0.1 Architecture Design

0.1.1 Overall design

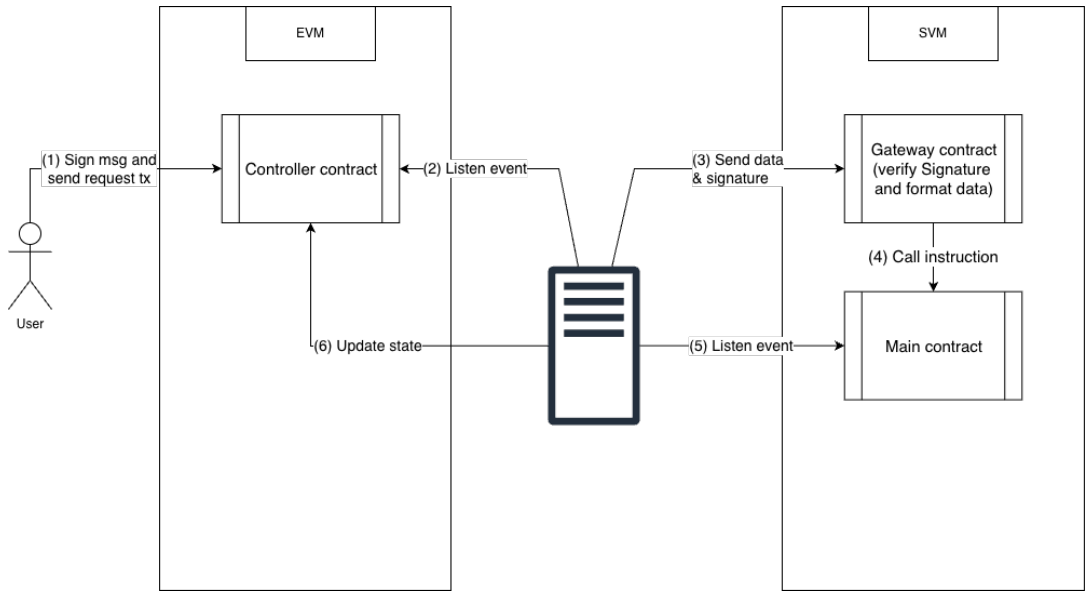
The architectural design of the Multi-chain Stablecoin Protocol is founded upon a Hybrid Event-Driven Microservices framework, structured within a Hub-and-Spoke topology. This sophisticated architecture is engineered to resolve the inherent challenges of liquidity fragmentation and state synchronization across heterogeneous blockchain networks. By strictly decoupling the Asset Custody Layer (residing on EVM chains) from the State Execution Layer (residing on the Solana SVM), the system achieves a high-performance, scalable solution that leverages the distinct advantages of each blockchain environment.

The architecture is composed of three primary execution environments, each functioning as an autonomous system component yet interconnected through a secure event-driven pipeline. The first environment is the EVM Spoke, which hosts the Controller Contract. This contract functions as the user's primary interface and asset vault, designed to be lightweight and gas-efficient. Its responsibilities are strictly limited to asset locking, event emission, and state updates based on authorized callbacks. The second environment is the Solana Hub (SVM), which acts as the system's "Brain." It hosts the Gateway Contract for cryptographic verification and data formatting, and the Main Contract for executing the core business logic, such as managing the Universal Wallet and calculating dynamic Health Factors. The third environment, bridging the deterministic worlds of these blockchains, is the Off-chain Guardian Infrastructure. This middleware operates as an active listener and orchestrator, ensuring that state transitions on one chain are accurately and securely reflected on the other.

To visualize the interaction between these components and the directional flow of data, Figure 0.1 presents the detailed system architecture diagram.

The operational workflow of the system, as depicted in the diagram, follows a rigorous six-step process designed to ensure atomicity, security, and eventual consistency.

Step 1: Initiation and Request Encapsulation The process begins on the EVM chain where the user intends to perform an action, such as depositing collateral. The user constructs a message M containing the action parameters (e.g., Token Address, Amount, Target Chain ID) and a unique nonce. Crucially, the user signs this message with their EVM private key, generating a signature σ. The user then submits a transaction to the **Controller Contract**. Upon receipt, the Controller



Hình 0.1: Detailed Architecture of the Multi-chain Stablecoin Protocol

does not immediately execute the cross-chain logic but performs two critical local actions: it locks the user's assets (in a Vault) and emits a 'RequestCreated' event containing the message M and signature σ. This emission acts as a signal flare to the off-chain infrastructure.

Step 2: Event Ingestion (The Listening Phase) Unlike traditional polling mechanisms that can be resource-intensive, the Guardian Server employs a reactive Event Listening model. It maintains an active WebSocket or HTTP connection to the EVM RPC nodes. When the 'RequestCreated' event is emitted by the Controller, the Guardian immediately captures the log data. This step represents the "Event" in the Event-Driven Architecture. The Guardian parses the log to extract the raw data necessary for the state transition on the destination chain.

Step 3: Relaying and Submission Once the data is captured, the Guardian packages the original message M and the signature σ into a new transaction payload compatible with the Solana runtime. It then submits this transaction to the **Gateway Contract** on the SVM. In this phase, the Guardian acts strictly as a courier (Relayer); it does not modify the message content, ensuring that the user's original intent remains tamper-proof. The Guardian also manages the payment of SOL gas fees, abstracting the complexity of holding multiple native tokens from the end-user.

Step 4: Verification and Instruction Execution The **Gateway Contract** serves as the entry point to the Solana environment. Its primary responsibility is secu-

rity. Before any business logic is executed, the Gateway invokes Solana's native 'Secp256k1' program to cryptographically verify that the signature σ corresponds to the user's EVM address contained in message M. This verification is performed on-chain, providing a trustless guarantee of identity. Upon successful verification, the Gateway formats the data into a structured Instruction and calls the **Main Contract**. The Main Contract then executes the core logic, such as creating a Universal Wallet PDA or updating the user's debt position.

Step 5: Outcome Observation Similar to the ingestion phase on the EVM side, the Guardian Server also maintains a listener on the Solana Blockchain. When the Main Contract finishes execution, it emits a specific event - either 'ExecutionSuccess' (indicating the state was updated) or 'ExecutionFailure' (indicating a logic error, such as low health factor). The Guardian listens for these specific outcome events to determine the final status of the cross-chain operation. This asynchronous confirmation step is vital for handling the probabilistic finality of blockchain networks.

Step 6: State Synchronization and Finalization Based on the event received from the Solana Main Contract, the Guardian initiates a final callback transaction to the EVM **Controller Contract** to close the loop. If the Solana execution was successful, the Guardian calls the 'updateState' function to finalize the process (e.g., minting stablecoins to the user). If the Solana execution failed, the Guardian triggers a rollback mechanism, unlocking the user's assets and resetting their nonce. This ensures that the system maintains data consistency between the Asset Layer and the State Layer, preventing any funds from being permanently locked in transit.

