Neo Service Layer Whitepaper

R3E Network

March 1, 2025

Contents

1 Neo Service Layer

1.1 Introduction

The Neo Service Layer represents a critical advancement in the Neo ecosystem, providing a comprehensive suite of services designed to enhance the capabilities of Neo Smart Contracts and improve the user experience. As blockchain technology continues to mature, the need for specialized services that bridge the gap between on-chain logic and off-chain resources becomes increasingly important. The Neo Service Layer addresses this need by offering a robust infrastructure that enables developers to build more powerful and versatile decentralized applications.

Unlike traditional Function-as-a-Service (FaaS) platforms that focus primarily on serverless computing, the Neo Service Layer is specifically designed for blockchain integration, with a particular emphasis on Neo N3's unique capabilities and requirements. This specialized approach ensures optimal performance, security, and compatibility with the Neo ecosystem, while providing developers with the tools they need to create innovative applications that leverage both on-chain and off-chain resources.

The Neo Service Layer is built on several core principles that guide its design and implementation. At its foundation is **Seamless Integration**, providing native compatibility with Neo N3 smart contracts and ecosystem components to ensure a cohesive developer experience. The layer delivers **Enhanced Functionality** by extending on-chain capabilities through secure off-chain services, enabling more complex and powerful applications. **Developer Accessibility** is prioritized through simplified interfaces that abstract away underlying complexity, allowing developers to focus on building applications rather than managing infrastructure details. **Security by Design** is implemented with comprehensive measures at every layer, protecting both user data and application integrity. The architecture emphasizes **Scalability**, with the ability to handle growing demands without compromising performance as adoption increases. Finally, **Interoperability** is built into the core design, supporting cross-chain and external system interactions to create a more connected blockchain ecosystem.

This section explores the architecture, components, and capabilities of the Neo Service Layer, demonstrating how it enhances the Neo ecosystem and enables new classes of decentralized applications.

1.2 Architectural Overview

The Neo Service Layer is designed with a modular architecture that provides both flexibility and robustness. This architecture consists of several key components that work together to deliver a comprehensive service infrastructure for Neo Smart Contracts and users.

The architecture consists of several interconnected layers that work together to provide a comprehensive service infrastructure. At the foundation is the **Neo N3 Blockchain**, which provides the secure, deterministic execution environment for smart contracts and digital assets. Above this sits the **Integration Layer**, which connects the Neo blockchain with the Service Layer by handling event monitoring, transaction management, and state synchronization. The **Service Layer** forms the core of the Neo Service Layer, providing specialized services such as Gas Bank, Meta Transactions, Oracle Services, and Trusted Execution Environments. The **API Layer** exposes service functionalities through standardized interfaces, including RESTful APIs, GraphQL endpoints, and WebSocket connections, making the services accessible to developers. At the top level are the **Applications**, user-facing software that leverages the Neo Service Layer to provide enhanced functionality and user experiences.

This layered approach ensures clear separation of concerns while enabling seamless integration between components. Each layer implements specific security controls, creating a defense-in-depth model that protects against threats at multiple levels while maintaining the performance characteristics necessary for commercial applications.

1.3 Core Services

The Neo Service Layer provides several core services that enhance the capabilities of Neo Smart Contracts and improve the user experience. These services address common challenges in blockchain application development and enable new use cases that would be difficult or impossible to implement using on-chain logic alone.

1.3.1 Gas Bank

The Gas Bank service provides a solution to one of the most significant barriers to blockchain adoption: the requirement for users to hold native tokens (GAS) to pay for transaction fees. This requirement creates friction for new users and limits the adoption of decentralized applications.

The Gas Bank allows application developers to create accounts that can

pay for transaction fees on behalf of their users, enabling a seamless user experience similar to traditional web applications. Users can interact with decentralized applications without needing to acquire GAS tokens first, removing a significant barrier to entry.

Definition 1 (Gas Bank Account) *A Gas Bank Account is a managed account that holds GAS tokens and can be used to pay for transaction fees on behalf of users. Each account has the following properties:*

- *Address: The unique identifier of the account*
- *Balance: The amount of GAS tokens available for fee payment*
- *Fee Model: The fee calculation model (fixed, percentage, dynamic, or free)*
- *Credit Limit: The maximum amount of credit that can be extended to the account*
- *Used Credit: The amount of credit currently used*
- *Status: The current status of the account (active, suspended, etc.)*

The Gas Bank supports multiple fee models to accommodate different application requirements. The **Fixed Fee** model applies a constant fee regardless of the transaction size or complexity, providing predictability for users and developers. The **Percentage Fee** model calculates fees as a percentage of the transaction amount, which is particularly useful for value transfer operations where the fee scales with the transaction value. For applications requiring more adaptive pricing, the **Dynamic Fee** model varies based on network conditions and transaction complexity, optimizing for both cost and confirmation time. Some applications may choose the **Free** model where no fee is charged to the user, with all costs absorbed by the application to provide a seamless user experience similar to traditional web applications.

The Gas Bank service also provides credit facilities, allowing applications to continue operating even when their Gas Bank account balance is temporarily depleted. This feature ensures uninterrupted service for users while giving application developers time to replenish their accounts.

Deposit and Withdrawal Operations The Gas Bank implements a comprehensive system for managing deposits and withdrawals. **User Deposits** allow individuals to deposit GAS tokens into the Gas Bank through direct blockchain transactions or application interfaces, with each deposit

verified on-chain and credited to the appropriate account. For developers, **Application Deposits** provide a way to fund their Gas Bank accounts through direct deposits or automated funding mechanisms, ensuring sufficient balance for covering user transaction fees. The **Withdrawal Process** enables account owners to withdraw their GAS tokens through a secure process that includes verification of ownership, balance checks, and transaction confirmation to prevent unauthorized access. To maintain operational continuity, **Automated Refills** can be configured by applications to trigger deposits from designated funding sources when account balances fall below specified thresholds, preventing service interruptions due to insufficient funds.

A formal specification of the Gas Bank operations protocol is provided in Section 1.23.

1.4 Gas Bank Operations Protocol

The Gas Bank Operations Protocol defines how users deposit and withdraw GAS tokens and how the Gas Bank manages transaction fees.

Protocol 1 (Gas Bank Deposit)

- ***Require:*** *User U, Gas Bank Account A, Amount amt, Blockchain B, Gas Bank Service G*
- ***Ensure:*** *Funds are securely deposited and credited to the account*
- ***Account Verification:***
- $exists \leftarrow G.accountExists(A.address)$
- ***If*** $\neg exists$
- ***return*** $\{error : "Account\ does\ not\ exist"\}$
- ***EndIf***
- ***Transaction Creation:***
- $tx \leftarrow B.createTransaction(\{from : U.address, to : G.depositAddress, amount : amt, data : A.address\})$
- $signedTx \leftarrow U.signTransaction(tx)$
- ***Transaction Submission:***
- $txHash \leftarrow B.submitTransaction(signedTx)$
- ***Transaction Monitoring:***
- $confirmed \leftarrow false$

- ***While*** $\neg confirmed$
- $receipt \leftarrow B.getTransactionReceipt(txHash)$
- ***If*** $receipt \neq null$
- ***If*** $receipt.status = \text{"success"}$
- $confirmed \leftarrow true$
- ***ElsIf*** $receipt.status = \text{"failed"}$
- ***return*** $\{error : \text{"Transaction failed"}, receipt : receipt\}$
- ***EndIf***
- ***EndIf***
- ***If*** $\neg confirmed$
- $Sleep(1\ second)$
- ***EndIf***
- ***EndWhile***
- ***Account Update:***
- $A.balance \leftarrow A.balance + amt$
- $G.updateAccount(A)$
- ***Event Emission:***
- $G.emitEvent(\text{"Deposit"}, \{account : A.address, amount : amt, txHash : txHash\})$
- ***return*** $\{success : true, account : A.address, amount : amt, txHash : txHash\}$

Protocol 2 (Gas Bank Withdrawal)

- ***Require:*** *User U, Gas Bank Account A, Amount amt, Blockchain B, Gas Bank Service G*
- ***Ensure:*** *Funds are securely withdrawn and transferred to the user*
- ***Account Verification:***
- $exists \leftarrow G.accountExists(A.address)$
- ***If*** $\neg exists$
- ***return*** $\{error : \text{"Account does not exist"}\}$
- ***EndIf***
- ***Authorization Check:***

- $isOwner \leftarrow G.isAccountOwner(U.address, A.address)$
- ***If*** $\neg isOwner$
- ***return*** $\{error : \text{"Unauthorized withdrawal attempt"}\}$
- ***EndIf***
- ***Balance Check:***
- ***If*** $A.balance < amt$
- ***return*** $\{error : \text{"Insufficient balance"}\}$
- ***EndIf***
- ***Withdrawal Processing:***
- $A.balance \leftarrow A.balance - amt$
- $G.updateAccount(A)$
- ***Transaction Creation:***
- $tx \leftarrow B.createTransaction(\{from : G.withdrawalAddress, to : U.address, amount : amt\})$
- $signedTx \leftarrow G.signTransaction(tx)$
- ***Transaction Submission:***
- $txHash \leftarrow B.submitTransaction(signedTx)$
- ***Transaction Monitoring:***
- $confirmed \leftarrow false$
- ***While*** $\neg confirmed$
- $receipt \leftarrow B.getTransactionReceipt(txHash)$
- ***If*** $receipt \neq null$
- ***If*** $receipt.status = \text{"success"}$
- $confirmed \leftarrow true$
- ***ElsIf*** $receipt.status = \text{"failed"}$
- $A.balance \leftarrow A.balance + amt$ *// Revert balance change*
- $G.updateAccount(A)$
- ***return*** $\{error : \text{"Transaction failed"}, receipt : receipt\}$
- ***EndIf***
- ***EndIf***
- ***If*** $\neg confirmed$

- *Sleep*(1 *second*)
- ***EndIf***
- ***EndWhile***
- ***Event Emission:***
- $G.emitEvent(\text{"Withdrawal"}, \{account : A.address, amount : amt, txHash : txHash\})$
- ***return*** $\{success : true, account : A.address, amount : amt, txHash : txHash\}$

Protocol 3 (Fee Payment)

- ***Require:*** *Application* P*, Gas Bank Account* A*, Transaction* T*, Blockchain* B*, Gas Bank Service* G
- ***Ensure:*** *Transaction fees are properly calculated and paid*
- ***Account Verification:***
- $exists \leftarrow G.accountExists(A.address)$
- ***If*** $\neg exists$
- ***return*** $\{error : \text{"Account does not exist"}\}$
- ***EndIf***
- ***Authorization Check:***
- $isAuthorized \leftarrow G.isApplicationAuthorized(P.id, A.address)$
- ***If*** $\neg isAuthorized$
- ***return*** $\{error : \text{"Application not authorized for this account"}\}$
- ***EndIf***
- ***Fee Calculation:***
- $feeModel \leftarrow A.feeModel$
- $estimatedFee \leftarrow B.estimateTransactionFee(T)$
- $fee \leftarrow CalculateFee(feeModel, estimatedFee, T)$
- ***Balance Check:***
- ***If*** $A.balance + A.creditLimit - A.usedCredit < fee$
- ***return*** $\{error : \text{"Insufficient balance and credit"}\}$
- ***EndIf***
- ***Fee Reservation:***

- ***If*** $A.balance \geq fee$
- $A.balance \leftarrow A.balance - fee$
- ***Else***
- $fromBalance \leftarrow A.balance$
- $fromCredit \leftarrow fee - fromBalance$
- $A.balance \leftarrow 0$
- $A.usedCredit \leftarrow A.usedCredit + fromCredit$
- ***EndIf***
- $G.updateAccount(A)$
- ***Transaction Submission:***
- $txHash \leftarrow B.submitTransaction(T)$
- ***Transaction Monitoring:***
- $confirmed \leftarrow false$
- ***While*** $\neg confirmed$
- $receipt \leftarrow B.getTransactionReceipt(txHash)$
- ***If*** $receipt \neq null$
- $confirmed \leftarrow true$
- $actualFee \leftarrow receipt.gasUsed \times receipt.gasPrice$
- ***If*** $actualFee < fee$
- $refund \leftarrow fee - actualFee$
- ***If*** $A.usedCredit > 0$
- $creditRefund \leftarrow \min(refund, A.usedCredit)$
- $A.usedCredit \leftarrow A.usedCredit - creditRefund$
- $refund \leftarrow refund - creditRefund$
- ***EndIf***
- ***If*** $refund > 0$
- $A.balance \leftarrow A.balance + refund$
- ***EndIf***
- $G.updateAccount(A)$
- ***EndIf***
- ***EndIf***

- ***If*** $\neg confirmed$
- $Sleep(1\ second)$
- ***EndIf***
- ***EndWhile***
- ***Event Emission:***
- $G.emitEvent(\text{"FeePaid"}, \{account : A.address, amount : actualFee, txHash : txHash\})$
- ***return*** $\{success : true, account : A.address, fee : actualFee, txHash : txHash\}$

Fee Calculation and Payment When a user interacts with an application that uses the Gas Bank:

1. The application requests fee payment from the Gas Bank service.
2. The Gas Bank calculates the required fee based on the configured fee model.
3. The Gas Bank verifies that the application's account has sufficient balance.
4. If sufficient balance exists, the Gas Bank authorizes the transaction and reserves the fee amount.
5. Once the transaction is confirmed on the blockchain, the Gas Bank deducts the actual fee from the reserved amount and releases any excess.
6. The application receives confirmation of the fee payment and can update its records accordingly.

Multi-Chain Support The Gas Bank service supports multiple blockchain networks to provide a unified fee management solution across different ecosystems. It offers **Neo N3** native support with comprehensive GAS token fee management tailored to the Neo blockchain's specific requirements. For projects operating on Ethereum or EVM-compatible chains, the service provides **Ethereum** support with ETH fee management capabilities, allowing developers to use the same service interface across different blockchain environments. The system also enables **Cross-Chain Operations** with the ability to manage fees across different blockchain

networks through integrated bridges and conversion mechanisms, creating a seamless experience for applications that operate in multi-chain environments.

This multi-chain capability enables applications to provide a consistent user experience across different blockchain ecosystems while centralizing fee management in a single service.

1.4.1 Meta Transaction Service

Meta transactions enable users to interact with smart contracts without directly paying for transaction fees, further reducing the barriers to blockchain adoption. The Meta Transaction Service acts as a relay that receives signed transaction requests from users, verifies their validity, and submits them to the blockchain on behalf of the users.

Definition 2 (Meta Transaction) *A Meta Transaction is a transaction request that is signed by a user but submitted to the blockchain by a relayer. The relayer pays for the transaction fees, while the transaction itself is executed as if it were submitted by the original user.*

The Meta Transaction Service supports multiple blockchain types and signature curves, enabling interoperability between Neo N3 and other blockchain ecosystems. It provides full support for **Neo N3 with secp256r1** curve for standard Neo N3 transactions and signatures, ensuring compatibility with the Neo ecosystem. Additionally, it supports **Ethereum with secp256k1** curve for Ethereum-compatible transactions and signatures, including EIP-712 typed data signing, which enables secure structured data signing for enhanced security and user experience.

This cross-chain compatibility is particularly valuable for applications that operate across multiple blockchain ecosystems, as it allows users to interact with contracts on different chains using a single signature mechanism.

The Meta Transaction Service integrates with the Gas Bank to pay for transaction fees, creating a comprehensive solution for fee abstraction. Application developers can configure the service to use specific Gas Bank accounts for different contracts or transaction types, enabling fine-grained control over fee allocation.

Meta Transaction Workflow The complete workflow for processing a meta transaction involves several steps:

1. **Transaction Creation**: The user creates a transaction request containing the target contract, method, parameters, and any other required information.

2. **Signature Generation**: The user signs the transaction request using their private key. For Neo N3, this uses the secp256r1 curve, while for Ethereum-compatible transactions, secp256k1 is used with EIP-712 typed data signing.

3. **Relay Submission**: The signed transaction request is submitted to the Meta Transaction Service through its API.

4. **Signature Verification**: The service verifies the signature to ensure the request was genuinely signed by the claimed user.

5. **Fee Estimation**: The service estimates the required transaction fee based on the current network conditions and transaction complexity.

6. **Gas Bank Integration**: The service requests fee payment from the Gas Bank service, specifying the application account that should cover the cost.

7. **Transaction Submission**: Upon successful fee authorization, the service constructs and submits the transaction to the blockchain network.

