PickYesNo

PREDICTION MARKET
WHITE PAPER

PickYesNo is an open source, decentralized prediction market platform built on the Polygon PoS blockchain, dedicated to ensuring 100% user asset safety under any circumstance — being hacked, insider malfeasance, or platform shutdown.

Leveraging Polygon PoS's fairness, transparency, and immutability, PickYesNo employs a robust suite of core smart contracts — Wallet Contract, Prediction Contract and Oracle Contract — to create a secure and trustworthy framework for asset management and interaction. PickYesNo prioritizes user trust and asset security above all, even at the cost of higher gas fees, enforcing stringent safety measures.

This whitepaper details the technical architecture and security mechanisms of PickYesNo, showcasing its resilience against threats and unwavering commitment to protecting user assets.

GLOSSARY

Smart Contract: Self-executing code deployed on the blockchain, managing user-platform interactions per predefined logic, ensuring transparency and immutability. Trustless, on-chain certainty, no middlemen.

Wallet Contract: Core contract for securely managing user assets, equipped with robust mechanisms to prevent unauthorized access or loss. Your funds, always under your control.

Prediction Contract: Enables users to trade on specific event outcomes, ensuring fair settlement and profit distribution based on correct results. On-chain predictions, dispute-free.

Oracle Chainlink Contract: An oracle contract based on on-chain price data provided by Chainlink, specifically used to deliver results for cryptocurrency-related predictions.

Oracle Contract: A contract that brings external data onto the blockchain, providing reliable and tamper-resistant real-world event outcomes for prediction contracts beyond cryptocurrencies.

Factory Contract: Generates Wallet and Prediction Contracts, ensuring consistent and secure contract code. Standardized deployment, no room for vulnerabilities.

EOA (Externally Owned Account): Blockchain account controlled by a private key, bound to Wallet Contracts to secure assets. Your keys, your control.

Gas Fees: Computational cost for executing transactions or smart contract operations on the blockchain. PickYesNo optimizes gas usage while prioritizing security, covering nearly all gas costs for users. No gas worries for you.

User Assets: USDC held by users on PickYesNo, protected by smart contracts against loss or unauthorized access. Your USDC, locked tight.

Hack: Unauthorized intrusion by external attackers targeting platform infrastructure or smart contracts to steal or manipulate assets. PickYesNo's decentralized design and rigorous security thwart such threats.

Insider Malfeasance: Dishonest actions by insiders attempting to misappropriate user assets. PickYesNo'stransparent, decentralized architecture eliminates this risk.

Platform Shutdown: Scenario where the platform ceases operations. PickYesNo's smart contracts ensure users retain full control and access to their assets. Platform down, your funds stay yours.

Blockchain Immutability: The unalterable nature of blockchain data, guaranteeing the integrity of transaction history and contract logic. On-chain records, forever untouchable.

