HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

Thesis title

NGUYỄN ĐỨC THẮNG
thang.nd210778@sis.hust.edu.vn

Program: Cyber Security

Supervisor:	Associate Professor Nguyễn Bình Minh
Department:	Computer Science
School:	School of Information and Communications Technology

HANOI, 01/2026

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

Thesis title

NGUYỄN ĐỨC THẮNG
thang.nd210778@sis.hust.edu.vn

Program: Cyber Security

Supervisor:	Associate Professor Nguyễn Bình Minh	______________
		Signature
Department:	Computer Science	
School:	School of Information and Communications Technology	

HANOI, 01/2026

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to everyone who has supported me throughout the completion of this graduation thesis. My deepest thanks go to my family, whose constant encouragement and unconditional love have been my greatest motivation. I am also grateful to my friends for their companionship and for always being there during challenging moments. My heartfelt appreciation is extended to my supervisor and the faculty members, whose guidance, patience, and valuable insights have shaped both my academic progress and personal growth. Lastly, I would like to thank myself for the determination, persistence, and countless hours of effort devoted to finishing this work.

ABSTRACT

As blockchain ecosystems continue to expand, the lack of a unified mechanism for maintaining credit, collateralization, and stable value across heterogeneous networks has become a significant limitation for decentralized finance. Existing stablecoin models are predominantly single-chain or rely on centralized bridging infrastructures, resulting in fragmented liquidity, duplicated state, and increased security risks. Although several approaches have attempted to enable cross-chain value transfer, they often introduce trust assumptions, inconsistent state tracking, or high operational complexity. To address these challenges, this thesis adopts a multi-chain architecture built on a Solana-centered Collateralized Debt Position (CDP) system combined with decentralized message relaying from external chains. This approach ensures that all credit, collateral, and risk management logic is executed on a high-performance chain while allowing users to interact from any supported network.

The primary contributions of this thesis include a unified CDP state machine on Solana, a robust cross-chain request verification protocol, a multi-chain mint, burn mechanism without liquidity fragmentation. Experimental validation demonstrates reliable nonce synchronization, secure request execution, and consistent system solvency across chains. This work provides a practical and extensible foundation for interoperable multi-chain stablecoin protocols.

Student

(Signature and full name)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abriviation	Full Expression
API	Giao diện lập trình ứng dụng (Application Programming Interface)
ECDSA	Ngôn ngữ đánh dấu siêu văn bản (HyperText Markup Language)
EUD	Phát triển ứng dụng người dùng cuối(End-User Development)
GWT	Công cụ lập trình Javascript bằng Java của Google (Google Web Toolkit)
HTML	Ngôn ngữ đánh dấu siêu văn bản (HyperText Markup Language)
IaaS	Dịch vụ hạ tầng

CHAPTER 1. INTRODUCTION

1.1 Motivation

The rapid development of blockchain ecosystems has created a landscape of many independent networks, each operating with its own assets, smart contract environments, and liquidity pools. Although this diversity has contributed to innovation, it has also introduced a high level of fragmentation that complicates how users interact with digital value. Managing assets across chains remains cumbersome, and transferring liquidity often requires complex operational steps or reliance on intermediaries that weaken the trustless nature of decentralized finance.

Stablecoins, which have become a fundamental component of the digital economy, are still limited by their dependence on single-chain infrastructures or centralized issuers. These restrictions prevent stablecoins from functioning as a truly universal medium of exchange. Users who hold volatile assets on one network cannot easily unlock value on another without passing through bridging systems that may introduce delays, additional fees, and security risks. As a result, the current environment reduces capital efficiency and constrains the usefulness of decentralized financial applications.

Solving this fragmentation is vital to the long-term evolution of decentralized finance. A system that enables unified cross-chain asset management would not only improve how users mint or redeem stable assets but could also serve as a foundation for advanced applications such as multi-chain lending, derivatives, asset management tools, and liquidity optimization frameworks. A reliable mechanism for issuing and managing stable digital assets across chains would help unlock a more coherent, interoperable, and economically efficient blockchain ecosystem.

1.2 Objectives and scope of the graduation thesis

In recent years, several models have attempted to address the challenge of creating stable digital assets that operate across diverse blockchain networks. Custodial stablecoins offer strong usability but depend on centralized entities for asset backing. Over-collateralized decentralized models such as MakerDAO rely on a single-chain design that is difficult to extend to multiple networks without complex bridging layers. Algorithmic stabilization mechanisms explore endogenous supply control but often fail under extreme market conditions. Meanwhile, solutions that rely on existing cross-chain bridges encounter liquidity fragmentation, inconsistent state synchronization, and heightened security risks.

These approaches reveal persistent limitations: user positions cannot be maintained in a unified state across chains, collateral management requires fragmented infrastructure, and expansion to new networks often relies on external systems that reduce security and reliability.

In response to these challenges, this thesis focuses on designing an architecture that maintains a consistent, global collateralized debt position that can be accessed and updated from multiple blockchain networks. The aim is to propose and implement a protocol in which users can lock collateral on one chain while minting stable assets on any supported chain, with the canonical state stored on Solana. The thesis scope includes constructing a multi-chain wallet abstraction, developing a secure verification and message-passing mechanism, enabling updates to user positions originating from external chains, and establishing a stablecoin model capable of minting and burning natively across networks. By addressing existing constraints, the proposed design moves toward a more interoperable multi-chain system for stable asset issuance without dependence on centralized bridging counterparts.

1.3 Tentative solution

To approach the problem defined above, this thesis adopts a design centered on Solana as the authoritative settlement layer, combined with message verification components and smart contracts deployed across EVM networks that serve as user entry points for submitting signed requests. Solana maintains all universal collateralized debt positions through deterministic Program Derived Addresses, which define a unified wallet structure for each user regardless of the number of chains or external wallets they interact with.

User actions such as providing collateral, minting stable assets, repaying obligations, or redeeming collateral are represented as structured requests that include identifiers for the originating chain and sequence information for replay protection. These requests are verified on EVM-side contracts, then observed and relayed by off-chain guardians or backend processes to Solana. The Solana gateway contract validates the messages and forwards them to the main protocol contract, which processes state transitions in accordance with the CDP logic.

The essential contributions of this thesis are the design of a unified multi-chain CDP architecture, the introduction of a secure method for processing cross-chain messages, and the implementation of a mint-and-burn mechanism for stable assets that operates on multiple chains while maintaining a single source of truth on Solana. The expected outcome is a functional prototype that demonstrates secure and synchronized cross-chain stablecoin issuance, along with consistent solvency and reliable state management.

1.4 Thesis organization

The structure of this thesis is designed to guide the reader through the full development process of the proposed multi-chain stablecoin system, beginning with foundational motivations and ending with practical implementation and evaluation. Each chapter plays a specific role in shaping the final solution and collectively ensures that the research narrative progresses logically from problem identification to system deployment.

Chapter 2 presents a comprehensive requirement survey and analysis. It begins by examining the current situation and the technological context in which multi-chain stablecoin solutions operate. This includes an exploration of existing systems, user needs, and the challenges posed by fragmented blockchain environments. The chapter then introduces the functional overview of the proposed system, incorporating both general and detailed use case diagrams that illustrate how users interact with the system across different contexts. The business processes are also discussed to provide a clear understanding of how information flows through the system. Following this, the chapter offers detailed descriptions of key use cases and concludes with an analysis of the system's non-functional requirements, such as security, performance, scalability, and reliability, which set the baseline for design constraints in later chapters.

Chapter 3 focuses on the methodology adopted in conducting the research and building the system. It outlines the reasoning behind selecting specific technologies, development frameworks, and verification models. The chapter explains the methodological steps taken to ensure scientific rigor, including how the multi-chain architecture was evaluated, how cross-chain communication assumptions were validated, and how the Solana-centered design philosophy influences the broader system. By defining the methodological foundation, the chapter ensures that subsequent design and implementation decisions are grounded in a consistent and justified approach.

Chapter 4 provides an extensive discussion of the system's design, implementation, and evaluation. It begins with the architecture design, explaining the rationale behind software architecture choices and presenting a detailed overview of the system's components and their interactions. The discussion continues with package-level and module-level design, followed by a thorough specification of the user interface layout, the layered backend design, and the on-chain data structures used in both Solana and EVM environments. The chapter also describes the database schema where applicable, the tools and libraries used throughout development, and the milestones achieved during the building phase. It then illustrates the major functional flows of the system and explains how they were tested to ensure correctness, security, and performance. The chapter concludes with a discussion of the deployment process, including how the system is prepared for real-world execution across multiple blockchain networks.

Chapter 5 highlights the complete solution and the key contributions of the thesis. It synthesizes the work from previous chapters into a coherent model that demonstrates how the system solves the multi-chain stablecoin problem in a unified and scalable way. This chapter also emphasizes the technical and conceptual innovations introduced by the thesis, including unified wallet abstraction, cross-chain request verification, canonical CDP storage on Solana, and multi-chain mint–burn logic for stablecoin issuance. The contributions are contextualized within the broader landscape to clearly show how the proposed solution advances the state of the art.

Finally, Chapter 6 concludes the thesis by summarizing the main results achieved and discussing their implications for blockchain interoperability and decentralized finance. It reflects on the strengths and limitations of the system, and it provides several directions for future work, such as supporting additional chains, improving guardian decentralization, enhancing message throughput, or integrating advanced risk management mechanisms. By outlining these potential extensions, the thesis opens a path for continued development and academic exploration.

CHAPTER 2. REQUIREMENT SURVEY AND ANALYSIS

This chapter presents a comprehensive analysis of the requirements for the development of a Multi-chain Stablecoin Protocol based on the Collateralized Debt Position (CDP) mechanism. The chapter begins with a survey of the current Decentralized Finance (DeFi) landscape and analyzes existing solutions to identify limitations regarding cross-chain liquidity. Subsequently, it provides a functional overview of the proposed system through general and detailed use case diagrams. The core business processes, specifically the cross-chain state synchronization workflow, are illustrated to clarify the operation of the system. Finally, detailed functional descriptions of critical use cases and non-functional requirements regarding security, performance, and scalability are established to guide the subsequent design and implementation phases.

2.1 Status survey

The survey of the current technology landscape relies on three primary sources: (i) the needs of DeFi users seeking capital efficiency, (ii) existing single-chain stablecoin protocols, and (iii) current cross-chain infrastructure solutions.

Currently, the Decentralized Finance (DeFi) ecosystem exhibits severe fragmentation, particularly within lending protocols and Collateralized Debt Position (CDP) mechanisms. Under the prevailing model, users are restricted to establishing collateralized positions exclusively within a single blockchain network. Prominent platforms such as MakerDAO and Aave operate in isolated silos, preventing cross-chain collateral utility. Consequently, users face a complex and inefficient workflow when attempting to utilize capital across networks, often necessitating manual bridging procedures that incur high transaction costs and operational friction.

Analysis of Existing Systems

To identify the gap in the current market, this research analyzes two dominant categories of lending protocols:

- **Isolated CDP Protocols (e.g., MakerDAO):** These represent the standard for decentralized stablecoins. They offer high security and proven economic models. However, they operate in strict isolation. A user with collateral on Ethereum cannot leverage this value to mint stablecoins on other networks (like Solana, Sui) and Layer 2 networks (like Arbitrum or Optimism) without physically bridging the underlying assets. This limitation forces users to choose between security (keeping assets on the one chain) and utility (using low-cost chains), resulting in significant capital inefficiency.