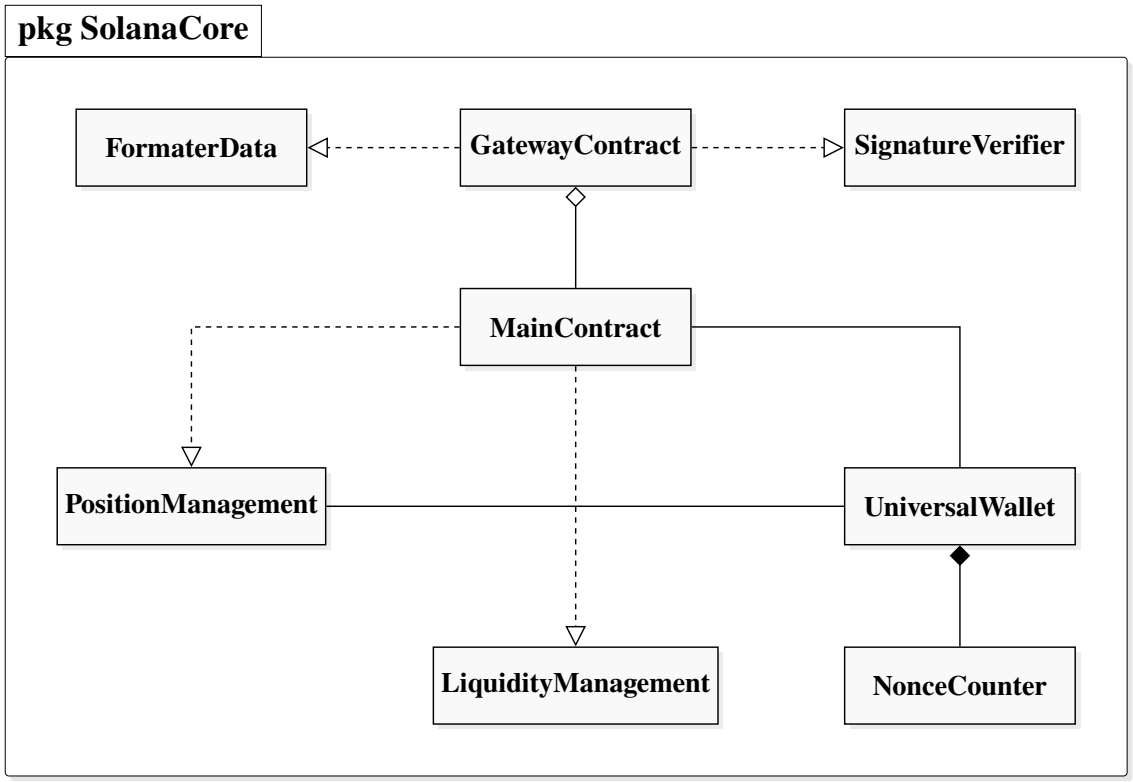
0.1.2 Detailed Package Design

Based on the overall architecture, this section details the internal design of the critical subsystems. The design is visualized using Class Diagrams grouped by their respective execution environments.

a, Solana Hub Package Design

The Solana Hub Package, designated as package SolanaCore, encapsulates the system's central business logic and state management. This package is architected to ensure strict separation between the interface layer (Gateway), the logic layer (Main Contract and Managers), and the data layer (Wallet and Counters). Figure 0.2 illustrates the internal structure and class relationships within this package.

Class Descriptions and Relationships:



Hình 0.2: Detailed Design of Solana Hub Package

- **GatewayContract:** This class acts as the single entry point for all cross-chain transactions initiated by the Guardian. It serves as an orchestrator that sanitizes inputs before passing control to the core logic.
  - Implementation Relationships: The Gateway implements logic from Formater Data class to deserialize incoming payloads and utilizes the Signature Verifier to perform cryptographic checks (Secp256k1 recovery) on the user's signature.
  - Aggregation: It maintains an aggregation relationship with the Main Contract, indicating that the Gateway coordinates the execution flow but delegates the actual financial state transitions to the Main Contract.
- **MainContract:** This is the core controller of the system. It manages the lifecycle of CDPs and coordinates liquidity across chains. To maintain modularity, it splits complex operations into specialized management modules:
  - It implements the Position Management module to handle collateral locking, debt minting, and health factor calculations.
  - It implements the Liquidity Management module to handle the rebalancing of assets between the Hub and Spokes during liquidations.
- **UniversalWallet:** This class represents the persistent state (Program Derived

Address - PDA) of a user. It stores the aggregated data of collateral and debt.

  - Association: Both the Main Contract and Position Management have direct associations with the Universal Wallet to read and modify the user's financial position.

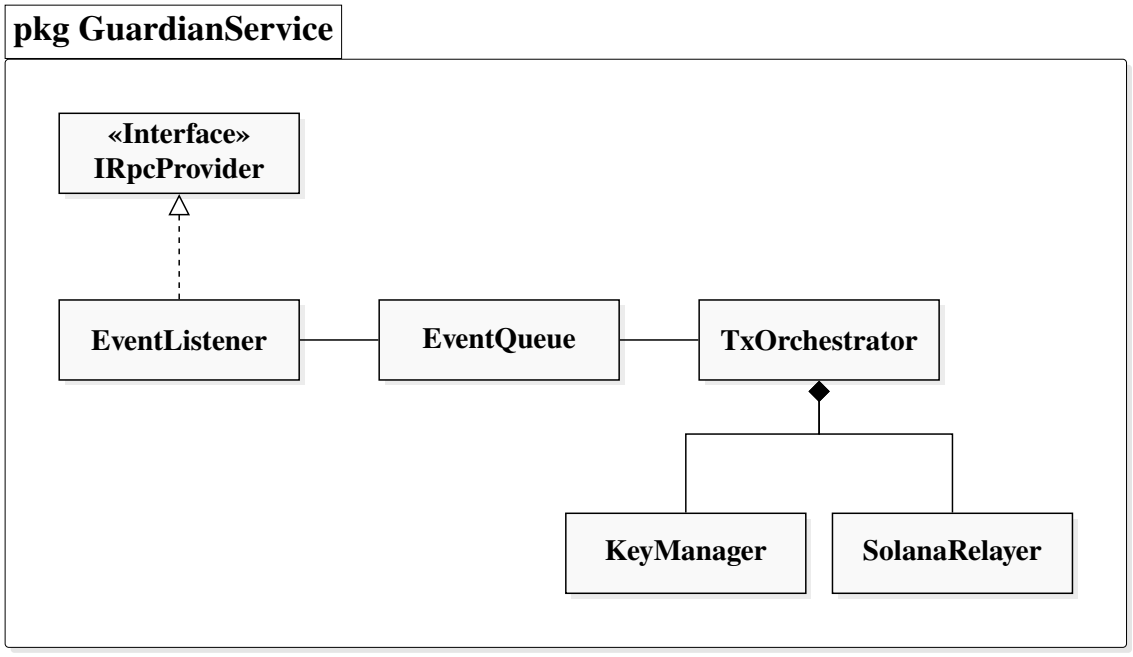
- **NonceCounter:** This class tracks the sequence number of transactions to prevent replay attacks.

  - Composition: The diagram defines a strict composition relationship (filled diamond) between Universal Wallet and Nonce Counter. This implies that the Nonce Counter is an intrinsic part of the Wallet; it cannot exist independently, and its lifecycle is bound to the existence of the Universal Wallet.

b, Guardian Middleware Package Design

The Guardian Middleware Package, designated as package GuardianService, acts as the off-chain orchestration layer. It is designed using an Event-Driven architecture to handle the asynchronous nature of cross-chain communication reliably.

Figure 0.3 depicts the internal class design of the Guardian node.



Hình 0.3: Detailed Design of Guardian Middleware Package

Class Descriptions and Relationships:

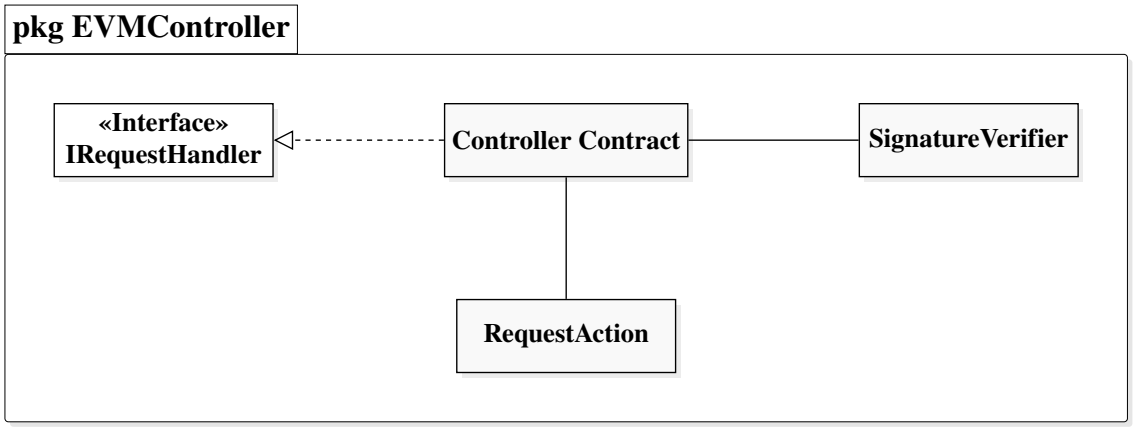
- **EventListener:** This component is responsible for monitoring blockchain networks. It implements the IRpcProvider interface to maintain agnostic connections to various EVM chains (Ethereum, BSC, Arbitrum). Its primary role is to detect RequestCreated logs and normalize them into a standard event format.