INTRODUCTION

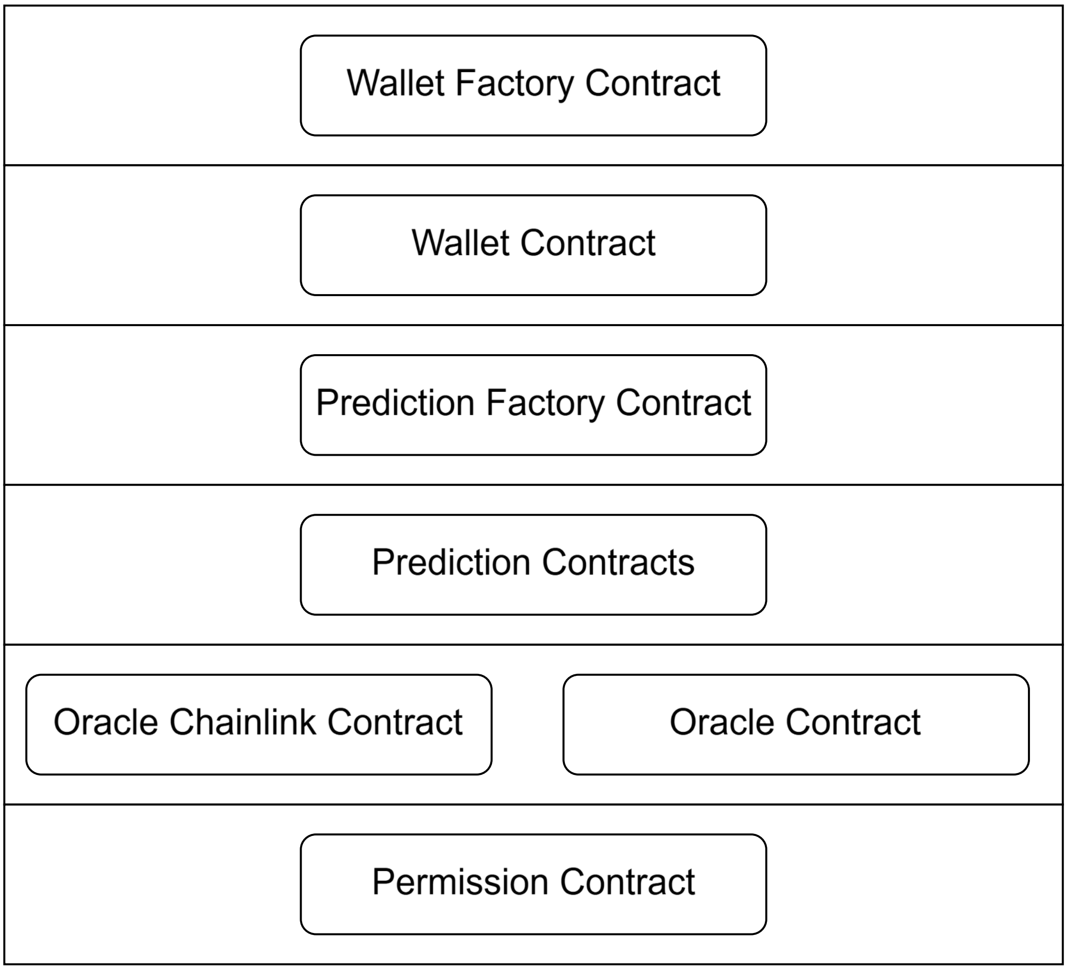
PickYesNo leverages blockchain technology to guarantee the absolute security of user assets. To achieve this, we've meticulously engineered our smart contracts to ensure zero loss of user assets under any extreme scenario, including but not limited to:

1. **Platform hacks**
2. **Insider malfeasance**
3. **Platform shutdown**

PickYesNo doesn't just "put things on-chain for the sake of it." We harness the fairness, transparency, and immutability of blockchain to safeguard user assets, even at the cost of higher gas fees, all to deliver a platform users can trust unequivocally.

Below, we dive into a technical breakdown of our smart contracts to verify whether we truly deliver on these security promises.

