VANNA FINANCE

Smart Contract Security Audit Report

Vanna Protocol V1 — Soroban (Stellar)

6	8	5	7	1	27
CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL	TOTAL FINDINGS

Prepared for	Platform	Report Date	Version
Vanna Finance	**Stellar / Soroban**	**July 2026**	**1.0**

1. Executive Summary

Security assessment of the Vanna Protocol V1 smart contracts for Soroban

This report presents the results of a security review of the Vanna Protocol V1 smart contracts, a margin-trading and lending protocol built on Soroban, the smart contract platform of the Stellar network. The review covered the protocol's core components, including the lending pools and vToken share accounting, the account manager and smart accounts, the risk engine, the oracle integration (Reflector), the interest rate model, the registry, and the protocol token contracts.

The assessment identified **27 findings**: **6 Critical**, **8 High**, **5 Medium**, **7 Low**, and **1 Informational**. The Critical findings include missing authorization on vToken supply changes, liquidation of healthy accounts due to a missing health check, a health-factor calculation that counts debt as collateral, a re-entrancy-lock conflict that makes accounts with external yield positions impossible to liquidate, arbitrary external contract execution through smart accounts, and double-counting of Blend collateral. Each of these directly threatens user funds or protocol solvency and should be remediated before deployment.

Findings are presented in Section 4, ordered by severity. Each finding includes a detailed description of the issue, its impact on the protocol, and a concrete recommendation for remediation.

SEVERITY CLASSIFICATION

SEVERITY	COUNT	DEFINITION
CRITICAL	6	Direct loss or theft of user funds, broken protocol solvency guarantees, or complete failure of a core security mechanism. Must be fixed before deployment.
HIGH	8	Significant risk to user funds or protocol accounting under realistic conditions. Should be fixed before deployment.
MEDIUM	5	Limited or conditional impact on funds or accounting; weakens protocol guarantees. Should be remediated.
LOW	7	Minor issues affecting robustness, consistency of controls, or integration behavior with no direct risk to funds.
INFORMATIONAL	1	Code-quality and maintainability observations with no security impact.

DISCLAIMER

This report describes issues identified during a time-boxed review of the source code made available at the time of the assessment. It is not a guarantee of the absence of vulnerabilities, nor an endorsement of the protocol's business model. Blockchain technology and smart contracts remain an area of active research; new attack vectors may be discovered after the publication of this report. A security audit is one component of a defense-in-depth strategy and should be complemented by testing, monitoring, and operational controls.

2. Summary of Findings

27 findings, ordered by severity

ID	TITLE	SEVERITY
C1	Missing Pool Authorization in `VToken::burn_from()` Enables Unauthorized Supply Manipulation and Breaks Pool Accounting	CRITICAL
C2	Missing Health Validation in `AccountManagerContract::liquidate()` Allows Unauthorized Liquidation of Healthy Accounts	CRITICAL
C3	Users Can Borrow More Than Their Collateral Should Allow	CRITICAL
C4	Accounts with External Yield Positions Cannot Be Liquidated	CRITICAL
C5	Missing Validation of External Protocol Addresses Allows Arbitrary Contract Execution	CRITICAL
C6	Duplicate Collateral Accounting Allows Users to Borrow Against the Same Assets Multiple Times	CRITICAL
H1	Missing Validation for Non-Positive Oracle Prices Leads to Incorrect Protocol Valuation	HIGH
H2	Missing Oracle Freshness Validation Allows Stale Prices to Be Used	HIGH
H3	Missing Oracle Price Validation Can Lead to Incorrect Borrow and Health Calculations	HIGH
H4	First Depositor Can Manipulate Share Pricing and Dilute Future Deposits	HIGH
H5	LP Collateral Is Not Properly Valued During Risk Assessment	HIGH
H6	Borrow State Is Updated Before Verifying Available Pool Liquidity	HIGH
H7	Incorrect `totalAssets` Calculation Results in Excessive vToken Minting	HIGH
H8	Accounts with Outstanding Debt Can Be Incorrectly Marked as Debt-Free	HIGH

2. Summary of Findings (continued)

ID	TITLE	SEVERITY
M1	Incorrect Share Accounting During Withdrawals May Lead to Accounting Inconsistencies	MEDIUM
M2	Missing Slippage Validation May Result in Unfavorable Asset Swaps	MEDIUM
M3	Trusting External Router Return Values May Result in Incorrect LP Accounting	MEDIUM
M4	Interest Rounding Favors Borrowers, Reducing Protocol Revenue	MEDIUM
M5	Missing Zero-Share Validation May Result in Incorrect Borrow Accounting	MEDIUM
L1	Zero-Balance Assets Remain Tracked After Complete Withdrawal	LOW
L2	Missing Balance Validation Before Repayment Results in Unnecessary Transaction Failures	LOW
L3	Frozen Accounts Can Continue Transferring Tokens	LOW
L4	Unsafe Integer Conversion May Result in Incorrect Arithmetic	LOW
L5	Frozen Accounts Can Continue Receiving Newly Minted Tokens	LOW
L6	Unsupported Assets Can Prevent Successful Account Settlement	LOW
L7	Getter Functions Revert Instead of Returning Zero for Non-Existent Borrow Positions	LOW
I1	Function Naming Inconsistency Reduces Code Readability	INFORMATIONAL

Full details for each finding, including impact analysis and remediation guidance, are provided in Section 4.

3. Scope

Components of the Vanna Protocol V1 Soroban codebase covered by this review

COMPONENT	CONTRACT	RESPONSIBILITY
Lending Pool	`LendingPoolContract`	Deposits, withdrawals, borrowing, repayment, and pool accounting
vToken	`VTokenContract / VToken`	Interest-bearing share token representing proportional pool ownership
Account Manager	`AccountManagerContract`	Margin account lifecycle, execution routing, liquidation, and settlement
Smart Account	`SmartAccountContract`	Per-user collateral custody and privileged external protocol execution
Risk Engine	`RiskEngineContract`	Collateral valuation, health factor computation, and borrow validation
Oracle	`OracleContract`	Price feeds sourced from the Reflector oracle
Rate Model	`RateModelContract`	Interest accrual and rate calculations
Registry	`RegistryContract`	Protocol address registry and configuration
Token	`TokenContract`	Protocol token with administrative freeze controls

METHODOLOGY

The review combined manual line-by-line inspection of the contract source code with invariant-driven analysis of the protocol's economic model. Particular attention was paid to: authorization and access control on supply-changing and privileged operations; the correctness of collateral valuation, health factor, and liquidation logic in the risk engine; oracle failure modes (missing, stale, zero, and negative prices); share-based accounting invariants in the lending pool and vToken contracts (including inflation and dilution attacks); rounding and fixed-point precision behavior; and the interaction of re-entrancy guards with internal protocol flows.