- **EventQueue:** Acting as a buffer, this class decouples the ingestion layer (Listener) from the processing layer (Orchestrator). It ensures that during high network traffic, events are not lost but queued for sequential processing.
- **TxOrchestrator:** This is the core logic unit of the middleware. It consumes events from the queue, validates the data integrity, and constructs the corresponding cross-chain transaction payloads.
- **KeyManager:** A security-critical class responsible for managing the Guardian's private keys.
  - Composition: The TxOrchestrator has a composition relationship with KeyManager, indicating that the orchestrator cannot function (cannot sign transactions) without the secure signing module.
- **SolanaRelayer:** This class handles the low-level networking required to broadcast signed transactions to the Solana cluster and confirm their finality.
  - Aggregation: The TxOrchestrator aggregates the SolanaRelayer, utilizing it as a service to dispatch the final outcome of its logic.

c, EVM Controller Package Design

The EVM Controller Package is designed to manage user interactions on the EVM chains, ensuring that requests are properly formatted, signed, and validated before being relayed to the Solana Hub. This package adheres to a layered design within the EVM environment, separating concerns related to request handling, data validation, and asset management.

Figure 0.4 illustrates the class structure within this package.



Hình 0.4: Detailed Design of EVM Controller Package

Design Explanation:

- **Controller Contract:** This is the core contract deployed on the EVM chain.

It manages user requests and orchestrates the cross-chain interaction flow.

  - Interface Implementation: The contract implements the IRequestHandler interface, defining the standard methods for processing external requests.
  - Usage Relationships: It has direct associations with RequestAction and SignatureVerifier classes. The Controller Contract utilizes RequestAction to structure and validate incoming request data and relies on SignatureVerifier to perform cryptographic checks on user signatures.

- **RequestAction:** A data structure class that encapsulates all the necessary parameters for a cross-chain request (e.g., nonce, action type, amount, destination chain ID). It is used by the Controller Contract to hold and process incoming requests.
- **SignatureVerifier:** This utility class is responsible for validating the authenticity of a user's signature. It takes the message M and the signature σ as input and verifies if they were generated by the owner of the intended EVM address. This is a critical security component that prevents unauthorized actions.

0.2 Detailed Design

This section presents the comprehensive design specification for the Multi-chain Stablecoin Protocol. It decomposes the system into three core layers: Smart Contract Design (On-chain), Server Application Design (Off-chain), and Database Design (Persistence). The level of detail provided herein serves as the blueprint for the implementation phase.

0.2.1 Smart Contract Design

The smart contract architecture serves as the immutable backbone of the Multi-chain Stablecoin Protocol. It is partitioned into two distinct environments: the EVM Controller (handling asset custody and user requests) and the Solana Hub (handling global state and core financial logic). The design focuses on data integrity, cryptographic security, and efficient state synchronization.

a, EVM Controller Design

The Controller contract, deployed on EVM-compatible networks, functions as the primary interface for user interaction and asset custody within the multi-chain ecosystem. Its design philosophy prioritizes security and simplicity, acting as a decentralized vault that locks collateral assets while delegating complex financial calculations to the central hub. The contract is architected to manage the lifecycle of user requests through a state machine model, ensuring that every cross-chain operation is atomic and reversible.

At the data level, the contract maintains a robust set of structures to track user positions and system integrity. Central to this design is the Request structure, which acts as a comprehensive data carrier. This structure encapsulates all necessary metadata for a transaction, including a unique request identifier, the destination chain identifier, and the specific action type such as deposit or withdrawal. Furthermore, it stores the cryptographic signature generated by the user, which serves as the immutable proof of intent required by the destination chain. To prevent replay attacks and ensure the strict ordering of operations, the contract implements a nonce management system, associating a monotonically increasing counter with each user address.

This is the data structures to manage the lifecycle of user requests:

- Request Struct: This is the fundamental data unit representing a user's intent. It encapsulates all necessary information for cross-chain processing:
  - requestId: A unique identifier for tracking the request across the system.
  - chainId: The ID of the destination chain where the action should be executed.
  - user: The EVM address of the user initiating the transaction.
  - actionType: An enumeration defining the operation (Deposit, Withdraw, Mint, Burn).
  - token: The address of the collateral token involved in the transaction.
  - amount: The quantity of tokens to be processed.
  - nonce: A sequential counter ensuring strict ordering and preventing replay attacks.
  - deadline: A timestamp after which the request is considered invalid, protecting against delayed execution.
  - signature: The cryptographic proof consisting of the components (r, s, v), generated by the user's private key.
- ActionType Enum: A predefined set of constants representing valid operations: Deposit, Withdraw, Mint, and Burn. This strict typing prevents invalid operations from being submitted.
- Link Wallet Request: A specialized structure used when a user wishes to link their EVM address to a Universal Wallet on a different chain. It stores the binding request until it is verified by the Solana Hub.

A critical aspect of the controller design is the concurrency control mechanism, implemented through a pessimistic locking strategy. When a user initiates a request, the contract enforces a mutex lock on their specific account state. This prevents the user from submitting multiple simultaneous requests, which could lead to race conditions or state desynchronization between the source and destination chains. The lock remains active until the off-chain guardian infrastructure explicitly confirms the finality of the operation on the remote chain. This design choice ensures linearizability, guaranteeing that the system state transitions occur in a predictable and secure sequence.

The operational logic of the contract is exposed through a restricted set of external functions. The primary entry point allows users to submit signed requests, triggering the asset transfer to the vault and the emission of an event log for off-chain listeners. Complementing this is the callback interface, accessible exclusively by the authenticated guardian address. This interface facilitates the finalization of the cross-chain lifecycle. Upon receiving a success signal from the guardian, the contract executes the final state transition, such as minting stablecoins to the user. Conversely, in the event of a remote failure, the contract provides a revert mechanism that releases the mutex lock and refunds the locked assets, ensuring that user funds are never permanently frozen due to network issues.

The contract exposes specific functions for User-System interaction and Guardian-System synchronization:

- Request (User-Facing): This function is the entry point for users. When called, it validates the input parameters, transfers the user's assets to the contract vault (in case of Deposit), and emits an event. Crucially, it locks the user's nonce to prevent concurrent requests, ensuring linearizability.
- Complete Request (Guardian-Only): This restricted function is invoked by the Guardian server upon successful execution on the Solana Hub. It accepts the final status and, if successful, finalizes the local state (e.g., minting stablecoins to the user's wallet).
- Revert Request (Guardian-Only): Serving as a fail-safe mechanism, this function is called if the cross-chain operation fails on Solana (e.g., due to insufficient health factor). It unlocks the user's assets and resets the nonce, returning the system to its pre-request state without loss of funds.

b, Solana Gateway Design

The Gateway contract on the Solana network serves as the secure ingress point for all cross-chain traffic, acting as a cryptographic firewall between the external environment and the internal financial logic. Its primary design objective is to sanitize and authenticate incoming data payloads before they can influence the system's state. Unlike typical smart contracts that focus on business rules, the Gateway is specialized for high-performance data verification, leveraging the parallel processing capabilities of the Solana runtime.

Data ingestion within the Gateway is handled through a specialized storage structure designed to accommodate the heterogeneous data formats of EVM chains. Since Ethereum uses 256-bit integers while Solana native programs are optimized for 64-bit or 128-bit operations, the Gateway implements a data normalization layer. When the off-chain guardian submits a request, the contract parses the raw byte stream, validating the integrity of the data layout and converting the parameters into a format compatible with the system's internal logic. This normalization process ensures that subsequent module calls operate on clean, type-safe data structures, preventing serialization errors deep within the call stack. There is one primary data structure within the Gateway:

- Request Data Storage: The contract defines a specific data layout to store the re-formatted request received from the EVM chain. Unlike the EVM struct, this data is optimized for the Solana BPF runtime (e.g., converting 256-bit integers to 64/128-bit where applicable) to minimize storage costs.

The most critical function of the Gateway is the execution of cryptographic proofs. The contract exposes a specific instruction set that allows the authorized guardian to submit the user's original signature along with the message payload. Instead of implementing complex elliptic curve mathematics in user-space code, which would be prohibitively expensive in terms of compute units, the Gateway utilizes Solana's native Secp256k1 program. By performing a cross-program invocation to this precompiled utility, the contract can efficiently verify that the signature provided was indeed generated by the claimed EVM address. Only upon successful verification does the Gateway permit the control flow to proceed to the Main Contract, thereby establishing a trustless link between the user's identity on Ethereum and their actions on Solana. The Gateway exposes the following key method for secure data handling:

- Set Request (Guardian-Only): This instruction allows the authorized Guardian to submit the raw data and signature from the EVM chain. The method per-

forms the heavy lifting of Secp256k1 signature verification to prove that the data originated from the claimed EVM user. Once verified, it formats the data into an internal instruction and invokes the Main Contract.

c, Solana Main Contract Design

The Main Contract functions as the central nervous system of the protocol, encapsulating the global financial state and executing the core logic of the Collateralized Debt Position mechanism. This component is architected to be the single source of truth for the entire multi-chain ecosystem, aggregating disparate asset data into a unified solvency model. The design prioritizes modularity and state isolation, ensuring that the complex interactions between collateral management, debt issuance, and liquidation are handled with precision and security.