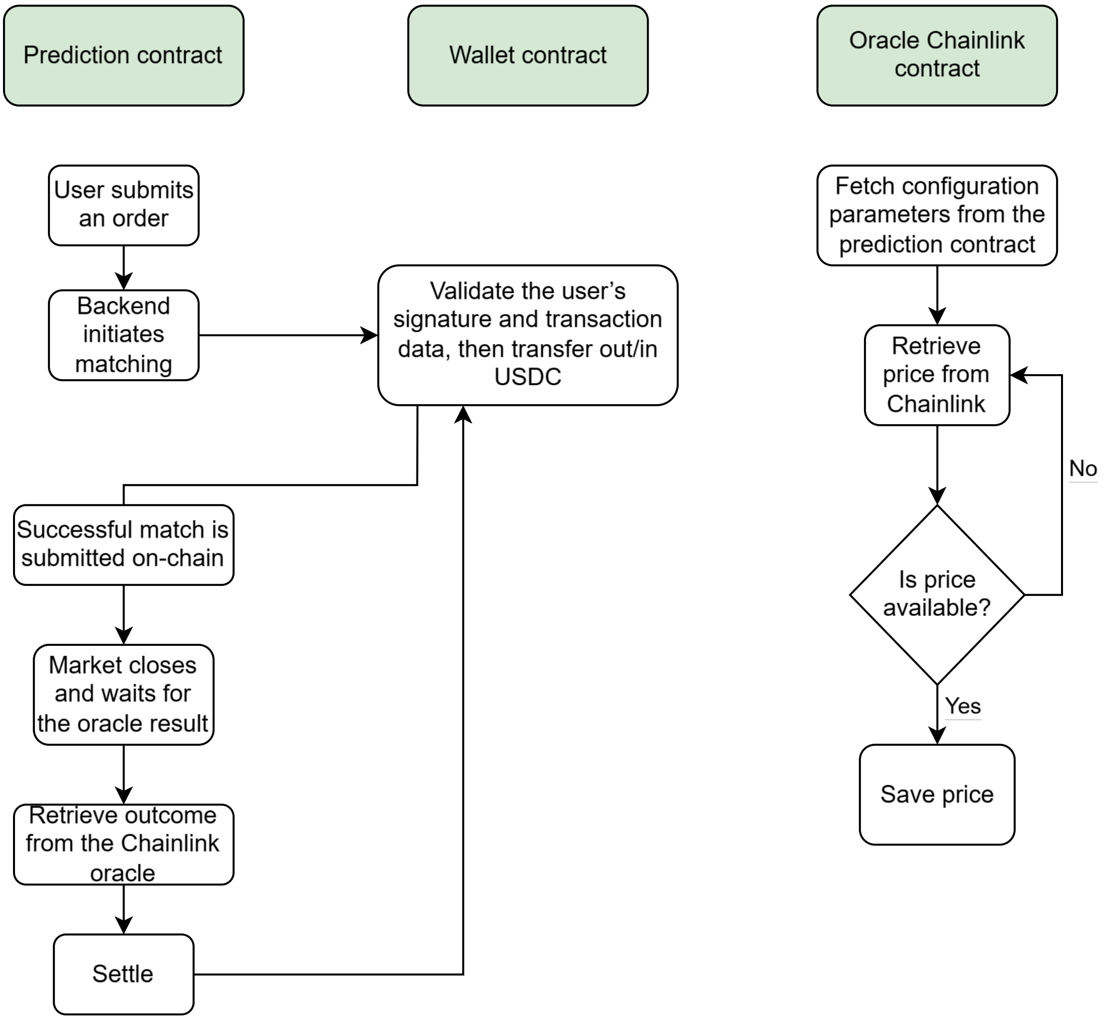
PickYesNo's core smart contract suite includes the **Wallet Contract, Prediction Contract and Oracle Contract**.