8. **Confirmation Monitoring**: The service monitors the transaction until it is confirmed on the blockchain.

9. **Result Notification**: The service notifies the application of the transaction result, including success status, transaction hash, and any return values.

A formal specification of the Meta Transaction protocol is provided in Section 1.24.

1.5 Meta Transaction Protocol

The Meta Transaction Protocol defines how users can interact with smart contracts without directly paying for transaction fees.

Protocol 4 (Meta Transaction Processing)

- ***Require:*** *User U, Transaction request R, Meta Transaction Service M, Gas Bank Service G, Blockchain B*
- ***Ensure:*** *Transaction is executed on behalf of the user without requiring user to pay fees*
- ***Transaction Creation:***

- $txData \leftarrow \{to : R.to, data : R.data, value : R.value, nonce : U.nonce, chainId : B.chainId\}$
- ***Signature Generation:***
- $message \leftarrow FormatMessage(txData)$
- $signature \leftarrow U.sign(message)$
- ***Meta Transaction Request:***
- $metaTx \leftarrow \{txData : txData, signature : signature, from : U.address\}$
- $M.submitTransaction(metaTx)$
- ***Signature Verification:***
- $recoveredAddress \leftarrow RecoverSigner(message, signature)$
- ***If*** $recoveredAddress \neq U.address$
- ***return*** $\{error : "Invalid\ signature"\}$
- ***EndIf***
- ***Nonce Verification:***
- $expectedNonce \leftarrow M.getNonce(U.address)$
- ***If*** $txData.nonce \neq expectedNonce$
- ***return*** $\{error : "Invalid\ nonce"\}$
- ***EndIf***
- ***Fee Estimation:***
- $gasLimit \leftarrow B.estimateGas(txData)$
- $gasPrice \leftarrow B.getGasPrice()$
- $fee \leftarrow gasLimit \times gasPrice$
- ***Gas Bank Integration:***
- $app \leftarrow M.getApplicationForContract(txData.to)$
- $account \leftarrow G.getAccountForApplication(app.id)$
- $feeApproval \leftarrow G.requestFeePayment(app, account, fee)$
- ***If*** $\neg feeApproval.approved$
- ***return*** $\{error : "Fee\ payment\ not\ approved", reason : feeApproval.reason\}$
- ***EndIf***
- ***Transaction Construction:***

- $rawTx \leftarrow B.createTransaction(\{from : M.relayerAddress, to : txData.to, data : EncodeWithSender(txData.data, U.address), value : txData.value, gasLimit : gasLimit, gasPrice : gasPrice\})$
- $signedTx \leftarrow M.signTransaction(rawTx)$
- ***Transaction Submission:***
- $txHash \leftarrow B.submitTransaction(signedTx)$
- ***Transaction Monitoring:***
- $confirmed \leftarrow false$
- ***While*** $\neg confirmed$
- $receipt \leftarrow B.getTransactionReceipt(txHash)$
- ***If*** $receipt \neq null$
- $confirmed \leftarrow true$
- ***If*** $receipt.status = "success"$
- $M.incrementNonce(U.address)$
- $G.confirmFeePayment(app, account, receipt.gasUsed \times receipt.gasPrice)$
- ***Else***
- $G.cancelFeePayment(app, account, feeApproval.id)$
- ***EndIf***
- ***EndIf***
- ***If*** $\neg confirmed$
- $Sleep(1\ second)$
- ***EndIf***
- ***EndWhile***
- ***Result Notification:***
- $M.notifyResult(U.address, txHash, receipt)$
- ***return*** $\{success : receipt.status = "success", txHash : txHash, receipt : receipt\}$

Protocol 5 (EIP-712 Signature Verification)

- ***Require:*** *Transaction data D, Signature S, Claimed signer A, Domain separator DS, Type hash TH*
- ***Ensure:*** *Signature is valid and matches the claimed signer*

- ***Type Hash Calculation:***
- $typeHash \leftarrow TH$
- ***Value Hash Calculation:***
- $encodedData \leftarrow EncodeData(D)$
- $valueHash \leftarrow keccak256(encodedData)$
- ***Message Hash Calculation:***
- $message \leftarrow$ "\x19\x01" $\|$ DS $\|$ $keccak256(typeHash \| valueHash)$
- $messageHash \leftarrow keccak256(message)$
- ***Signature Components:***
- $r \leftarrow S[0:32]$
- $s \leftarrow S[32:64]$
- $v \leftarrow S[64]$
- ***Address Recovery:***
- $recoveredAddress \leftarrow ecrecover(messageHash, v, r, s)$
- ***If*** $recoveredAddress = A$
- ***return*** *true*
- ***Else***
- ***return*** *false*
- ***EndIf***

Signature Verification Protocol The Meta Transaction Service implements a robust signature verification protocol that supports multiple signature schemes. For Neo N3 transactions, the service performs **Neo N3 Verification** by verifying secp256r1 signatures against the transaction hash, ensuring the signature matches the claimed sender's public key and maintaining the security model of the Neo blockchain. When handling Ethereum-compatible transactions, the service implements **EIP-712 Verification** using the EIP-712 standard for typed data signing, which provides enhanced security by having users sign structured data rather than raw transaction bytes, making the signing process more transparent and secure for users. For applications that operate across multiple blockchain networks, the service supports **Cross-Chain Verification** by verifying signatures using the appropriate scheme for each chain, enabling seamless cross-chain interactions while maintaining the security guarantees of each underlying blockchain.

Replay Protection To prevent replay attacks, where a valid transaction is maliciously resubmitted, the Meta Transaction Service implements several comprehensive protection mechanisms. The service employs **Nonce Tracking** where each user account has an associated nonce that must be included in the transaction and incremented with each submission, ensuring transactions can only be processed once in the correct order. For time-sensitive operations, **Timestamp Validation** is implemented, where transactions include a timestamp and are rejected if they are too old or too far in the future, creating a time window for valid submissions. To prevent cross-chain replay attacks, **Chain ID Inclusion** ensures that transactions include a chain identifier, preventing the same transaction from being replayed on different blockchain networks. Additionally, the service maintains a **Transaction Hash Registry** of processed transaction hashes to provide an extra layer of protection against duplicate submissions, even in edge cases where other protection mechanisms might not catch a replay attempt.

1.5.1 Oracle Service

Blockchain smart contracts operate in a deterministic environment isolated from external data sources. This isolation ensures security and predictability but limits the types of applications that can be built using smart contracts alone. The Oracle Service bridges this gap by providing a secure and reliable way for smart contracts to access external data.

Definition 3 (Oracle Request) *An Oracle Request is a query from a smart contract to an external data source, processed through the Oracle Service. Each request includes:*

- *Request Type: The category of data being requested (price, random, weather, sports, custom)*
- *Data: The specific parameters of the request*
- *Callback: An optional endpoint to receive the response*
- *Requester: The identity of the entity making the request*

The Oracle Service supports various types of data requests to meet the diverse needs of decentralized applications. It provides **Price Data** with real-time and historical price information for cryptocurrencies, commodities, stocks, and other financial instruments, essential for DeFi applications and financial services. For gaming and lottery applications, it offers **Random**

Numbers that are verifiably random, ensuring fair and unpredictable outcomes that cannot be manipulated. Applications requiring location-specific information can access **Weather Data** with current and forecasted weather conditions for specific locations, enabling weather-dependent smart contracts. For sports betting and related applications, the service delivers **Sports Results** with scores, statistics, and outcomes from sporting events through reliable data feeds. Beyond these standard offerings, the service supports **Custom Data** retrieval from specified APIs or data sources, allowing developers to access virtually any external data needed for their specific use cases.

To ensure the reliability and security of the data provided, the Oracle Service implements several key features that work together to create a robust data delivery system. The service utilizes **Multiple Data Sources**, aggregating data from multiple providers to reduce the risk of manipulation and ensure that no single source can unduly influence the results. For data integrity verification, **Cryptographic Verification** is employed with signed responses that can be verified on-chain, allowing smart contracts to independently confirm the authenticity of the data they receive. The service maintains a **Reputation System** that continuously tracks provider reliability and accuracy, adjusting their influence on aggregated results based on historical performance. When discrepancies or questions arise about provided data, the system offers **Dispute Resolution** mechanisms for challenging and resolving disputed data, ensuring that errors can be identified and corrected in a transparent manner.

The Oracle Service is designed to be extensible, allowing for the addition of new data types and sources as the ecosystem evolves. This flexibility ensures that the service can adapt to the changing needs of decentralized applications and their users.

Price Data Feed System The Price Data Feed is one of the most critical components of the Oracle Service, providing reliable and timely price information for various assets. The system employs **Multi-Source Aggregation** by retrieving price data from multiple sources, including major exchanges (Binance, Coinbase, etc.), aggregation services (CoinGecko, CoinMarketCap), and specialized price feeds, ensuring comprehensive market coverage and resilience against single-source failures. To handle the complexity of asset identification across different platforms, a **Price Index Registry** serves as a sophisticated mapping system that maintains relationships between different symbol representations across various data sources, ensuring consistent identification of assets regardless of the specific notation used by

different providers. Performance optimization is achieved through **Dynamic Caching** which implements an intelligent caching mechanism that balances data freshness with response time, with cache invalidation strategies tailored to different asset types based on their volatility and trading volume. To maintain data quality, **Outlier Detection** algorithms identify and filter out anomalous price data that could result from market manipulation, technical issues, or other irregularities, protecting applications from acting on incorrect information. Each price point is assigned a **Confidence Scoring** based on factors such as source reliability, data recency, and consistency across sources, allowing applications to make informed decisions based on the quality of the price data they receive.

A formal specification of the Price Data Feed protocol is provided in Section 1.26.

1.6 Price Data Feed Protocol

The Price Data Feed Protocol defines how price data is collected, aggregated, and provided to smart contracts and applications.

Protocol 6 (Price Data Collection)

- ***Require:*** *Asset symbol S, Price Provider P, Price Registry R, Cache C*
- ***Ensure:*** *Accurate and up-to-date price data is collected and stored*
- ***Symbol Resolution:***
- $mappings \leftarrow R.getSymbolMappings(S)$
- ***If*** $mappings = \emptyset$
- ***return*** $\{error : "Unknown\ symbol"\}$
- ***EndIf***
- ***Cache Check:***
- $cachedData \leftarrow C.get(S)$
- ***If*** $cachedData \neq null \land now() - cachedData.timestamp < cachedData.ttl$
- ***return*** $cachedData.price$
- ***EndIf***
- ***Source Selection:***
- $sources \leftarrow P.getSourcesForAsset(S)$

- $prices \leftarrow \emptyset$
- ***Parallel Data Retrieval:***
- ***For*** *each* $source \in sources$ ***in parallel***
- $sourceSymbol \leftarrow mappings[source.name]$
- $rawPrice \leftarrow source.fetchPrice(sourceSymbol)$
- ***If*** $rawPrice.success$
- $prices \leftarrow prices \cup \{(source.name, rawPrice.price, rawPrice.timestamp, source.weight)\}$
- ***EndIf***
- ***EndFor***
- ***If*** $|prices| < P.minimumSources$
- ***return*** $\{error : \text{"Insufficient price sources"}, available : |prices|\}$
- ***EndIf***
- ***Outlier Detection:***
- $validPrices \leftarrow FilterOutliers(prices)$
- ***Price Aggregation:***
- $aggregatedPrice \leftarrow AggregateWeightedPrice(validPrices)$
- $confidence \leftarrow CalculateConfidence(validPrices, aggregatedPrice)$
- ***Cache Update:***
- $ttl \leftarrow P.getTTLForAsset(S)$
- $C.set(S, \{price : aggregatedPrice, timestamp : now(), ttl : ttl, confidence : confidence, sources : validPrices\})$
- ***return*** $\{price : aggregatedPrice, timestamp : now(), confidence : confidence\}$

Protocol 7 (Oracle Price Request)

- ***Require:*** *Requester* R*, Asset symbol* S*, Callback information* C*, Oracle Service* O*, Price Provider* P
- ***Ensure:*** *Price data is delivered to the requester through the specified callback mechanism*
- ***Request Validation:***
- $valid \leftarrow ValidateRequest(R, S, C)$
- ***If*** $\neg valid$

- ***return*** $\{error : \text{"Invalid request"}\}$
- ***EndIf***
- ***Request Registration:***
- $requestId \leftarrow GenerateUniqueID()$
- $O.registerRequest(requestId, R, S, C)$
- ***Price Collection:***
- $priceResult \leftarrow P.collectPrice(S)$
- ***If*** $priceResult.error \neq null$
- $O.updateRequestStatus(requestId, \text{"failed"}, priceResult.error)$
- ***If*** $C.type = \text{"blockchain"}$
- $O.sendErrorCallback(C.contract, C.method, requestId, priceResult.error)$
- ***ElsIf*** $C.type = \text{"http"}$
- $O.sendHttpErrorResponse(C.url, requestId, priceResult.error)$
- ***EndIf***
- ***return*** $\{error : priceResult.error, requestId : requestId\}$
- ***EndIf***
- ***Response Signing:***
- $response \leftarrow \{requestId : requestId, symbol : S, price : priceResult.price, timestamp : priceResult.timestamp, confidence : priceResult.confidence\}$
- $signature \leftarrow O.signResponse(response)$
- $signedResponse \leftarrow \{response : response, signature : signature\}$
- ***Callback Execution:***
- ***If*** $C.type = \text{"blockchain"}$
- $tx \leftarrow O.createCallbackTransaction(C.contract, C.method, signedResponse)$
- $txHash \leftarrow O.submitTransaction(tx)$
- $O.monitorTransaction(txHash)$
- ***ElsIf*** $C.type = \text{"http"}$
- $O.sendHttpResponse(C.url, signedResponse)$
- ***EndIf***
- ***Request Completion:***
- $O.updateRequestStatus(requestId, \text{"completed"}, signedResponse)$
- ***return*** $\{success : true, requestId : requestId, response : signedResponse\}$

Oracle Request Processing Protocol The Oracle Service implements a comprehensive protocol for processing requests that ensures reliability, security, and efficiency. The process begins with **Request Validation**, where incoming requests are thoroughly validated for proper formatting, parameter completeness, and requester authorization to prevent malformed or unauthorized requests from entering the system. Once validated, **Request Classification** categorizes requests by type and priority to ensure appropriate handling and resource allocation, allowing critical requests to be processed with higher priority. The system then performs **Source Selection**, where based on the request type and parameters, appropriate data sources are selected from the available provider pool, considering factors such as reliability, specialization, and historical performance. To optimize performance, **Parallel Data Retrieval** is employed, retrieving data from multiple sources simultaneously to minimize latency and ensure redundancy in case any individual source fails. The collected data undergoes **Data Aggregation** using appropriate algorithms such as median, volume-weighted average, or other statistical methods based on the data type, producing a consensus result that mitigates outliers and anomalies. Before delivery, **Result Verification** checks the aggregated results against predefined validity criteria and historical patterns to detect potential errors or manipulation. For security, **Response Signing** cryptographically signs the final result to ensure its authenticity and integrity, allowing recipients to verify its origin. The system then handles **Response Delivery**, where the signed result is delivered to the requester through the specified callback mechanism or stored for retrieval, with support for various delivery methods including blockchain transactions, API callbacks, and message queues. For blockchain-based callbacks, **Transaction Confirmation** monitors the transaction until confirmation, ensuring that the data is successfully recorded on-chain. Finally, **Request Archiving** stores request details and results for audit, dispute resolution, and performance analysis, creating a comprehensive record of all oracle operations.

Blockchain Gateway Integration The Oracle Service integrates with blockchain networks through specialized gateways that enable seamless interaction with different blockchain ecosystems. The **Neo N3 Gateway** provides native integration with the Neo N3 blockchain, supporting direct interaction with NeoContract smart contracts through optimized interfaces that leverage Neo's unique features and capabilities. For projects operating on Ethereum or its derivatives, the **Ethereum Gateway** offers integration with Ethereum and EVM-compatible chains, supporting interaction with Solidity smart contracts through standardized interfaces that comply with

Ethereum's design patterns and conventions. To address the growing need for cross-chain functionality, the service includes a **Cross-Chain Oracle** with specialized components that enable oracle data to be used across multiple blockchain networks, maintaining consistency and reliability through sophisticated synchronization and verification mechanisms that ensure data integrity across different consensus systems.