4. Detailed Findings

Each finding includes a description of the issue, its impact, and remediation guidance.

FINDING C1

Missing Pool Authorization in `VToken::burn_from()` Enables Unauthorized Supply Manipulation and Breaks Pool Accounting

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`VToken::burn_from()`

DESCRIPTION

The `VToken::burn_from()` function allows any vToken holder to burn their own shares without requiring authorization from the associated lending pool. Since vTokens represent a lender's proportional ownership of the pool's assets, the lending pool must remain the sole authority responsible for managing the token supply. Every mint and burn operation is expected to occur as part of the pool's deposit and withdrawal lifecycle so that the pool's internal accounting remains synchronized with the outstanding share supply.

By allowing arbitrary token holders to invoke `burn_from()` directly, the protocol breaks this fundamental accounting invariant. A user can permanently reduce the circulating vToken supply without the lending pool updating its corresponding internal state. This desynchronization causes the pool's recorded shares and the actual token supply to diverge, corrupting exchange-rate calculations and invalidating assumptions made throughout the lending protocol.

Because the lending pool relies on the vToken supply to determine ownership, withdrawals, and asset distribution, unauthorized supply manipulation can result in incorrect redemption amounts, unfair allocation of pool assets, denial of withdrawals for legitimate lenders, or permanent accounting inconsistencies that threaten the solvency of the lending pool. Since this vulnerability compromises a core protocol invariant and directly impacts custody and distribution of user funds, it should be classified as **Critical**.

RECOMMENDATION

Restrict `VToken::burn_from()` so that it can only be invoked by the corresponding lending pool or another explicitly authorized protocol contract. End users should never be able to arbitrarily modify the vToken supply outside the controlled deposit and withdrawal flow.

Additionally, enforce the invariant that every supply-changing operation is accompanied by a corresponding update to the lending pool's internal accounting. Keeping the vToken supply and pool state synchronized at all times preserves correct share accounting, exchange-rate calculations, and the integrity of lender fund distribution.

FINDING C2

Missing Health Validation in `AccountManagerContract::liquidate()` Allows Unauthorized Liquidation of Healthy Accounts

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`AccountManagerContract::liquidate()`

DESCRIPTION

The `AccountManagerContract::liquidate()` function does not verify whether a margin account is actually undercollateralized before executing the liquidation process. Instead, the function proceeds as long as the account has an outstanding debt position, without confirming that the account's health factor has fallen below the required liquidation threshold.

As a result, any account with an active borrow position, including those that remain sufficiently collateralized, can be liquidated. This allows healthy users to have their collateral seized despite satisfying the protocol's collateralization requirements, resulting in an unauthorized and irreversible loss of user funds.

Because liquidation transfers collateral from borrowers to liquidators, failing to validate account health breaks one of the protocol's most fundamental security guarantees. An attacker can target solvent positions and force premature liquidation, causing direct theft of collateral and financial loss for honest users. Since this vulnerability enables unauthorized seizure of user assets and compromises the integrity of the protocol's liquidation mechanism, it should be classified as **Critical**.

RECOMMENDATION

The `AccountManagerContract::liquidate()` function should validate the target account's health through the Risk Engine before initiating liquidation. Liquidation should only be permitted when the account's collateral value falls below the protocol's required health threshold, ensuring that only genuinely undercollateralized positions are eligible for liquidation.

Additionally, the liquidation eligibility check should be performed immediately before executing any state-changing operations to prevent invalid liquidations caused by stale or manipulated state. Enforcing this invariant guarantees that collateral can only be seized from accounts that legitimately qualify for liquidation.

FINDING C3

Users Can Borrow More Than Their Collateral Should Allow

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`RiskEngineContract::is_borrow_allowed()`

DESCRIPTION

The `RiskEngineContract::is_borrow_allowed()` function incorrectly incorporates the borrower's existing outstanding debt into the collateral side of the health factor calculation when evaluating a new borrow request. Rather than computing the health factor exclusively from the account's collateral value relative to its total liabilities, the implementation adds the existing debt to the numerator, effectively treating outstanding liabilities as additional collateral.

This calculation violates a fundamental invariant of the protocol's risk model: debt must only reduce an account's health, never improve it. Consequently, every subsequent borrow performed by an account with an existing debt position artificially increases its calculated health factor instead of decreasing it. An attacker can repeatedly borrow against the same collateral while continuously satisfying the protocol's collateralization checks, despite becoming increasingly undercollateralized.

Because the protocol relies on this validation to enforce maximum borrowing capacity, this flaw allows borrowers to obtain significantly more assets than their collateral should permit. The resulting bad debt may exceed the value of the underlying collateral, leaving the lending pool with unrecoverable losses even after liquidation. Since this vulnerability directly compromises the protocol's primary solvency guarantee and can lead to permanent loss of lender funds, it should be classified as **Critical**.

RECOMMENDATION

The borrow validation logic should compute the health factor using only the account's collateral value against its total outstanding liabilities, including the newly requested borrow amount. Existing debt must never contribute positively to the collateral valuation, as doing so fundamentally breaks the protocol's collateralization model and overestimates a borrower's borrowing capacity.

Additionally, ensure the health factor calculation follows the invariant that increasing debt can only decrease (or at best leave unchanged) an account's health. Enforcing this invariant preserves accurate borrow validation, prevents excessive leverage, and protects the lending pool from the accumulation of bad debt.

FINDING C4

Accounts with External Yield Positions Cannot Be Liquidated

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`AccountManagerContract::liquidate()`

DESCRIPTION

The `AccountManagerContract::liquidate()` function automatically unwinds a user's external Blend and LP positions before transferring the remaining collateral to the liquidator. However, the liquidation flow invokes the public `execute()` function while the account's reentrancy lock is already held.

