privacy, and a sustainable economic model.

**Core Principles:**

* **Decentralization:** Minimize reliance on centralized services for core functionality (identity, data storage, voting logic).
* **Security:** Implement robust cryptographic practices and leverage hardware security modules.
* **Privacy:** Ensure user anonymity and unlinkability through pseudonymous DIDs and ZKPs.
* **Transparency:** All core voting processes must be publicly auditable on the blockchain.
* **User Control:** Users must retain full control over their identity and credentials.
* **Feasibility:** Acknowledge the complexity and propose realistic, step-by-step solutions for real-world implementation.

### Phase 1: Foundational Blockchain Migration & Core Logic

This phase replaces the current Firebase backend with a blockchain and establishes the fundamental smart contract architecture.

1. **Blockchain Network Selection:**
   * **Requirement:** Choose a public, permissionless, EVM-compatible blockchain network or Layer 2 (L2) scaling solution.
   * **Justification:** Prioritize low gas fees, high transaction throughput, and strong security. Ethereum L2s (e.g., Polygon zkEVM, Arbitrum, Optimism, Base, zkSync Era) are preferred for their balance of scalability and security inheritance from Ethereum.
   * **Action:** Explicitly state the chosen network (e.g., "Polygon zkEVM Testnet for initial development, transitioning to Mainnet").
2. **Smart Contract Development (Solidity):**
   * **Requirement:** Develop and deploy a suite of audited Solidity smart contracts.
   * **Action:**
     + CoinsensusToken.sol **(ERC-20):**
       - Implement a standard ERC-20 token (using OpenZeppelin's ERC20 for security).
       - Define a fixed or controlled total supply.
       - Include functions for token transfers, approvals, and potentially a mechanism for initial distribution (e.g., a faucet for testnet, or a controlled mint for mainnet).
       - This token will be used for campaign launch fees and voter rewards.
     + DIDRegistry.sol**:**
       - A smart contract to register and manage user's Master DIDs (blockchain addresses as DIDs, e.g., did:pkh or did:ethr).
       - Functions for registering a new DID, updating DID Document hashes (if applicable), and resolving DIDs.
     + CoinsensusCampaignFactory.sol**:**
       - A factory contract responsible for deploying new VotingCampaign.sol instances.
       - It must verify the caller's eligibility (e.g., by checking for an "Entity VC" via ZKP verification, or allowing "Basic Voters" to launch Public/General campaigns with a fee).
       - It will collect campaign launch fees in CoinsensusToken.
       - Emit an event (CampaignCreated) upon successful deployment, including the new campaign contract's address and key metadata.
     + VotingCampaign.sol **(Upgradeable):**
       - This will be the core contract for each individual voting campaign.
       - **Upgradeability:** Must be implemented using an upgradeable pattern (e.g., OpenZeppelin UUPS proxies) to allow for bug fixes and feature additions post-deployment.
       - **Data Storage:** Store campaign title, description, type (Public/General, Multiple Position Ballot, Weighted), options/positions/weights, requiredVCs (as hashes or references), start/end times, and current vote results.
       - **Functions:**
         * castVote(bytes calldata \_proof, bytes calldata \_publicSignals, ...): Accepts a ZKP from the voter to prove eligibility and vote choice (for private ballots).
         * getResults(): Public view function to retrieve current vote tallies.
         * hasVoted(address voterAddress): Public view function to check if an address has already voted in this specific campaign.
       - **Eligibility Logic:** Implement on-chain logic to verify voter eligibility based on requiredVCs (initially, simple address checks; later, ZKP verification).
       - **Vote Aggregation:** Securely update vote counts, considering campaign type (single, multiple, weighted, multi-position).
       - **Events:** Emit events for VoteCast, ResultsUpdated, etc., for off-chain indexing.
     + **Security Best Practices:** All smart contracts must adhere to Solidity best practices (fixed compiler versions, reentrancy guards, checks-effects-interactions pattern, explicit visibility, etc.). Use OpenZeppelin contracts extensively.
3. **Frontend Web3 Wallet Integration (React Native):**
   * **Requirement:** Replace Firebase authentication with real Web3 wallet interaction.
   * **Action:**
     + Integrate a Web3 library like ethers.js or web3.js for blockchain interaction.
     + Implement a **"Connect Wallet"** feature using a library like wagmi or web3modal to support popular mobile wallets (MetaMask Mobile, Trust Wallet, WalletConnect).
     + Display the connected user's **wallet address** (which now acts as their primary on-chain identifier).
     + All user actions that modify blockchain state (creating campaign, casting vote) must trigger a wallet transaction request for user confirmation and signature.

### Phase 2: Advanced Decentralized Identity & Privacy

This phase introduces the sophisticated identity and privacy features.