1.6.1 Event Messaging System

The Event Messaging System is a core component of the Neo Service Layer that enables asynchronous communication between different parts of the platform. It provides a reliable and scalable mechanism for detecting, processing, and responding to events from various sources, including the Neo N3 blockchain, external systems, and internal services.

Definition 4 (Event) *An Event is a discrete occurrence or notification that represents a change in state or a significant action within the system. Each event has:*

- *Type: The category of the event (blockchain, system, custom)*
- *Source: The origin of the event*
- *Payload: The data associated with the event*
- *Timestamp: When the event occurred*
- *ID: A unique identifier for the event*

The Event Messaging System consists of several key components that work together to provide a comprehensive event processing pipeline. **Event Sources** serve as connectors that monitor and capture events from various origins, including blockchain networks, external APIs, and internal services, providing the raw event data that enters the system. The **Event Registry** functions as a centralized repository that maintains information about event types, their schemas, and associated triggers, ensuring that events are properly categorized and matched with appropriate handlers. When events enter the system, the **Trigger Evaluator** evaluates them against registered triggers to determine which functions should be executed in response, applying complex matching rules and conditions to ensure precise function selection. Once triggers are matched, the **Event Processor** handles events by enriching them with additional context and routing them to the appropriate handlers, transforming raw events into actionable information. Finally, the

Function Executor executes the functions triggered by events, managing their lifecycle from initialization to completion and handling their results, which may include storing data, sending notifications, or triggering additional events.

Event Processing Protocol The Event Messaging System implements a comprehensive protocol for processing events that ensures reliable and efficient event handling. The process begins with **Event Detection**, where specialized monitors observe various sources such as blockchain networks, message queues, and system components to identify relevant events as they occur. Once detected, **Event Validation** checks these events against their schema definitions to ensure they contain all required fields and conform to the expected format, preventing malformed events from entering the processing pipeline. To enhance the event's utility, **Event Enrichment** adds additional context such as related data from other systems, historical information, or derived attributes, creating a more comprehensive event representation. The enriched events undergo **Trigger Matching**, where they are compared against registered triggers using pattern matching, condition evaluation, and other criteria to determine which actions should be taken in response. When a trigger matches an event, **Function Selection** identifies the associated functions that should be executed, creating a mapping between events and their handlers. Before execution, **Execution Planning** determines the optimal execution order, parallelization strategy, and resource allocation for the selected functions, ensuring efficient processing. The system then performs **Function Invocation**, where selected functions are called with the event data and any additional context required for processing, executing the business logic associated with the event. After execution, **Result Handling** processes the function results, which may include storing data, sending notifications, or triggering additional events, creating a chain of event-driven actions. Finally, **Event Archiving** stores processed events and their handling results for auditing, analytics, and debugging purposes, creating a comprehensive record of system activity.

A formal specification of the event processing protocol is provided in Section 1.19.

1.7 Event Messaging Protocol

The Event Messaging Protocol defines how events are detected, processed, and routed to appropriate handlers.

Protocol 8 (Event Processing)

- ***Require:*** *Event source S, Event registry R, Trigger evaluator T, Function registry F*
- ***Ensure:*** *Proper event processing and function execution*
- ***Event Detection:***
- $e \leftarrow DetectEvent(S)$ *// Detect event from source*
- $valid \leftarrow ValidateEvent(e, schema(e.type))$
- ***If*** $\neg valid$
- ***reject*** e
- ***EndIf***
- ***Event Enrichment:***
- $e' \leftarrow EnrichEvent(e, context(e))$ *// Add context information*
- ***Trigger Matching:***
- $triggers \leftarrow \emptyset$
- ***For*** *each* $t \in R.getTriggers(e'.type)$
- ***If*** $EvaluateTrigger(t, e')$
- $triggers \leftarrow triggers \cup \{t\}$
- ***EndIf***
- ***EndFor***
- ***Function Selection:***
- $functions \leftarrow \emptyset$
- ***For*** *each* $t \in triggers$
- $functions \leftarrow functions \cup F.getFunctions(t)$
- ***EndFor***
- ***Execution Planning:***
- $plan \leftarrow CreateExecutionPlan(functions, e')$
- ***Function Execution:***
- ***For*** *each* $f \in plan.getOrderedFunctions()$
- $result_f \leftarrow ExecuteFunction(f, e', context(f))$
- $ProcessResult(result_f, f, e')$
- ***EndFor***
- ***Event Archiving:***
- $ArchiveEvent(e', triggers, functions, \{result_f\})$

Blockchain Event Integration The Event Messaging System provides specialized integration with the Neo N3 blockchain through a comprehensive set of event types that capture different aspects of blockchain activity. The system processes **Block Events** which provide notifications for new blocks, including block height, timestamp, and transaction count, enabling applications to track blockchain progression and synchronize with the latest state. For transaction monitoring, **Transaction Events** deliver notifications for transactions, including transaction hash, sender, receiver, and amount, allowing applications to track fund movements and transaction confirmations in real-time. Smart contract interactions are captured through **Contract Events** which provide notifications for smart contract events, including contract hash, event name, and parameters, enabling applications to respond to specific contract state changes and user interactions. The system also monitors **State Changes** with notifications for modifications to the blockchain state, such as balance updates, contract deployments, and storage modifications, providing a comprehensive view of the evolving blockchain ecosystem.

Event Filtering and Routing The system provides sophisticated filtering and routing capabilities that enable precise control over event processing flows. **Content-Based Filtering** allows events to be filtered based on their content, such as specific field values or patterns, enabling applications to focus on only the most relevant events for their use case. For more dynamic routing decisions, **Context-Based Routing** directs events to different handlers based on contextual information such as user identity, application, or environment, creating adaptive processing pipelines that can respond to changing conditions. The system implements **Priority-Based Processing** where events can be assigned priorities that determine their processing order and resource allocation, ensuring that critical events are handled promptly even during high load periods. To protect system stability and ensure fair resource distribution, **Rate Limiting** mechanisms can be applied to event processing, preventing any single source or event type from overwhelming the system while maintaining responsiveness for all applications.

1.7.1 Function Deployment and Execution

The Neo Service Layer provides a comprehensive system for deploying, managing, and executing serverless functions. This system enables developers to create and deploy code that responds to events, processes data, and interacts with blockchain networks without managing the underlying infrastructure.

Definition 5 (Function) *A Function is a unit of code that performs a spe-*

cific task when invoked. Each function has:

- *Name: A unique identifier for the function*
- *Code: The executable code that implements the function logic*
- *Runtime: The environment in which the function executes (e.g., JavaScript, TypeScript)*
- *Configuration: Settings that control the function's behavior and resource allocation*
- *Triggers: Conditions that cause the function to be executed*
- *Permissions: Access controls that determine what resources the function can use*

Function Deployment Process The deployment process involves several interconnected steps to ensure functions are properly registered, stored, and made available for execution. The process begins with **Function Submission**, where developers submit functions through the API Service, providing the function code, configuration, and metadata necessary for deployment. Once submitted, **Code Validation** thoroughly checks the code for syntax correctness, security vulnerabilities, and compliance with platform policies, ensuring that only secure and well-formed functions enter the system. For functions with external dependencies, **Dependency Resolution** identifies, validates, and prepares any required dependencies for inclusion in the execution environment, ensuring that functions have access to all necessary libraries and resources. The function is then formally added to the system through **Function Registration** in the Function Registry, which maintains comprehensive information about all available functions, including their interfaces, permissions, and metadata. For persistent storage, the **Storage** step securely saves the function code and configuration in the Storage System, which provides durable and versioned storage with rollback capabilities. To enable event-driven execution, **Trigger Registration** records any triggers associated with the function in the Event Registry, enabling event-based invocation when specified conditions are met. Finally, **Distribution** deploys the function to Worker Nodes, which prepare the execution environment and cache the function for efficient invocation, optimizing performance by reducing cold-start times.

A formal specification of the function deployment and execution protocol is provided in Section 1.20.

1.8 Function Deployment and Execution Protocol

The Function Deployment and Execution Protocol defines how functions are deployed, managed, and executed within the Neo Service Layer.

Protocol 9 (Function Deployment)

- ***Require:*** *Function code C, Configuration conf, Metadata M, Function Registry R, Storage System S*
- ***Ensure:*** *Function is properly deployed and available for execution*
- ***Function Submission:***
- $F \leftarrow \{code : C, config : conf, metadata : M\}$
- ***Code Validation:***
- $valid \leftarrow ValidateCode(F.code)$
- ***If*** $\neg valid$
- ***reject*** F
- ***EndIf***
- ***Dependency Resolution:***
- $deps \leftarrow ResolveDependencies(F.code, F.config)$
- $F.dependencies \leftarrow deps$
- ***Function Registration:***
- $id \leftarrow GenerateUniqueID(F.metadata.name)$
- $F.id \leftarrow id$
- $R.registerFunction(F)$
- ***Storage:***
- $S.storeFunction(F)$
- ***Trigger Registration:***
- ***For*** *each* $t \in F.config.triggers$
- $EventRegistry.registerTrigger(t, F.id)$
- ***EndFor***
- ***Distribution:***
- ***For*** *each* $node \in WorkerNodes$
- $DistributeFunction(F, node)$

- ***EndFor***
- ***return*** *F.id*

Protocol 10 (Function Execution)

- ***Require:*** *Function ID id, Input data D, Context ctx, Scheduler sched, Worker Nodes W*
- ***Ensure:*** *Function is executed securely and results are properly handled*
- ***Function Lookup:***
- $F \leftarrow FunctionRegistry.getFunction(id)$
- ***Worker Assignment:***
- $worker \leftarrow sched.assignWorker(F, D, ctx)$
- ***Resource Allocation:***
- $resources \leftarrow worker.allocateResources(F.config.resources)$
- ***Sandbox Creation:***
- $sandbox \leftarrow worker.createSandbox(resources)$
- ***Runtime Initialization:***
- $runtime \leftarrow sandbox.initializeRuntime(F.config.runtime)$
- $runtime.setPermissions(F.config.permissions)$
- ***Function Loading:***
- $runtime.loadFunction(F.code, F.dependencies)$
- ***Execution:***
- $result \leftarrow runtime.executeFunction(F.metadata.entrypoint, D, ctx)$
- ***Monitoring:***
- ***While*** $runtime.isExecuting()$
- $metrics \leftarrow runtime.getMetrics()$
- ***If*** $metrics.exceedsLimits(F.config.limits)$
- $runtime.terminate()$
- ***return*** $\{error : \text{"Resource limits exceeded"}\}$
- ***EndIf***
- ***EndWhile***
- ***Result Capture:***

- $output \leftarrow runtime.getOutput()$
- $errors \leftarrow runtime.getErrors()$
- $logs \leftarrow runtime.getLogs()$
- ***Resource Cleanup:***
- $sandbox.cleanup()$
- $worker.releaseResources(resources)$
- ***Result Processing:***
- $ProcessFunctionResult(output, errors, logs, F, D, ctx)$
- ***return*** $\{output : output, errors : errors, logs : logs\}$

Function Execution Flow When a function is triggered for execution, the system follows a comprehensive flow to ensure reliable and secure processing. The process begins with **Trigger Evaluation**, where an event or direct invocation initiates the function execution process, determining which function needs to be executed and with what parameters. Once triggered, **Scheduler Assignment** occurs, where the Scheduler assigns the function execution task to an appropriate Worker Node based on factors such as load, locality, and resource requirements, optimizing resource utilization across the platform. The selected Worker Node then undergoes **Worker Preparation**, allocating resources and initializing the execution environment to ensure it has sufficient capacity to handle the function. For security, **Sandbox Creation** establishes a secure sandbox environment to isolate the function execution from other functions and the host system, preventing potential security breaches or resource conflicts. Within this sandbox, **Runtime Initialization** prepares the JavaScript runtime (based on Deno Core) with appropriate permissions and API access, creating a controlled execution context with precisely defined capabilities. The system then performs **Function Loading**, where the function code and dependencies are loaded into the runtime environment, making them available for execution. The core step of **Execution** runs the function with the provided input data and context, executing the business logic defined by the developer. During execution, **Monitoring** tracks resource usage, execution time, and other metrics to ensure compliance with limits and detect anomalies, preventing runaway processes or resource exhaustion. After execution completes, **Result Capture** collects the function's output, including return values, errors, and side effects, providing comprehensive information about the execution outcome. The system then performs **Resource Cleanup**, releasing resources and cleaning up or

recycling the sandbox environment to maintain system efficiency. Finally, **Result Processing** handles the execution results according to the function configuration, which may include storing data, sending notifications, or triggering additional functions, creating chains of function executions that can implement complex workflows.

Resource Management The function execution system implements sophisticated resource management to ensure efficient and fair allocation across all functions and applications. The system enforces **CPU Limits** by allocating specific CPU quotas to functions, preventing excessive usage by any single function and ensuring fair sharing of computational resources across all workloads. To prevent memory-related issues, **Memory Allocation** is carefully monitored and limited to prevent memory leaks and excessive consumption, ensuring system stability even under high load conditions. For operational safety, **Execution Timeouts** are implemented where functions have maximum execution time limits to prevent infinite loops and resource hogging, automatically terminating functions that exceed their allocated time. The platform optimizes overall throughput through **Concurrency Control**, managing the number of concurrent function executions to balance responsiveness with system stability, preventing overload while maximizing resource utilization. To adapt to changing demand patterns, **Auto-scaling** capabilities allow Worker Nodes to be dynamically scaled based on demand, automatically adding resources during peak periods and reducing them during low-demand periods to handle varying workloads efficiently while optimizing operational costs.

1.8.1 Secret Storage and Management

The Secret Storage and Management system provides a secure way for functions to store and access sensitive information such as API keys, credentials, and private keys. This system ensures that secrets are protected from unauthorized access while remaining available to authorized functions.

Definition 6 (Secret) *A Secret is a piece of sensitive information that requires protection from unauthorized access. Each secret has:*

- *Key: A unique identifier for the secret*
- *Value: The sensitive data being stored*
- *Owner: The entity that controls access to the secret*
- *Permissions: Rules that determine which functions can access the secret*

- *Metadata: Additional information about the secret, such as creation time and expiration*

Secret Storage Architecture The Secret Storage system implements a multi-layered architecture to ensure comprehensive security throughout the secret management lifecycle. At the front end, the **API Layer** provides a standardized interface for functions to store, retrieve, and manage secrets, offering a consistent and secure access point for all secret operations. Behind this interface, the **Service Layer** implements the core logic for secret management, including access control, encryption, and lifecycle management, coordinating the various components involved in secret handling. For data protection, the **Encryption Layer** ensures that all secrets are encrypted at rest using strong cryptographic algorithms, preventing unauthorized access even if the underlying storage is compromised. The **Storage Layer** provides durable and reliable storage for encrypted secrets, ensuring that secrets remain available and intact even in the face of system failures. Throughout the system, **Access Control** mechanisms enforce permissions and policies that determine who can access which secrets, implementing the principle of least privilege to minimize potential security risks. For security monitoring and compliance, comprehensive **Audit Logging** records all access attempts and operations, creating an immutable trail of all interactions with the secret storage system that can be used for security analysis and regulatory compliance.

Secret Management Protocol The system implements a comprehensive protocol for managing secrets throughout their lifecycle, ensuring security at every stage. The process begins with **Secret Creation**, where a new secret is assigned a unique identifier, encrypted with a master key, and stored in the storage layer, establishing the foundation for secure secret management. Following creation, **Access Control Configuration** allows the owner to define precise access permissions, specifying which functions or users can access the secret, implementing the principle of least privilege. When a function requires access to a secret, **Secret Retrieval** is initiated, where the function sends a request to the Secret API with its authentication credentials, initiating the secure access process. Upon receiving a request, the system performs an **Authorization Check**, verifying that the requesting function has permission to access the requested secret, preventing unauthorized access attempts. For authorized requests, **Decryption** occurs where the service retrieves the encrypted secret from storage, decrypts it using the master key, and returns it to the function, ensuring that secrets are only exposed to authorized en-

tities. To maintain a comprehensive audit trail, **Usage Tracking** logs all access to secrets, including the requester, timestamp, and operation, providing visibility into secret usage patterns and potential security incidents. To minimize the risk of long-term compromise, **Secret Rotation** automatically updates secrets according to configured policies, ensuring that even if a secret is compromised, the window of vulnerability is limited. In cases of suspected compromise, **Secret Revocation** provides immediate invalidation of secrets, preventing further access and mitigating potential damage from unauthorized use.

A formal specification of the secret management protocol is provided in Section 1.21.