Since `execute()` attempts to acquire the same reentrancy lock and also enforces trader authorization intended for externally initiated transactions, the internal call always fails whenever the account holds an external Blend or LP position. Consequently, the liquidation transaction reverts before the external positions can be unwound, preventing the protocol from recovering collateral from these accounts.

This creates a denial-of-service condition for a class of undercollateralized borrowers. Any account utilizing supported external yield strategies becomes effectively immune to liquidation despite falling below the required collateralization threshold. As debt continues accruing while liquidation remains impossible, the protocol can accumulate unrecoverable bad debt, directly threatening lender funds and overall protocol solvency. Because this vulnerability prevents enforcement of the protocol's primary risk mitigation mechanism, it should be classified as **Critical**.

RECOMMENDATION

The liquidation flow should avoid invoking the public `execute()` entry point during internal protocol operations. Instead, implement a dedicated internal execution routine that bypasses redundant authorization and reentrancy checks while preserving the underlying business logic required to unwind external positions.

Additionally, separate externally accessible entry points from internal execution paths so that protocol-controlled operations can safely compose functionality without conflicting with security mechanisms designed exclusively for user-initiated transactions. This ensures that all undercollateralized accounts remain liquidatable regardless of whether their collateral is deployed into external yield strategies.

FINDING C5

Missing Validation of External Protocol Addresses Allows Arbitrary Contract Execution

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`AccountManagerContract::execute()`, `SmartAccountContract::execute()`

DESCRIPTION

The `AccountManagerContract::execute()` function forwards user-supplied execution requests to the associated `SmartAccountContract`, which subsequently performs privileged interactions with external protocols on behalf of the user's margin account. However, the implementation does not validate whether the supplied target address corresponds to a trusted protocol registered by the protocol before forwarding the external call.

As a result, a malicious or unintended contract can be specified as the execution target and invoked with the Smart Account's privileges. Because the Smart Account maintains custody of the user's collateral and is authorized to manage protocol-owned assets on behalf of the account, arbitrary contract execution breaks the protocol's trust boundary. A malicious target contract may exploit these privileges to manipulate state, initiate unauthorized asset transfers, or perform unexpected operations against the Smart Account's managed assets.

This vulnerability effectively extends the Smart Account's authority to any user-specified contract without enforcing protocol-level trust assumptions. Since privileged execution against arbitrary contracts can directly compromise user collateral and bypass the intended integration model, the issue represents a critical security risk capable of resulting in theft or permanent loss of funds.

RECOMMENDATION

The protocol should strictly validate every external execution target against a whitelist of approved protocol addresses maintained by the Registry contract before forwarding the call to the `SmartAccountContract`. Any address that is not explicitly registered as a trusted integration should immediately cause the transaction to revert.

Additionally, separate trusted protocol interactions from arbitrary contract execution and ensure that all privileged Smart Account operations can only invoke audited integrations explicitly approved by governance. Enforcing a strict allowlist preserves the protocol's trust boundary and prevents privileged execution from being delegated to untrusted contracts.

FINDING C6

Duplicate Collateral Accounting Allows Users to Borrow Against the Same Assets Multiple Times

SEVERITY

CRITICAL

AFFECTED FUNCTION(S)

`RiskEngineContract::get_total_balance()`

DESCRIPTION

The `RiskEngineContract::get_total_balance()` function incorrectly accounts for collateral deposited into Blend by including the same economic position multiple times during collateral valuation. Specifically, both the tracking token representation and the corresponding underlying collateral position are included when calculating an account's total collateral value, causing a single deposit to be counted more than once.

This violates one of the protocol's fundamental accounting invariants: each unit of collateral must contribute exactly once to an account's borrowing capacity. By double-counting the same economic position, the Risk Engine artificially inflates the collateral available to a borrower, causing the calculated health factor to significantly exceed its true value.

As a result, users with Blend positions can appear substantially overcollateralized and obtain additional leverage beyond the protocol's intended collateralization limits. Because borrow validation and liquidation eligibility both depend on the inflated collateral value, the protocol may approve loans that should be rejected while delaying or entirely preventing liquidation of genuinely undercollateralized positions. The resulting bad debt can exceed the available collateral, directly threatening lender funds and the protocol's overall solvency. Since this vulnerability fundamentally compromises the correctness of collateral accounting and enables borrowing against the same assets multiple times, it should be classified as **Critical**.

RECOMMENDATION

Each collateral position should have a single authoritative source of truth during valuation. The Risk Engine should ensure that collateral represented through tracking tokens is not simultaneously valued through its underlying asset position. Every economic position must contribute exactly once to an account's total collateral regardless of how it is represented internally.

Additionally, centralize collateral valuation logic so that all collateral types follow a consistent accounting model that prevents duplicate inclusion across protocol integrations. Enforcing a one-to-one mapping between economic positions and collateral value preserves accurate health factor calculations, borrow validation, and liquidation behavior while preventing users from borrowing against the same assets multiple times.

FINDING H1

Missing Validation for Non-Positive Oracle Prices Leads to Incorrect Protocol Valuation

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`OracleContract::get_latest_price()`

DESCRIPTION

The `OracleContract::get_latest_price()` function retrieves asset prices from the Reflector oracle as signed `i128` values and directly converts them into unsigned integers without validating that the returned price is strictly greater than zero. Consequently, if the oracle returns a negative or zero price due to an upstream malfunction, stale configuration, or unexpected oracle behavior, the protocol continues processing an invalid value as if it were a legitimate market price.

Since the oracle serves as the primary source of truth for asset valuation, every downstream component—including the `RiskEngineContract`, `AccountManagerContract`, and lending pools—relies on this value when evaluating collateral, calculating borrow capacity, determining account health, and validating liquidations. A negative value cast to an unsigned integer can become an extremely large number, while a zero price causes collateral and debt calculations to become mathematically invalid, resulting in incorrect protocol state.

Because the invalid price propagates throughout the entire protocol, an attacker may obtain excessive borrowing power, avoid liquidation, or cause incorrect liquidation of healthy positions. As a result, a single malformed oracle response can compromise the protocol's accounting guarantees and ultimately threaten overall protocol solvency.

RECOMMENDATION

The `OracleContract::get_latest_price()` function should validate that every oracle price is strictly greater than zero before performing any type conversion or returning the value to downstream contracts. If the returned price is zero or negative, the transaction should immediately revert, preventing invalid market data from propagating into the protocol's risk calculations.