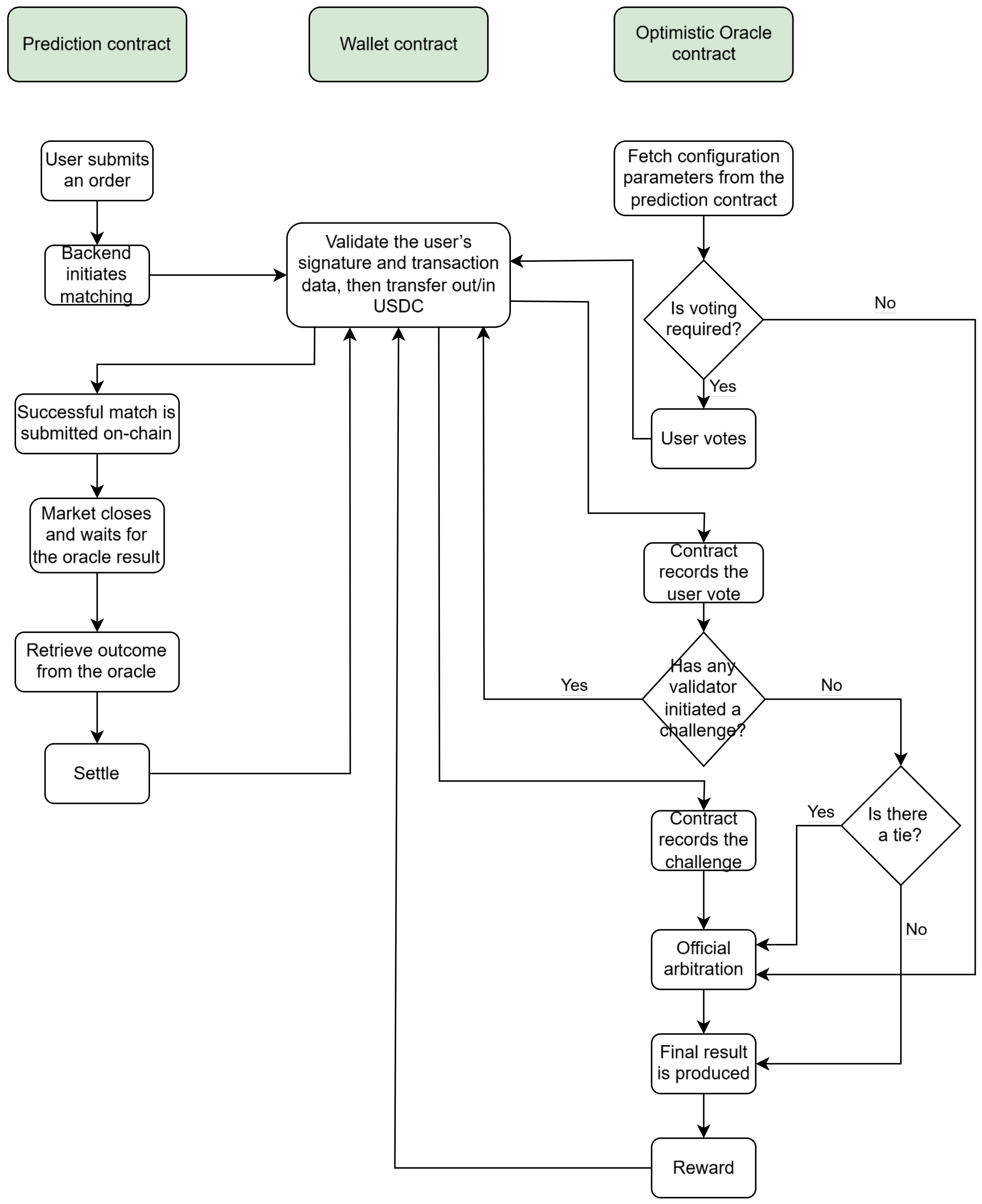

Wallet Factory Contract
Wallet Contract
Prediction Factory Contract
Prediction Contracts
Oracle Chainlink Contract
Oracle Contract
Permission Contract

Overall Process Overview

The following is an overview of how the PickYesNo platform operates—from the creation of a prediction market, order placement, order matching, to the oracle producing the final result and completing settlement.

Cryptocurrency prediction markets and general prediction markets use different types of oracles. The former relies on Chainlink oracles, while the latter uses a voting-based optimistic oracle. The prediction contracts and wallet contracts are identical in both cases. These contracts are described in greater detail in their respective sections.


Prediction contract
Wallet contract
Oracle Chainlink contract
User submits an order
Backend initiates matching
Validate the user’s signature and transaction data, then transfer out/in USDC
Successful match is submitted on-chain
Market closes and waits for the oracle result
Retrieve outcome from the Chainlink oracle
Settle
Fetch configuration parameters from the prediction contract
Retrieve price from Chainlink
Is price available?
No
Yes
Save price


Prediction contract
Wallet contract
Optimistic Oracle contract
User submits an order
Backend initiates matching
Validate the user's signature and transaction data, then transfer out/in USDC
Fetch configuration parameters from the prediction contract
Is voting required?
No
Yes
User votes
Successful match is submitted on-chain
Market closes and waits for the oracle result
Contract records the user vote
Has any validator initiated a challenge?
Yes
No
Retrieve outcome from the oracle
Settle
Contract records the challenge
Yes
Is there a tie?
No
Official arbitration
Final result is produced
Reward

FACTORY WALLET CONTRACT

Before diving into wallet contracts, it's essential to understand PickYesNo's factory wallet contract. Each call to this contract deploys a new wallet contract at a new address; all wallet contracts are generated by it, ensuring identical code and security. We only invoke it when a new user creates an account.