1.9 Secret Storage and Management Protocol

The Secret Storage and Management Protocol defines how sensitive information is securely stored, accessed, and managed within the Neo Service Layer.

Protocol 11 (Secret Storage)

- ***Require:*** *Secret key k, Secret value v, Owner o, Permissions P, Secret Service S*
- ***Ensure:*** *Secret is securely stored and accessible only to authorized entities*
- ***Secret Creation:***
- $metadata \leftarrow \{created : now(), owner : o, permissions : P\}$
- $secret \leftarrow \{k, v, metadata\}$
- ***Access Control Setup:***
- $acl \leftarrow CreateAccessControlList(o, P)$
- $S.registerACL(k, acl)$
- ***Encryption:***
- $dataKey \leftarrow GenerateDataKey()$
- $encryptedValue \leftarrow Encrypt(v, dataKey)$
- $encryptedDataKey \leftarrow Encrypt(dataKey, S.masterKey)$
- ***Storage:***
- $record \leftarrow \{k, encryptedValue, encryptedDataKey, metadata\}$
- $S.storage.store(record)$
- ***return*** k

Protocol 12 (Secret Retrieval)

- ***Require:*** *Secret key k, Requester r, Secret Service S*
- ***Ensure:*** *Secret is only provided to authorized requesters*
- ***Authentication:***
- $authenticated \leftarrow AuthenticateRequester(r)$
- ***If*** $\neg authenticated$
- ***reject*** *request*
- ***EndIf***
- ***Authorization:***
- $acl \leftarrow S.getACL(k)$
- $authorized \leftarrow CheckAuthorization(r, acl)$
- ***If*** $\neg authorized$
- ***reject*** *request*
- ***EndIf***
- ***Retrieval:***
- $record \leftarrow S.storage.retrieve(k)$
- ***If*** $record = null$
- ***return*** $\{error : "Secret\ not\ found"\}$
- ***EndIf***
- ***Decryption:***
- $dataKey \leftarrow Decrypt(record.encryptedDataKey, S.masterKey)$
- $v \leftarrow Decrypt(record.encryptedValue, dataKey)$
- ***Audit Logging:***
- $LogAccess(k, r, "retrieve", now())$
- ***return*** v

Protocol 13 (Secret Rotation)

- ***Require:*** *Secret key k, New value v', Requester r, Secret Service S*
- ***Ensure:*** *Secret is securely updated while maintaining access controls*
- ***Authentication and Authorization:***
- $authenticated \leftarrow AuthenticateRequester(r)$

- $acl \leftarrow S.getACL(k)$
- $authorized \leftarrow CheckAuthorization(r, acl, \text{"update"})$
- ***If*** $\neg authenticated \vee \neg authorized$
- ***reject*** *request*
- ***EndIf***
- ***Retrieval:***
- $record \leftarrow S.storage.retrieve(k)$
- ***If*** $record = null$
- ***return*** $\{error : \text{"Secret not found"}\}$
- ***EndIf***
- ***New Encryption:***
- $dataKey \leftarrow GenerateDataKey()$
- $encryptedValue \leftarrow Encrypt(v', dataKey)$
- $encryptedDataKey \leftarrow Encrypt(dataKey, S.masterKey)$
- ***Update:***
- $record.encryptedValue \leftarrow encryptedValue$
- $record.encryptedDataKey \leftarrow encryptedDataKey$
- $record.metadata.updated \leftarrow now()$
- $S.storage.update(record)$
- ***Audit Logging:***
- $LogAccess(k, r, \text{"rotate"}, now())$
- ***return*** *true*

Encryption Technologies The Secret Storage system employs multiple encryption technologies to provide comprehensive protection for secrets throughout their lifecycle. The system implements **Envelope Encryption** where secrets are encrypted with data keys, which are themselves encrypted with a master key, providing multiple layers of protection that limit exposure of the master key and enable efficient key rotation. For the highest level of security, **Hardware Security Modules (HSMs)** can be used to store master keys, providing hardware-level protection against extraction and ensuring that cryptographic operations occur within a secure, tamper-resistant environment. To minimize the impact of potential key compromise, **Key**

Rotation mechanisms regularly update encryption keys according to configurable policies, ensuring that even if a key is compromised, its utility is limited to a specific time window. The system also employs **Secure Key Derivation** techniques where keys are derived using secure algorithms that incorporate multiple factors, such as user credentials and environmental attributes, creating context-specific keys that are more resistant to brute force attacks and unauthorized access attempts.

1.9.1 Sandbox Security

The Sandbox Security system provides a secure and isolated environment for executing functions, protecting both the platform from malicious functions and functions from each other. This system ensures that functions can only access the resources they are explicitly authorized to use.

Definition 7 (Sandbox) *A Sandbox is a controlled execution environment that restricts the capabilities of the code running within it. Each sandbox has:*

- *Isolation Boundaries: Limits that separate the sandbox from the host system and other sandboxes*
- *Resource Limits: Constraints on CPU, memory, network, and other resources*
- *Permission Model: Rules that determine what APIs and system resources the code can access*
- *Monitoring: Mechanisms for observing and controlling the behavior of the code*

Sandbox Architecture The Sandbox system implements a multi-layered architecture to provide comprehensive security throughout the function execution lifecycle. At the infrastructure level, the **Worker Node** serves as the physical or virtual machine that hosts the sandbox environment, providing the foundational hardware and operating system resources. Within each worker, the **Sandbox Environment** creates an isolated container or virtual machine that provides the first level of isolation, separating function execution from the host system and other functions. For code execution, the **JavaScript Runtime** based on Deno Core executes the function code with controlled access to system resources, implementing a secure-by-default approach to permissions. To prevent resource abuse, the **Resource Limiter** continuously monitors and enforces limits on CPU, memory, and other resource usage, ensuring that functions cannot consume excessive resources or

interfere with other workloads. Access to external systems is controlled by the **Permission Manager**, which governs access to APIs, network resources, and other capabilities based on the function's explicitly granted permissions, implementing the principle of least privilege. At the core of the JavaScript execution, the **V8 Engine** provides the underlying JavaScript engine with additional isolation through its security model, including memory sandboxing and context separation that prevent code from accessing memory outside its designated areas.

Sandbox Security Mechanisms The system implements multiple security mechanisms to ensure comprehensive protection throughout the function execution lifecycle. For execution isolation, **Process Isolation** ensures that functions run in separate processes to prevent interference and limit the impact of crashes, creating strong boundaries between different functions. To control resource access, **Namespace Isolation** provides each sandbox with its own namespace for files, network, and other resources, preventing access to resources outside the sandbox and creating a contained environment for execution. The system implements **Capability-Based Security** where functions must explicitly request and be granted permissions for specific capabilities, such as network access or file system operations, enforcing the principle of least privilege at a granular level. To prevent resource monopolization, **Resource Quotas** impose strict limits on CPU time, memory usage, network bandwidth, and other resources to prevent denial-of-service attacks, ensuring fair resource allocation across all functions. For operational safety, **Time Limits** enforce maximum execution time limits to prevent infinite loops and resource exhaustion, automatically terminating functions that exceed their allocated time. Network security is maintained through **Network Filtering** where network access is controlled through fine-grained rules that specify which hosts and ports can be accessed, preventing unauthorized network communication and limiting potential attack vectors.

A formal specification of the sandbox security protocol is provided in Section 1.22.

1.10 Sandbox Security Protocol

The Sandbox Security Protocol defines how function execution is isolated and secured to protect both the platform and other functions.

Protocol 14 (Sandbox Creation and Isolation) •

Require: *Function F, Resource limits L, Permissions P, Worker Node W*

- ***Ensure:*** *Secure and isolated execution environment*
- ***Resource Allocation:***
- $resources \leftarrow W.allocateResources(L)$
- ***Container Creation:***
- $container \leftarrow W.createContainer(resources)$
- $container.setNamespaces(\{"pid", "net", "ipc", "mnt", "uts"\})$
- ***Filesystem Setup:***
- $rootfs \leftarrow W.createIsolatedFilesystem()$
- $container.mountFilesystem(rootfs)$
- ***Network Configuration:***
- ***If*** $P.allowsNetwork()$
- $network \leftarrow W.createNetworkNamespace(P.networkRules)$
- $container.setNetwork(network)$
- ***Else***
- $container.disableNetwork()$
- ***EndIf***
- ***Resource Limits Setup:***
- $container.setCPULimit(L.cpu)$
- $container.setMemoryLimit(L.memory)$
- $container.setDiskIOLimit(L.diskIO)$
- $container.setNetworkIOLimit(L.networkIO)$
- ***Runtime Initialization:***
- $runtime \leftarrow container.initializeRuntime(F.config.runtime)$
- $runtime.setPermissions(P)$
- ***Security Policies:***
- $seccomp \leftarrow CreateSeccompProfile(P)$
- $container.applySeccompProfile(seccomp)$
- $apparmor \leftarrow CreateAppArmorProfile(P)$
- $container.applyAppArmorProfile(apparmor)$
- ***Monitoring Setup:***
- $monitor \leftarrow W.createResourceMonitor(container, L)$

- $monitor.start()$
- ***return*** $\{container : container, runtime : runtime, monitor : monitor\}$

Protocol 15 (Sandbox Execution Control)

- ***Require:*** *Sandbox* S*, Function* F*, Input data* D*, Context ctx, Timeout* T
- ***Ensure:*** *Secure execution with proper resource control*
- ***Function Loading:***
- $S.runtime.loadFunction(F.code, F.dependencies)$
- ***API Access Configuration:***
- ***For*** *each* $api \in F.config.apis$
- ***If*** $F.permissions.allowsAPI(api)$
- $S.runtime.enableAPI(api)$
- ***EndIf***
- ***EndFor***
- ***Execution Timer:***
- $timer \leftarrow StartTimer(T)$
- ***Execution:***
- $executionPromise \leftarrow S.runtime.executeAsync(F.entrypoint, D, ctx)$
- ***Monitoring Loop:***
- ***While*** $\neg executionPromise.isResolved() \wedge \neg timer.isExpired()$
- $metrics \leftarrow S.monitor.getMetrics()$
- ***If*** $metrics.exceedsLimits(F.config.limits)$
- $S.runtime.terminate()$
- ***return*** $\{error : \text{"Resource limits exceeded"}, metrics : metrics\}$
- ***EndIf***
- $Sleep(10\ ms)$
- ***EndWhile***
- ***If*** $timer.isExpired()$
- $S.runtime.terminate()$
- ***return*** $\{error : \text{"Execution timeout"}, timeout : T\}$

- ***EndIf***
- ***Result Capture:***
- *result* ← *executionPromise.getResult*()
- ***Cleanup:***
- *S.runtime.cleanup*()
- *S.monitor.stop*()
- ***return*** *result*

JavaScript Runtime Security The Deno Core JavaScript runtime provides additional security features that enhance the overall sandbox protection model. The runtime implements a **Secure Module System** where modules must be explicitly imported and can only access the capabilities they are granted, preventing unauthorized code execution and limiting the potential attack surface. For improved code quality and security, **TypeScript Support** provides static type checking that can catch potential security issues at compile time, reducing the risk of type-related vulnerabilities and improving code reliability. Access to potentially dangerous functionality is controlled through **Controlled API Access** where access to sensitive APIs is managed through a permission system that requires explicit grants, ensuring that functions can only use the specific capabilities they need. The system follows **Secure Defaults** principles where, by default, functions have no access to the file system, network, or environment variables, requiring explicit permission for each capability, implementing a zero-trust security model that minimizes the risk of unauthorized access.

1.10.1 Trusted Execution Environment (TEE)

The Trusted Execution Environment (TEE) service provides a secure and isolated environment for executing sensitive computations off-chain. This service is particularly valuable for applications that require privacy, high computational resources, or access to sensitive data that cannot be exposed on the public blockchain.

Definition 8 (Trusted Execution Environment) *A Trusted Execution Environment is a secure area within a processor that ensures the confidentiality and integrity of code and data loaded inside it. TEEs provide hardware-level isolation, protecting sensitive operations from the host operating system and other applications.*

The TEE service supports multiple platforms and security levels to accommodate diverse hardware environments and security requirements. The service provides support for **Intel SGX**, which offers secure enclaves within Intel processors that create isolated memory regions protected from the rest of the system, including privileged software. For AMD-based systems, **AMD SEV** (Secure Encrypted Virtualization) is supported, providing memory encryption for virtual machines to protect against physical memory attacks and hypervisor vulnerabilities. Mobile and embedded applications can leverage **ARM TrustZone**, a secure execution environment for ARM processors that creates a hardware-separated secure world for sensitive operations. For cloud deployments, the service integrates with **Cloud TEEs**, which are confidential computing offerings from major cloud providers that enable secure processing in shared cloud environments while maintaining data confidentiality.

Applications can leverage the TEE service for various use cases that require enhanced security and privacy guarantees. For sensitive data processing, **Private Computation** enables applications to process sensitive data without revealing it to the blockchain, maintaining confidentiality while still allowing verification of computation results. In scenarios involving multiple stakeholders, **Secure Multi-Party Computation** facilitates collaborative computation between mutually distrusting parties, allowing them to jointly compute results without revealing their individual inputs to each other. For blockchain applications requiring privacy, **Confidential Smart Contracts** enable the execution of contracts with private state and logic, keeping sensitive business logic and transaction details confidential while still leveraging blockchain security. Critical security operations benefit from **Secure Key Management** capabilities, providing robust protection of cryptographic keys and credentials against both software and hardware attacks, ensuring that even if the host system is compromised, the keys remain secure.

The TEE service provides attestation reports that verify the integrity and authenticity of the execution environment. These reports can be verified on-chain, allowing smart contracts to trust the results of off-chain computations.

TEE Execution Protocol The TEE service implements a comprehensive protocol for secure execution that ensures confidentiality and integrity throughout the computation lifecycle. The process begins with **Code Submission**, where the application submits the code to be executed in the TEE, along with any required input data, initiating the secure computation process. Based on the application's requirements, **Environment Selection** occurs, where the service selects an appropriate TEE environment based on

the security requirements, availability, and capabilities, ensuring optimal execution conditions. The selected environment then undergoes **Enclave Initialization**, where the TEE initializes a secure enclave, loading the code and preparing the execution environment with the necessary resources and security configurations. To establish trust, **Remote Attestation** generates an attestation report that proves the authenticity and integrity of the enclave to the application, providing cryptographic evidence that the code is running in a genuine and unmodified TEE. Before execution, **Secure Input Processing** ensures that input data is securely transferred to the enclave, often using encryption to protect sensitive information during transit, preventing exposure of confidential inputs. The core computation occurs during **Protected Execution**, where the code executes within the enclave with hardware-level protection against external observation or interference, ensuring that even privileged software cannot access the computation. After execution, **Result Sealing** cryptographically protects the execution results to ensure they can only be accessed by authorized parties, maintaining confidentiality beyond the execution phase. The protected results undergo **Secure Output Delivery**, where the sealed results are securely delivered to the requesting application or stored for later retrieval, ensuring end-to-end protection of sensitive outputs. Finally, **Verification** allows the application to verify the authenticity of the results using the attestation report and cryptographic proofs, establishing a chain of trust from execution to result consumption.

TEE Integration with Blockchain The TEE service provides specialized integration with blockchain networks, creating a powerful bridge between on-chain security and off-chain confidentiality. Through **On-Chain Verification**, smart contracts can verify TEE attestation reports directly on the blockchain, enabling trust in off-chain computation results and creating a verifiable link between private computations and public consensus. For applications requiring data privacy, **Confidential State** capabilities allow TEEs to maintain confidential state that is not visible on the public blockchain but can be used in computations, enabling applications to leverage blockchain security without sacrificing data confidentiality. To enhance transaction privacy, **Private Transactions** processing allows TEEs to handle private transaction data off-chain while publishing only necessary verification information on-chain, striking a balance between privacy and verifiability. For external data integration, **Secure Oracles** functionality enables TEEs to serve as secure oracles that access external data sources while providing cryptographic guarantees of data integrity, creating a trusted channel for bringing real-world data onto the blockchain with strong security assurances.

1.10.2 Smart Contract Integration

The Neo Service Layer provides comprehensive support for smart contract development, deployment, and integration. This system enables developers to create powerful decentralized applications that leverage both on-chain security and off-chain capabilities.