Additionally, the protocol should consistently fail closed whenever an oracle response is missing, stale, or invalid. Rejecting malformed price data is significantly safer than allowing downstream contracts to operate on corrupted asset valuations, thereby preserving the integrity of collateral accounting, health factor calculations, and liquidation logic.

FINDING H2

Missing Oracle Freshness Validation Allows Stale Prices to Be Used

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`OracleContract::get_latest_price()`

DESCRIPTION

The `OracleContract::get_latest_price()` function retrieves both the asset price and its associated timestamp from the Reflector oracle. However, it does not verify whether the returned price is sufficiently recent before supplying it to downstream protocol components.

As a result, if the oracle stops updating due to network issues, oracle downtime, or operational failures, the protocol will continue using outdated prices for collateral valuation, borrow validation, health factor calculations, and liquidation decisions. During periods of market volatility, stale prices may cause accounts to be incorrectly evaluated, potentially allowing excessive borrowing, preventing necessary liquidations, or triggering liquidations based on outdated market conditions.

RECOMMENDATION

Validate the timestamp returned by the oracle and reject prices that exceed a predefined maximum staleness threshold. Ensuring that only recent oracle data is used prevents outdated market information from affecting the protocol's risk management and lending operations.

FINDING H3

Missing Oracle Price Validation Can Lead to Incorrect Borrow and Health Calculations

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`RiskEngineContract::get_total_borrows()` / `RiskEngineContract::get_current_total_borrows()`

DESCRIPTION

The `RiskEngineContract::get_total_borrows()` and `RiskEngineContract::get_current_total_borrows()` functions assume that a valid oracle price exists for every supported asset. However, when a price lookup fails, the implementation defaults the asset price to `0` instead of rejecting the operation or treating the missing price as an invalid state.

As a result, borrowed assets without a valid oracle price are excluded from the account's total debt valuation, causing the Risk Engine to underestimate the account's outstanding liabilities. This leads to inaccurate borrow validation, incorrect health factor calculations, and improper liquidation decisions. A borrower may therefore appear significantly healthier than they actually are, allowing additional borrowing or preventing liquidation when the account is in fact undercollateralized.

Although exploitation depends on a missing or misconfigured oracle price feed, silently treating an unavailable price as zero violates the protocol's core accounting assumptions. Since the Risk Engine is responsible for enforcing solvency across the lending protocol, excluding debt from risk calculations can expose lending pools to unrecoverable bad debt and undermine the integrity of the protocol's collateralization model.

RECOMMENDATION

The Risk Engine should require a valid oracle price for every supported asset before performing any borrow, health factor, or liquidation calculation. If a price is unavailable, the operation should immediately revert instead of treating the asset's value as zero.

Additionally, the protocol should adopt a fail-closed approach for all oracle dependencies, ensuring that missing or invalid price data cannot silently influence risk-sensitive calculations. Requiring complete and valid pricing information preserves accurate debt accounting, health factor computation, and liquidation decisions while protecting the protocol from bad debt resulting from incomplete oracle data.

FINDING H4

First Depositor Can Manipulate Share Pricing and Dilute Future Deposits

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`LendingPoolContract::deposit()`

DESCRIPTION

The `LendingPoolContract::deposit()` function calculates the number of shares to mint based on the current ratio between the pool's assets and total share supply. However, when the pool is initialized with little or no liquidity, the share pricing mechanism becomes susceptible to a first-depositor inflation attack.

An attacker can become the initial depositor by depositing a minimal amount of assets and subsequently transferring additional assets directly to the lending pool without minting corresponding shares. Since these donated assets increase the pool's balance without increasing the total share supply, the exchange rate becomes artificially inflated. As a result, subsequent users receive significantly fewer shares than they should for their deposits, effectively transferring ownership of their assets to the attacker.

This attack exploits the absence of protections around the initial share price and allows the first liquidity provider to permanently manipulate the vault's exchange rate. While the attack is limited to newly deployed or completely emptied pools, it can result in substantial losses for early depositors and permanently distort share distribution. Because the vulnerability enables theft of value from future liquidity providers during pool initialization, it should be classified as **High**.

RECOMMENDATION

Protect the initial share pricing mechanism by introducing virtual assets and virtual shares, or by permanently locking a minimum amount of liquidity during pool initialization. These techniques establish a non-manipulable initial exchange rate and prevent unsolicited asset donations from disproportionately increasing the share price.

Additionally, ensure that direct token transfers to the lending pool cannot influence share pricing without minting corresponding shares. Preserving the integrity of the asset-to-share conversion prevents first-depositor inflation attacks and guarantees fair share allocation for all liquidity providers.

FINDING H5

LP Collateral Is Not Properly Valued During Risk Assessment

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`RiskEngineContract::get_current_total_balance()`, `RiskEngineContract::get_total_balance()`

DESCRIPTION

The `RiskEngineContract::get_current_total_balance()` and `RiskEngineContract::get_total_balance()` functions do not correctly account for the value of LP positions when calculating an account's total collateral. Rather than determining the fair market value of the underlying assets represented by an LP position, the implementation either treats these positions as having no value or attempts to rely on a direct oracle price that is unavailable for the LP token itself.

As a result, accounts that supply LP tokens as collateral are evaluated using an inaccurate collateral value during borrow validation, withdrawal checks, health factor calculations, and liquidation decisions. Depending on the implementation path, the protocol may significantly undervalue or entirely ignore LP collateral, causing users to be prevented from borrowing against legitimate collateral or leading to incorrect account health assessments.

Because the Risk Engine depends on accurate collateral valuation to enforce the protocol's solvency guarantees, incorrectly pricing LP positions can produce invalid borrowing decisions, premature liquidations, or delayed liquidation of genuinely undercollateralized accounts. Over time, these accounting errors can accumulate into protocol bad debt while undermining the correctness of every risk-sensitive operation involving LP collateral.

RECOMMENDATION

LP positions should be valued according to their proportional ownership of the underlying pool reserves rather than relying on a direct oracle price for the LP token itself. The Risk Engine should determine the quantity of each underlying asset represented by the LP position, retrieve the oracle price for each asset, and compute the aggregate value before performing any borrow, withdrawal, health factor, or liquidation calculation.