If you discover that your wallet contract was not generated by the official factory wallet contract, something is definitely wrong! In such a case, immediately stop interacting with that wallet contract and contact us as soon as possible.

For your security, please ensure your wallet factory contract matches the one shown on our official website. This is the only way to guarantee your assets are protected by our security protocols. Please visit https://pickyesno.com/docs/contract/contract-address for the official factory wallet contract address.

WALLET CONTRACT

For optimal security, we strongly recommend that users immediately link their allocated wallet contract to an Externally Owned Account (EOA) where they control the private keys upon registration. This step is crucial for unlocking the highest level of security protection offered by PickYesNo. Detailed instructions for this binding process can be found in our "**Binding Your Own EOA**" guide.

PickYesNo is committed to fostering the widespread adoption of Web3 applications, making it easy for people who have never been exposed to blockchain to get started and enjoy the convenience of Web3. To do this, we've meticulously designed our wallet contracts so that users can safely interact with the chain without needing any

POL (Polygon's native token). This is true even though it means the platform has to cover the extra gas fees.

BINDING YOUR WALLET

Binding your wallet is the most critical step to ensure your asset security. This involves linking your EOA, where you control the private keys. Once bound, your wallet contract falls under the control of your EOA.

If you're not concerned with technical details, simply follow these steps when registering your account to ensure 100% asset security:

1. Confirm that the wallet contract you are about to bind was generated by the official factory wallet contract. For verification methods, refer to "**Factory Wallet Contract**."
2. Click the [Bind Wallet] function and follow the prompts to link your EOA wallet.
3. Confirm that the third parameter of the WalletBound event emitted by the wallet contract matches the EOA you entered. You can verify the event by following these steps:
   - Go to https://polygonscan.com
   - Look up the transaction hash using your wallet contract address.
   - Use the transaction hash to find the logs.

- Confirm that the third parameter of the WalletBound event matches your bound EOA

4. Once successfully bound, no one other than you can transfer your assets.

After the wallet is bound, any operation that transfers assets out of the wallet requires your signature (following the EIP-712 standard).

Only after verifying that the signature is indeed yours can the transaction proceed. If the signature is invalid, the entire transaction will revert.

Once the wallet is bound, neither the platform nor any attacker can move assets out of your wallet. Only you have the authority to access and utilize the funds inside.

Reference Code:

function name: bindWallet

input:
1. uint256 requestId (Server ID for this transaction)
2. address wallet (User's desired EOA address to bind)
3. bytes signature (Signature of bound EOA)
output:
no return value

The bindWallet function is used to bind the EOA. The initial call must be executed by the platform, with gas fees covered by the platform. Once bound,only the bound EOA can freely invoke the function. If the platform needs to invoke it again in the future, it must provide a signature from the bound EOA to proceed.

Step 1 above, verifying the wallet contract's origin from the official factory wallet contract, is crucial to prevent backend hacks or malicious insiders from providing fake wallet contracts. Given the immutability of blockchain data, this is the safest approach.

For the same reasons, Step 3 requires confirmation of the on-chain event to prevent tampering by hackers or insiders. On-chain logs verification ensures that the EOA actually bound by the contract matches your input. **REBINDING**

If you wish to change the bound EOA after your wallet contract has been successfully bound, you must either manually call the bindWallet function in the wallet contract using your currently bound EOA account, or provide a valid signature to the platform. Only with your signature does the platform have permission to perform the binding operation on your behalf and cover the gas fees for the transaction.

Reference Code:

function name: bindWallet

input:
1. uint256 requestId (Server ID for this transaction. If interacting directly with the smart contract (not via platform), any uint256 can be input without affecting system or account security)
2. address wallet (New EOA address to rebind)
3. bytes signature (Signature of bound EOA)
output:
no return value