Definition 9 (Smart Contract Integration) *Smart Contract Integration refers to the seamless connection between blockchain-based smart contracts and the Neo Service Layer components. This integration enables contracts to access off-chain services while maintaining security and decentralization.*

Smart Contract Development The Neo Service Layer provides tools and libraries that simplify smart contract development, enabling developers to create sophisticated blockchain applications with minimal friction. Developers can leverage **Contract Templates**, which are pre-built templates for common contract patterns that integrate with Service Layer components, accelerating development by providing proven, secure implementations of frequently used functionality. To facilitate integration, **Service Layer SDKs** offer language-specific libraries that enable seamless interaction between smart contracts and Service Layer services, providing type-safe interfaces and abstractions that hide the complexity of the underlying protocols. For quality assurance, a comprehensive **Testing Framework** provides specialized tools for simulating contract interactions with Service Layer components, allowing developers to verify functionality and security before deployment to production environments. The development experience is enhanced through an **Development Environment** with integrated debugging capabilities specifically designed for contracts that use Service Layer features, enabling developers to efficiently identify and resolve issues throughout the development lifecycle.

Smart Contract Deployment Process The deployment process for smart contracts involves several interconnected steps that ensure proper integration with the Neo ecosystem. The process begins with **Contract Development**, where developers write smart contracts using Neo's supported languages (C#, Python, Go, TypeScript) with Service Layer integration, leveraging the specialized libraries and SDKs to create contracts that can interact with off-chain services. Once the contract code is complete, **Compilation** transforms the high-level code into Neo Executable Format (NEF) files, which

contain the bytecode that runs on the Neo Virtual Machine (NeoVM), optimizing the code for on-chain execution. To define the contract's capabilities and interfaces, **Manifest Creation** generates a manifest file that describes the contract's interface, permissions, and other metadata, establishing the contract's identity and access rights within the blockchain ecosystem. For contracts that leverage Service Layer functionality, **Service Registration** connects them with the relevant services to establish the necessary connections, enabling seamless interaction between on-chain and off-chain components. The actual blockchain integration occurs during **Deployment**, where the NEF file and manifest are deployed to the Neo N3 blockchain through a transaction, making the contract available for interaction on the network. After deployment, **Verification** ensures that the deployed contract matches the source code and functions as expected, confirming that no errors occurred during the compilation and deployment process. Finally, **Integration Configuration** sets up Service Layer components to interact with the deployed contract, establishing event listeners, callbacks, and other integration points that enable the contract to leverage the full capabilities of the Neo Service Layer.

A formal specification of the Smart Contract Integration protocol is provided in Section 1.25.

1.11 Smart Contract Integration Protocol

The Smart Contract Integration Protocol defines how smart contracts are deployed and integrated with the Neo Service Layer.

Protocol 16 (Smart Contract Deployment)

- ***Require:*** *Contract source code C, Developer D, Compiler comp, Blockchain B*
- ***Ensure:*** *Contract is properly deployed and registered with Service Layer*
- ***Contract Compilation:***
- $nef \leftarrow comp.compile(C)$
- $manifest \leftarrow comp.generateManifest(C)$
- ***Contract Validation:***
- $valid \leftarrow ValidateContract(nef, manifest)$
- ***If*** $\neg valid$
- ***return*** $\{error : "Invalid\ contract"\}$

- ***EndIf***
- ***Service Layer Integration:***
- ***For*** *each service* $\in$ *manifest.services*
- *ServiceRegistry.registerContractService*(*service.name*, *service.config*)
- ***EndFor***
- ***Deployment Transaction:***
- $tx \leftarrow B.createDeployTransaction(nef, manifest, D.address)$
- $signedTx \leftarrow D.signTransaction(tx)$
- ***Transaction Submission:***
- $txHash \leftarrow B.submitTransaction(signedTx)$
- ***Transaction Monitoring:***
- $confirmed \leftarrow false$
- ***While*** $\neg confirmed$
- $receipt \leftarrow B.getTransactionReceipt(txHash)$
- ***If*** $receipt \neq null$
- $confirmed \leftarrow true$
- ***If*** $receipt.status = \text{"success"}$
- $contractHash \leftarrow receipt.contractHash$
- ***Else***
- ***return*** $\{error : \text{"Deployment failed"}, receipt : receipt\}$
- ***EndIf***
- ***EndIf***
- ***If*** $\neg confirmed$
- *Sleep*(1 *second*)
- ***EndIf***
- ***EndWhile***
- ***Service Layer Registration:***
- *ContractRegistry.registerContract*(*contractHash*, *manifest*, *nef*)
- ***Event Listener Setup:***
- ***For*** *each event* $\in$ *manifest.events*
- *EventSystem.registerEventListener*(*contractHash*, *event.name*)

- ***EndFor***
- ***return*** $\{success : true, contractHash : contractHash, txHash : txHash\}$

Protocol 17 (Service Layer Contract Integration)

- ***Require:*** *Contract hash H, Service type T, Integration configuration I, Service Layer S*
- ***Ensure:*** *Contract is properly integrated with the specified Service Layer component*
- ***Contract Verification:***
- $contract \leftarrow ContractRegistry.getContract(H)$
- ***If*** $contract = null$
- ***return*** $\{error : "Contract\ not\ found"\}$
- ***EndIf***
- ***Service Verification:***
- $service \leftarrow S.getService(T)$
- ***If*** $service = null$
- ***return*** $\{error : "Service\ not\ found"\}$
- ***EndIf***
- ***Permission Check:***
- $hasPermission \leftarrow contract.manifest.permissions.hasPermission(T)$
- ***If*** $\neg hasPermission$
- ***return*** $\{error : "Contract\ does\ not\ have\ permission\ for\ this\ service"\}$
- ***EndIf***
- ***Integration Configuration:***
- $config \leftarrow service.createIntegrationConfig(H, I)$
- ***Integration Type-Specific Setup:***
- ***If*** $T = "GasBank"$
- $account \leftarrow service.createAccount(H, I.feeModel)$
- $config.accountAddress \leftarrow account.address$
- ***ElsIf*** $T = "MetaTransaction"$
- $service.registerContractForRelaying(H, I.methods)$

- ***ElsIf*** $T =$ *"Oracle"*
- *service.registerCallbackContract*$(H, I.callbackMethod)$
- ***ElsIf*** $T =$ *"TEE"*
- *service.registerAttestationVerifier*$(H, I.verifierMethod)$
- ***EndIf***
- ***Event Listener Setup:***
- ***For*** *each event* $\in I.events$
- *EventSystem.registerServiceEventHandler*$(T, H,$ *event.name*, *event.handler*$)$
- ***EndFor***
- ***Integration Storage:***
- *IntegrationRegistry.storeIntegration*$(H, T,$ *config*$)$
- ***return*** $\{success : true, contract : H, service : T, config : config\}$

Service Layer Integration Patterns Smart contracts can integrate with the Service Layer through several patterns that enable powerful combinations of on-chain and off-chain capabilities. Through **Event-Driven Integration**, contracts emit events that trigger Service Layer functions, enabling off-chain processing of on-chain actions and creating responsive systems that can react to blockchain state changes with complex off-chain logic. For accessing external data, **Oracle Integration** allows contracts to request information from the Oracle Service, enabling access to real-world information not natively available on the blockchain such as price feeds, weather data, or sports results. To improve user experience, **Gas Bank Integration** enables contracts to interact with the Gas Bank service to implement gas-free transactions for users, removing the friction of requiring users to hold cryptocurrency for transaction fees. Similarly, **Meta Transaction Integration** allows contracts to support meta transactions, enabling users to interact with blockchain applications without directly paying for transaction fees, with costs covered by application developers or subsidized through other mechanisms. For privacy-sensitive operations, **TEE Integration** allows contracts to delegate sensitive computations to the Trusted Execution Environment service, enabling private processing with on-chain verification that maintains confidentiality while leveraging blockchain security. Advanced user experiences are enabled through **Abstract Account Integration**, where contracts interact with the Abstract Account service to support sophisticated account management features such as social recovery, multi-signature control, and programmable authorization policies.

Smart Contract Security The Neo Service Layer enhances smart contract security through several mechanisms that work together to create a comprehensive security framework. For mathematical assurance of correctness, **Formal Verification** tools enable developers to formally verify contract behavior and security properties, providing mathematical proofs that contracts behave as intended under all possible conditions. To identify potential vulnerabilities early in the development process, **Security Scanning** provides automated analysis for common vulnerabilities and security issues, detecting patterns that could lead to exploits or unintended behavior before deployment. Developers are guided by **Secure Development Guidelines** that provide best practices and patterns for developing secure contracts that integrate with Service Layer components, establishing a foundation of security knowledge that helps prevent common mistakes. For thorough security assessment, **Auditing Tools** offer specialized capabilities for auditing contracts that use Service Layer features, enabling security professionals to comprehensively evaluate contract security with awareness of Service Layer integration points. After deployment, **Monitoring and Alerting** provides real-time monitoring of contract behavior with alerts for suspicious activities, enabling rapid response to potential security incidents and unusual patterns that might indicate an attack in progress.

1.11.1 Advanced Cryptographic Services

The Neo Service Layer includes advanced cryptographic services that provide cutting-edge privacy, security, and computational capabilities. These services enable applications to leverage sophisticated cryptographic techniques without requiring deep expertise in cryptography.

Definition 10 (Advanced Cryptographic Service) *An Advanced Cryptographic Service is a specialized component that implements sophisticated cryptographic protocols to enable secure computation, privacy-preserving operations, or enhanced security features beyond what is available in standard blockchain environments.*

The Neo Service Layer provides three main categories of advanced cryptographic services that enable sophisticated privacy and security capabilities. **Trusted Execution Environment (TEE)** technology offers hardware-based secure enclaves for confidential computation, creating isolated execution environments protected by the processor itself to ensure that even privileged software cannot access sensitive data or code. For computations on sensitive data, **Fully Homomorphic Encryption (FHE)** enables computation on encrypted data without requiring decryption, allowing operations

to be performed while maintaining data confidentiality throughout the entire processing lifecycle. To prove statements without revealing underlying information, **Zero-Knowledge Computing (ZK)** facilitates the generation and verification of proofs that demonstrate knowledge without exposing the underlying information, enabling privacy-preserving verification of claims, computations, and identities.

Fully Homomorphic Encryption Service The FHE service enables computation on encrypted data without requiring decryption, providing powerful privacy-preserving capabilities for sensitive applications. The service supports multiple **Encryption Schemes**, including TFHE, BFV, and CKKS, each optimized for different types of operations, allowing developers to select the most appropriate scheme for their specific use case requirements. For computational flexibility, the service provides various **Operation Types** on encrypted data, including arithmetic operations, boolean operations, and approximate operations on real numbers, enabling a wide range of privacy-preserving computations from simple calculations to complex algorithms. Secure **Key Management** is a core component of the service, providing secure generation, storage, and distribution of encryption keys, including support for threshold key generation and multi-party computation, ensuring that encryption keys remain protected throughout their lifecycle. To address the computational intensity of FHE, the service implements **Performance Optimization** techniques for enhancing FHE performance, including batching, parallelization, and hardware acceleration, making privacy-preserving computation practical for real-world applications. For blockchain applications, the service offers **Integration Patterns** for combining FHE with blockchain systems, including on-chain verification of off-chain FHE computations, enabling applications to leverage both the privacy benefits of FHE and the security guarantees of blockchain technology.

Zero-Knowledge Computing Service The ZK service enables the generation and verification of zero-knowledge proofs, providing powerful privacy-preserving capabilities for blockchain applications. The service supports multiple **Proof Systems**, including SNARKs, STARKs, and Bulletproofs, each with different trade-offs in terms of setup requirements, proof size, and verification time, allowing developers to select the most appropriate system for their specific requirements. For developer accessibility, **Circuit Compilation** tools facilitate the conversion of high-level program descriptions into arithmetic circuits suitable for zero-knowledge proofs, abstracting away the complex mathematical details and enabling developers to focus on applica-

tion logic. The core functionality includes **Proof Generation** capabilities for efficiently creating proofs that demonstrate the correctness of computations without revealing inputs or intermediate values, maintaining privacy while ensuring verifiability. For blockchain integration, **On-Chain Verification** provides seamless integration with Neo N3 smart contracts for on-chain verification of zero-knowledge proofs, enabling trustless verification of private computations. The service supports various **Privacy-Preserving Applications** with pre-built patterns for common privacy-preserving use cases, such as confidential transactions, private voting, and anonymous credentials, accelerating the development of privacy-focused blockchain applications.

Cross-Service Integration The advanced cryptographic services can be combined to create powerful hybrid solutions that leverage the complementary strengths of different cryptographic approaches. The **TEE-FHE Hybrid** integration uses Trusted Execution Environments to accelerate Fully Homomorphic Encryption operations while providing additional security guarantees, addressing the performance limitations of FHE while enhancing its security properties through hardware protection. For privacy-preserving proof generation, the **TEE-ZK Hybrid** approach uses TEEs to generate zero-knowledge proofs more efficiently while ensuring the privacy of the witness data, significantly improving the performance of complex zero-knowledge proof generation while maintaining strong privacy guarantees. To verify encrypted computations, the **FHE-ZK Hybrid** combines zero-knowledge proofs to verify the correctness of FHE computations without revealing the encryption keys or the computation itself, creating a verifiable yet private computational framework. For collaborative applications, **Multi-Party Computation** integrations combine multiple cryptographic techniques to enable secure computation among mutually distrusting parties, allowing organizations to jointly compute results without revealing their sensitive inputs to each other.

A formal specification of the Advanced Cryptographic Services protocol is provided in Section 1.27.

1.12 Advanced Cryptographic Services Protocol

The Advanced Cryptographic Services Protocol defines how the TEE, FHE, and ZK services operate and interact with applications.

Protocol 18 (TEE Secure Execution) • ***Require:*** *Code C, Input data D, TEE Service T, Requester R*

- ***Ensure:*** *Code is executed securely with confidentiality and integrity guarantees*
- ***TEE Selection:***
- $teeType \leftarrow T.selectTEEType(C.requirements)$
- $teeInstance \leftarrow T.allocateTEEInstance(teeType)$
- ***Enclave Initialization:***
- $enclave \leftarrow teeInstance.createEnclave()$
- $enclaveID \leftarrow enclave.getID()$
- ***Remote Attestation:***
- $quote \leftarrow enclave.generateQuote(C.hash)$
- $attestationReport \leftarrow T.verifyQuote(quote, teeType)$
- $T.storeAttestationReport(enclaveID, attestationReport)$
- ***Code Loading:***
- $enclave.loadCode(C)$
- ***Secure Input Processing:***
- $encryptedInput \leftarrow T.encryptForEnclave(D, enclave.publicKey)$
- $enclave.setInput(encryptedInput)$
- ***Secure Execution:***
- $enclave.execute()$
- ***Result Retrieval:***
- $encryptedResult \leftarrow enclave.getOutput()$
- $result \leftarrow T.decryptFromEnclave(encryptedResult, R.privateKey)$
- ***Result Verification:***
- $resultHash \leftarrow Hash(result)$
- $signature \leftarrow enclave.signResult(resultHash)$
- $verified \leftarrow T.verifyEnclaveSignature(enclaveID, resultHash, signature)$
- ***If*** $\neg verified$
- ***return*** $\{error : \text{"Result verification failed"}\}$
- ***EndIf***
- ***Enclave Cleanup:***
- $enclave.destroy()$

- $T.releaseTEEInstance(teeInstance)$
- ***return*** $\{success : true, result : result, attestation : attestationReport, signature : signature\}$

Protocol 19 (FHE Computation)

- ***Require:*** *Function F, Encrypted inputs E, FHE Service S, FHE scheme scheme*
- ***Ensure:*** *Computation is performed on encrypted data without revealing the plaintext*
- ***Scheme Verification:***
- $supported \leftarrow S.supportsScheme(scheme)$
- ***If*** $\neg supported$
- ***return*** $\{error : "Unsupported\ FHE\ scheme"\}$
- ***EndIf***
- ***Function Compilation:***
- $compiledFunction \leftarrow S.compileFunction(F, scheme)$
- ***Input Validation:***
- ***For*** $each\ input \in E$
- $valid \leftarrow S.validateEncryptedInput(input, scheme)$
- ***If*** $\neg valid$
- ***return*** $\{error : "Invalid\ encrypted\ input"\}$
- ***EndIf***
- ***EndFor***
- ***Execution Planning:***
- $executionPlan \leftarrow S.createExecutionPlan(compiledFunction, E)$
- ***Homomorphic Execution:***
- $encryptedResult \leftarrow S.executeHomomorphically(executionPlan)$
- ***Result Verification:***
- $valid \leftarrow S.validateEncryptedResult(encryptedResult, scheme)$
- ***If*** $\neg valid$
- ***return*** $\{error : "Invalid\ encrypted\ result"\}$
- ***EndIf***
- ***return*** $\{success : true, encryptedResult : encryptedResult\}$