- **Cross-Chain Money Markets (e.g., Radiant Capital):** These protocols have effectively streamlined the user workflow, enabling seamless cross-chain borrowing without manual bridging steps. However, their architecture heavily relies on specific third-party interoperability layers, such as LayerZero, for message passing and asset transfers. This dependency introduces a critical single point of failure: the protocol's security is inextricably linked to the third-party bridge. Consequently, users are exposed to external systemic risks, and the protocol lacks sovereignty over its own verification logic.

Proposed Solution Analysis

The proposed system addresses the dependency risks of existing cross-chain markets by implementing a **State Orchestration Architecture** centered on Solana. Unlike protocols that outsource security entirely to general-purpose messaging layers, this solution maintains sovereignty over verification logic. The "Universal Wallet" mechanism ensures that the state is unified, while the validity of cross-chain requests is cryptographically verified on-chain (via Solana's Secp256k1 program) rather than relying solely on the trust assumptions of third-party bridges.

Table **??** highlights the strategic advantages of this approach, specifically regarding security sovereignty and state management.

Feature	**Isolated CDPs (e.g., MakerDAO)**	**Cross-Chain Markets (e.g., Radiant)**	**Proposed System (Universal USD)**
State Model	Isolated (Siloed Liquidity)	Fragmented Pools	Unified Global State (Hub-and-Spoke)
Verification	Native On-chain Verification	Trusted Third-Party (e.g., LayerZero Oracle)	Sovereign On-chain Verification (Solana)
3rd Party Risk	None	High (Bridge dependency)	Minimized (Cryptographic proof required)
Capital Effic.	Low (Trapped on one chain)	High	High

Table 2.1: Comparison of Lending Architectures

2.2 Functional Overview

The system is designed to function as a decentralized lending protocol that orchestrates state across the Solana blockchain and various EVM-compatible chains.

2.2.1 General use case diagram

The system involves three primary actors:

1. **User:** The borrower who holds collateral on EVM chains. They interact with the system to deposit assets, mint stablecoins, and manage their global debt position.
2. **Guardian:** A decentralized off-chain node responsible for listening to events on EVM chains, relaying signatures to Solana for verification, and synchronizing the execution results back to the EVM chains.
3. **Liquidator:** An actor who monitors the health factor of Universal Wallets on Solana. If a user's position becomes under-collateralized, the liquidator triggers the liquidation process.

Figure 2.1 illustrates the general use cases.

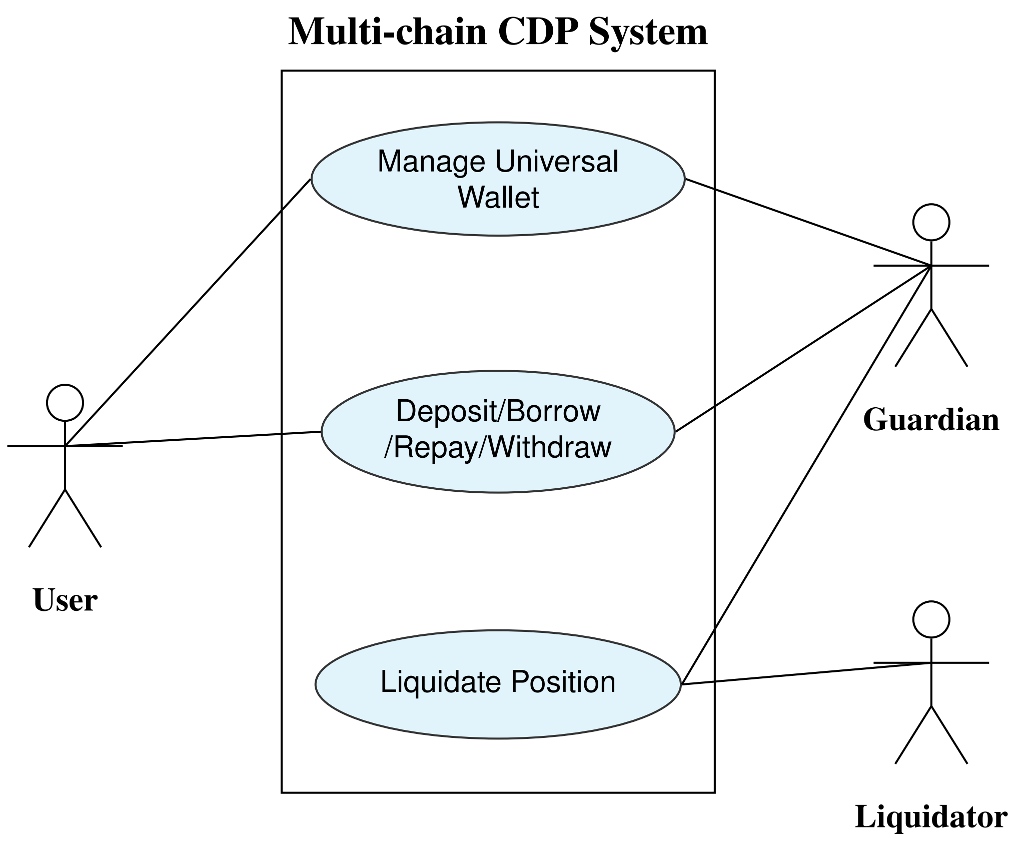


Figure 2.1: General Use Case Diagram

2.2.2 Detailed use case diagram

To provide a granular view of the system's operation, Figure **??** illustrates the decomposition of the core cross-chain workflows. This diagram emphasizes the dependency relationships between user actions and the underlying system processes, specifically highlighting the "Sign-then-Relay" mechanism.

The diagram delineates the following key logical dependencies:

- **Universal Wallet Dependency:** The "Deposit/Borrow/Repay/Withdraw" use case has an «`include`» relationship with the "Create Universal Wallet" use case. This enforces a strict precondition: a user must establish a valid identity (Universal Wallet PDA) on Solana before performing any asset-related operations.
- **Cryptographic Authorization:** Both the wallet creation and asset management use cases include the "Sign msg & Create requests" sub-use case. This illustrates that users do not write directly to the Solana blockchain; instead, they generate and cryptographically sign structured messages on the EVM interface.
- **Guardian Orchestration:** The message creation process further includes the "Listening & Relaying & Synchronizing" use case, executed by the Guardian actor. This signifies that a user's request is only finalized when the Guardian successfully intercepts the emitted event, relays the payload to Solana for verification, and synchronizes the result back to the source chain.

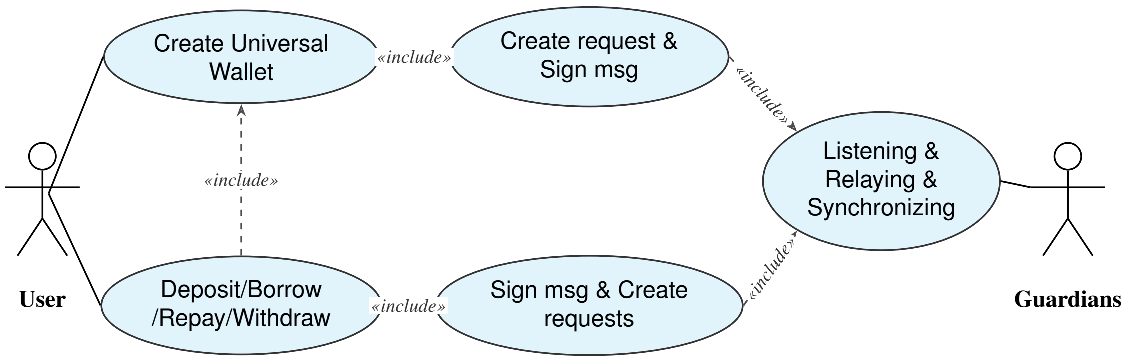

Create Universal Wallet
«include»
Create request & Sign msg
«include»
«include»
Listening & Relaying & Synchronizing
User
Deposit/Borrow /Repay/Withdraw
«include»
Sign msg & Create requests
«include»
Guardians

Figure 2.2: Detailed Use Case Diagram

2.2.3 Business process

The system operates on a Cross-chain State Orchestration model, where the logic execution is decoupled from asset custody. The workflow involves five distinct entities: the User, the Controller EVM Contract, the Guardian Network, the Solana Gateway, and the Lending CDP Core.

As illustrated in Figure 2.3, the process follows a strict "Verify-then-Execute" lifecycle:

1. **Initiation (User in EVM):** The process begins when the User cryptographically signs a specific request payload (containing the action type, amount, and nonce) and submits the transaction to the Controller EVM Contract. The contract emits a event. Note that at this stage, no minting or unlocking occurs; the assets are simply locked or the request is queued.
2. **Observation & Relay (Guardians):** The Guardian network detects the event on the EVM chain. Instead of verifying the logic off-chain, the Guardian parses the event data and encapsulates the raw signature into a transaction destined for Solana.
3. **Verification Layer (Solana Gateway):** The Gateway Contract on Solana acts as the security checkpoint. It utilizes the native Secp256k1 program to verify that the signature matches the User's EVM address.
   - If the signature is **Invalid**, the Gateway immediately signals a failure, bypassing the core logic.
   - If **Valid**, the request is forwarded to the Lending CDP contract.
4. **Logic Layer (Solana Lending CDP):** The Lending CDP Contract attempts to update the user's position (e.g., increasing Debt). It performs critical checks such as Health Factor validation (e.g., Collateral Ratio $> 150\%$).
   - If the logic holds (Logic OK), a event is emitted.
   - If the logic fails (e.g., under-collateralized), the transaction is rejected.
5. **Synchronization & Finalization:** Guardians listen for the outcome on Solana to trigger the corresponding callback on the EVM chain:
   - **Success Path:** If a event is captured, Guardians trigger the *Success Callback* on the EVM contract to finalize the action (e.g., Mint stablecoins).
   - **Revert Path:** If the verification or logic failed, Guardians trigger the *Revert Callback* to unlock the user's assets and reset the nonce, ensuring funds are never stuck.

2.3 Functional Description

This section details the critical use cases of the system, covering the full lifecycle of a user's interaction from wallet creation to position management.