DEDUCTION

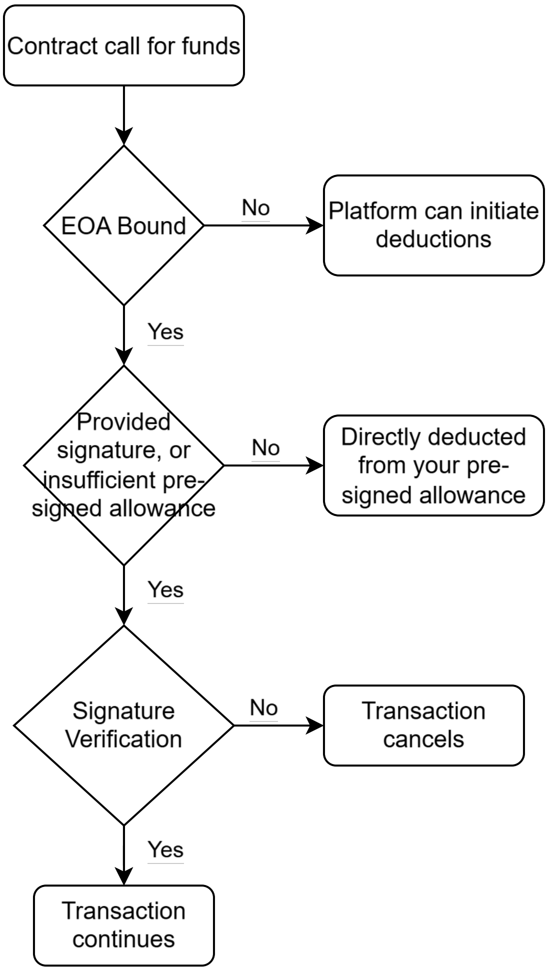
When you use PickYesNo’s services, two types of smart contracts can deduct funds from the wallet contract: Prediction Contracts and Oracle Contracts.

- When **EOA is not bound** , all two types of smart contracts can deduct from the wallet contract.
- Once **EOA is bound**, no one, including the platform's smart contracts, can use your assets in the wallet contract without your consent.

If EOA is bound:

- If a signature is provided or if the pre-signed allowance is insufficient, the system will verify that your signature content matches the current transaction information.
- Conversely, if no signature is provided and the pre-signed allowance is sufficient, the pre-signed allowance will be used.

The deduction flowchart can be summarized as:


Contract call for funds
EOA Bound
No
Platform can initiate deductions
Yes
Provided signature, or insufficient pre-signed allowance
No
Directly deducted from your pre-signed allowance
Yes
Signature Verification
No
Transaction cancels
Yes
Transaction continues

The structure of the signed content (encodedData) varies for different contract types (Prediction, Oracle Contracts). Details can be found in their respective sections.

The **process for verifying signature and content consistency** (applicable to all contracts) is as follows:

1. Parse , , and from the signature.
2. PickYesNo follows **EIP-712**, which allows for the recovery of the sender's address.

3. Compare the recovered address with your bound EOA. If they match, the transaction proceeds; otherwise, the transaction reverts.

This mechanism aligns with Ethereum's standard transaction validation method: verifying the sender's wallet address through signature and data to ensure the transaction is indeed authorized by you.

ENTRUST

You can transfer USDC to the wallet contract without paying any on-chain gas fees. As long as you provide an EIP-2612–compliant signature (r, s, v), the platform will transfer the USDC on your behalf and cover all on-chain gas costs.

Reference Code:

Function name: entrust

Input:
1. uint256 requestId (Used by the server to identify this transaction)
2. address from (The address from which your USDC will be transferred to the wallet contract)
3. uint256 amount (The amount of USDC to be transferred into the wallet
Note that USDC uses 6 decimals. Therefore, if you interact with the contract directly and input 50000 (5×10^4), the actual transferred amount is 0.05 USDC. If you want to transfer 3 USDC, you must input 3000000 (3×10^6))
4. uint256 deadline (The expiration time of the signature in EIP-2612, expressed as a Unix timestamp)
5. uint8 v (The v value of the EIP-2612 signature)
6. bytes32 r (The r value of the EIP-2612 signature)
7. bytes32 s (The s value of the EIP-2612 signature)

Output:

None.

REDEEM

When you use the redeem function, your assets from the wallet contract are transferred to a specified address.