Protocol 20 (Zero-Knowledge Proof Generation) •

Require: *Circuit C, Witness W, Public inputs P, ZK Service Z, Proof system system*

- ***Ensure:*** *A valid zero-knowledge proof is generated that can be verified without revealing the witness*
- ***System Verification:***
- $supported \leftarrow Z.supportsProofSystem(system)$
- ***If*** $\neg supported$
- ***return*** $\{error : "Unsupported\ proof\ system"\}$
- ***EndIf***
- ***Circuit Compilation:***
- $compiledCircuit \leftarrow Z.compileCircuit(C, system)$
- ***Setup Phase:***
- ***If*** $system.requiresTrustedSetup()$
- $setupParams \leftarrow Z.getSetupParameters(compiledCircuit)$
- ***If*** $setupParams = null$
- $setupParams \leftarrow Z.performTrustedSetup(compiledCircuit)$
- $Z.storeSetupParameters(compiledCircuit.hash, setupParams)$
- ***EndIf***
- ***Else***
- $setupParams \leftarrow Z.generateVerificationKey(compiledCircuit)$
- ***EndIf***
- ***Witness Validation:***
- $valid \leftarrow Z.validateWitness(W, compiledCircuit)$
- ***If*** $\neg valid$
- ***return*** $\{error : "Invalid\ witness"\}$
- ***EndIf***
- ***Proof Generation:***
- $proof \leftarrow Z.generateProof(compiledCircuit, W, P, setupParams)$
- ***Proof Verification:***
- $verified \leftarrow Z.verifyProof(proof, P, setupParams.verificationKey)$
- ***If*** $\neg verified$

- ***return*** $\{error : \text{"Generated proof verification failed"}\}$
- ***EndIf***
- ***return*** $\{success : true, proof : proof, verificationKey : setupParams.verificationKey\}$

Application Domains The advanced cryptographic services enable a wide range of privacy-preserving and secure applications across diverse domains. In the financial sector, **Private DeFi** applications preserve user privacy while ensuring compliance with regulations, enabling financial transactions that protect sensitive information while still meeting regulatory requirements. For identity management, **Confidential Identity** systems allow selective disclosure of attributes without revealing unnecessary information, giving users control over their personal data while still providing necessary verification to service providers. In data analytics, **Secure Multi-Party Analytics** enables collaborative analysis on sensitive data without exposing the raw data to any party, allowing organizations to derive insights from combined datasets without compromising data confidentiality. For organizational decision-making, **Private Governance** systems protect voter privacy while ensuring correct tallying, enabling transparent yet private voting processes for corporate governance, DAOs, and other collective decision-making contexts. In logistics and manufacturing, **Confidential Supply Chain** tracking preserves commercial confidentiality while enabling verification of claims, allowing businesses to validate supply chain information without exposing proprietary details about suppliers, pricing, or volumes.

1.12.1 Abstract Account Service

The Abstract Account service provides advanced account management capabilities beyond what is possible with standard blockchain accounts. This service enables features such as multi-signature control, account recovery, and programmable authorization policies, creating a flexible foundation for sophisticated account management.

Definition 11 (Abstract Account) *An Abstract Account is a smart contract-based account that implements advanced control and authorization mechanisms. Each abstract account has a unique Address that serves as its identifier in the blockchain ecosystem, an Owner who acts as the primary controller of the account, Controllers who are additional entities authorized to perform operations on the account, a Policy that defines the rules governing how operations are authorized, and a Contract that implements the underlying smart contract logic for the account.*

The Abstract Account service supports various account operations that enable flexible and secure account management. Users can modify account access by **Adding/Removing Controllers**, which allows adjusting the set of entities that can control the account, enabling dynamic access management as organizational needs change. Security policies can be adapted through **Updating Policies**, which allows changing the rules for operation authorization, such as modifying signature thresholds or adding time-based restrictions. For protection against key loss, **Account Recovery** mechanisms enable recovering access to an account after key loss through predefined recovery procedures like social recovery or backup key activation. Beyond standard operations, **Custom Operations** support application-specific functions defined by the account contract, allowing developers to extend account functionality for specific use cases such as scheduled transactions or conditional approvals.

This service is particularly valuable for applications requiring enhanced security, corporate governance, or complex authorization workflows. By abstracting account management into a dedicated service, applications can implement sophisticated control mechanisms without burdening users with technical complexity.

1.13 Integration with Neo Ecosystem

The Neo Service Layer is designed to integrate seamlessly with the broader Neo ecosystem, enhancing the capabilities of existing components and enabling new synergies between different parts of the platform.

1.13.1 Integration with NeoContract

The Neo Service Layer extends the capabilities of NeoContract by providing services that complement on-chain logic with off-chain resources. Smart contracts can interact with the Service Layer through standardized interfaces, allowing developers to combine the security and determinism of on-chain execution with the flexibility and power of off-chain services.

For example, a decentralized finance application might use NeoContract for core financial logic while leveraging the Oracle Service for price data, the TEE Service for private computations, and the Gas Bank for fee abstraction. This combination enables sophisticated applications that would be difficult or impossible to implement using on-chain logic alone.

The integration with NeoContract is facilitated through several mechanisms that create a seamless bridge between on-chain and off-chain components. **Native Contracts** provide system-level contracts that offer on-chain

interfaces to Service Layer capabilities, allowing smart contracts to interact with off-chain services through standardized and secure interfaces. For event-driven architectures, **Event Monitoring** components within the Service Layer listen for and respond to events emitted by smart contracts, enabling reactive off-chain processing triggered by on-chain state changes. To handle asynchronous operations, **Callback Patterns** establish standardized approaches for asynchronous interaction between contracts and services, allowing contracts to request off-chain operations and receive results when they become available. For maintaining trust in off-chain computations, **Verification Protocols** implement cryptographic methods for verifying off-chain computations on-chain, ensuring that results from Service Layer components can be cryptographically verified within the blockchain's trust model.

1.13.2 Integration with Digital Identity

The Neo Service Layer leverages Neo's digital identity framework to provide secure and compliant services. Each service component can verify user identities and enforce access controls based on identity attributes, enabling applications that comply with regulatory requirements while preserving user privacy.

The integration with digital identity enables several key capabilities that enhance security while respecting user privacy. For secure access control, **Authentication** mechanisms verify user identities for service access, ensuring that only legitimate users can interact with sensitive services and functions. Once authenticated, **Authorization** systems control service permissions based on identity attributes, implementing fine-grained access control that restricts users to only the specific operations they are entitled to perform. For compliance and security monitoring, **Audit** functionality tracks service usage for compliance and security purposes, creating immutable records of who accessed what services and when, without compromising user privacy. To protect sensitive personal information, **Privacy-Preserving Verification** enables selective disclosure of identity attributes, allowing users to prove specific claims about their identity without revealing unnecessary personal information.

1.13.3 Integration with Oracle and Interoperability Services

The Oracle Service within the Neo Service Layer complements Neo's native oracle capabilities, providing enhanced functionality and additional data sources. Similarly, the Service Layer's interoperability features extend Neo's

cross-chain capabilities, enabling more sophisticated interactions with external systems.

This integration creates a comprehensive solution for connecting Neo applications with external data and systems, enabling a diverse range of cross-domain applications. In the financial sector, **Cross-Chain DeFi** applications can operate across multiple blockchain ecosystems, allowing for liquidity pooling, asset transfers, and financial operations that span different blockchain networks. For physical asset representation, **Real-World Asset Tokenization** enables the representation of physical assets on the blockchain with real-time data updates, creating digital twins of real-world assets that maintain current information about their status and value. In the connected devices space, **IoT Integration** facilitates the connection of blockchain applications with Internet of Things devices and data, enabling secure and transparent recording of sensor data and automated contract execution based on real-world events. For enterprise adoption, **Enterprise System Integration** bridges blockchain applications with traditional enterprise systems, allowing organizations to leverage blockchain capabilities while maintaining compatibility with existing IT infrastructure and business processes.

1.14 Security and Privacy Considerations

Security and privacy are fundamental considerations in the design and implementation of the Neo Service Layer. As a bridge between on-chain and off-chain environments, the Service Layer must maintain the security guarantees of the blockchain while addressing the unique challenges of off-chain services.

1.14.1 Security Architecture

The Neo Service Layer implements a comprehensive security architecture based on fundamental security principles that work together to create a robust defense system. The architecture employs **Defense in Depth** by implementing multiple layers of security controls that protect against different types of threats, ensuring that the compromise of a single security control does not lead to a complete system breach. Access management follows the principle of **Least Privilege**, where services and components operate with the minimum permissions necessary to perform their functions, reducing the potential impact of compromised components. The system is designed to be **Secure by Default**, with security features enabled by default and requiring explicit action to disable, preventing security gaps from misconfiguration or

oversight. For community trust, **Transparent Security** ensures that security mechanisms are thoroughly documented and open to scrutiny, allowing independent verification of security claims and facilitating community-driven security improvements. Operational security is maintained through **Continuous Monitoring** with ongoing surveillance for security events and anomalies, enabling rapid detection and response to potential security incidents.

Specific security measures implemented across the Service Layer provide comprehensive protection at multiple levels. For data integrity, **Cryptographic Verification** ensures that all service responses are cryptographically signed and can be verified on-chain, creating a verifiable chain of trust for off-chain computations and data. Communication security is maintained through **Secure Communication** where all interactions between components use encrypted channels, protecting sensitive data in transit from interception or tampering. Access to services is controlled through **Access Control** mechanisms that implement granular permissions to determine who can access which services and operations, enforcing the principle of least privilege at the service level. For accountability and compliance, **Audit Logging** provides comprehensive recording of all security-relevant events, creating an immutable trail of actions for forensic analysis and regulatory compliance. The foundation of cryptographic security is protected through **Secure Key Management** with robust protection of cryptographic keys using hardware security modules or secure enclaves, ensuring that even if other security measures are compromised, the cryptographic foundations remain secure.

1.14.2 Privacy Features

The Neo Service Layer includes several features designed to enhance privacy while maintaining compliance with regulatory requirements, creating a balanced approach to data protection. For sensitive data processing, **Private Computation** leverages the TEE Service to enable computation on sensitive data without exposing the data itself, allowing applications to process confidential information while maintaining its privacy. User interactions are protected through **Selective Disclosure** mechanisms that allow users to reveal only the minimum information necessary for a particular interaction, giving them fine-grained control over what personal data is shared in each context. Advanced cryptography is employed through **Zero-Knowledge Proofs**, which provide cryptographic techniques that prove statements without revealing underlying data, enabling verification of claims without exposing sensitive information. The platform follows **Data Minimization** principles where services collect and store only the data necessary for their operation, reducing privacy risks by limiting the scope of potentially sensitive

information in the system. Throughout all interactions, **User Control** ensures that users maintain control over their data and how it is used, providing transparency and choice regarding data usage and sharing.

1.14.3 Compliance Framework

The Neo Service Layer is designed to facilitate compliance with relevant regulations while preserving the decentralized nature of blockchain applications. The compliance framework includes several integrated components that work together to create a comprehensive approach to regulatory requirements. For user verification, **Identity Verification** integrates with Neo's digital identity system for KYC/AML compliance, enabling applications to meet regulatory requirements for user identification while maintaining privacy. Regulatory oversight is supported through **Audit Trails** that provide comprehensive logging for regulatory reporting and auditing, creating immutable records of system activities that can be used for compliance verification. To address regional variations in regulations, **Configurable Controls** offer adjustable security and compliance settings for different jurisdictions, allowing applications to adapt to specific regulatory environments without requiring code changes. The system architecture implements **Privacy by Design** principles from the ground up, ensuring that privacy considerations are built into every component rather than added as an afterthought. For operational transparency, **Transparency Reports** provide regular reporting on service operation and compliance measures, creating accountability and building trust with users, regulators, and the broader community.

1.15 Developer Experience

The Neo Service Layer is designed to provide a seamless and productive experience for developers building applications on the Neo ecosystem. This focus on developer experience is reflected in several key aspects of the Service Layer:

1.15.1 API Design

The Service Layer exposes its functionality through well-designed, consistent APIs that follow modern best practices, providing developers with multiple options for integration. For standard web integration, **RESTful Endpoints** provide standard HTTP interfaces for service interaction, following familiar patterns that web developers can easily understand and implement. Applications with complex data requirements benefit from **GraphQL Support**,

which enables flexible queries that allow clients to request exactly the data they need in a single request, reducing over-fetching and under-fetching of data. For applications requiring real-time updates, **WebSocket Connections** enable real-time updates and notifications, allowing clients to maintain persistent connections and receive immediate updates when relevant events occur. To accommodate blockchain developers, **JSON-RPC Compatibility** provides familiar interfaces that align with existing blockchain API standards, reducing the learning curve for developers already working in the blockchain space. All APIs are thoroughly documented, with interactive documentation, code examples, and SDKs available in multiple programming languages, ensuring that developers can quickly understand and implement the APIs regardless of their technology stack.

1.15.2 SDK and Libraries

The Neo Service Layer provides comprehensive SDKs and libraries that simplify integration with popular programming languages and frameworks, reducing the development effort required to build applications. For diverse development environments, **Language Support** includes SDKs for JavaScript/TypeScript, Python, Rust, Go, Java, and C#, ensuring that developers can work with the Service Layer using their preferred programming language. To accelerate web and mobile development, **Framework Integration** offers plugins for popular web and mobile frameworks, providing seamless integration with existing application architectures and development workflows. For rapid smart contract development, **Smart Contract Templates** provide pre-built contract templates that integrate with Service Layer components, allowing developers to quickly implement common patterns and functionality. To minimize boilerplate code, **Code Generation** tools automatically generate client code from API specifications, ensuring that client implementations are always in sync with the latest API definitions. These SDKs abstract away the complexity of service interaction, allowing developers to focus on their application logic rather than infrastructure details, significantly reducing the learning curve and development time required to build Neo applications.

1.15.3 Developer Tools

The Neo Service Layer includes a suite of tools designed to streamline the development, testing, and deployment of applications, creating a comprehensive developer experience. For isolated testing, a **Local Development Environment** provides a self-contained environment for local testing, allowing developers to build and test applications without requiring access to

production infrastructure. To simulate service interactions, **Service Simulators** offer mock implementations of services for testing, enabling developers to test their applications against simulated service responses without requiring actual service deployments. When issues arise, **Debugging Tools** provide specialized utilities for troubleshooting service interactions, helping developers quickly identify and resolve integration issues. For operational visibility, a **Monitoring Dashboard** delivers real-time insights into service performance and usage, allowing developers to monitor their applications in production and identify potential issues before they impact users. To simplify production deployment, **Deployment Automation** tools facilitate deploying applications to production environments, streamlining the transition from development to production with automated workflows and validation checks.

1.16 Use Cases and Applications

The Neo Service Layer enables a wide range of applications that leverage the security and transparency of blockchain technology while overcoming its traditional limitations. This section explores some of the key use cases and applications that the Service Layer makes possible.

1.16.1 Decentralized Finance (DeFi)

The Neo Service Layer enhances DeFi applications with capabilities that address common challenges in the space, creating a more accessible and powerful financial ecosystem. For improved user experience, **Gas-Free Transactions** leverage the Gas Bank and Meta Transaction services to enable fee-less user experiences, reducing friction for DeFi users and eliminating the need for users to hold native tokens solely for transaction fees. Financial applications benefit from **Real-Time Price Feeds** through the Oracle Service, which provides accurate and timely price data for financial calculations, enabling reliable pricing for trading, lending, and derivatives applications. For institutional and professional traders, **Private Trading Strategies** can be implemented using the TEE Service, which allows the execution of trading algorithms without revealing proprietary strategies, protecting valuable intellectual property while still leveraging on-chain liquidity. The platform's **Cross-Chain Liquidity** capabilities integrate with multiple blockchain ecosystems to enable access to liquidity across different platforms, creating a more efficient and unified market across blockchain boundaries. To address legal requirements, **Regulatory Compliance** features through the Abstract Account Service and identity integration facilitate compliant DeFi applications, enabling financial services that can operate within regulatory

frameworks while maintaining the benefits of decentralization.