At the heart of the state management strategy is the Universal Wallet, a sophisticated data structure stored as a Program Derived Address. This structure aggregates the user's entire portfolio, tracking the collateral balances deposited across various connected chains and the total debt issued by the protocol. By maintaining this global view, the contract can calculate a unified health factor for each user in real-time, allowing for capital efficiency that isolated lending protocols cannot match. Complementing individual user states is the Depository, a global singleton account that tracks system-wide parameters such as the total value locked, the aggregate debt ceiling, and the risk configurations for each supported asset class. The Main Contract defines several critical data structures to manage user positions and system integrity:

- Universal Wallet: The central PDA (Program Derived Address) that aggregates a user's entire portfolio. It stores the total collateral balance and total debt across all connected chains, serving as the single source of truth for solvency calculations.
- Depository: A global state tracking the system-wide parameters, such as total protocol debt, total value locked (TVL), and risk parameters (LTV ratios) for each supported asset.
- Loan: A data structure representing the specific debt position of a user. It tracks the principal amount borrowed and the accrued interest over time.
- Link Wallet Request: Stores pending requests for wallet linkage. When a user wants to control their Solana Universal Wallet from a new EVM address, this attribute temporarily holds the request until proof of ownership is established.

The operational logic of the contract is exposed through a set of polymorphic

instructions capable of handling requests from diverse origins. Fundamental operations such as deposit, mint, and withdraw are designed to be agnostic to the caller's source. Whether the instruction originates from a native Solana user or is relayed by the Guardian on behalf of an EVM user, the underlying business logic remains consistent. This unification simplifies the codebase and reduces the attack surface. Additionally, the contract implements a specialized liquidation engine that runs natively on Solana. When a user's position becomes insolvent due to market volatility, this engine allows liquidators to repay debt and seize collateral directly on the hub chain, ensuring that the protocol remains solvent without relying on asynchronous cross-chain calls for critical risk management. The Main Contract exposes the following key methods for user interaction and system maintenance:

- Deposit, Mint, Burn, Withdraw: These are the fundamental CDP operations. They are designed to be polymorphic, capable of being invoked either by the Guardian (relaying an action from an EVM user) or directly by a Solana Native User. This unified interface ensures that the business logic remains consistent regardless of the user's origin chain.
- Interact Universal Wallet (Guardian-Only): A specialized administrative method allowing the Guardian to update the mapping within a Universal Wallet, such as adding a new linked chain address after successful verification.
- Liquidate: This critical function maintains system solvency. It is executed natively on the Solana chain. When a user's Health Factor drops below the threshold, a liquidator can call this method. The liquidator repays the user's debt (in stablecoins) directly on Solana and, in return, receives a portion of the user's collateral stored in the Solana vault (or claims rights to cross-chain collateral via the Depository).

0.2.2 Server Application Design

The Off-chain Infrastructure, architected as the Guardian Middleware, functions as the central nervous system of the protocol, facilitating the secure and reliable transmission of state between the EVM asset layer and the Solana execution layer. Implemented within a Node.js runtime environment, the server is designed not merely as a passive relay, but as an active orchestration engine capable of handling network instability, data verification, and state synchronization. The design of this application is comprehensive, addressing five critical functional domains: network connectivity, data persistence, business logic orchestration, cryptographic security, and transaction finality management.

The first critical function of the server is the robust ingestion of blockchain

events through the Network Connectivity and Event Monitoring module. The application initializes and maintains persistent connections to multiple JSON-RPC providers for each supported chain. This multi-provider strategy is essential to mitigate the risk of single-point failures where a specific node provider might experience downtime or latency spikes. The event listener operates on a polling mechanism that queries the EVM Controller contract for specific logs, such as the request creation events. To ensure data integrity, the listener implements a block confirmation delay strategy, waiting for a configurable number of block confirmations before ingesting an event. This precaution is necessary to protect the system from interacting with reorganized blocks or chain forks, ensuring that the Guardian only acts upon immutable ledger states.

Once an event is securely ingested, the system relies on the Data Persistence and Reliability module to manage the asynchronous processing flow. Recognizing that cross-chain operations involve probabilistic latency, the server avoids in-memory processing which is volatile and prone to data loss during system restarts. Instead, all incoming requests are serialized and pushed into a persistent First-In-First-Out queue backed by the local file system or a dedicated database. This queuing mechanism serves as a buffer that decouples the high-throughput ingestion layer from the transaction submission layer. It allows the system to absorb traffic spikes without overwhelming the Solana RPC endpoints. Furthermore, this module integrates with a Redis key-value store to implement idempotency checks. By caching the unique nonce of every processed request, the system ensures that a specific user action is never executed twice, even if the source chain emits duplicate events due to network retries.

Following the queuing phase, the Business Logic Orchestration module takes responsibility for interpreting and transforming the raw data. Since the EVM and Solana environments utilize different data standards—for instance, the handling of large integers and address formats—this module performs the necessary normalization. It parses the binary payload from the EVM logs, extracting critical parameters such as the user identity, token address, and action type. The logic then maps these parameters to the corresponding Solana-specific data structures defined in the Anchor program IDL. This phase also involves the validation of business rules off-chain, such as checking if the user's request exceeds the system's global debt ceiling or if the requested token is currently supported by the protocol. This precomputation layer reduces the burden on the on-chain smart contracts, ensuring that only structurally valid transactions are attempted.

Integral to the safety of the protocol is the Cryptographic Security and Verifi-

cation module. Before the Guardian constructs a transaction to update the state on the Solana Hub, it must cryptographically prove that the request originated from the rightful owner. The server implements a signature verification utility that reconstructs the message hash from the request parameters and performs an elliptic curve recovery on the user's provided signature. This off-chain verification acts as a filter to discard malicious or forged requests immediately, saving the Guardian from paying gas fees for invalid transactions. Additionally, this module manages the Guardian's own sensitive private keys through a secure Key Management System. The signing process is isolated from the logic layer, ensuring that the private keys are never exposed in logs or memory dumps, strictly adhering to the principle of least privilege.

The final functional component is the Transaction Dispatch and State Synchronization module, which closes the feedback loop of the cross-chain interaction. After constructing and signing the Solana transaction, this module broadcasts it to the cluster and enters an observation state. Unlike standard "fire-and-forget" mechanisms, this system implements a sophisticated polling logic to track the transaction's lifecycle until it reaches a finalized status. Upon confirmation of the execution result on Solana, the module constructs a corresponding callback transaction directed back to the EVM Controller. This callback carries the success or failure status, triggering the asset layer to either mint the stablecoins or revert the locked collateral. This bidirectional synchronization ensures that the distributed system maintains eventual consistency, preventing any scenario where user funds remain indefinitely locked in transit due to partial failures.

0.2.3 Database Design

While the blockchain ledger serves as the immutable system of record, the architecture incorporates an off-chain database layer within the Guardian infrastructure. This database functions as a high-performance indexer and state cache, facilitating rapid data retrieval for the frontend interface and supporting complex queries required for the liquidation engine. The schema is designed to mirror the on-chain state, organized into three primary domains: Identity Management, User Position Tracking, and Protocol Liquidity Control.

a, Universal Wallet Collection

The fundamental entity within the database is the Universal Wallet collection, which manages the cross-chain identity mapping. The primary objective of this entity is to prevent identity collisions and enforce the one-to-one relationship between a user's primary EVM address and their Solana Program Derived Address.