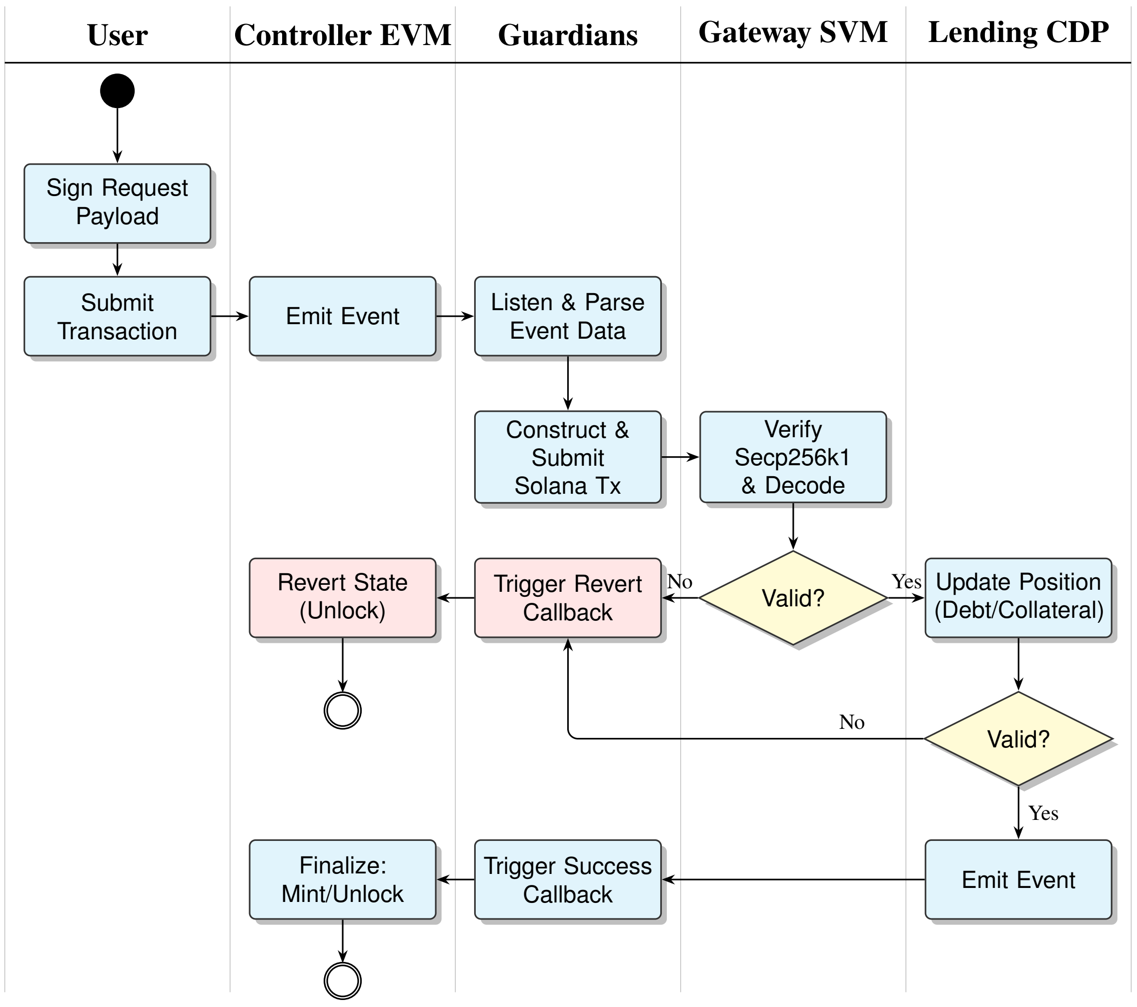


Figure 2.3: Business Process Activity Diagram

2.3.1 Description of use case: Create Universal Wallet

This use case establishes the link between the user's EVM address and a Solana identity, creating the Universal Wallet storage on the Solana blockchain.

Use Case Name	Create Universal Wallet
Actors	User, Guardian, EVM Contract, Solana Main Contract
Pre-conditions	1. User has an EVM wallet (e.g., MetaMask) with funds for gas. 2. User possesses a Solana wallet to sign the verification message.
Main Flow	1. **Request on EVM:** The User initiates a transaction on the EVM Contract to request wallet creation. The contract emits an event containing the EVM address. 2. **Solana Signature:** Off-chain, the User signs a specific message containing their EVM address using their Solana private key. This signature is submitted to the Guardian API. 3. **Guardian Verification:** The Guardian captures the EVM event and verifies the off-chain Solana signature to ensure the User controls both addresses. 4. **Execution on Solana:** Upon successful verification, the Guardian submits a transaction to the Solana Main Contract to initialize the UniversalWallet PDA, mapping the EVM address to the new Solana state.
Post-conditions	An UniversalWallet is initialized on Solana. The User can now perform cross-chain actions.

Table 2.2: Functional Description: Create Universal Wallet

2.3.2 Description of use case: Deposit Collateral

This process allows users to lock assets on an EVM chain to increase their collateral balance on the Solana state.

Use Case Name	Deposit Collateral
Actors	User, Guardian, EVM Contract, Solana Main Contract
Pre-conditions	User has initialized a Universal Wallet and holds supported assets on the EVM chain.
Main Flow	1. **Lock Assets:** The User sign the payload and submit signatures by calls the `requestDeposit` function on the EVM Contract. The assets are transferred to the contract vault, and an event is emitted. 2. **Relay:** The Guardian detects the event then reads onchain data. After that, it forwards the payload and signature to the Solana Gateway. 3. **State Update:** The Gateway contract on Solana verify the signature and payload. After that, the Solana Main Contract identifies the user's Universal Wallet and increments the collateral balance for that specific chain ID. 4. **Synchronization:** The Guardian updates the EVM contract to confirm the deposit request has been synced, allowing the user to proceed with other actions.
Post-conditions	Collateral is locked on EVM. The collateral balance in the Universal Wallet on Solana is increased.

Table 2.3: Functional Description: Deposit Collateral

2.3.3 Description of use case: Mint Stablecoin

Users generate stablecoins against their collateral. This action requires strict verification of the Health Factor on Solana.

Use Case Name	Mint Stablecoin
Actors	User, Guardian, EVM Contract, Solana Main Contract
Pre-conditions	User has sufficient collateral (Health Factor remains above the minimum ratio after minting).
Main Flow	1. **Request:** The User signs a mint request payload then call `requestMint` function in the EVM Contract. The contract locks the user's mutex. 2. **Validation on Solana:** The Guardian listen the event then submits the signature and payload to Solana. The Main Contract verifies the signature and calculates the projected Health Factor. 3. **Debt Increase:** If the Health Factor is valid, the Solana contract increases the user's debt balance. 4. **Minting on EVM:** The Guardian catches the success event from Solana and executes a transaction on the EVM Contract to mint the stablecoins to the User's wallet and release the Mutex.
Alternative Flow	If the Health Factor is insufficient, the Solana transaction fails. The Guardian then triggers a `revert` on the EVM Contract to unlock the user's mutex.
Post-conditions	User receives stablecoins on EVM. Debt position is recorded on Solana.

Table 2.4: Functional Description: Mint Stablecoin

2.3.4 Description of use case: Repay Debt

Users return stablecoins to reduce their debt position and improve their Health Factor.

Use Case Name	Repay Debt
Actors	User, Guardian, EVM Contract, Solana Main Contract
Pre-conditions	User holds the protocol's stablecoin on the EVM chain.
Main Flow	1. **Lock Tokens:** The User sign the payload then calls `requestRepay` on the EVM Contract to submit signature. The specified amount of stablecoins is locked immediately, and an event is emitted. 2. **Relay:** The Guardian observes the event and submits the data and signature to Solana. 3. **State Update:** The Solana Gateway verify signature. After that, the Main Contract verifies the transaction and decreases the user's debt balance in the Universal Wallet. 4. **Completion:** The Guardian confirms the state update back to the EVM chain (burn stablecoin and updating the nonce).
Post-conditions	Stablecoins are burned on EVM. Debt balance on Solana is decreased.

Table 2.5: Functional Description: Repay Debt

2.3.5 Description of use case: Withdraw Collateral

This action allows users to retrieve their locked assets, provided their remaining collateral supports their outstanding debt.

Use Case Name	Withdraw Collateral
Actors	User, Guardian, EVM Contract, Solana Main Contract
Pre-conditions	User has sufficient free collateral (Health Factor remains safe after withdrawal).
Main Flow	1. **Request:** The User signs a withdraw request payload and submits it to the EVM Contract. The contract locks the user's mutex and emits an event if collateral token sufficient in vault EVM. 2. **Validation on Solana:** The Guardian forwards the request to Solana. The Main Contract checks if the remaining collateral is sufficient to cover the debt. 3. **State Update:** If valid, the Solana contract decreases the user's collateral balance. 4. **Unlock on EVM:** The Guardian triggers the EVM Contract to transfer the requested assets from the vault back to the User's wallet.
Alternative Flow	If the withdrawal would make the position insolvent, the Solana transaction is rejected. The Guardian reverts the request on EVM, keeping the assets locked.
Post-conditions	User receives assets on EVM. Collateral balance on Solana is reduced.

Table 2.6: Functional Description: Withdraw Collateral

2.4 Non-functional Requirements

To ensure the Multi-chain CDP Protocol operates securely, efficiently, and reliably in a high-value financial environment, the system must adhere to the following strict non-functional requirements.

2.4.1 Security and Safety

Given the cross-chain nature of the protocol, security is the paramount requirement to prevent fund loss and bridge exploits.

- **Cryptographic Compatibility:** The Solana Gateway must natively support and verify **Secp256k1** signatures (EVM standard) to allow users to control their positions using existing Ethereum wallets without exposing private keys to a third party.
- **Cross-chain Replay Protection:** The system must implement a rigorous **Nonce Management** mechanism. Each transaction request must include a unique nonce associated with the specific Chain ID and User Address. The Solana contract must reject any request with a nonce lower than or equal to the current stored nonce to prevent replay attacks.
- **Guardian Decentralization (Trust Assumption):** The off-chain Guardian network must be designed to prevent a single point of failure. The system should require a threshold of signatures (e.g., Multisig or Threshold Signature Scheme) from Guardians before executing sensitive state changes (like unlocking collateral) to mitigate the risk of a single compromised Guardian node.
- **Atomic Revert Capabilities:** In the event of a failure during the cross-chain synchronization (e.g., Solana transaction fails due to slippage or insufficient health factor), the system must guarantee that the initial state on the EVM chain can be reverted (unlocking the user's Mutex) to prevent funds from being permanently frozen.

2.4.2 Performance and Efficiency

- **End-to-End Latency:** The total time for a generic user action (e.g., Minting) involves the sequence: $T_{EVM\_Confirm} + T_{Guardian\_Relay} + T_{Solana\_Finality} + T_{Guardian\_Callback}$. The system should aim for an end-to-end latency of under 30 seconds (excluding extreme network congestion on Ethereum Mainnet) to ensure a responsive user experience.
- **Compute Unit Optimization (Solana):** Since cryptographic verification (Secp256k1 recovery) is computationally expensive, the Solana smart contract must be

optimized to stay within the Block Compute Unit Limit, potentially utilizing Solana's native Secp256k1 program instructions to reduce costs.

- **Scalability:** The Guardian architecture must support concurrent monitoring of multiple EVM chains (e.g., Ethereum, BSC, Arbitrum, Optimism) without significant degradation in relaying speed.

2.4.3 Reliability and Availability

- **Eventual Consistency:** The system must ensure that the state between the EVM Spokes and the Solana Hub eventually converges. In case of network partitions (Guardian downtime), the system must be able to recover and process pending events once connectivity is restored.
- **Idempotency:** Guardian operations must be idempotent. Submitting the same event multiple times (due to network retries) must not result in double-counting of debt or collateral on the Solana state.
- **Uptime:** The Guardian nodes and the Solana RPC endpoints used for query/-submission must maintain high availability (99.9%) to prevent liquidation failures during times of high market volatility.