Additionally, the protocol should ensure that all collateral types are evaluated through a consistent valuation framework so that risk calculations accurately reflect the true economic value of every supported asset. This preserves correct collateral accounting, improves liquidation accuracy, and protects the lending pools from losses caused by incorrect LP valuation.

FINDING H6

Borrow State Is Updated Before Verifying Available Pool Liquidity

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`LendingPoolContract::borrow()`

DESCRIPTION

The `LendingPoolContract::borrow()` function updates the borrower's debt position and the lending pool's internal accounting before verifying that sufficient liquidity is available to satisfy the requested borrow amount. Consequently, the protocol performs state transitions prior to confirming that the borrow operation can be successfully completed.

If the lending pool lacks sufficient available liquidity or the asset transfer subsequently fails, the borrow request cannot be completed even though portions of the protocol's accounting have already been processed. Updating protocol state before validating execution prerequisites violates the checks-effects-interactions principle and increases the risk of inconsistent accounting, particularly when execution is interrupted by downstream failures.

Because borrow accounting is one of the protocol's core financial invariants, processing state changes before confirming liquidity availability can leave the protocol in an unexpected intermediate state or introduce inconsistencies between recorded debt and actual asset transfers. Such inconsistencies may affect future borrowing, repayment, interest accrual, and liquidation logic, ultimately weakening the protocol's accounting guarantees.

RECOMMENDATION

The contract should verify that the lending pool has sufficient available liquidity before updating any borrow-related state. All prerequisite validations, including liquidity availability and any conditions required for a successful asset transfer, should be completed before modifying protocol accounting.

Additionally, structure the borrow flow so that all validation logic executes before any state mutation, ensuring that protocol accounting changes occur only when the borrow operation can be completed successfully. Following this ordering preserves accounting consistency and adheres to established secure smart contract design practices.

FINDING H7

Incorrect `totalAssets` Calculation Results in Excessive vToken Minting

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`LendingPoolContract::deposit()`, `VTokenContract::convert_to_shares()` (or the internal `totalAssets` calculation)

DESCRIPTION

The lending pool calculates the number of vTokens to mint for each deposit based on the pool's `totalAssets` value. However, the current implementation only considers the assets physically held by the lending pool while excluding assets that have been borrowed by users but remain owned by the protocol as outstanding loans.

As borrowers withdraw liquidity from the pool, the reported `totalAssets` decreases despite the protocol's overall economic value remaining unchanged. Because the share conversion logic relies on this underestimated asset value, new deposits are issued more vTokens than they should receive for the same deposit amount. Consequently, new depositors obtain a disproportionately large ownership stake in the lending pool at the expense of existing lenders.

This violates the core accounting invariant that the total vToken supply must accurately represent proportional ownership of all protocol-owned assets, including outstanding loans and accrued interest. Over time, repeated deposits made while liquidity is borrowed can significantly dilute existing lenders, distort the asset-to-share exchange rate, and cause permanent inconsistencies between vToken ownership and the protocol's true economic value.

RECOMMENDATION

The lending pool's `totalAssets` calculation should include every asset economically owned by the protocol rather than only the liquidity currently available within the pool. This includes idle liquidity, all outstanding borrowed principal, accrued interest, and any other assets that remain part of the lending pool's balance sheet.

Additionally, ensure that every share conversion function derives its exchange rate from the protocol's complete economic value instead of its immediately available liquidity. Maintaining an accurate `totalAssets` value preserves fair share issuance, prevents dilution of existing lenders, and guarantees that vToken ownership remains proportional to the protocol's underlying assets throughout the lending lifecycle.

FINDING H8

Accounts with Outstanding Debt Can Be Incorrectly Marked as Debt-Free

SEVERITY

HIGH

AFFECTED FUNCTION(S)

`AccountManagerContract::liquidate()`

DESCRIPTION

The `AccountManagerContract::liquidate()` function unconditionally clears an account's debt status by invoking `set_has_debt(false)` after completing the repayment loop, without verifying that every outstanding borrow position has been fully repaid. If a repayment leaves residual debt due to rounding, partial repayment, accrued interest, or any other condition that prevents complete settlement, the account may continue to hold outstanding liabilities while being marked internally as debt-free.

This violates a fundamental accounting invariant of the lending protocol: the account's debt status must accurately reflect whether any liabilities remain outstanding. By relying solely on completion of the liquidation flow rather than the actual debt balances, the protocol can desynchronize its bookkeeping from the underlying financial state.

As a result, subsequent protocol operations that depend on the `has_debt` flag may incorrectly assume the account has no remaining liabilities. Residual debt may therefore bypass debt-related validation, allow margin accounts to be closed or recycled prematurely, interfere with future liquidation eligibility, or leave borrow positions effectively orphaned within the protocol. These inconsistencies undermine debt accounting and can ultimately contribute to unrecoverable bad debt if outstanding liabilities are no longer tracked or enforced.

RECOMMENDATION

The protocol should clear an account's debt status only after verifying that every outstanding borrow position has been completely repaid. Following liquidation, the implementation should explicitly confirm that no remaining debt exists before invoking `set_has_debt(false)`.

Additionally, derive the debt status directly from the account's outstanding borrow positions whenever practical rather than maintaining it as an independently managed flag. Basing the account state on the actual debt balances eliminates the possibility of bookkeeping inconsistencies and ensures that liquidation, account lifecycle management, and borrow validation always operate on accurate debt information.

FINDING M1

Incorrect Share Accounting During Withdrawals May Lead to Accounting Inconsistencies

SEVERITY

MEDIUM

AFFECTED FUNCTION(S)

`LendingPoolContract::withdraw()`

DESCRIPTION

The `LendingPoolContract::withdraw()` function burns vTokens and updates the pool's internal accounting based on the requested withdrawal amount rather than the amount of assets actually transferred to the user after rounding and asset conversion. Because asset transfers may be subject to precision loss during decimal conversion, the amount ultimately transferred can differ slightly from the value used to update the protocol's accounting.

As a result, the lending pool's recorded asset balance and outstanding vToken supply may gradually diverge from the protocol's true economic state. Although the discrepancy introduced by a single withdrawal is typically limited to rounding precision, repeated deposits and withdrawals can accumulate these accounting differences over time, resulting in inaccurate asset-to-share conversion rates and degraded accounting precision.