The schema for this entity stores the unique Universal Wallet address derived from the Solana blockchain, acting as the primary key. Associated with this key is the Owner field, recording the initial EVM address that created the position. To support multi-chain interoperability, the entity includes a Wallets array or relation, listing all secondary addresses from different chains that have been cryptographically linked to this profile. This structure allows the Guardian to instantly verify if an incoming request originates from an authorized address associated with an existing universal profile, thereby preventing duplicate wallet creation attempts. The Universal Wallet collection includes the following key attributes:

- **Universal wallet address (Primary Key):** The unique Solana address (PDA) generated by the program. This serves as the global identifier for the user's account across the entire system.
- **Owner address:** The initial EVM address that created the wallet. This field is used to authenticate the root ownership rights.
- **Linked wallets:** An array or relational table storing all secondary EVM addresses linked to this Universal Wallet. This allows the system to recognize a user interacting from different chains (e.g., Polygon, BSC) as the same entity.
- **Creation tx hash:** The transaction hash on the source chain that triggered the wallet creation. This serves as an immutable audit trail for the account's origin.
- **Status:** A status flag (e.g., *Active, Pending Sign, Blacklisted*) used to manage the operational state of the wallet.

b, User Collection (Loans and Requests)

The second critical domain is the User Position and Transaction History, which tracks the financial health and interaction logs of individual users. This domain is essential for both the user dashboard and the automated liquidation bots. The User entity aggregates the financial data, storing the Loan details such as total minted debt and the current composite collateral value. Crucially, it maintains a computed Health Factor field, updated in real-time based on oracle price feeds, which allows the system to query for under-collateralized positions efficiently without scanning the entire blockchain. Linked to the user profile is the Transaction Requests collection. This entity logs every lifecycle event, capturing attributes such as the request ID, action type, token amount, and the associated transaction hashes for both the source and destination chains. This historical log serves as an audit trail, enabling the system to track the status of asynchronous cross-chain operations and detect

any stuck or failed requests that require manual intervention. The User Collection encompasses two main sub-entities:

- **Loan Position State:**
  - **Total debt:** The aggregate amount of stablecoins minted by the user, denominated in the protocol's base unit.
  - **Total collateral value:** The real-time USD value of all assets locked by the user across all chains. This is frequently updated by the price oracle workers.
  - **Health factor:** A computed index derived from the (**??**) formula. This field is indexed to allow liquidator bots to query where Health factor < 1.0 efficiently.
  - **Liquidation threshold:** The minimum safe ratio required for the specific basket of assets held by the user.
- **Request Action History:**
  - **Request id:** A unique composite key (typically chainId + nonce) identifying a specific user action.
  - **Action type:** Specifies the intent of the transaction (e.g., *Deposit, Withdraw, Mint, Repay*).
  - **Source chain id** & **Dest chain id**: Tracks the origin and destination networks of the request.
  - **Token address** & **Amount**: Details the asset and quantity involved in the transaction.
  - **EVM tx hash:** The transaction hash on the EVM side (Locking/Burning assets).
  - **Solana tx hash:** The corresponding transaction hash on the Solana side (State Update). This links the two asynchronous events together.
  - **Process status:** Indicates the current lifecycle stage: *Pending, Processing, Completed, Failed, Reverted.*

c, Controller & Liquidity Collection

Finally, the Protocol Controller domain manages the aggregate liquidity and security parameters of the system. This entity tracks the global state of the protocol across all connected spokes. It records the Total Value Locked (TVL) per chain and the total circulating supply of the stablecoin. The schema includes specific fields

for liquidity management, monitoring the available capacity of each EVM vault to ensure that withdrawal requests can be honored. Furthermore, this domain stores system-wide configuration parameters, such as the current Loan-To-Value ratios for supported collateral assets and the operational status of each bridge connection. By centralizing this data off-chain, the Guardian can perform complex analytics to detect liquidity imbalances or potential security threats, allowing for proactive rebalancing or emergency pausing of specific controller contracts if anomalies are detected. The Controller & Liquidity Collection includes the following critical attributes:

- **Chain id:** The identifier for the specific blockchain network (e.g., 1 for Ethereum).
- **Total value locked (TVL):** The aggregate amount of collateral assets currently held in the protocol on each specific chain.
- **Circulating supply:** The total amount of stablecoins minted and currently active in the market from each chain.
- **Available liquidity:** Monitors the free capital available in the vault. This is crucial for approving withdrawal requests; if a vault is empty, the Guardian must pause withdrawals or trigger a rebalance.

0.3 Application Client and Illustration of main functions

This section demonstrates the implementation of the client-side application, focusing on the user experience flows for identity management and cross-chain financial operations.

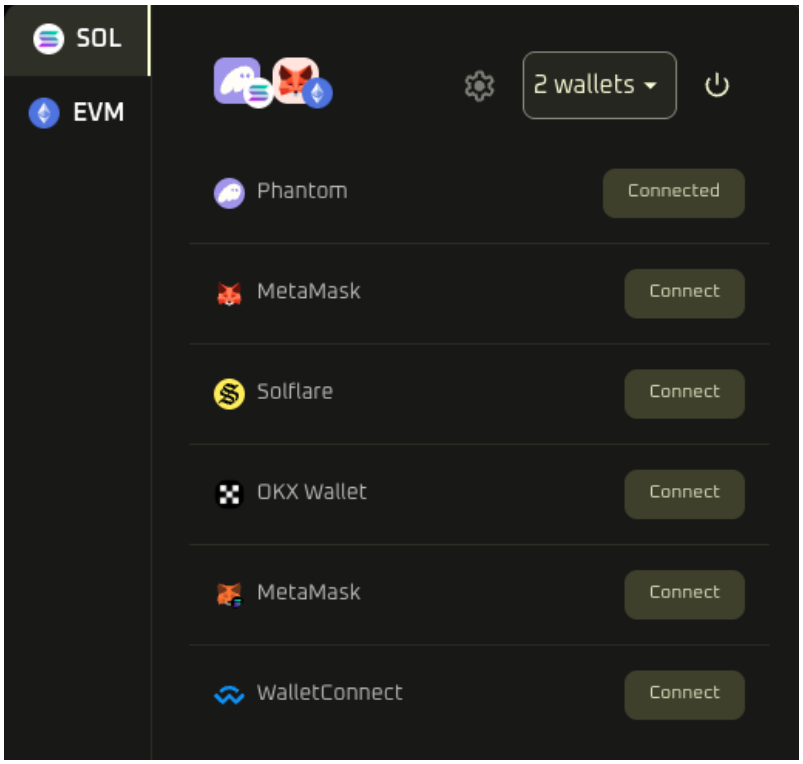
0.3.1 User Interface for Universal Wallet

The Universal Wallet interface serves as the foundational layer of the application, enabling users to aggregate their fragmented blockchain identities into a single manageable profile. This interface handles the complexities of multi-chain authentication and state synchronization.

a, Wallet Connectivity

The initial interaction with the protocol begins with the wallet connection module. To support the hybrid architecture, the interface is designed to accommodate distinct blockchain standards simultaneously. The connectivity panel categorizes providers into specific network groups, allowing users to select between Solana-native wallets like Phantom or EVM-compatible wallets such as MetaMask. A distinguishing feature of this implementation is the support for concurrent multi-wallet connections. Users can maintain active sessions with multiple providers at the same time, which is visualized by the status indicators next to each provider.

This capability is essential for the protocol, as it allows the application to read balances and request signatures from different chains without forcing the user to constantly switch the active network in their browser extension.

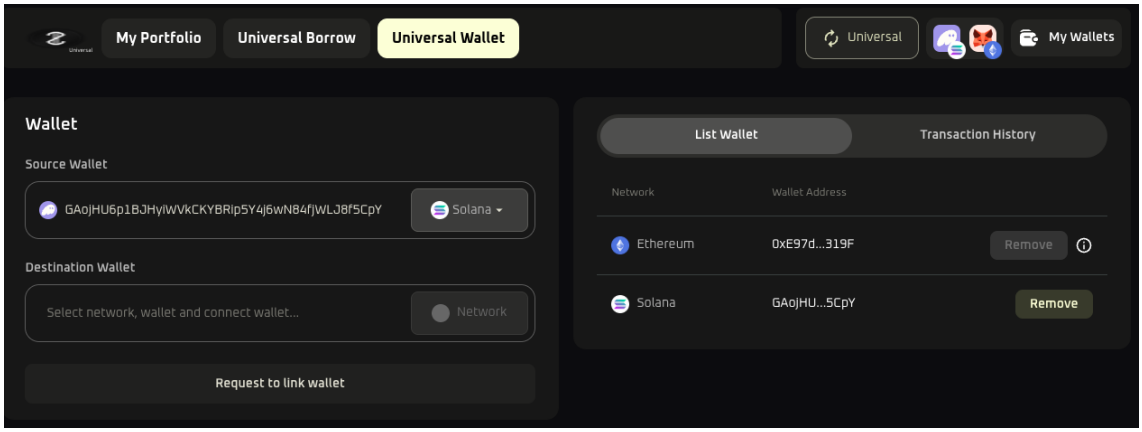


Hình 0.5: Multi-chain Wallet Connectivity Interface

b, Link Wallet Workflow

Once the wallets are connected, the user interacts with the linking interface to establish the Universal Wallet. The design presents a dual-field form requiring the selection of a source wallet and a destination wallet. The application enforces a rigorous proof-of-ownership protocol during this process. The workflow initiates when the user selects the request option, triggering a transaction on the source chain. This on-chain action serves as the intent declaration. Upon successful confirmation of the source transaction, the interface automatically prompts the user to sign a cryptographically secure message using the destination wallet. This two-step verification ensures that the link is established only when the user possesses the private keys for both addresses, effectively binding the identities across the two networks.