The ability to call this function is limited to your bound EOA and the platform's system. You might be concerned that since the platform can also call the redeem function, there could be a security risk. Here's a breakdown of the two scenarios to explain why your assets are safe:

1. If your **EOA is not yet bound**

   Theoretically, the platform could transfer your assets to any address. Although the platform would never do this, a certain level of security risk does exist in this state.

2. If your **EOA is bound**

   Binding your wallet contract to your EOA completely secures your assets. This is because:

   - If a redemption is initiated by the platform, and the transfer target is not the bound EOA, your signature is required.

Without your signature, the platform can only transfer the redeemed amount to your bound EOA.

- If the redemption is initiated by your bound EOA, no signature is needed, since it can be confirmed that the operation is performed by you personally.

By linking your wallet contract to your EOA, your assets are strictly protected, and only transactions you authorize can be executed, ensuring your assets are completely secure.

<u>Reference Code:</u>

function name: redeem

input:
1. uint256 requestId (Server ID for this transaction. If interacting directly with the smart contract (not via platform), any uint256 can be input without affecting security)
2. address wallet (Target address for USDC transfer. If EOA is bound, this address must equal your bound EOA or signed address, otherwise the function will revert)
3. uint256 amount (Amount of USDC to transfer. Note: USDC has 6 decimals. So, inputting 50000 ($5*10^4$) directly to the contract transfers 0.05 USDC; to transfer 3 USDC, input 3000000 ($3*10^6$))
4. bytes signature (The signature is used to verify that a transaction is from the bound EOA. This is only necessary when an EOA is bound to the wallet contract and the transfer address is different from the bound EOA)

output:
no return value

MARKETING CAMPAIGN BONUSES (present)

Let us first define Assets and Marketing Bonuses. Marketing bonuses can only be used to interact with Prediction Market, or Oracle contracts, whereas Assets can also be redeemed (redemption is explained in a later section).

During certain special campaigns — for example, Lunar New Year or Anniversary events — the platform may distribute marketing bonuses to users' wallets. You may use these funds to participate in prediction markets, or oracle interactions. If you make a profit from these campaigns, the marketing bonus will be converted into your Assets, which can then be freely withdrawn or spent. Conversely, if you incur losses during these activities, your Assets will remain unaffected — only your available marketing bonus will decrease.

It is important to note that changes to your marketing bonus value (whether after a win or a loss) will not take effect immediately. The update must be actively initiated by the platform.

Next, let's talk about the security mechanism of marketing bonuses.

Whenever the platform wants to increase a player's marketing bonus, it must transfer an equivalent amount of USDC to the player's wallet.

This prevents the platform from maliciously inflating marketing bonuses in a way that would reduce the user's redeemable asset value.

Without transferring USDC to your wallet, the platform can only decrease the value of the marketing bonus, or simultaneously decrease the marketing bonus while transferring out the same amount of USDC.

Your assets will not be affected, because the platform can only withdraw the previously transferred marketing bonus funds.

In short, the marketing bonus mechanism ensures that your assets can only increase (when you profit using marketing bonuses) and will never decrease, guaranteeing the full safety of user funds.

Reference Code:

function name: present

input:

1. uint256 requestId (The server uses this to identify the current transaction)

2. uint256 amount (The value by which the marketing bonus will be increased. An equivalent amount of USDC must be transferred into the user's wallet; otherwise, the transaction will revert)

3. address from (The address from which USDC will be transferred into the user's wallet

4. bytes32 code (The marketing code that grants the execution EOA permission to transfer the marketing bonus into your wallet)

5. bytes signature (The signature that grants the execution EOA permission to transfer the marketing bonus into your wallet)

6. uint256 unlocked (When you earn profits using marketing bonuses, the platform will use this parameter to convert your bonus profits into your personal assets)

7. uint256 reclaimed (The platform can use this parameter to reclaim marketing bonuses. It can only decrease the marketing bonus balance and will never reduce your assets)

PRE-SIGNATURE(preSign)

If you have a high level of trust in us and wish to simplify operations by avoiding manual signatures for every transaction, you can use the **Pre-Signature** feature.

By using a pre-signature, you authorize the platform for a specified allowance, enabling it to directly stake your USDC into the corresponding contract when an order is successfully matched or when you participate in voting, without requiring you to sign each transaction individually. Once the authorized limit is used up, you will need to submit a new pre-signature. You can also revoke the **Pre-Signature** by setting the amount to 0.