While this issue is unlikely to directly result in an immediate loss of funds, persistent accounting drift can negatively affect share pricing, interest calculations, and future deposits or withdrawals. Maintaining precise synchronization between protocol accounting and actual asset balances is essential to preserving the correctness of the lending pool's financial state.

RECOMMENDATION

The withdrawal logic should perform all accounting updates using the final amount of assets actually transferred to the user rather than the originally requested withdrawal amount. The completed transfer amount should serve as the single source of truth for both asset accounting and share redemption.

Additionally, ensure that every asset conversion and rounding operation is completed before any accounting variables are updated. Basing protocol accounting on finalized transfer values eliminates cumulative rounding drift and guarantees that the lending pool's recorded balances remain consistent with its actual holdings over time.

FINDING M2

Missing Slippage Validation May Result in Unfavorable Asset Swaps

SEVERITY

MEDIUM

AFFECTED FUNCTION(S)

`AccountManagerContract::execute()`

DESCRIPTION

The `AccountManagerContract::execute()` function forwards swap operations to integrated decentralized exchanges without enforcing a minimum acceptable output amount. Consequently, the protocol assumes that the execution price returned by the external protocol is always acceptable, regardless of market conditions or price movement occurring between transaction submission and execution.

Without slippage validation, swap transactions become vulnerable to adverse price movement and miner/maximal extractable value (MEV) attacks, including sandwich attacks. An attacker can manipulate the market immediately before the swap executes, causing the transaction to settle at a significantly less favorable exchange rate than anticipated. As a result, users may receive substantially fewer assets than expected, unnecessarily reducing the value of their collateral or trading proceeds.

Although this issue does not directly compromise protocol solvency, it weakens execution guarantees for users and exposes protocol operations to avoidable value loss whenever swaps are executed under volatile market conditions. Proper slippage protection is a fundamental safeguard for any protocol interacting with decentralized exchanges.

RECOMMENDATION

Require every swap operation to specify a minimum acceptable output amount (`minAmountOut`) and verify that the actual assets received satisfy this threshold before completing the transaction. If the received amount falls below the specified minimum, the transaction should revert.

Additionally, propagate user-defined slippage tolerances throughout the execution flow and ensure that all integrated DEX interactions enforce these limits consistently. This prevents excessive price impact, protects users from MEV-based manipulation, and guarantees that swaps execute only within acceptable pricing bounds.

FINDING M3

Trusting External Router Return Values May Result in Incorrect LP Accounting

SEVERITY

MEDIUM

AFFECTED FUNCTION(S)

`AccountManagerContract::execute()`

DESCRIPTION

The `AccountManagerContract::execute()` function relies on the liquidity amount returned by an external router after adding liquidity and subsequently uses this value to update the user's collateral position. However, the implementation accepts the router's returned value without independently verifying the number of LP tokens actually received by the Smart Account.

This introduces an unnecessary trust dependency on an external integration. If the router returns an incorrect value due to an implementation bug, unexpected behavior, integration mismatch, or future protocol upgrade, the protocol's internal accounting may no longer reflect the Smart Account's actual LP token balance. Consequently, users may receive more or fewer tracking tokens than warranted, causing collateral balances maintained by the protocol to diverge from the underlying assets actually held.

Although exploitation generally depends on faulty behavior from an integrated external protocol rather than a direct vulnerability in the protocol itself, inaccurate LP accounting can propagate throughout the Risk Engine, affecting collateral valuation, health factor calculations, borrowing capacity, and liquidation decisions. Verifying externally reported values before incorporating them into protocol accounting is essential to maintaining accurate financial state.

RECOMMENDATION

The protocol should independently determine the amount of LP tokens received by measuring the Smart Account's LP token balance before and after the liquidity operation rather than relying exclusively on the value returned by the external router.

Additionally, treat externally returned values as untrusted input whenever they influence protocol accounting. Deriving accounting updates from observed on-chain balances instead of external return values ensures that collateral records always reflect the assets actually owned by the Smart Account and prevents inconsistencies caused by unexpected router behavior.

FINDING M4

Interest Rounding Favors Borrowers, Reducing Protocol Revenue

SEVERITY

MEDIUM

AFFECTED FUNCTION(S)

`RateModelContract::calculate_interest()`

DESCRIPTION

The `RateModelContract::calculate_interest()` function computes accrued interest using integer arithmetic that consistently rounds fractional values down during debt accrual. Whenever the calculated interest cannot be represented exactly due to fixed-point precision limitations, the implementation truncates the fractional component, systematically reducing the amount of interest added to the borrower's outstanding debt.

Although the rounding error introduced by any individual calculation is typically insignificant, the effect compounds across every active borrow position and every interest accrual event throughout the lifetime of the protocol. Over time, this systematic downward bias causes borrowers to underpay interest relative to the intended interest model, reducing lender yield, understating outstanding debt, and gradually decreasing protocol revenue.

While the issue does not immediately threaten protocol solvency, persistent under-accrual of interest weakens the protocol's economic model by transferring value from lenders to borrowers. Ensuring unbiased interest calculations is essential for maintaining accurate debt accounting, predictable yield generation, and long-term financial correctness.

RECOMMENDATION

Interest calculations should adopt a rounding strategy that preserves the protocol's intended economic model and avoids introducing a systematic bias toward either borrowers or lenders. Where precision loss is unavoidable, rounding behavior should be deterministic and should not consistently reduce accrued debt.

Additionally, consider using higher-precision fixed-point arithmetic or accumulating fractional interest across accrual periods before applying rounding. These approaches minimize cumulative precision loss while ensuring accurate debt growth, fair interest distribution, and consistent protocol revenue over time.

FINDING M5

Missing Zero-Share Validation May Result in Incorrect Borrow Accounting

SEVERITY

MEDIUM

AFFECTED FUNCTION(S)

`LendingPoolContract::lend_to()`

DESCRIPTION

The `LendingPoolContract::lend_to()` function calculates the number of borrow shares to mint for a newly created borrow position but does not verify that the computed `borrow_shares_wad` is greater than zero before updating the lending pool's accounting. Unlike the corresponding repayment logic implemented in `collect_from()`, which explicitly handles zero-share conditions, the borrow flow proceeds without validating that a valid share representation has been created.