1. **Biometric Key Management & "One Account-One Key":**
   * **Requirement:** Securely generate and manage a private key tied to biometrics, ensuring a single account per user.
   * **Action:**
     + **Secure Hardware Integration:** Leverage device-level Hardware Security Modules (HSMs) like Apple's Secure Enclave (iOS) and Android's TrustZone/StrongBox.
     + **Mechanism:**
       - **On First Use (Registration):** The dApp initiates a request to the HSM. The HSM securely generates a unique private key within its environment. The biometric scan (fingerprint, Face ID, etc.) then acts as an unlock mechanism for this key. The dApp then takes the public key derived from this HSM-managed private key and attempts to register it as a new Master DID on the DIDRegistry smart contract.
       - **On Subsequent Uses (Login):** The dApp requests the HSM to use the previously generated private key (unlocked by biometrics) to sign a challenge. If successful, the dApp checks if the corresponding public key/DID is already registered on the DIDRegistry. If found, the user is logged into the existing account. If not, it indicates a new user flow.
     + **No Biometric Data Storage:** Emphasize that raw biometric data, or the derived seed/private key, is **never stored centrally** or transmitted off the device. Only the public key (or DID) is registered on-chain.
2. **Verifiable Credentials (VCs) Implementation:**
   * **Requirement:** Enable users to acquire, store, and present VCs for eligibility, with privacy.
   * **Action:**
     + **VC Issuance Flow (Real-World Simulation):**
       - **Issuer Backend:** Describe a conceptual "Issuer Service" (e.g., a real government, school) that performs off-chain KYC/KYB.
       - **VC Generation:** Upon successful off-chain verification, the Issuer Service cryptographically signs and issues a VC (W3C VC format, JSON-LD) to the user's Master DID.
       - **Temporary DIDs:** The Issuer Service may receive a temporary DID (derived from the Master DID, unlinkable on-chain) during the off-chain verification process to protect the Master DID's privacy.
     + **On-Device VC Wallet:** The dApp must securely store the user's issued VCs locally on their device (e.g., encrypted storage, device keystore).
     + VCRegistry.sol **(Optional/Advanced):** A smart contract to register public keys of trusted VC issuers and potentially their revocation registries.
3. **Zero-Knowledge Proofs (ZKPs) for Privacy & Eligibility:**
   * **Requirement:** Use ZKPs to prove VC possession and voting eligibility without revealing personal identity or VC contents.
   * **Action:**
     + **ZKP Circuit Design:** Develop specific ZKP circuits (e.g., using circom for ZK-SNARKs) for each type of eligibility proof:
       - "I possess a valid 'government-voter' VC issued by 'Alpha Country'."
       - "I am eligible to launch an 'Entity' campaign based on my 'Company Entity VC'."
       - "I have not voted in this specific campaign before." (This can also be handled by smart contract state).
     + **On-Device Proving:** Integrate a ZKP proving library (snarkjs or similar) into the mobile dApp. When a user attempts to vote in a VC-gated campaign, the dApp generates the relevant ZKP on the user's device.
     + **On-Chain ZKP Verification:** Update the VotingCampaign.sol and CoinsensusCampaignFactory.sol contracts to include ZKP verifier functions. These functions will accept the ZKP and public signals, verifying the proof on-chain before allowing the action (vote/campaign launch).

### Phase 3: Scalability, UX, and Robustness

This phase optimizes performance and builds resilience.