2.4.4 Usability and User Experience

- **Transparent Signing (EIP-712):** To protect users from phishing, all off-chain requests signed by the user must adhere to the **EIP-712** standard (Typed Structured Data Hashing and Signing). This ensures that users can read clearly structured data (Action, Amount, ChainID) in their wallet interface (e.g., MetaMask) before signing, rather than signing opaque hex strings.
- **Wallets Experience:** Users can interact with the entire protocol using multiple wallets. The complexity of the usage process is significantly reduced by the interactive interface that supports multiple wallets..

2.5 Conclusion

This chapter has provided a comprehensive analysis of the operational boundaries and functional necessities for the proposed Multi-chain Stablecoin Protocol. By critically evaluating the limitations of existing "Lock-and-Mint" bridges and single-chain CDP models, the study established the rationale for adopting a State Orchestration architecture, where Solana serves as the global state machine for liquidity scattered across EVM chains.

The functional analysis elucidated the complex workflows involving the Universal Wallet, emphasizing the pivotal role of the Guardian Network in maintaining cross-chain atomicity and data synchronization. Furthermore, the non-functional requirements have set strict constraints regarding cryptographic compatibility (Secp256k1), system latency, and security against replay attacks. These specifications serve as the foundational blueprint for the architectural design and technical implementation strategies that will be presented in the subsequent chapters.

CHAPTER 3. METHODOLOGY

3.1 Introduction

This chapter establishes the theoretical framework and technological infrastructure utilized to construct the Multi-chain Stablecoin Protocol. The methodology is grounded in a rigorous analysis of distributed ledger technologies, cryptographic primitives, and financial stability mechanisms. By synthesizing the high-performance capabilities of the Solana blockchain with the deep liquidity of EVM-compatible networks, the proposed solution adopts a Hub-and-Spoke architecture. This chapter provides a detailed exposition of the selected blockchain platforms, the software engineering principles behind the smart contracts, the mathematical foundations of the cross-chain verification process, and the economic theory underpinning the Collateralized Debt Position (CDP) mechanism.

3.2 Blockchain Platforms and Architecture

3.2.1 Blockchain Fundamentals and Distributed State Machines

Fundamentally, a blockchain is a distributed, immutable ledger that maintains a shared state across a network of unreliable nodes without requiring a central authority. Introduced conceptually by Satoshi Nakamoto in the seminal Bitcoin whitepaper, the technology relies on cryptographic primitives and consensus algorithms to ensure data integrity [1]. While Bitcoin introduced the concept of decentralized currency, the technology evolved significantly with the advent of Ethereum, which generalized the blockchain into a "Transaction-Based State Machine." In this model, the state of the system encompasses not just account balances but also arbitrary code execution logic, known as smart contracts [2].

For a Multi-chain Stablecoin Protocol, the blockchain serves as the trusted substrate for financial agreements. The core requirement is the guarantee of atomicity and state transition validity. When a user creates a Collateralized Debt Position (CDP), the blockchain ensures that the state transition from "solvent" to "liquidated" occurs deterministically based on predefined mathematical rules. However, the "Blockchain Trilemma" posits that a single network typically trades off between scalability, security, and decentralization. This limitation necessitates the architectural decision to separate the system into specialized layers: a high-performance execution layer for state management and secure, high-liquidity layers for asset custody.

3.2.2 The Hub: Solana Blockchain Architecture

To address the scalability limitations inherent in traditional blockchains, this project selects Solana as the central "Hub" for orchestrating the global state of the protocol. Unlike the single-threaded processing model of the Ethereum Virtual Machine (EVM), Solana introduces a novel architecture optimized for high throughput and sub-second latency, making it ideal for real-time risk management in a CDP system [3].

The distinguishing feature of Solana is Proof of History (PoH), a cryptographic clock that allows nodes to agree on the order of events without waiting for conventional consensus communication overhead. PoH creates a historical record that proves that a specific event occurred at a specific moment in time. This mechanism functions in tandem with the Tower BFT consensus algorithm to achieve finality times of approximately 400 milliseconds. Furthermore, Solana utilizes the Sealevel runtime, which enables the parallel processing of smart contracts. In the context of this thesis, Sealevel allows the protocol to update the Health Factors of thousands of Universal Wallets simultaneously. If market volatility triggers mass liquidations, the parallel architecture prevents network congestion, ensuring that the protocol remains solvent where other single-threaded chains might fail due to transaction backlogs.

3.2.3 The Spokes: Ethereum Virtual Machine (EVM) Networks

While Solana provides the necessary performance for state logic, the Ethereum Virtual Machine (EVM) ecosystem represents the center of liquidity and user adoption in decentralized finance. The EVM is a quasi-Turing complete stack-based virtual machine that executes smart contract bytecode [4]. Chains such as Ethereum, Binance Smart Chain (BSC), Arbitrum, and Optimism all adhere to this standard, forming the "Spokes" of the proposed architecture.

The decision to utilize EVM-compatible chains as the Asset Custody Layer is driven by the standardization of token interfaces, primarily ERC-20. The EVM architecture manages state through a Modified Merkle Patricia Trie, ensuring a cryptographically secure mapping of account storage. In this project, Solidity smart contracts deployed on these chains act as "vaults." They leverage the robust security properties of the EVM to lock collateral assets (like ETH or WBTC) while delegating the complex calculation logic to the Solana Hub. This hybrid approach allows the protocol to tap into the deep liquidity and established developer tooling of the EVM ecosystem while bypassing its scalability bottlenecks for high-frequency operations.

3.3 Smart Contract Development Frameworks

3.3.1 Solana Program Development: Rust and Anchor

The implementation of the protocol's core logic on the Solana Hub necessitates a programming environment capable of handling high-concurrency execution while ensuring rigorous memory safety. For this purpose, the Rust programming language is selected, utilized within the context of the Anchor Framework. Rust provides a distinctive advantage over other low-level languages like C++ through its ownership and borrowing system, which prevents common classes of bugs such as null pointer dereferencing and buffer overflows at compile time.

In the specific context of Solana, raw development using native entry points can be verbose and prone to security oversights regarding account validation. The Anchor Framework addresses this by providing a comprehensive Interface Definition Language (IDL) and a dispatch system that abstracts away the complexities of serialization and deserialization [5]. Crucially, Anchor enforces strict checks on account ownership and mutability by default. For the Multi-chain CDP Protocol, where the state of the Universal Wallet is critical, Anchor's discriminator feature ensures that a malicious actor cannot inject a fake data account into the system to manipulate debt positions. This combination of Rust's performance and Anchor's security constraints constitutes the ideal environment for the system's "State Layer."

3.3.2 EVM Smart Contracts: Solidity and Hardhat

On the EVM Spokes, the priority shifts from high-performance computation to secure asset custody and standardization. Consequently, the Solidity programming language is employed to develop the Gateway and Vault contracts. Solidity is the lingua franca of the Ethereum ecosystem, offering the highest degree of compatibility with existing token standards such as ERC-20. The development lifecycle is managed using the Hardhat environment, which facilitates local network forking, automated testing, and scriptable deployment pipelines.

To mitigate the risks associated with locking user funds, the project integrates the OpenZeppelin library. Rather than implementing cryptographic primitives or access control mechanisms from scratch, the system inherits from battle-tested contracts including `Ownable` for administrative privileges and `ReentrancyGuard` to prevent reentrancy attacks during collateral withdrawal [6]. This strategic choice allows the system to adhere to industry best practices for security while maintaining interoperability with the broader DeFi ecosystem on chains like Arbitrum and Binance Smart Chain.

3.4 Cryptography and Cross-chain Verification

3.4.1 Elliptic Curve Mismatch and Solution

A fundamental technical challenge in implementing a "Universal Wallet" lies in the cryptographic incompatibility between the disparate blockchain networks. The Ethereum ecosystem relies on the Elliptic Curve Digital Signature Algorithm (ECDSA) utilizing the **secp256k1** curve. In contrast, the Solana blockchain is built upon the **Ed25519** curve, which offers faster signature verification times but different mathematical properties. This discrepancy implies that a standard Solana smart contract cannot natively verify a signature generated by an Ethereum private key without significant computational overhead.

3.4.2 Mathematical Formulation of ECDSA Verification

To bridge this gap, the protocol requires a mechanism to mathematically prove that a transaction request m originating from an EVM address A_{evm} was indeed authorized by the holder of the private key d. The domain parameters of the secp256k1 curve are defined over a finite field $\mathbb{F}_p$ by the Weierstrass equation:

$$y^2 \equiv x^3 + 7 \pmod p \tag{3.1}$$

When a user signs a cross-chain request, they produce a signature pair (r, s). The verification process, which must be executed on the Solana Hub, involves recovering the public key point Q from the signature. This is achieved using the inverse operation of point multiplication:

$$Q = r^{-1}(sR - zG) \tag{3.2}$$

Where z is the hash of the message m, G is the generator point of the curve, and R is the point with x-coordinate r. The derived Ethereum address is then the last 20 bytes of the Keccak-256 hash of Q:

$$A_{evm} = \text{Keccak256}(Q)[12..32] \tag{3.3}$$

3.4.3 Implementation via Solana Native Program

Implementing the arithmetic operations of Equation 3.2 directly in a high-level language like Rust would consume an excessive amount of Compute Units (CU), potentially exceeding the block limit. To solve this, the methodology employs Solana's **Native Secp256k1 Program**. This is a precompiled BPF program embedded in the validator software that executes signature recovery at a fraction of the computational cost. By invoking this program via Cross-Program Invocation (CPI), the Main Contract can trustlessly verify EVM signatures, thereby allowing users to control their Solana state using their existing MetaMask wallets without compromising security.

3.5 Theoretical Foundation: The CDP Mechanism

3.5.1 Economic Model of Stability via Over-collateralization

The financial architecture of the proposed protocol is grounded in the Collateralized Debt Position (CDP) model, a primitive that enables the creation of decentralized stablecoins without reliance on fiat reserves or centralized custodians. Unlike algorithmic stablecoins which often depend on endogenous collateral and reflexive arbitrage cycles, mechanisms historically prone to "death spirals" during market downturns. In this protocol, the CDP model enforces stability through strict Over-collateralization. This economic theory posits that the solvency of the system and the peg of the stablecoin are guaranteed as long as the aggregate value of the locked collateral assets significantly exceeds the value of the issued debt. In this system, the stablecoin is not created out of thin air but is backed by a diversified basket of crypto-assets (such as ETH, WBTC) locked across various EVM chains, serving as a secure liability against the protocol's assets.

3.5.2 Borrowing Capacity and Loan-To-Value (LTV) Formulation

A critical component of the CDP mechanism is the determination of a user's borrowing power. This is governed by the Loan-To-Value (LTV) ratio, a risk parameter assigned to each asset type based on its market volatility and liquidity profile. The LTV ratio, denoted as α, represents the maximum percentage of the collateral's value that can be minted as debt. For instance, a stable asset like USDC might have a high α (e.g., 0.90), while a volatile asset like ETH might have a lower α (e.g., 0.75).