There is one primary screen for the Universal Wallet, as shown in Figure 0.6. The interface is divided into two main panels: the wallet connectivity panel on the left and the wallet management panel on the right.



Hình 0.6: Universal Wallet Management Interface (Link and List View)

c, Remove Wallet Logic

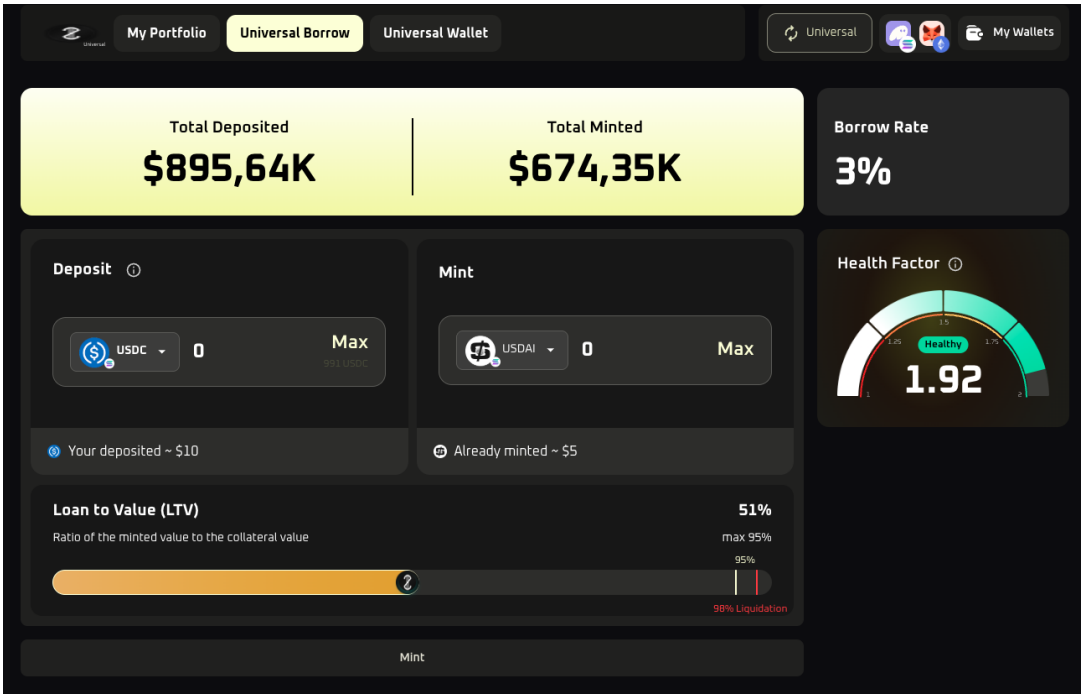
The management of associated addresses is handled through the wallet list panel, where users can view and disassociate their linked accounts. The interface provides a removal function for each entry; however, the execution of this action is governed by strict logic to preserve system integrity. The protocol distinguishes between the primary wallet, which acts as the root identifier, and secondary wallets. The interface restricts the removal of the first wallet, enforcing a rule that it can only be dissociated after all secondary wallets have been removed. Additionally, the system performs a solvency check before allowing any removal operation. If the user has outstanding debt obligations, the interface prevents the removal of any wallet that contains collateral necessary to maintain a safe health factor, thereby securing the protocol against under-collateralization risks.

0.3.2 User Interface for Cross-chain Borrow

The borrowing interface acts as the central command center for the user's financial activities. It is designed to provide immediate visual feedback on the user's solvency while offering granular control over their asset allocation across different chains. Figure 0.7 illustrates the consolidated dashboard view.

a, Overview Panel: Total Deposit and Minted

At the top of the interface, the system aggregates the user's global financial position into two primary metrics: Total Deposited and Total Minted. Unlike single-chain applications that only show local balances, these values represent the summation of assets across all connected networks (e.g., Ethereum, Solana, Arbitrum), normalized to a USD base currency. This high-level summary provides users with an instant snapshot of their portfolio's scale and their outstanding liabilities, which is crucial for making informed borrowing decisions. To the right, the current Bor-



Hình 0.7: Cross-chain Borrowing and Position Management Interface

row Rate is prominently displayed, ensuring transparency regarding the cost of debt.

b, Action Panel: Deposit and Mint Operations

The core interaction occurs within the dual-pane action modules for Deposit and Mint.

- **Deposit Module:** This component allows users to lock collateral. It features a chain-agnostic token selector, enabling the user to deposit assets like USDC or ETH from any supported network without leaving the dashboard. The interface automatically fetches the user's available balance and provides a "Max" button for convenience.
- **Mint Module:** Correspondingly, the minting module allows the user to generate the protocol's stablecoin (USDAI) on any supported blockchain. It is tightly coupled with the deposit module; as the user inputs a deposit amount, the interface dynamically updates the borrowing capacity in the mint section, providing real-time feedback on how much debt can be safely issued against the new collateral.

c, Risk Management Controls: LTV and Health Factor

To mitigate the risk of liquidation, the interface incorporates advanced visualization tools for risk management.

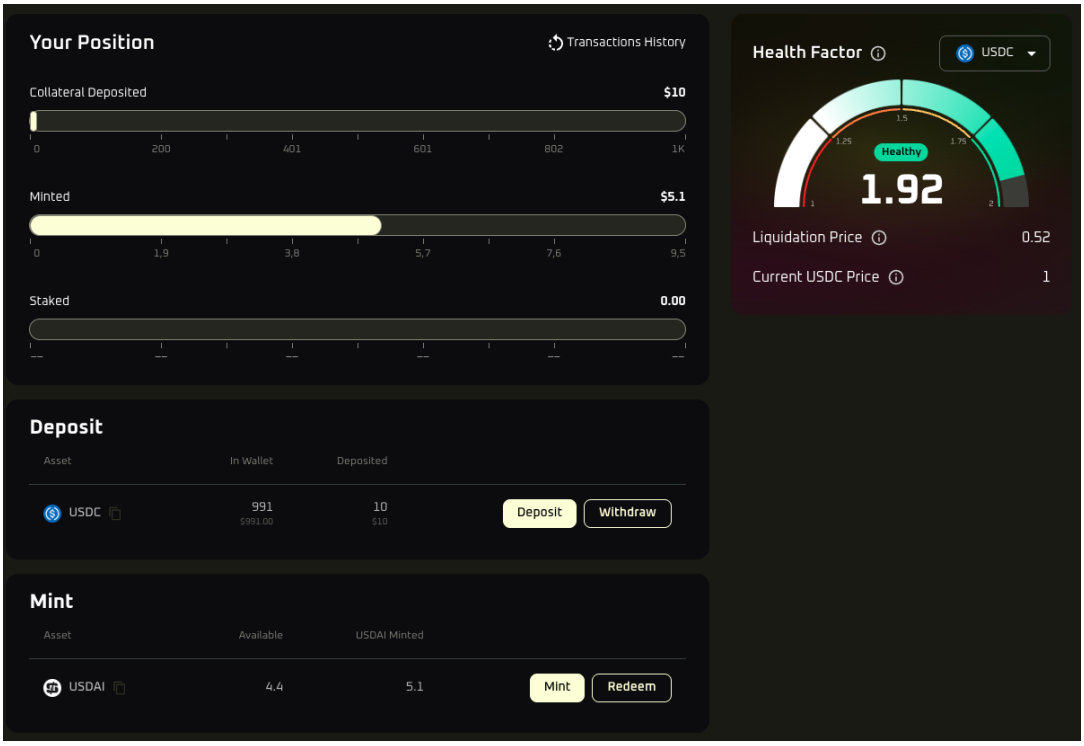
Loan-To-Value (LTV) Slider: A distinctive feature of the design is the inter-

active LTV slider bar. This control visualizes the ratio between the minted value and the collateral value. As users adjust the amount of debt they wish to mint, the progress bar fills up towards the liquidation threshold (marked in red at 98%). This visual cue helps users intuitively understand their position's risk level relative to the maximum allowable limit (95%), preventing accidental over-leveraging.