1. **Decentralized Data Indexing & Querying:**
   * **Requirement:** Efficiently query on-chain data without scanning the entire blockchain.
   * **Action:**
     + **The Graph Protocol:** Develop and deploy a **subgraph** for your dApp.
       - The subgraph will listen to events emitted by your CoinsensusCampaignFactory (for new campaign deployments) and individual VotingCampaign contracts (for vote casts and result updates).
       - It will index all relevant campaign metadata, vote counts, and transaction IDs.
     + **Frontend Queries:** Update the CampaignList and ExistingCampaignsViewer components to query The Graph's GraphQL API for campaign data, search functionality, and live results. This replaces Firebase's querying.
2. **Enhanced Frontend UX for Blockchain Interactions:**
   * **Requirement:** Abstract blockchain complexity for a smooth user experience.
   * **Action:**
     + **Gas Estimation:** Before any transaction, provide real-time gas fee estimates (using Web3 library functions) and display them clearly to the user in confirmation modals.
     + **Transaction Status Feedback:** Implement comprehensive UI states for pending, confirmed, and failed transactions, with clear messages and links to block explorers.
     + **Robust Error Handling:** Catch and display specific blockchain errors (e.g., "insufficient funds," "user rejected transaction," smart contract reverts) in user-friendly modals.
     + **Confirmation Modals:** Ensure all critical on-chain actions have a clear confirmation modal showing the action details and estimated fees.
3. **Key Recovery Mechanisms:**
   * **Requirement:** Provide secure, decentralized ways for users to regain account access if their device is lost/damaged or biometrics change.
   * **Action:**
     + **Smart Contract Wallet (Account Abstraction):** Implement user accounts as smart contract wallets (e.g., leveraging ERC-4337 or a custom implementation). This allows for programmable recovery logic.
     + **Social Recovery:** Integrate a social recovery module. Users designate a set of "guardians" (other DIDs/addresses). A threshold of guardians can approve a transaction to change the account's controller to a new biometric-derived key on a new device.
     + **Multi-Factor/Multi-Signature (Optional):** Allow users to link additional keys (e.g., hardware wallet, cloud backup of an encrypted seed phrase) to their smart contract wallet, requiring multiple signatures for critical actions or recovery.
4. **Account/Credential Revocability:**
   * **Requirement:** Enable issuers to revoke VCs and for the dApp to recognize these revocations.
   * **Action:**
     + **Decentralized Revocation Registry:** Implement an on-chain "Revocation Registry" (or leverage a W3C-compliant Credential Revocation List - CRL) where issuers can publish the revocation status of VCs they've issued.
     + **Smart Contract Verification:** Update VotingCampaign smart contracts to query this revocation registry (via on-chain calls) when verifying ZKPs of VCs, denying eligibility if the VC is revoked.
     + **Timed VCs:** For temporary roles (e.g., student, employee), issue VCs with built-in expiry dates, automatically invalidating them after the period.

### Phase 4: Production Readiness & Beyond

This phase ensures the dApp is robust, secure, and sustainable for real-world use.

1. **Refined Tokenomics & Incentives:**
   * **Requirement:** Establish a clear and sustainable economic model for the CoinsensusToken.
   * **Action:**
     + **Detailed Fee Structure:** Define precise CoinsensusToken fees for launching different campaign types (Multiple Position Ballot, Weighted) based on the simulated eligibleVoterCount (which would be derived from on-chain VC data in a real setup).
     + **Voter Rewards:** Implement smart contract logic to distribute a portion of campaign launch fees (or a pre-allocated pool of CoinsensusToken) as rewards to eligible voters upon successful vote casting or campaign completion.
     + **Treasury Management:** Consider a protocol treasury (governed by a DAO) to fund ongoing development, security, and maintenance from a small percentage of fees.
2. **Comprehensive Security Measures:**
   * **Requirement:** Ensure the entire dApp stack is secure against known vulnerabilities.
   * **Action:**
     + **Professional Smart Contract Audits:** Mandatory before mainnet deployment. Engage reputable blockchain security firms.
     + **Rigorous Testing:** Extensive unit, integration, and fuzz testing for all smart contracts and ZKP circuits.
     + **OpenZeppelin Standards:** Continue to leverage battle-tested OpenZeppelin contracts for all standard components.
     + **Bug Bounty Program:** Launch a public bug bounty program to incentivize white-hat hackers.
     + **Secure Frontend Practices:** Adhere to secure coding guidelines for React Native (input validation, XSS/CSRF prevention).
3. **Decentralized Frontend Hosting:**
   * **Requirement:** Host the dApp frontend in a censorship-resistant and immutable manner.
   * **Action:**
     + Deploy the static build of the React Native dApp (compiled for web, if applicable, or served via IPFS for mobile app updates) to **IPFS or Arweave**.
     + Link a traditional domain name to the decentralized content hash (CID) using services like ENS (Ethereum Name Service) or IPFS gateways.
4. **Monitoring, Analytics, and Maintenance:**
   * **Requirement:** Ensure continuous operation, performance tracking, and timely updates.
   * **Action:**
     + Implement comprehensive monitoring for smart contract events, transaction statuses, gas prices, and dApp performance.
     + Set up alerts for unusual activity or potential issues.
     + Plan for regular smart contract upgrades (via proxy patterns) and dApp updates.

**Important Considerations for the AI/Development Team:**

* **Iterative Development:** This is a multi-year project. Prioritize building and testing each phase iteratively.
* **Mobile-First Design:** All UI/UX decisions should prioritize the mobile experience.
* **Performance:** Optimize smart contracts for gas efficiency and frontend for responsiveness.
* **User Education:** The dApp must simplify complex blockchain concepts for mainstream users.
* **Legal & Regulatory:** Acknowledge that legal counsel will be required for real-world deployment, especially concerning tokenomics, biometrics, and voting legality.

This prompt provides a detailed blueprint for building a cutting-edge decentralized application that aligns with your vision for Coinsensus.

The following code is only a concept based on a Firebase context. Replit agent should only consider this as basis, and not an actual production-ready code. Use a sample Biometric seed for simulation purposes to enable account creation process without undergoing actual biometrics scan. If necessary, and for convinience, use blockchain testnet token environment.