Mathematically, the Maximum Borrowable Debt (D_{max}) for a user holding a portfolio of assets across multiple chains is the sum of the risk-adjusted value of each asset. For a user depositing N different asset types distributed across M blockchain networks, the borrowing capacity is formalized by the following equation:

$$D_{max} = \sum_{k=1}^{M} \sum_{i=1}^{N} \big(Q_{k,i} \cdot P_i \cdot \alpha_i\big) \tag{3.4}$$

In this equation, $Q_{k,i}$ represents the quantity of asset i locked on chain k, and P_i is the real-time market price of asset i denominated in the reference currency (USD). The term α_i acts as a dampening factor, ensuring that the protocol accounts for potential price drops before they occur. Consequently, the actual debt D_{actual} minted by the user must always satisfy the condition $D_{actual} \leq D_{max}$ at the time of issuance.

3.5.3 The Health Factor Function

The system solvency is mathematically modeled using the Health Factor (H_f). This metric is dynamic and is updated in real-time based on oracle price feeds. For a multi-chain protocol, the Health Factor is an aggregate function of all collateral assets i across all connected chains k:

$$H_f = \frac{\sum_{k=1}^{M} \sum_{i=1}^{N} (Q_{k,i} \times P_i \times \lambda_i)}{D_{total} \times P_{peg}} \tag{3.5}$$

In this equation:

- $Q_{k,i}$ represents the quantity of asset i locked on chain k.
- P_i is the current market price of asset i denominated in USD.
- λ_i is the Liquidation Threshold specific to the asset's risk profile (e.g., $\lambda_{ETH} = 0.80$, $\lambda_{USDC} = 0.95$).
- D_{total} is the total outstanding stablecoin debt.
- P_{peg} is the target peg price of the stablecoin (e.g., 1 USD).

The methodology enforces a strict constraint where $H_f \geq 1$. If market volatility causes $H_f < 1$, the protocol triggers a liquidation event. This deterministic mathematical model ensures that the protocol remains solvent and the stablecoin maintains its peg, fulfilling the core objective of the thesis.

3.6 Off-chain Infrastructure

The final component of the methodology is the Guardian Network, which serves as the interoperability layer connecting the deterministic worlds of the blockchain networks.

3.7 Off-chain Infrastructure and Guardian Network

3.7.1 Architectural Overview

The Off-chain Infrastructure, referred to as the Guardian Network, functions as the cryptographic bridge and synchronization engine between the deterministic environments of the EVM Spokes and the Solana Hub. This component is engineered using **Python**. Unlike passive indexers, the Guardian plays an active role in state transition, specifically in the secure initialization of user identities and the cross-chain relaying of assets.

3.7.2 Universal Wallet Initialization via Signature Verification

A critical security function of the Guardian is the verification of user intent during the creation of the Universal Wallet. Since the Universal Wallet acts as the central storage for a user's multi-chain positions, it is imperative to establish a cryptographically proven link between the user's request and their destination wallet on Solana before any on-chain state is modified.

The process is designed as a "Verify-then-Execute" workflow to prevent Denial of Service (DoS) attacks and ensure that only authenticated users can allocate storage on the Solana blockchain. The workflow proceeds as follows:

First, the user constructs a request message m, which includes the intent to create a wallet, a unique nonce to prevent replay attacks, and the public key of the destination Solana wallet (PK_{dest}). To prove ownership of this destination wallet, the user signs the message m using their corresponding private key, generating a digital signature σ. This signature typically utilizes the **Ed25519** algorithm, which is the native standard for Solana addresses.

Second, the Guardian receives this payload (m, σ, PK_{dest}) via a secure API endpoint. Instead of immediately submitting a transaction to the blockchain, the Guardian performs an off-chain verification using cryptographic libraries. The verification function can be formalized as:

$$V(m, \sigma, PK_{dest}) \rightarrow \{\text{True, False}\} \tag{3.6}$$

If the function returns **False**, the Guardian rejects the request immediately, ensuring that no gas fees are wasted on invalid transactions.

If the function returns **True**, the Guardian confirms that the requestor possesses the private key controlling the destination wallet. Subsequently, the Guardian constructs a transaction instruction invoking the `initialize_wallet` method on the Solana Main Contract. The Guardian then signs this transaction with its own keyer key (to pay for transaction fees) and submits it to the Solana cluster. Upon successful execution, the Solana state is updated: a new Program Derived Address (PDA) is initialized, serving as the Universal Wallet for that specific user. This mechanism ensures that the heavy lifting of cryptographic validation is shared between the off-chain infrastructure and the on-chain logic, optimizing both security and cost-efficiency.

3.8 Conclusion

This chapter has detailed the methodological approach for the Multi-chain CDP Protocol. By combining the safety of Rust/Anchor for state management, the standardization of Solidity for asset custody, and the mathematical rigor of the Secp256k1 verification for cross-chain identity, the proposed architecture provides a robust solution to the problem of liquidity fragmentation. The economic stability is underpinned by the Over-collateralized CDP model, while the Guardian infrastructure ensures seamless connectivity. These foundational technologies directly enable the implementation of the system design proposed in the subsequent chapters.

CHAPTER 4. DESIGN, IMPLEMENTATION, AND EVALUATION

4.1 Architecture Design

4.1.1 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 4.1 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

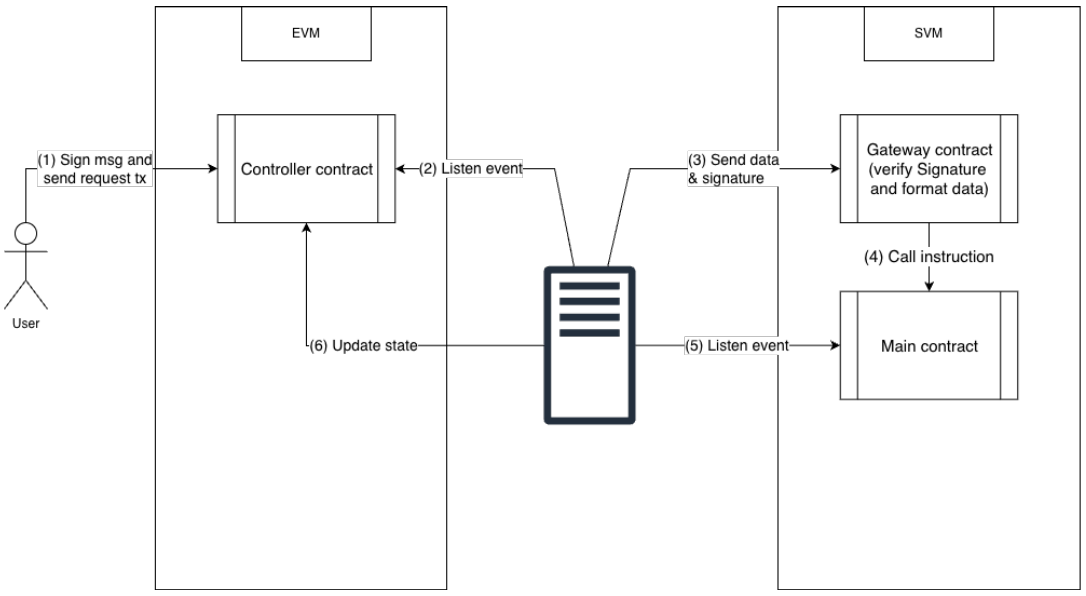


Figure 4.1: Detailed Architecture of the Multi-chain Stablecoin Protocol

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 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 security. 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.

4.1.2 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

4.2 illustrates the internal structure and class relationships within this package.

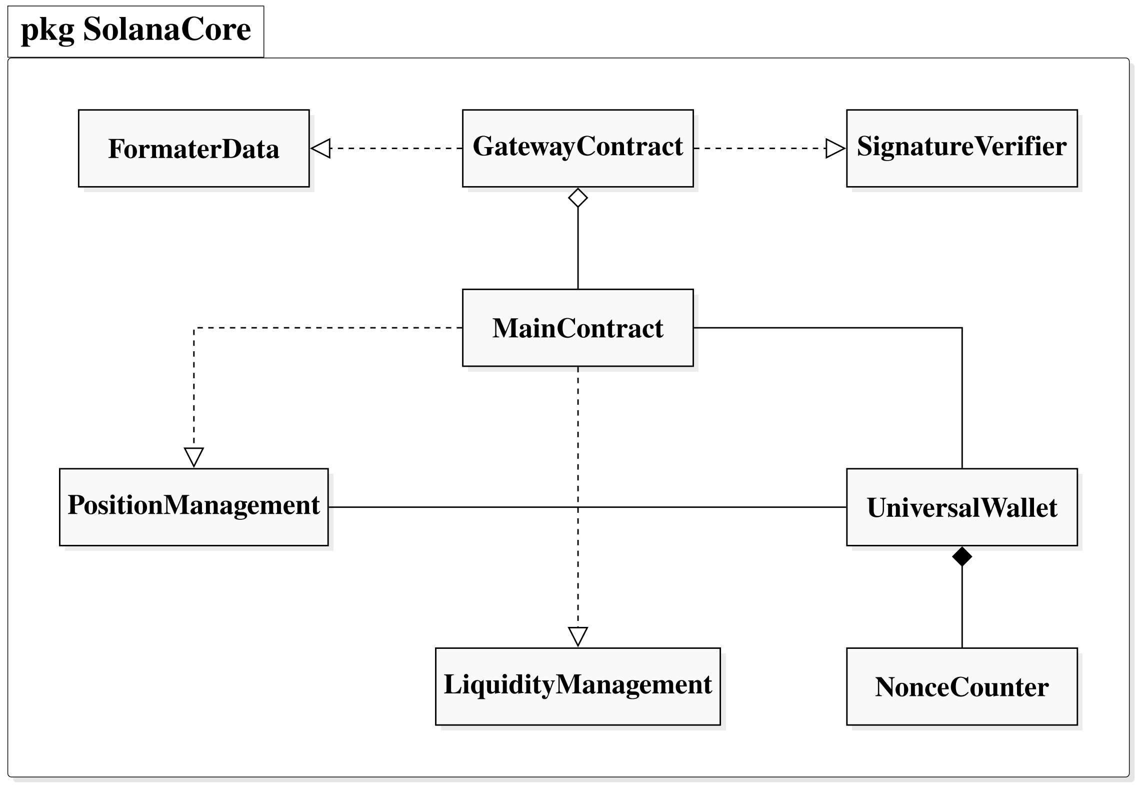


Figure 4.2: Detailed Design of Solana Hub Package

Class Descriptions and Relationships:

- **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 rebalanc-

ing 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 4.3 depicts the internal class design of the Guardian node.