1.16.2 Gaming and NFTs

Blockchain gaming and NFT applications benefit from several Service Layer capabilities that collectively enhance the player experience and ecosystem functionality. For new users, **Seamless Onboarding** through gas abstraction removes the need for new players to acquire tokens before playing, eliminating a significant barrier to entry and allowing immediate engagement with blockchain games. Game fairness is ensured through **Verifiable Randomness** where the Oracle Service provides fair and transparent random numbers for game mechanics, creating trust in critical gameplay elements like loot drops, combat outcomes, and procedural generation. Performance constraints are addressed through **Off-Chain Computation** where the TEE Service enables complex game logic that would be too expensive to run entirely on-chain, allowing for sophisticated gameplay while maintaining blockchain integration for critical elements. The value of digital assets is enhanced through **Cross-Game Assets** where integration across different games and platforms allows NFTs to be used in multiple contexts, creating an interconnected ecosystem where player investments retain value across experiences. Player security is strengthened through **Social Recovery** mechanisms where the Abstract Account Service provides recovery options for valuable gaming accounts and NFT collections, protecting players from permanent loss due to key management issues.

1.16.3 Enterprise Applications

Enterprise blockchain applications can leverage the Neo Service Layer to address specific business requirements, enabling secure and efficient integration with existing systems. For operational visibility, **Supply Chain Tracking** integrates with IoT devices and external systems for end-to-end visibility, creating transparent and immutable records of product journeys from manufacturing to delivery. Intellectual property protection is enhanced through **Confidential Business Logic** where the TEE Service enables execution of proprietary business rules without exposing them publicly, allowing enterprises to leverage blockchain transparency while protecting competitive advantages. Compliance requirements are addressed through **Regulatory Reporting** with automated compliance reporting through secure off-chain processing, streamlining regulatory obligations while maintaining data privacy and security. Enterprise security frameworks are extended through **Identity and Access Management** integration with enterprise identity

systems for secure and compliant blockchain access, allowing organizations to maintain existing security policies while adopting blockchain technology. Technological transition is facilitated through **Legacy System Integration** by bridging between blockchain applications and traditional enterprise systems, enabling gradual adoption without requiring complete infrastructure replacement.

1.16.4 Public Goods and Infrastructure

The Neo Service Layer can support public infrastructure and social impact applications, enabling transparent and efficient systems for societal benefit. For democratic processes, **Transparent Governance** provides secure voting and decision-making processes for public organizations, creating auditable records of governance activities while protecting voter privacy. Humanitarian efforts are enhanced through **Aid Distribution** systems that enable efficient and transparent distribution of humanitarian aid using blockchain verification, ensuring that resources reach intended recipients and reducing opportunities for fraud or diversion. Environmental initiatives benefit from **Carbon Credits** functionality that supports verification and trading of carbon offsets with real-world data integration, creating trusted markets for environmental assets based on verifiable data. Civic infrastructure is improved through **Public Records** systems that provide secure and accessible storage of public records with privacy protections, balancing transparency with individual privacy rights. Educational systems are enhanced through **Educational Credentials** verification that enables verifiable academic and professional credentials with selective disclosure, allowing individuals to prove their qualifications without revealing unnecessary personal information.

1.17 Future Directions

The Neo Service Layer is designed to evolve alongside the broader blockchain ecosystem, adapting to new technologies, use cases, and requirements. Several key directions for future development include:

1.17.1 Advanced Cryptographic Techniques

The Service Layer will incorporate emerging cryptographic technologies to enhance privacy, security, and functionality, pushing the boundaries of what's possible in blockchain applications. For secure data processing, **Fully Homomorphic Encryption (FHE)** enables computation on encrypted data without decryption, allowing sensitive data to be processed while remaining

encrypted throughout the entire computation pipeline. Distributed security is enhanced through **Threshold Signatures** that enable distributed signing without reconstructing private keys, improving security by eliminating single points of failure in signature generation. Future-proofing against emerging threats is addressed through **Post-Quantum Cryptography** that provides resistance to quantum computing attacks, ensuring that blockchain applications remain secure even as quantum computing capabilities advance. Computational scalability is improved through **Recursive Zero-Knowledge Proofs** that enable scalable verification of complex computations, allowing for efficient validation of extensive calculations without requiring validators to repeat the entire computation process.

1.17.2 Enhanced Interoperability

Future versions of the Service Layer will expand interoperability capabilities, creating a more connected and integrated blockchain ecosystem. For cross-platform communication, **Cross-Chain Messaging** will provide standardized protocols for communication between different blockchain networks, enabling applications to seamlessly interact with multiple chains without requiring custom integration for each one. Asset mobility will be enhanced through **Universal Asset Bridge** functionality that enables seamless transfer of assets across multiple blockchain ecosystems, allowing users to utilize their digital assets wherever they provide the most value. Identity systems will be unified through **Decentralized Identity Federation** that creates interoperable identity verification across different systems, enabling users to maintain a single digital identity that works across the entire blockchain ecosystem. Developer experience will be improved through **API Standardization** with common interfaces for blockchain services across platforms, reducing the learning curve for developers working with multiple blockchain technologies and enabling the creation of tools that work across different platforms.

1.17.3 Decentralized Service Provision

The Neo Service Layer will progressively decentralize its components, moving toward a more distributed and community-driven infrastructure. For competitive service delivery, **Service Marketplaces** will create competitive markets for service provision, allowing multiple providers to offer compatible services with different performance characteristics and pricing models. Service quality will be maintained through **Staking and Reputation** systems that provide economic incentives for reliable service operation, requiring ser-

vice providers to stake assets that can be slashed for poor performance while building reputation scores based on historical reliability. Community involvement will be enhanced through **Governance Mechanisms** that enable community control over service parameters and upgrades, allowing stakeholders to participate in decision-making about the evolution of the Service Layer. Infrastructure decentralization will be achieved through **Distributed Execution** where service components run across decentralized infrastructure, reducing reliance on centralized providers and increasing resilience against failures and censorship attempts.

1.17.4 AI Integration

Integration with artificial intelligence technologies will create new capabilities, enabling a new generation of intelligent blockchain applications. For on-chain intelligence, **On-Chain AI Models** will enable deployment of lightweight AI models directly on the blockchain, allowing for deterministic AI-driven smart contracts and decentralized autonomous organizations with embedded intelligence. Computational security will be enhanced through **Secure AI Computation** that enables execution of AI workloads in trusted environments, allowing sensitive data to be used for AI inference without exposing the underlying information. Result integrity will be ensured through **Verifiable AI Results** with cryptographic verification of AI model outputs, creating provable records of AI decisions that can be independently verified without requiring re-execution of the models. Collaborative intelligence will be fostered through **Decentralized Training** that enables collaborative training of AI models while preserving data privacy, allowing multiple parties to contribute to model improvement without sharing their raw data.

1.18 Formal Protocol Specifications

This section provides formal specifications for the protocols that govern the operation of the Neo Service Layer components. These specifications define the precise behavior, interactions, and security properties of each component, ensuring consistent implementation and operation across different environments.