As a result, when the calculated borrow shares round down to zero due to fixed-point precision limitations or an extremely small borrow amount, the protocol may successfully issue borrowed assets without minting the corresponding borrow shares. This violates the fundamental accounting invariant that every outstanding debt position must be represented by an equivalent amount of borrow shares.

Although the issue generally requires edge-case borrow amounts, permitting debt to exist without an associated share representation can lead to inconsistencies in interest accrual, repayment accounting, debt tracking, and pool utilization calculations. Maintaining a one-to-one correspondence between borrow balances and borrow shares is essential for preserving the correctness of the lending pool's accounting model.

RECOMMENDATION

The `lend_to()` function should verify that the calculated `borrow_shares_wad` is greater than zero before updating any borrow-related accounting or transferring assets to the borrower. If the computed share amount is zero, the transaction should immediately revert.

Additionally, enforce the invariant that every borrow operation must create a corresponding borrow share representation. Applying the same validation logic used by `collect_from()` ensures consistent handling of precision edge cases throughout the borrowing lifecycle and preserves the integrity of the protocol's debt accounting.

FINDING L1

Zero-Balance Assets Remain Tracked After Complete Withdrawal

SEVERITY

LOW

AFFECTED FUNCTION(S)

`SmartAccountContract::remove_collateral()` *(or the equivalent collateral management function)*

DESCRIPTION

The `SmartAccountContract::remove_collateral()` function updates a user's collateral balance following a withdrawal but does not remove the corresponding asset entry when the remaining balance becomes zero. Consequently, collateral assets with no remaining balance continue to be tracked as active positions within the Smart Account.

Although this behavior does not directly affect protocol security, solvency, or collateral accounting, it causes obsolete asset entries to persist in storage after they are no longer relevant. Subsequent operations that iterate over a user's collateral positions must continue processing these zero-balance entries despite their inability to influence borrow limits, health factor calculations, or liquidation eligibility.

Over time, the accumulation of stale collateral entries increases unnecessary storage consumption and introduces avoidable computation during account-related operations. While the impact is limited to operational efficiency and storage hygiene, removing inactive entries improves maintainability and ensures that account state accurately reflects the user's active collateral positions.

RECOMMENDATION

The collateral management logic should automatically remove an asset from the tracked collateral list whenever its balance becomes zero following a withdrawal or collateral reduction.

Additionally, ensure that any auxiliary data structures used for collateral enumeration remain synchronized when entries are removed. Maintaining only active collateral positions improves storage efficiency, reduces unnecessary iteration during protocol operations, and keeps account state consistent with the user's actual holdings.

FINDING L2

Missing Balance Validation Before Repayment Results in Unnecessary Transaction Failures

SEVERITY	AFFECTED FUNCTION(S)
LOW	`LendingPoolContract::repay()`

DESCRIPTION

The `LendingPoolContract::repay()` function attempts to transfer repayment assets from the borrower before verifying that the payer possesses a sufficient token balance to satisfy the requested repayment amount. Instead of performing an explicit balance check, the implementation relies on the underlying token transfer operation to fail when the account lacks sufficient funds.

Although the repayment transaction ultimately reverts and no incorrect protocol state is introduced, the absence of an upfront validation causes unnecessary external contract interactions and results in generic token transfer failures rather than protocol-specific error conditions. This makes transaction failures less informative for users, wallets, and protocol integrations while also performing avoidable execution before detecting an invalid repayment request.

The issue primarily affects usability, developer experience, and execution efficiency rather than protocol security or financial correctness. Providing deterministic validation before interacting with external token contracts improves error handling and aligns the repayment flow with established smart contract design practices.

RECOMMENDATION

The repayment logic should verify that the payer possesses a sufficient token balance before initiating any external token transfer. Performing this validation as part of the protocol's precondition checks ensures that invalid repayment requests fail immediately with a clear, protocol-defined error.

Additionally, complete all internal validation before interacting with external contracts to reduce unnecessary execution and improve the predictability of transaction failures. This approach enhances integration with wallets and off-chain applications while preserving efficient and deterministic protocol behavior.

FINDING L3

Frozen Accounts Can Continue Transferring Tokens

SEVERITY

LOW

AFFECTED FUNCTION(S)

`TokenContract::transfer()`

DESCRIPTION

The `TokenContract::transfer()` function does not verify whether the sender or recipient has been marked as frozen before processing token transfers. Although the protocol provides an administrative mechanism for freezing accounts, the transfer logic does not enforce this restriction, allowing frozen accounts to continue participating in token transfers.

As a result, the account freeze mechanism cannot reliably prevent the movement of tokens from or to restricted accounts. This weakens the effectiveness of administrative controls and undermines assumptions that frozen accounts are incapable of transferring assets while subject to protocol-imposed restrictions.

While this issue does not directly threaten protocol solvency or allow unauthorized minting or burning of tokens, it reduces the effectiveness of operational controls used for incident response, compliance requirements, or emergency intervention. Consistently enforcing freeze restrictions across all transfer operations is necessary to preserve the intended security properties of the token contract.

RECOMMENDATION

The transfer logic should validate the freeze status of all relevant accounts before executing any token transfer. If either the sender or recipient is marked as frozen according to the protocol's intended policy, the transaction should immediately revert.

Additionally, ensure that every token movement pathway—including direct transfers and any transfer helper functions—enforces the same freeze checks. Centralizing freeze validation guarantees consistent administrative behavior and preserves the intended security and operational guarantees of the account freeze mechanism.

FINDING L4

Unsafe Integer Conversion May Result in Incorrect Arithmetic

SEVERITY

LOW

AFFECTED FUNCTION(S)

`OracleContract::get_latest_price()` *(or the equivalent function performing integer conversion)*

DESCRIPTION

The `OracleContract::get_latest_price()` function performs integer type conversions without explicitly validating that the source value falls within the representable range of the destination type. If a converted value exceeds the bounds supported by the target integer type, the conversion may overflow, underflow, or otherwise produce an unintended result depending on the language and conversion semantics.

Although the current implementation may rely on trusted oracle inputs and protocol assumptions, omitting explicit range validation weakens the robustness of the protocol against unexpected inputs, future code changes, or integration with additional oracle providers. Defensive validation around integer conversions is particularly important for price-related calculations because incorrect values can propagate throughout downstream accounting and risk-management logic.