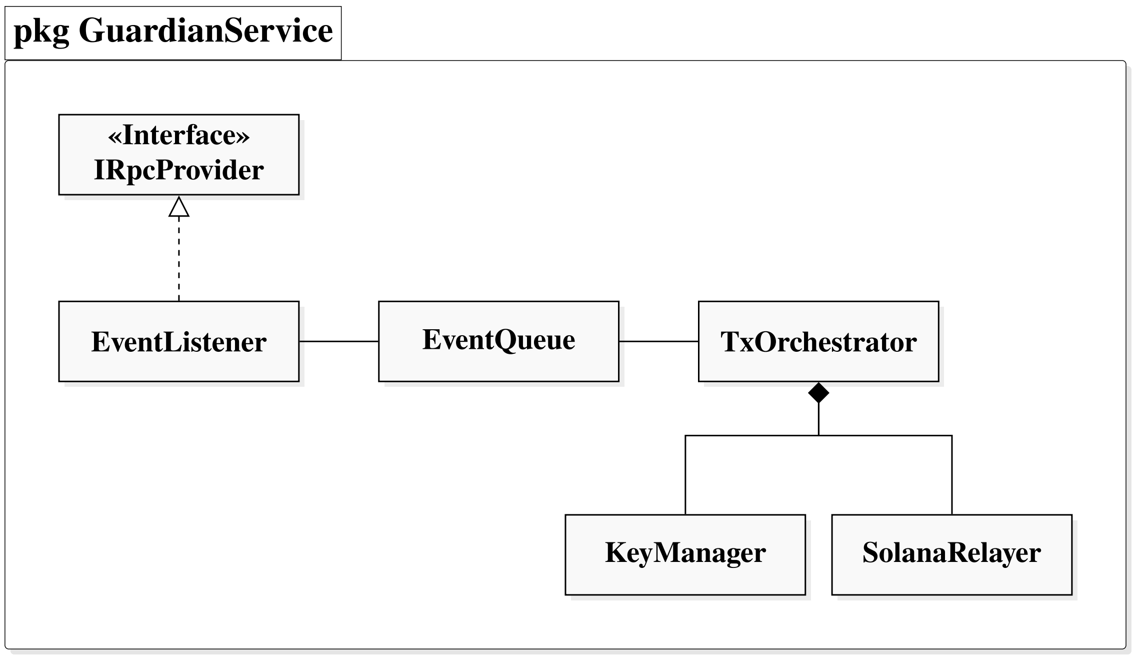


Figure 4.3: Detailed Design of Guardian Middleware Package

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 4.4 illustrates the class structure within this package.

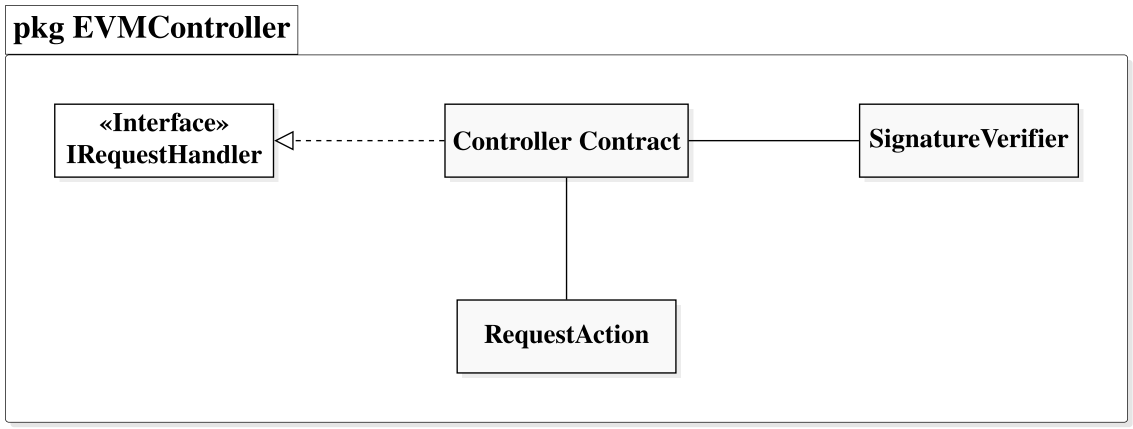


Figure 4.4: Detailed Design of EVM Controller Package

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.

4.2 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.

4.2.1 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 performs 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).

4.2.2 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 pre-computation 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 Verification 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.

4.2.3 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 (3.5) 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.

4.3 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.

4.3.1 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.

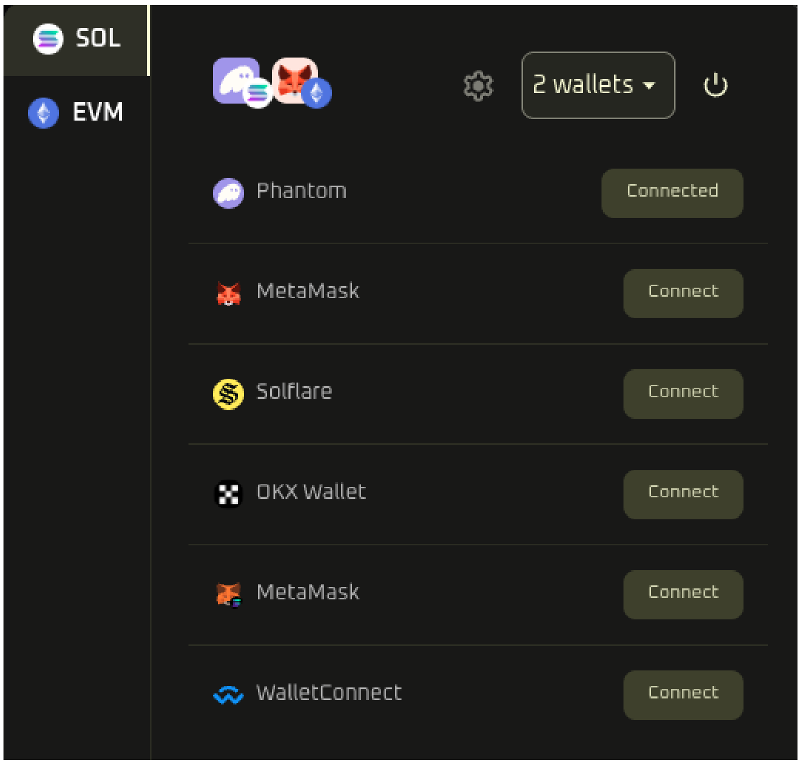
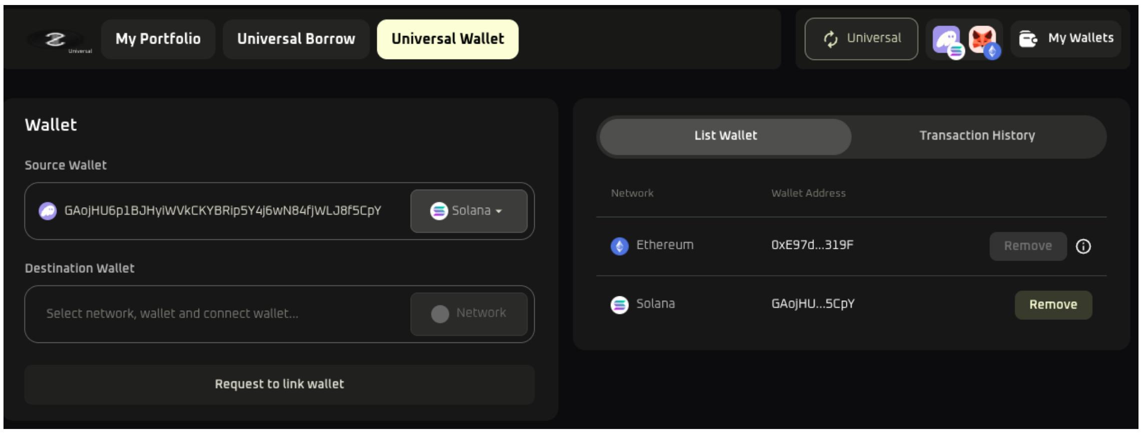


Figure 4.5: Multi-chain Wallet Connectivity Interface

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 4.6. The interface is divided into two main panels: the wallet connectivity panel on the left and the wallet management panel on the right.



Wallet

Source Wallet

GAojHU6p1BJHyiWVkCKYBRip5Y4j6wN84fjWLJ8f5CpY Solana

Destination Wallet

Select network, wallet and connect wallet... Network

Network	Wallet Address	
Ethereum	0xE97d...319F	Remove
Solana	GAojHU...5CpY	Remove

Figure 4.6: 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.

4.3.2 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 4.7 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-

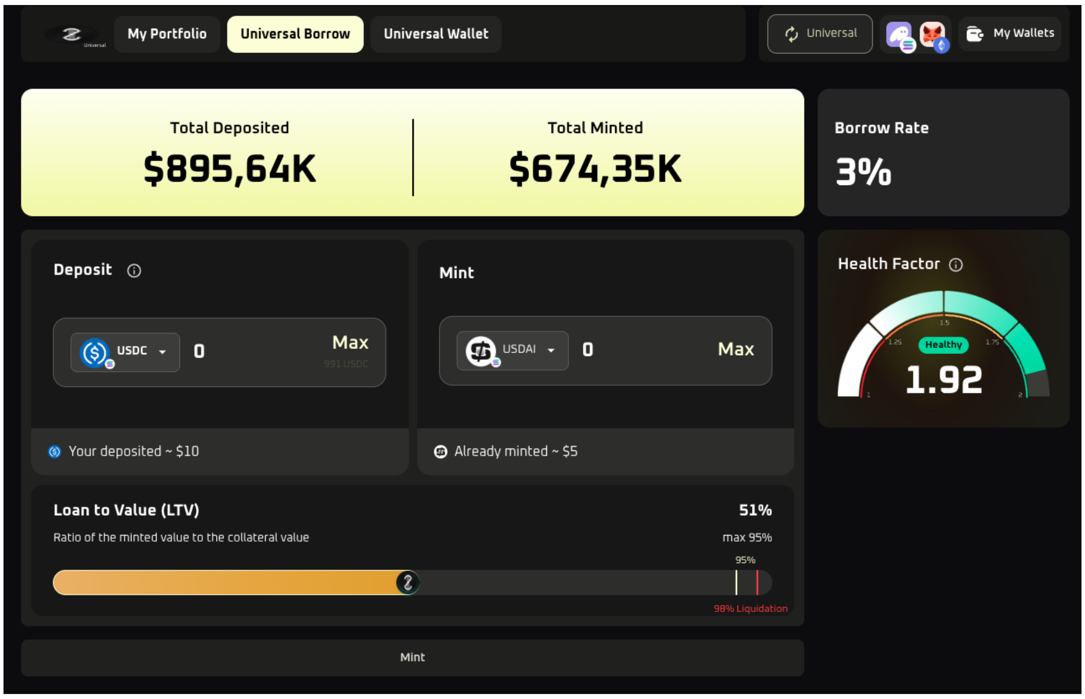


Figure 4.7: Cross-chain Borrowing and Position Management Interface

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

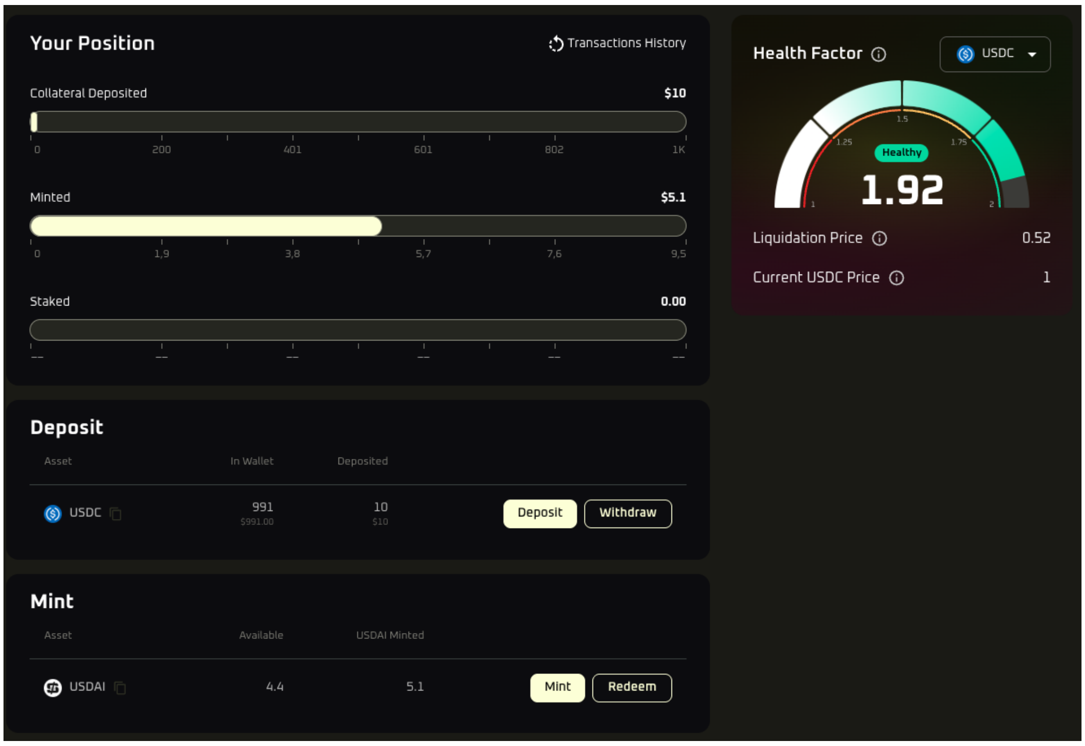
To mitigate the risk of liquidation, the interface incorporates advanced visualization tools for risk management.