Health Factor Gauge: The Health Factor is the most critical metric for position safety, and it is visualized using a semi-circular gauge. The interface categorizes the health status into color-coded zones: red for danger (close to 1.0), yellow for caution, and green for healthy (e.g., > 1.5). In the example shown, a Health Factor of 1.92 indicates a robust position. This gauge updates instantaneously as the user modifies the deposit or mint amounts in the action panel, allowing them to simulate the impact of their transaction on their solvency before committing to the blockchain.

0.3.3 User Interface for Portfolio

The Portfolio interface is engineered to provide a comprehensive, real-time overview of the user's active financial engagements within the protocol. Unlike the borrowing interface which focuses on initiating new actions, the portfolio view is dedicated to the ongoing management and monitoring of established debt positions. Figure 0.8 displays the detailed layout of this section.



Hình 0.8: User Portfolio and Position Management Interface

a, Position Visualization

The "Your Position" panel offers a granular breakdown of the user's capital allocation. It utilizes horizontal progress bars to visually represent the magnitude of assets relative to the user's total capacity.

- **Collateral Deposited:** This bar displays the total value of assets locked in the protocol (e.g., $110). It provides a quick visual reference for the user's capital commitment.
- **Minted Debt:** This bar indicates the outstanding stablecoin debt (e.g., $5.1). By placing these bars in proximity, the interface allows users to easily compare their debt load against their collateral base.
- **Staked Assets:** A dedicated section tracks yield-bearing activities, ensuring that all forms of capital deployment are centralized in one view.

b, Solvency Indicators and Liquidation Risk

To the right of the position summary, the Health Factor gauge reappears as a persistent safety monitor. In the portfolio context, this gauge is augmented with critical market data essential for risk assessment.

- **Liquidation Price:** This metric calculates the specific price point of the collateral asset (e.g., USDC) at which the position would become insolvent. In the example, a liquidation price of 0.04 against a current price of 1 indicates a highly safe position.
- **Current Price Feed:** Real-time price updates from the Oracle are displayed, allowing users to benchmark the market movement against their liquidation threshold instantly.

c, Operational Controls: Deposit, Withdraw, Mint, Redeem

The lower section of the portfolio interface integrates the functional controls directly with the asset list. This design pattern minimizes navigation friction, allowing users to react swiftly to market changes.

- **Deposit & Withdraw:** Located within the asset row (e.g., USDC), these buttons allow users to add more collateral to strengthen their health factor or withdraw excess capital when the market is favorable. The interface displays both the wallet balance and the currently deposited amount to facilitate decision-making.
- **Mint & Redeem:** Similarly, the stablecoin section (USDAI) provides controls to increase leverage (Mint) or repay debt (Redeem/Burn). The "Available"

field informs the user of their remaining borrowing power, while "Minted" tracks the current obligation.

0.4 Application Building

This section details the technological ecosystem utilized to construct the Multi-chain Stablecoin Protocol. The development stack is categorized into development environments, smart contract frameworks, frontend interfaces, backend infrastructure, and testing suites.

0.4.1 Libraries and Tools

a, IDE & Extensions

The development environment is centered around Visual Studio Code, augmented with specific extensions to support syntax highlighting, linting, and formatting for the diverse languages used (Rust, Solidity, TypeScript, Python).

Tool / Extension	**Purpose**	**URL**
Visual Studio Code	Primary Integrated Development Environment.	`https://code.visualstudio.com/`
rust-analyzer	Language support for Rust (code completion, goto definition).	`https://github.com/rust-lang/rust-analyzer`
Solidity (Juan Blanco)	Syntax highlighting and snippets for Ethereum smart contracts.	`https://github.com/juanfranblanco/vscode-solidity`

Bảng 1: Development Environment Tools

b, Smart Contract Tools & Libraries

This category encompasses the core frameworks required to develop, compile, and deploy logic on both the Solana Virtual Machine (SVM) and the Ethereum Virtual Machine (EVM).

Library / Framework	Purpose	URL
Rust (v1.79.0)	Systems programming language for Solana programs.	https://www.rust-lang.org/
Anchor Framework (v0.30.0)	Sealevel runtime framework for Solana, handling serialization/IDL.	https://www.anchor-lang.com/
Solidity (v0.8.28)	Object-oriented language for EVM Controller contracts.	https://soliditylang.org/
OpenZeppelin Contracts	Standard secure contract components (ERC20, Ownable).	https://www.openzeppelin.com/
Hardhat (v2.25.0)	Ethereum development environment for compiling and deployment.	https://hardhat.org/

Bảng 2: Smart Contract Development Stack

c, Frontend Client

The user interface is engineered using a high-performance React stack powered by Vite. The integration of blockchain interactions relies on modern libraries like Viem/Wagmi for EVM and the Solana Web3 suite for SVM.

Library (Version)	Purpose / Usage	URL
React (v18.3.1)	Component-based UI library for building the application interface.	https://react.dev/
TypeScript (∼5.6.2)	Strictly typed superset of JavaScript ensuring type safety across the codebase.	https://www.typescriptlang.org/
Vite (v6.0.5)	Next-generation frontend build tool providing fast HMR and optimized bundling.	https://vitejs.dev/
Wagmi (v2.16.9)	React Hooks library for Ethereum, simplifying wallet connection and state management.	https://wagmi.sh/
Viem (v2.x)	Low-level TypeScript interface for Ethereum, replacing Ethers.js for better performance.	https://viem.sh/
@solana/web3.js (v1.98.0)	Core library for interacting with the Solana blockchain (RPC, Transactions).	https://solana.com/docs
@coral-xyz/anchor (v0.30.1)	Client library to interact with Anchor-based Solana programs (IDL, Accounts).	https://www.anchor-lang.com/
@solana/spl-token (v0.4.12)	Utility library for managing SPL tokens (minting, transferring) on Solana.	https://spl.solana.com/token
Jotai (v2.11.0)	Primitive and flexible state management library for React (Global State).	https://jotai.org/

Bång 3: Frontend Development Libraries

d, Backend (Guardian Infrastructure)

The Guardian infrastructure operates on a Python 3.9 runtime, utilizing asynchronous frameworks to handle high-throughput event processing. The core libraries facilitate interaction with both Ethereum and Solana networks alongside persistent storage.

Library (Version)	Purpose / Usage	URL
Python (v3.9)	Runtime environment for the backend logic and scripting.	https://www.python.org/
Sanic ($\geq$ v22.12.0)	High-performance asynchronous web server and framework for building fast APIs.	https://sanic.dev/
Web3.py (v6.15.1)	Comprehensive Python library for interacting with Ethereum nodes and contracts.	https://web3py.readthedocs.io/
Eth-account (v0.11.3)	Library for signing transactions and managing Ethereum accounts securely.	https://eth-account.readthedocs.io/
Solana.py (v0.36.6)	Python client for interacting with the Solana blockchain JSON RPC API.	https://michaelhly.github.io/solana-py/
Solders (v0.26.0)	High-performance Python binding for Solana primitives (Keypairs, Pubkeys).	https://github.com/kevinheavey/solders
AnchorPy (v0.21.0)	Python client for Anchor-based Solana programs, enabling IDL interaction.	https://kevinheavey.github.io/anchorpy/
PyMongo (v4.10.1)	Synchronous driver for MongoDB, used for persisting user request logs.	https://pymongo.readthedocs.io/

Bảng 4: Backend Development Libraries

e, Testing Tools

Quality assurance is maintained through a combination of unit testing frameworks and local blockchain simulators.

Tool	Purpose	URL
Mocha / Chai	JavaScript test framework and assertion library.	https://mochajs.org/
Hardhat	Local Ethereum node for forking mainnet state during tests.	https://hardhat.org/

Bảng 5: Testing and Simulation Tools

0.4.2 Achievement

The development phase culminated in a fully operational Cross-chain Stablecoin Protocol, demonstrating the feasibility of the Hub-and-Spoke architecture for decentralized finance. The project successfully delivered three distinct yet integrated components, each fulfilling a critical role in the system:

1. **The Multi-chain DApp Client:** A unified interface that abstracts the complexity of cross-chain interactions. It allows users to manage their Universal Wallets and execute financial operations seamlessly across heterogeneous networks without manually bridging assets.
2. **The Hybrid Smart Contract System:**
   - The *Solana Hub* successfully manages the global financial state, proving that a high-performance chain can serve as the settlement layer for assets on slower chains.
   - The *EVM Controllers* function effectively as decentralized vaults, ensuring secure asset custody with minimized gas costs.
3. **The Guardian Infrastructure:** The Python-based middleware proved robust in synchronizing state. It successfully handles event ingestion, cryptographic verification, and transaction relaying, ensuring eventual consistency between the EVM and SVM environments.

a, Project Statistics

The scale and complexity of the implementation are reflected in the codebase metrics. Table 6 provides a detailed breakdown of the lines of code (LOC) across different modules.