The issue is primarily defensive in nature, as exploitation generally depends on upstream assumptions being violated. Nevertheless, validating numeric bounds before performing type conversions improves protocol resilience and prevents unexpected arithmetic behavior if invalid values are ever introduced.

RECOMMENDATION

Validate that every integer value falls within the valid range of the destination type before performing any type conversion. If the value cannot be represented safely, the operation should immediately revert rather than proceeding with an invalid conversion.

Additionally, prefer checked conversion routines or language-provided safe conversion utilities wherever available. Explicitly validating conversion boundaries preserves arithmetic correctness, prevents unexpected numeric behavior, and improves the overall robustness of protocol calculations.

FINDING L5

Frozen Accounts Can Continue Receiving Newly Minted Tokens

SEVERITY

LOW

AFFECTED FUNCTION(S)

`TokenContract::mint()`

DESCRIPTION

The `TokenContract::mint()` function does not verify whether the recipient account has been marked as frozen before issuing newly minted tokens. Although the protocol provides an administrative account freeze mechanism intended to restrict interactions with designated accounts, this restriction is not enforced during token issuance.

As a result, newly minted tokens can continue to be issued to frozen accounts despite the protocol explicitly classifying those accounts as restricted. This creates inconsistent enforcement of the freeze mechanism, as frozen accounts are prevented from certain operations while remaining eligible to receive newly created tokens.

While this issue does not directly threaten protocol solvency or allow unauthorized minting, it weakens the effectiveness of the protocol's administrative controls and may interfere with incident response, regulatory compliance, or emergency operational procedures. Administrative restrictions should be enforced consistently across every token lifecycle operation to preserve the intended semantics of account freezing.

RECOMMENDATION

The mint operation should validate the recipient's freeze status before issuing any new tokens. If the recipient is marked as frozen, the transaction should revert instead of completing the mint operation.

Additionally, centralize freeze validation across all token operations—including minting, burning, and transfers—to ensure that administrative restrictions are applied consistently throughout the token contract. Uniform enforcement improves operational reliability and preserves the intended security guarantees of the account freeze mechanism.

FINDING L6

Unsupported Assets Can Prevent Successful Account Settlement

SEVERITY

LOW

AFFECTED FUNCTION(S)

`AccountManagerContract::settle_account()`

DESCRIPTION

The `AccountManagerContract::settle_account()` function assumes that every asset associated with a margin account is registered and supported by the protocol. During settlement, the implementation expects each asset to be recognized and processed successfully. If an unsupported or unregistered asset is encountered, the function reverts instead of handling the condition gracefully.

As a result, the presence of a single unsupported asset can prevent the entire settlement process from completing, even when all remaining collateral and debt positions are valid. This unnecessarily blocks account settlement and forces otherwise recoverable positions to remain unresolved until the invalid asset is removed or manually corrected.

Although this issue does not directly expose protocol funds or compromise accounting correctness, it reduces the robustness and fault tolerance of the settlement process. A single invalid asset entry should not prevent unrelated assets from being processed successfully, particularly during administrative or recovery operations.

RECOMMENDATION

The settlement logic should gracefully handle unsupported or unregistered assets encountered during iteration. Invalid entries should either be skipped or processed through an appropriate fallback mechanism without interrupting settlement of the remaining valid positions.

Additionally, consider validating asset registration when collateral is initially added to an account so that unsupported assets cannot enter the settlement pipeline. Combining preventative validation with resilient settlement logic improves protocol robustness and ensures isolated asset inconsistencies do not prevent successful account resolution.

FINDING L7

Getter Functions Revert Instead of Returning Zero for Non-Existent Borrow Positions

SEVERITY

LOW

AFFECTED FUNCTION(S)

`LendingPoolContract::get_borrow_balance()`, `LendingPoolContract::get_total_borrow_shares()`

DESCRIPTION

The `LendingPoolContract::get_borrow_balance()` and `LendingPoolContract::get_total_borrow_shares()` functions retrieve borrow share records using `unwrap()`, assuming that the requested entry always exists. However, accounts that have never borrowed do not possess an associated borrow share record. In these cases, the lookup returns `None`, causing the subsequent `unwrap()` call to panic and revert the execution.

For accounts without outstanding debt, the absence of a borrow share record represents a valid protocol state rather than an exceptional condition. Reverting in this scenario unnecessarily complicates integrations and forces callers to handle avoidable failures when querying debt information for accounts that simply have no borrow history.

Although this issue does not affect protocol security or accounting correctness, it reduces the robustness of the contract's read interface and introduces unnecessary failure cases for frontends, indexers, SDKs, and other protocol integrations. Getter functions should return sensible default values whenever possible instead of reverting for expected states.

RECOMMENDATION

The getter functions should treat missing borrow share records as representing a zero borrow balance rather than an exceptional condition. Replace `unwrap()` with `unwrap_or(0)` (or an equivalent default-value mechanism) so that accounts without borrow history return a debt balance of zero.

Additionally, ensure that all read-only query functions gracefully handle uninitialized storage whenever a missing record represents a valid protocol state. Returning deterministic default values improves developer experience, simplifies integrations, and produces a more resilient query interface.

FINDING I1

Function Naming Inconsistency Reduces Code Readability

SEVERITY

INFORMATIONAL

AFFECTED FUNCTION(S)

`RegistryContract::get_xlm_contract_adddress()`

DESCRIPTION

The `RegistryContract::get_xlm_contract_adddress()` function contains a typographical error in its name (`adddress` instead of `address`). Although this issue does not affect the runtime behavior or security of the protocol, inconsistent naming conventions reduce code readability and make the contract interface more difficult for developers to understand and maintain.

Typographical inconsistencies in public APIs can also propagate into downstream integrations, documentation, SDKs, and developer tooling. Maintaining clear and consistent naming conventions improves overall code quality, reduces cognitive overhead during development, and minimizes the likelihood of implementation mistakes when interacting with protocol contracts.

RECOMMENDATION

Rename the function to follow the protocol's established naming convention (for example, `get_xlm_contract_address()`) and update all internal references, interfaces, documentation, and downstream integrations accordingly.

Additionally, consider incorporating automated linting or code review checks to detect naming inconsistencies during development. Enforcing consistent naming conventions improves maintainability, provides a clearer developer experience, and results in a more professional and self-documenting codebase.