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

4.3.3 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 4.8 displays the detailed layout of this section.



Your Position

Collateral Deposited: $10

Minted: $5.1

Staked: 0.00

Deposit

Asset	In Wallet	Deposited
USDC	991 ($991.00)	10 ($10)

Mint

Asset	Available	USDAI Minted
USDAI	4.4	5.1

Health Factor: 1.92 (Healthy)

Liquidation Price: 0.52

Current USDC Price: 1

Figure 4.8: 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.

4.4 Application Building

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

4.4.1 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`

Table 4.1: 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/`

Table 4.2: 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/`

Table 4.3: 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/`

Table 4.4: 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/`

Table 4.5: Testing and Simulation Tools

4.4.2 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 4.6 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**	

Table 4.6: Source Code Statistics by Component

4.5 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.

4.5.1 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.

4.5.2 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

Table 4.7: 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

Table 4.8: 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: $2000 \to 1600$	**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

Table 4.9: Test Cases for Liquidation Logic

4.5.3 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.

4.6 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.)

CHAPTER 5. SOLUTION AND CONTRIBUTION

Chương này có độ dài tối thiểu 5 trang, tối đa không giới hạn.[1] Sinh viên cần trình bày tất cả những nội dung đóng góp mà mình thấy tâm đắc nhất trong suốt quá trình làm ĐATN. Đó có thể là một loạt các vấn đề khó khăn mà sinh viên đã từng bước giải quyết được, là giải thuật cho một bài toán cụ thể, là giải pháp tổng quát cho một lớp bài toán, hoặc là mô hình/kiến trúc hữu hiệu nào đó được sinh viên thiết kế.

Chương này **là cơ sở quan trọng** để các thầy cô đánh giá sinh viên. Vì vậy, sinh viên cần phát huy tính sáng tạo, khả năng phân tích, phản biện, lập luận, tổng quát hóa vấn đề và tập trung viết cho thật tốt. Mỗi giải pháp hoặc đóng góp của sinh viên cần được trình bày trong một mục độc lập bao gồm ba mục con: (i) dẫn dắt/giới thiệu về bài toán/vấn đề, (ii) giải pháp, và (iii) kết quả đạt được (nếu có).

Sinh viên lưu ý **không trình bày lặp lại nội dung**. Những nội dung đã trình bày chi tiết trong các chương trước không được trình bày lại trong chương này. Vì vậy, với nội dung hay, mang tính đóng góp/giải pháp, sinh viên chỉ nên tóm lược/mô tả sơ bộ trong các chương trước, đồng thời tạo tham chiếu chéo tới đề mục tương ứng trong Chương 5 này. Chi tiết thông tin về đóng góp/giải pháp được trình bày trong mục đó.

Ví dụ, trong Chương 4, sinh viên có thiết kế được kiến trúc đáng lưu ý gì đó, là sự kết hợp của các kiến trúc MVC, MVP, SOA, v.v. Khi đó, sinh viên sẽ chỉ mô tả ngắn gọn kiến trúc đó ở Chương 4, rồi thêm các câu có dạng: “Chi tiết về kiến trúc này sẽ được trình bày trong phần 5.1".

[1]Trong trường hợp phần này dưới 5 trang thì sinh viên nên gộp vào phần kết luận, không tách ra một chương riêng rẽ nữa.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

Sinh viên so sánh kết quả nghiên cứu hoặc sản phẩm của mình với các nghiên cứu hoặc sản phẩm tương tự.

Sinh viên phân tích trong suốt quá trình thực hiện ĐATN, mình đã làm được gì, chưa làm được gì, các đóng góp nổi bật là gì, và tổng hợp những bài học kinh nghiệm rút ra nếu có.

6.2 Future work

Trong phần này, sinh viên trình bày định hướng công việc trong tương lai để hoàn thiện sản phẩm hoặc nghiên cứu của mình.

Trước tiên, sinh viên trình bày các công việc cần thiết để hoàn thiện các chức năng/nhiệm vụ đã làm. Sau đó sinh viên phân tích các hướng đi mới cho phép cải thiện và nâng cấp các chức năng/nhiệm vụ đã làm.

SHORT NOTICES ON REFERENCE

Lưu ý: Sinh viên không được đưa bài giảng/slide, các trang Wikipedia, hoặc các trang web thông thường làm tài liệu tham khảo.

Một trang web được phép dùng làm tài liệu tham khảo **chỉ khi** nó là công bố chính thống của cá nhân hoặc tổ chức nào đó. Ví dụ, trang web đặc tả ngôn ngữ XML của tổ chức W3C `https://www.w3.org/TR/2008/REC-xml-20081126/` là TLTK hợp lệ.

Có năm loại tài liệu tham khảo mà sinh viên phải tuân thủ đúng quy định về cách thức liệt kê thông tin như sau. Lưu ý: các phần văn bản trong cặp dấu < > dưới đây chỉ là hướng dẫn khai báo cho từng loại tài liệu tham khảo; sinh viên cần xóa các phần văn bản này trong ĐATN của mình.

<**Bài báo đăng trên tạp chí khoa học**: Tên tác giả, tên bài báo, tên tạp chí, volume, từ trang đến trang (nếu có), nhà xuất bản, năm xuất bản >

hovy1993automated E. H. Hovy, “Automated discourse generation using discourse structure rela- tions," *Artificial intelligence*, vol. 63, no. 1-2, pp. 341–385, 1993

<**Sách**: Tên tác giả, tên sách, volume (nếu có), lần tái bản (nếu có), nhà xuất bản, năm xuất bản>

peterson2007computer L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*. Elsevier, 2007.

NguyenThucHai N. T. Hải, *Mạng máy tính và các hệ thống mở*. Nhà xuất bản giáo dục, 1999.

<**Tập san Báo cáo Hội nghị Khoa học**: Tên tác giả, tên báo cáo, tên hội nghị, ngày (nếu có), địa điểm hội nghị, năm xuất bản>

poesio2001discourse M. Poesio and B. Di Eugenio, “Discourse structure and anaphoric accessibil- ity," in *ESSLLI workshop on information structure, discourse structure and discourse semantics*, Copenhagen, Denmark, 2001, pp. 129–143.

<**Đồ án tốt nghiệp, Luận văn Thạc sĩ, Tiến sĩ**: Tên tác giả, tên đồ án/luận văn, loại đồ án/luận văn, tên trường, địa điểm, năm xuất bản>

knott1996data A. Knott, “A data-driven methodology for motivating a set of coherence relations," Ph.D. dissertation, The University of Edinburgh, UK, 1996.

<**Tài liệu tham khảo từ Internet**: Tên tác giả (nếu có), tựa đề, cơ quan (nếu

có), địa chỉ trang web, thời gian lần cuối truy cập trang web>

BernersTim T. Berners-Lee, *Hypertext transfer protocol (HTTP)*. [Online]. Available: `ftp:/info.cern.ch/pub/www/doc/http-spec.txt.Z` (visited on 09/30/2010).

LectureA Princeton University, *Wordnet*. [Online]. Available: `http://www.cogsci.princeton.edu/~wn/index.shtml` (visited on 09/30/2010).

REFERENCE

[1] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, `https://bitcoin.org/bitcoin.pdf`, Accessed: 2024, 2008.

[2] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, `https://ethereum.org/en/whitepaper/`, 2014.

[3] A. Yakovenko, *Solana: A new architecture for a high performance blockchain v0.8.13*, `https://solana.com/solana-whitepaper.pdf`, Whitepaper, 2018.

[4] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[5] A. Ferrante and Coral Team, *Anchor framework: A framework for solana's sealevel runtime*, `https://www.anchor-lang.com/`, Documentation, 2021.

[6] OpenZeppelin, *Openzeppelin contracts: The standard for secure blockchain applications*, `https://www.openzeppelin.com/contracts`, Library Documentation, 2023.

APPENDIX

A. THESIS WRITING GUIDELINE

A.1 General Regulations

Dưới đây là một số quy định và hướng dẫn viết đồ án tốt nghiệp mà bắt buộc sinh viên phải đọc kỹ và tuân thủ nghiêm ngặt.

Sinh viên cần đảm bảo tính thống nhất toàn báo cáo (font chữ, căn dòng hai bên, hình ảnh, bảng, margin trang, đánh số trang, v.v.). Để làm được như vậy, sinh viên chỉ cần sử dụng các định dạng theo đúng template ĐATN này. Khi paste nội dung văn bản từ tài liệu khác của mình, sinh viên cần chọn kiểu Copy là "Text Only" để định dạng văn bản của template không bị phá vỡ/vi phạm.

Tuyệt đối cấm sinh viên đạo văn. Sinh viên cần ghi rõ nguồn cho tất cả những gì không tự mình viết/vẽ lên, bao gồm các câu trích dẫn, các hình ảnh, bảng biểu, v.v. Khi bị phát hiện, sinh viên sẽ không được phép bảo vệ ĐATN.

Tất cả các hình vẽ, bảng biểu, công thức, và tài liệu tham khảo trong ĐATN nhất thiết phải được SV giải thích và tham chiếu tới ít nhất một lần. Không chấp nhận các trường hợp sinh viên đưa ra hình ảnh, bảng biểu tùy hứng và không có lời mô tả/giải thích nào.

Sinh viên tuyệt đối không trình bày ĐATN theo kiểu viết ý hoặc gạch đầu dòng. ĐATN không phải là một slide thuyết trình; khi người đọc không hiểu sẽ không có ai giải thích hộ. Sinh viên cần viết thành các đoạn văn và phân tích, diễn giải đầy đủ, rõ ràng. Câu văn cần đúng ngữ pháp, đầy đủ chủ ngữ, vị ngữ và các thành phần câu. Khi thực sự cần liệt kê, sinh viên nên liệt kê theo phong cách khoa học với các ký tự La Mã. Ví dụ, nhiều sinh viên luôn cảm thấy hối hận vì (i) chưa cố gắng hết mình, (ii) chưa sắp xếp thời gian học/chơi một cách hợp lý, (iii) chưa tìm được người yêu để chia sẻ quãng đời sinh viên vất vả, và (iv) viết ĐATN một cách cẩu thả.