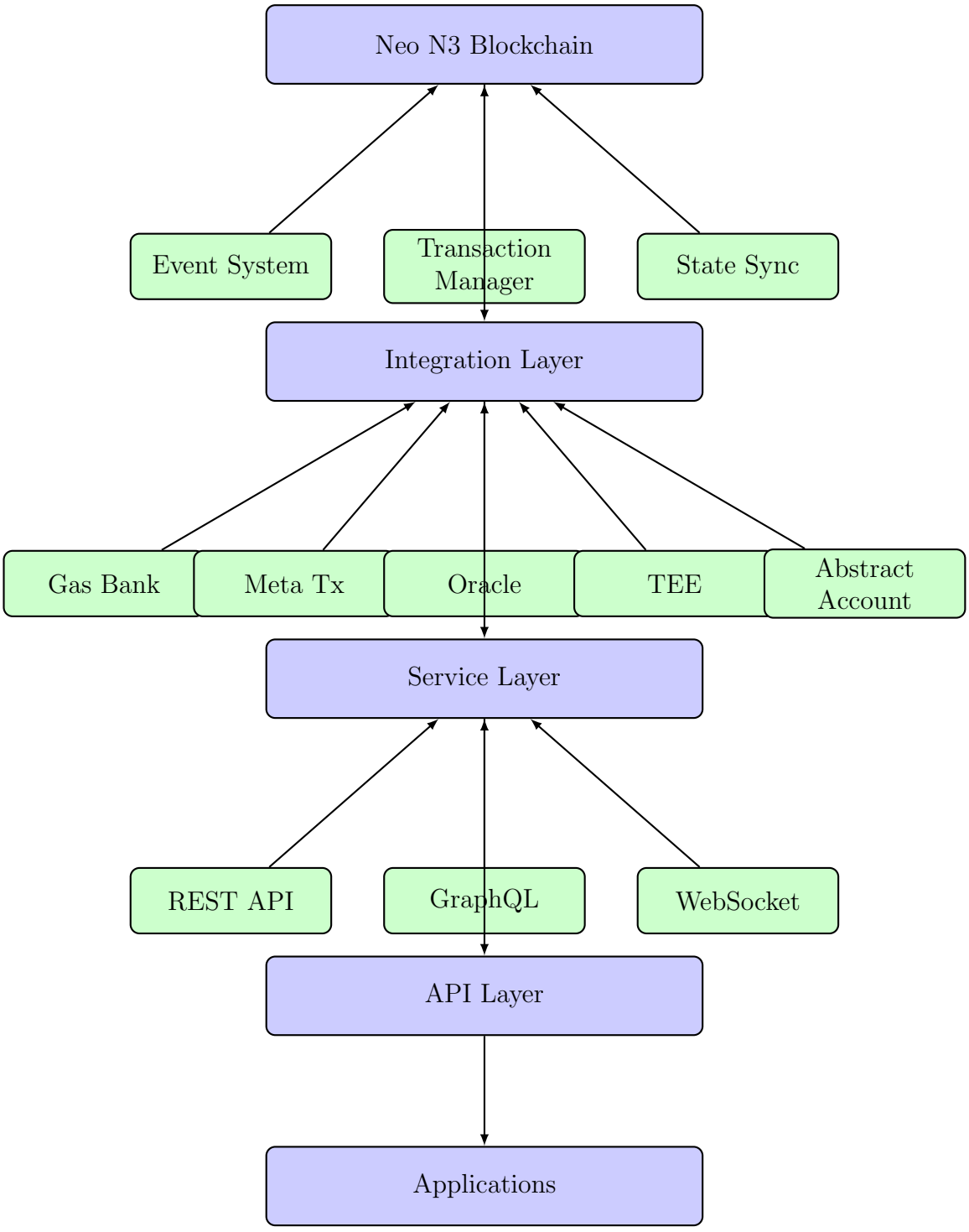

Figure 1: Neo Service Layer Architecture

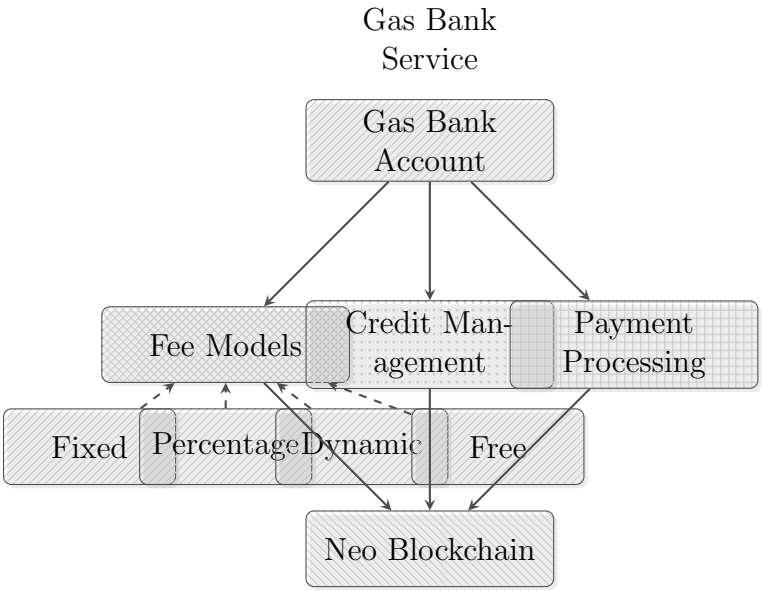

Figure 2: Gas Bank Service Architecture

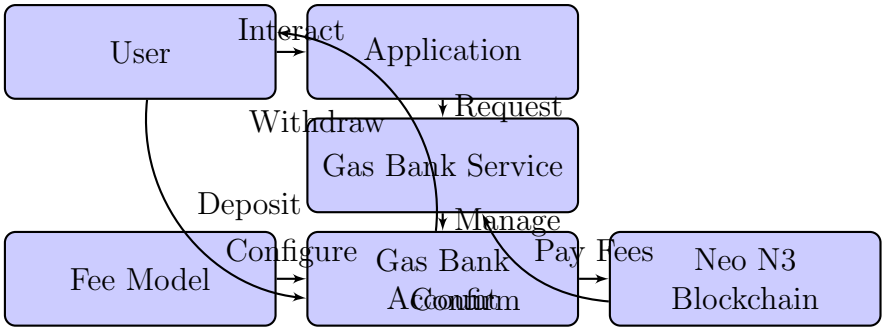

Figure 3: Gas Bank Operations in Neo Service Layer

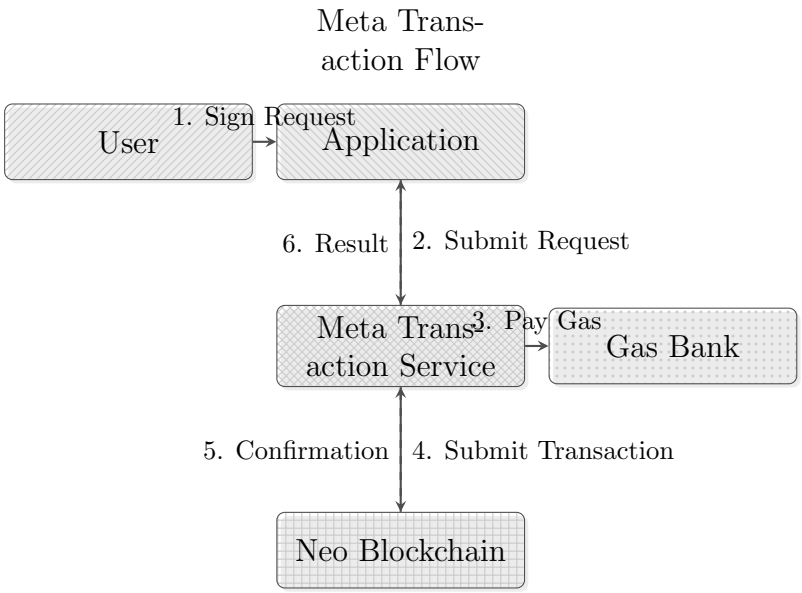

Figure 4: Meta Transaction Service Flow

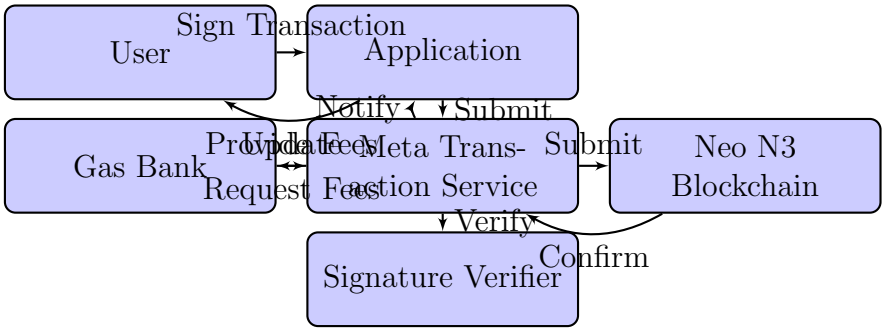

Figure 5: Meta Transaction Flow in Neo Service Layer

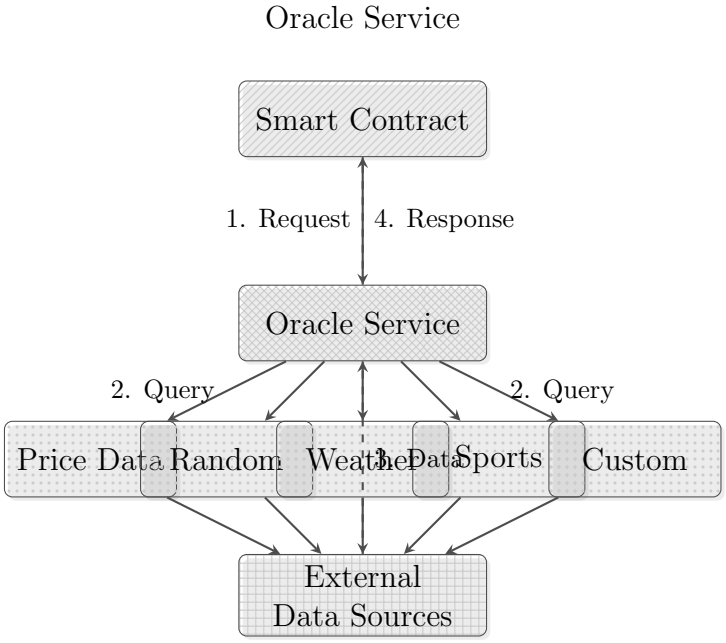

Figure 6: Oracle Service Architecture

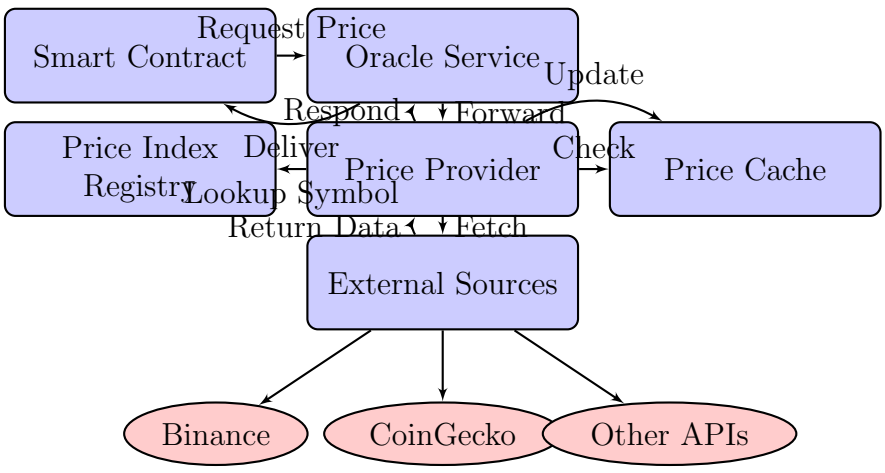

Figure 7: Price Data Feed System in Neo Service Layer

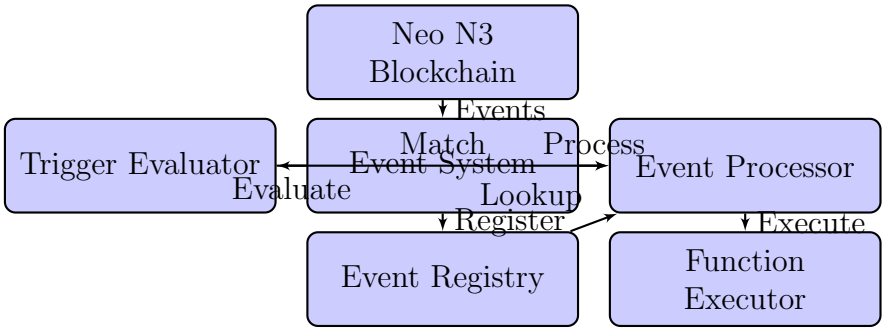

Figure 8: Neo Service Layer Event Messaging System

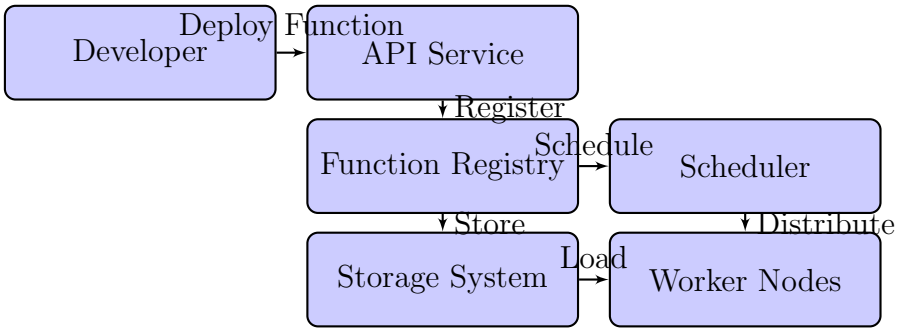

Figure 9: Function Deployment Process in Neo Service Layer

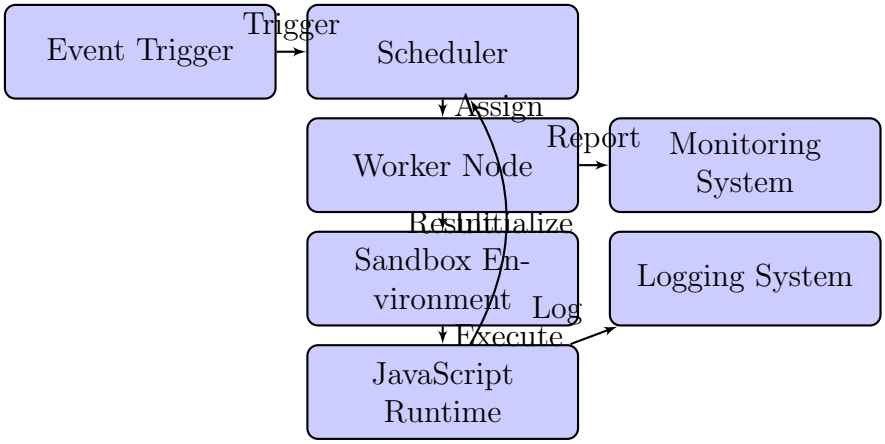

Figure 10: Function Execution Flow in Neo Service Layer

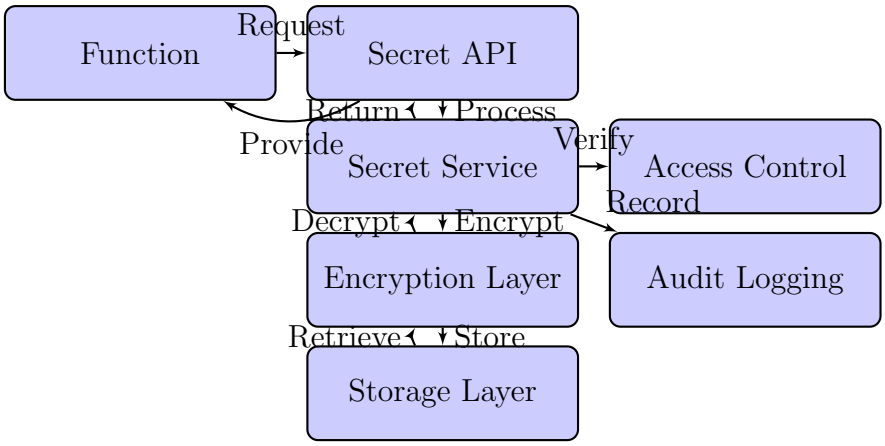

Figure 11: Secret Storage System in Neo Service Layer

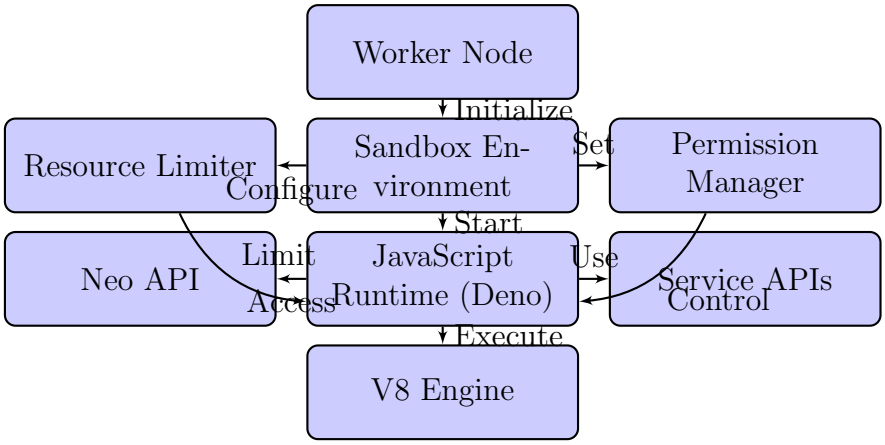

Figure 12: Sandbox Architecture in Neo Service Layer

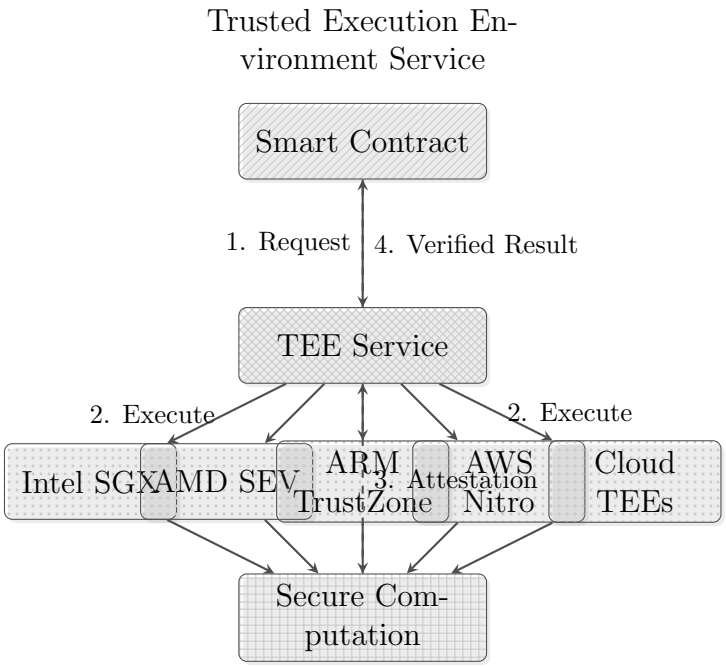

Figure 13: TEE Service Architecture

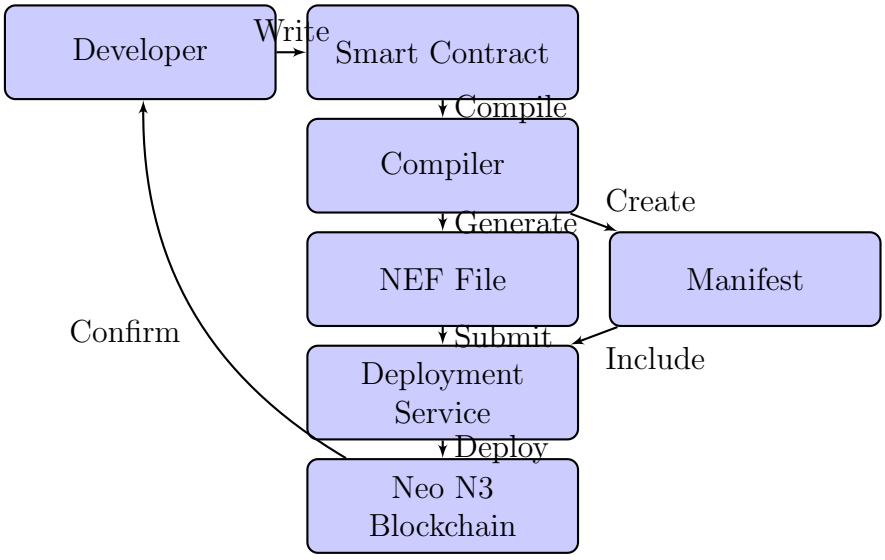

Figure 14: Smart Contract Deployment in Neo Service Layer

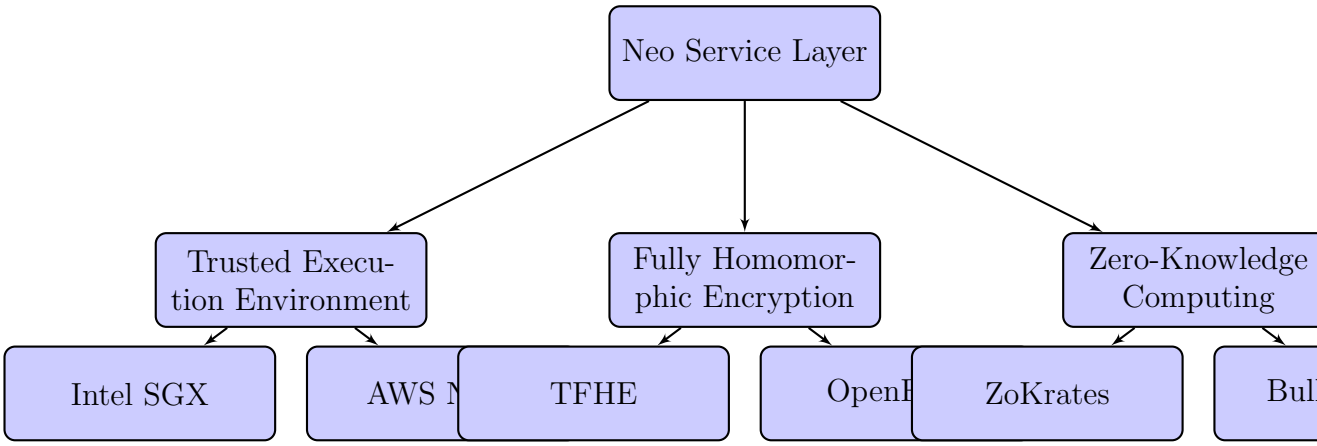

Figure 15: Advanced Cryptographic Services in Neo Service Layer

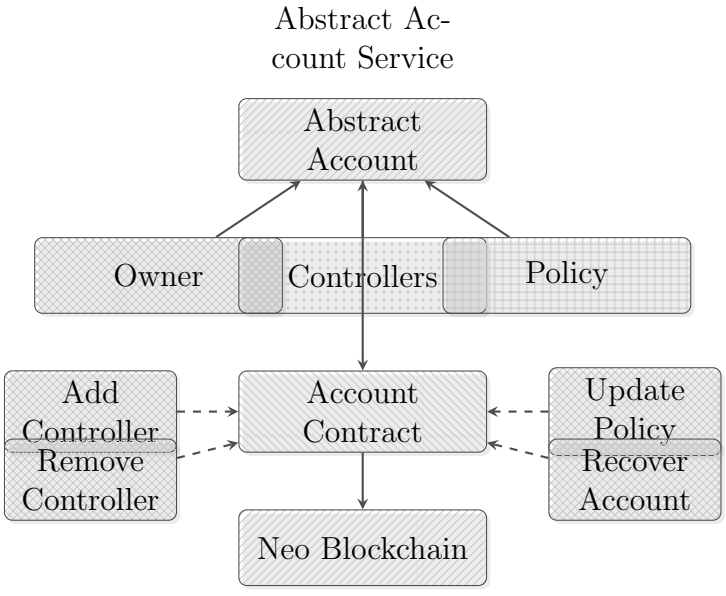

Figure 16: Abstract Account Service Architecture

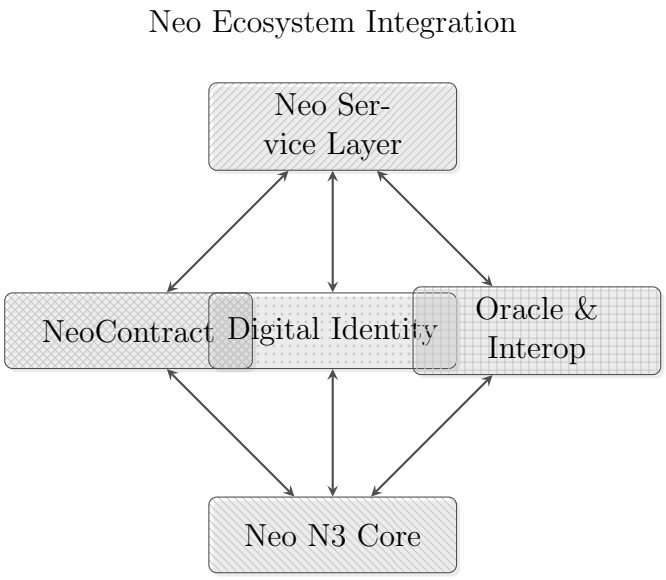

Figure 17: Neo Service Layer Integration with Neo Ecosystem