Component	Language / Technology	Lines of Code (LOC)
Frontend Client	TypeScript	42,355
	TypeScript JSX (React Components)	19,174
Smart Contracts (SVM)	Rust (Anchor Framework)	10,633
	TypeScript (Integration Tests)	$\approx$ 34,000
Smart Contracts (EVM)	Solidity	733
Backend (Guardian)	Python	$\approx$ 36,000
Total Project Size	$\approx$ **142,895 LOC**	

Bảng 6: Source Code Statistics by Component

0.5 Testing

Ensuring the reliability and security of smart contracts is the paramount objective of the testing phase, particularly for a DeFi protocol handling user assets across multiple blockchains. The testing strategy employed for this project is comprehensive, moving from atomic unit tests of individual functions to complex integration scenarios that simulate the entire cross-chain lifecycle. This section details the methodologies used, the specific test cases designed for critical functionalities, and the final evaluation of the system's robustness.

0.5.1 Testing Methodology

The testing framework leverages a combination of industry-standard techniques to maximize code coverage and vulnerability detection:

- **Unit Testing:** This is the first line of defense. Each function within the Solana `MainContract` (Rust) and EVM `Controller` (Solidity) is tested in isolation. We utilized the *Anchor* framework's testing suite (TypeScript) to verify Solana logic and *Hardhat/Chai* for EVM logic. The goal is to ensure that mathematical calculations (e.g., Health Factor) and state transitions occur exactly as specified.
- **Integration Testing (End-to-End):** Given the hybrid architecture, unit tests alone are insufficient. We simulated the cross-chain environment using a local testnet setup (Solana Bankrun + Anvil Fork). These tests involve mocking the Guardian's role to verify the interaction flow: *User locks on EVM* $\rightarrow$ *Event Emitted* $\rightarrow$ *Mock Relayer submits to Solana* $\rightarrow$ *State Updated*.
- **Fuzz Testing (Property-based Testing):** To detect edge cases that manual test cases might miss (such as integer overflows or rounding errors), we employed fuzzing techniques. This involves feeding the smart contracts with thousands of random inputs (e.g., random amounts, random chain IDs) to ensure the system never enters an undefined state or panics unexpectedly.

0.5.2 Test Cases for Critical Functions

The testing focus was prioritized on three high-risk areas: Cross-chain Minting (Solvency), Cryptographic Verification (Security), and Liquidation (System Health).

a, Function 1: Cross-chain Deposit and Minting

This function is the core value proposition of the protocol. It involves state changes on both chains and requires strict solvency checks.

Test Case ID	Test Scenario & Input	Detailed Procedure & Expected Result	Status
TC-MINT-01 (Happy Path)	**Scenario:** User deposits valid collateral and mints within LTV. **Input:** - Collateral: 10 ETH - Mint: 5000 USDAI - LTV Limit: 80%	**1.** User calls `requestAction` on EVM. **2.** Check EVM Vault balance: increases by 10 ETH. **3.** Mock Guardian submits valid signature to Solana. **4. Expected:** Solana state updates UniversalWallet debt to 5000. Health Factor remains > 1.0. Transaction succeeds.	Passed
TC-MINT-02 (Over-minting)	**Scenario:** User attempts to mint debt exceeding the collateral value. **Input:** - Collateral: 1 ETH ($2000) - Mint: 3000 USDAI - Max LTV: 80%	**1.** Construct request on EVM (this passes as EVM doesn't check price). **2.** Guardian relays to Solana. **3.** Solana contract calculates projected Health Factor. **4. Expected:** Calculation shows $HF < 1.0$. Transaction on Solana reverts with custom error `RiskThresholdExceeded`.	Passed
TC-MINT-03 (Zero Amount)	**Scenario:** User sends a request with 0 amount. **Input:** Amount = 0	**1.** Call `requestAction`. **2. Expected:** EVM Contract reverts immediately with `InvalidAmount` error to save gas and prevent spam.	Passed

Bảng 7: Test Cases for Cross-chain Minting Logic

b, Function 2: Cryptographic Security (Signature Verification)

Since the Solana state is updated based on messages relayed by an off-chain server, verifying that the original user actually signed the message is critical to prevent spoofing.

Test Case ID	Test Scenario & Input	Detailed Procedure & Expected Result	Status
TC-SEC-01 (Valid Sig)	**Scenario:** Guardian submits a payload with a valid Secp256k1 signature. **Input:** - Msg Hash: $H(M)$ - Sig: Sign($User_{priv}$, $H(M)$)	**1.** Solana Gateway invokes native `secp256k1_recover`. **2.** Recovered address is compared with `UniversalWallet.owner`. **3. Expected:** Addresses match. Execution proceeds to Main Contract.	Passed
TC-SEC-02 (Forgery)	**Scenario:** Attacker (or compromised Guardian) tries to mint debt for a Victim using Attacker's signature. **Input:** - Msg: "Mint for Victim" - Sig: Sign($Attacker_{priv}$, Msg)	**1.** Gateway recovers the signer address from the signature. **2.** Contract compares Recovered ($Attacker$) vs Wallet Owner ($Victim$). **3. Expected:** Assertion fails. Transaction reverts with `InvalidSignature`.	Passed
TC-SEC-03 (Replay Attack)	**Scenario:** Attacker resubmits a previously valid Mint request to double the debt. **Input:** A valid payload (M, σ) used in block N.	**1.** First submission succeeds; `UniversalWallet.nonce` increments from k to $k+1$. **2.** Second submission sends same payload (nonce k). **3. Expected:** Contract checks $Payload.nonce(k) == Wallet.nonce(k+1)$. Check fails. Revert with `InvalidNonce`.	Passed

Bảng 8: Test Cases for Security and Verification

c, Function 3: Liquidation Engine

This module ensures the protocol remains solvent by allowing third parties to liquidate bad debt. Testing this requires manipulating price feeds.

Test Case ID	Test Scenario & Input	Detailed Procedure & Expected Result	Status
TC-LIQ-01 (Valid Liquidation)	**Scenario:** Oracle price drops, making user insolvent. **Input:** - Collateral: 1 ETH - Debt: 1500 USD - Price drops: $2000 \rightarrow 1600$	**1.** Mock Oracle updates price to \$1600. Health Factor drops below 1.1. **2.** Liquidator calls `liquidate()`. **3. Expected:** Liquidator pays debt. User's collateral is transferred to Liquidator + Bonus. User's debt is reduced.	Passed
TC-LIQ-02 (Healthy Position)	**Scenario:** Liquidator tries to liquidate a healthy user for profit. **Input:** Health Factor = 1.5	**1.** Call `liquidate()`. **2. Expected:** Contract verifies $HF > Threshold$. Reverts with `PositionHealthy`. No assets are transferred.	Passed

Bảng 9: Test Cases for Liquidation Logic

0.5.3 Evaluation and Results

The testing phase has yielded highly positive results, validating the architectural decisions made during the design phase.

a, Quantitative Summary

- **Total Test Suites:** 12 (Covering EVM Vaults, Solana State, Gateway, and Math Libraries).
- **Total Test Cases Executed:** 84 individual atomic tests.
- **Code Coverage:**
  - Solana (Rust): 92% Statement Coverage.
  - EVM (Solidity): 100% Branch Coverage (due to smaller codebase).

b, Qualitative Analysis

The testing process uncovered a critical issue in the early development phase regarding *decimal precision loss* when converting between EVM (18 decimals) and Solana (9 decimals). This was rectified by implementing a `DecimalScaling` library in the Gateway contract, which was subsequently verified by Test Case `TC-MINT-01`. Furthermore, the integration tests confirmed that the `Mutex` locking mechanism on the EVM controller effectively prevents race conditions, ensuring that user funds are secure even if the Guardian node experiences temporary latency.

0.6 Deployment

Sinh viên trình bày mô hình và/hoặc cách thức triển khai thử nghiệm/thực tế. Ứng dụng của sinh viên được triển khai trên server/thiết bị gì, cấu hình như thế nào. Kết quả triển khai thử nghiệm nếu có (số lượng người dùng, số lượng truy cập, thời gian phản hồi, phản hồi người dùng, khả năng chịu tải, các thống kê, v.v.)