Trong một số trường hợp nhất thiết phải dùng các bullet để liệt kê, sinh viên cần thống nhất Style cho toàn bộ các bullet các cấp mà mình sử dụng đến trong báo cáo. Nếu dùng bullet cấp 1 là hình tròn đen, toàn bộ báo cáo cần thống nhất cách dùng như vậy; ví dụ như sau:

- Đây là mục 1 – Thực sự không còn cách nào khác tôi mới dùng đến việc bullet trong báo cáo.
- Đây là mục 2 – Nghĩ lại thì tôi có thể không cần dùng bullet cũng được. Nên tôi sẽ xóa bullet và tổ chức lại hai mục này trong báo cáo của mình cho khoa học hơn. Tôi muốn thầy cô và người đọc cảm nhận được tâm huyết của tôi

trong từng trang báo cáo ĐATN.

A.2 Majoring

Sinh viên lưu ý viết đúng ngành/chuyên ngành trên bìa và trên gáy theo đúng quy định của Trường. Ngành học hay chuyên ngành học phụ thuộc vào ngành học mà sinh viên đăng ký. Sinh viên có thể đăng nhập trên trang quản lý học tập của mình để xem lại chính xác ngành học của mình.

Một số ví dụ sinh viên có thể tham khảo dưới đây, trong trường hợp có chuyên ngành thì sinh viên không cần ghi chuyên ngành:

- Đối với kỹ sư chính quy:
  - Từ K61 trở về trước: Ngành Kỹ thuật phần mềm
  - Từ K62 trở về sau: Ngành Khoa học máy tính
- Đối với cử nhân:
  - Ngành Công nghệ thông tin
- Đối với chương trình EliteTech:
  - Chương trình Việt Nhật/KSTN: Ngành Công nghệ thông tin
  - Chương trình ICT Global: Ngành Information Technology
  - Chương trình DS&AI: Ngành Khoa học dữ liệu

A.3 Bulleting and Numbering

Việc sử dụng danh sách trong LaTeX khá đơn giản và không yêu cầu sinh viên phải thêm bất kỳ gói bổ sung nào. LaTeX cung cấp hai môi trường liệt kê đó là:

- Đánh dấu (bullet) là kiểu liệt kê không có thứ tự. Để sử dụng kiểu liệt kê đánh dấu, chúng ta khai báo như sau
  ```
  \begin{itemize}
  \item Nội dung thứ nhất được viết ở đây.
  \item Nội dung thứ hai được viết ở đây.
  \item ...
  \end{itemize}
  ```
- Đánh số (numering) là kiểu liệt kê có thứ tự. Để sử dụng kiểu liệt kê đánh số, chúng ta khai báo như sau
  ```
  \begin{enumerate}
  \item Nội dung thứ nhất được viết ở đây.
  \item Nội dung thứ hai được viết ở đây.
  \item ...
  ```

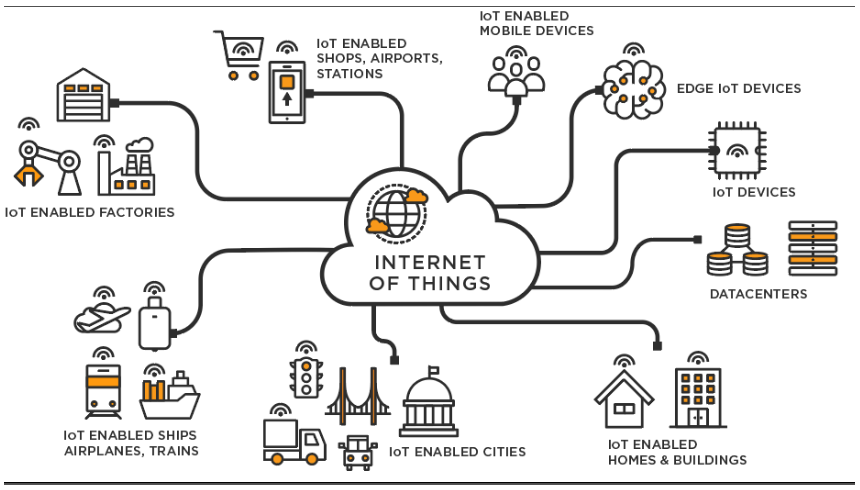

IoT ENABLED SHOPS, AIRPORTS, STATIONS
IoT ENABLED MOBILE DEVICES
EDGE IoT DEVICES
IoT DEVICES
IoT ENABLED FACTORIES
INTERNET OF THINGS
DATACENTERS
IoT ENABLED SHIPS AIRPLANES, TRAINS
IoT ENABLED CITIES
IoT ENABLED HOMES & BUILDINGS

Figure A.1: Internet vạn vật

```
\end{enumerate}
```

Chú ý các nội dung trình bày trong cả hai môi trường liệt kê theo sau lệnh `\item`. Ngoài ra LaTeX còn cung cấp một số kiểu liệt kê khác, sinh viên có thể tham khảo tại `https://www.overleaf.com/learn/latex/Lists`

A.4 Table insertion

Col1	Col2	Col2	Col3
1	6	87837	787
2	7	78	5415
3	545	778	7507
4	545	18744	7560
5	88	788	6344

Table A.1: Table to test captions and labels.

Bảng A.1 là ví dụ về cách tạo bảng. Tất cả các bảng biểu phải được đề cập đến trong phần nội dung và phải được phân tích và bình luận. Chú ý: Tạo bảng trong Latex khá phức tạp và mất thời gian, vì vậy sinh viên có thể sử dụng các công cụ hỗ trợ tạo bảng (Ví dụ: `https://www.tablesgenerator.com/`). Sinh viên có thể tìm hiểu sâu hơn về cách chèn ảnh trong Latex tại link `https://www.overleaf.com/learn/latex/Tables`.

A.5 Figure Insertion

Hình A.1 là ví dụ về cách chèn ảnh. Lưu ý chú thích của hình vẽ được đặt ngay dưới hình vẽ. Sinh viên có thể tìm hiểu sâu hơn về cách chèn ảnh trong Latex tại `https://www.overleaf.com/learn/latex/Inserting_Images`.

Chú ý, tất cả các hình vẽ phải được đề cập đến trong phần nội dung và phải được phân tích và bình luận.

A.6 Reference

Listing

Áp dụng cách liệt kê theo quy định của IEEE. Ví dụ của việc trích dẫn như sau **scott2013sdn**. Cụ thể, sinh viên sử dụng lệnh `\cite{}` như sau **ashton2009internet**. Chỉ những tài liệu được trích dẫn thì mới xuất hiện trong phần Tài liệu tham khảo. Tài liệu tham khảo cần có nguồn gốc rõ ràng và phải từ nguồn đáng tin cậy. Hạn chế trích dẫn tài liệu tham khảo từ các website, từ wikipedia.

Types of Reference

Các nguồn tài liệu tham khảo chính là sách, bài báo trong các tạp chí, bài báo trong các hội nghị khoa học và các tài liệu tham khảo khác trên internet.

A.7 Equations

Các gói amsmath, amssymb, amsfonts hỗ trợ viết phương trình/công thức toán học đã được bổ sung sẵn ở phần đầu của file main.tex. Một ví dụ về tạo phương trình (A.1) như sau

$$F(x) = \int_b^a \frac{1}{3}x^3 \tag{A.1}$$

Phương trình A.1 là ví dụ về phương trình tích phân. Một phương trình khác không được đánh số thứ tự (gán nhãn)

$$x[t_n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X[f_k] e^{j2\pi nk/N}$$

Phương trình này thể hiện phép biến đổi Fourier rời rạc ngược (IDFT).

A.8 Qui cách đóng quyển

Phần bìa trước chế bản theo qui định; bìa trước và bìa sau là giấy liền khổ. Sử dụng keo nhiệt để dán gáy khi đóng quyển thay vì sử dụng băng dính và dập ghim như mô tả ở Figure A.3 Phần gáy ĐATN cần ghi các thông tin tóm tắt sau: Kỳ làm ĐATN - Ngành đào tạo - Họ và tên sinh viên - Mã số sinh viên. Ví dụ:

2022.1 - KỸ THUẬT MÁY TÍNH - NGUYỄN VĂN A - 20221234

Qui cách ghi chữ phần gáy như hình dưới đây:

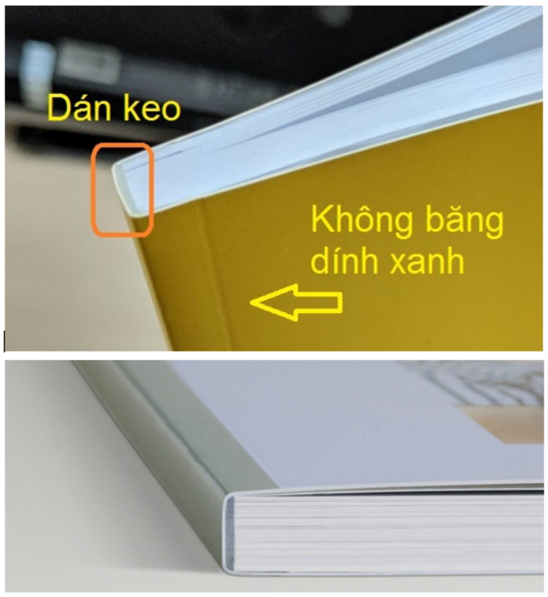


Figure A.2: Qui cách đóng quyển đồ án

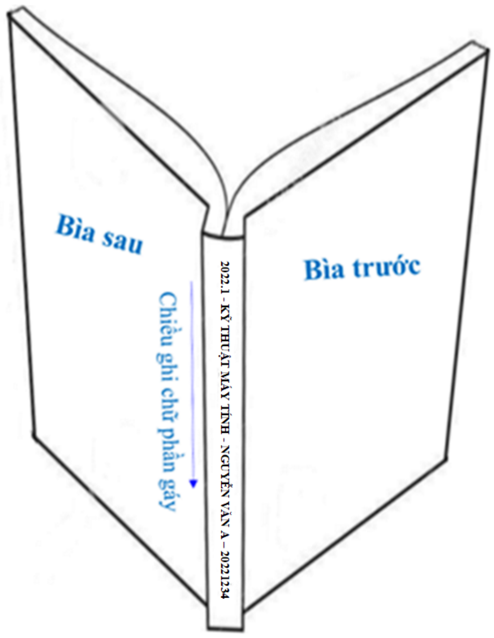


Figure A.3: Qui cách đóng quyển đồ án

B. USE CASE DESCRIPTIONS

Nếu trong nội dung chính không đủ không gian cho các use case khác (ngoài các use case nghiệp vụ chính) thì đặc tả thêm cho các use case đó ở đây.

B.1 Đặc tả use case “Thống kê tình hình mượn sách”

. . .

B.2 Đặc tả use case “Đăng ký làm thẻ mượn”

. . .