Here's a key point about pre-signed allowances. The amount deducted from your allowance is often more than the USDC that's actually moved. For example, when you submit a market order on the prediction market and the trade executes at 50 cents, only 50 cents of USDC will be transferred. However, we will deduct 99.9 cents from your pre-signed allowance.

Don't worry, this doesn't result in any loss of assets for you. The USDC amount transferred will always be equal to the actual execution price. The reason we do this is that we can't know the exact final price when you submit a market order. Therefore, we initially deduct the maximum possible transaction price for that prediction market—99.9 cents—to prevent the trade from reverting due to an insufficient pre-signed allowance.

Reference Code:

function name: preSign

input:
1. uint256 requestId (Server ID for this transaction)
2. uint64 amount (Amount of USDC you are willing to authorize. Can be set higher than current wallet balance, but insufficient balance will still cause transaction failure (revert) during actual deduction)
3. bytes signature (Your provided offline signature to verify consent. Each signature includes a unique nonce to prevent replay attacks)

output:
no return value

When your EOA is not bound to the wallet contract, the pre-signed function cannot be called. However, once it's bound, only your EOA and the platform can call it.

No matter who calls the pre-sign function, a valid signature from the bound EOA must be provided to prevent the platform from abusing its privileges or arbitrarily increasing your pre-signed allowance. To prevent replay attacks, the nonce and request details are signed together. After a successful pre-signature, the nonce within the wallet contract automatically increases by one. (+1)

Note that the pre-signed authorization amount can be greater than the current USDC balance in the wallet contract, but during actual use, if the balance is insufficient, the transaction will still fail (revert).

Rescuing ERC-20 Tokens

If you accidentally transfer an ERC-20 token other than USDC into your wallet contract, we provide a rescue token function that allows you to transfer the token out.

For security reasons, not all ERC20 tokens are eligible for rescue. Only suitable tokens will be included in the whitelist. The platform reserves the right to modify the whitelist at any time.

Only the platform's executor EOA and your bound EOA can call this function.

<u>Reference Code:</u>

function name: rescueToken

input:
1. uint256 requestId (Server ID for this transaction)
2. address from (The contract address of the ERC20 token)
3. address to (The address to transfer the tokens to)
output:
no return value

RECYCLING

The main purpose of this mechanism is to protect users who have lost access to their wallets.

If you forget your account password or lose your private key, making it impossible to operate your wallet contract that still contains funds, the platform can, upon verification of ownership, extract the assets from the contract and return them to you.

If the wallet contract does not meet the recovery conditions, even if the platform executes the recycleWallet function, your wallet will not be recycled, and its assets will remain untouched.

For wallets that do meet the recycling conditions, the platform will not execute a recycle unless a “forgotten password” claim has been filed — even though the platform holds the technical authority to do so.

For active users, you do not need to worry about your wallet contract being recycling without cause. Recycling is only possible when the wallet has had no entrust or trading order activity for an extended period. The specific recycling rules are as follows:

- If the wallet contract holds USDC:
  - And there is a history of entrust or trading activity:

    The wallet may be recycled only if more than one year has passed since the last entrust or trading activity.
  - And there is no history of entrust or trading activity:

    The wallet may be recycled only if, within one year after the recycleWallet function is called, there is still no entrust or trading activity.
- If the wallet contract holds no USDC:
  - And there is a history of entrust or trading activity:

    The wallet may be recycled only if more than 180 days have passed since the last entrust or trading activity.
  - And there is no history of entrust or trading activity:

The wallet may be recycled if, within 3 days after the recycleWallet function is called, there is still no entrust or trading activity.***Reference Code:***

function name: recycleWallet

input:
1. uint256 requestId (Server ID for this transaction)

output:
no return value

UPGRADE WALLET

This function provides three capabilities:

(1) upgrading the wallet contract’s logic,

(2) adding a new prediction factory contract, and

(3) adding a new oracle contract.

1. Upgrading the wallet contract logic

PickYesNo’s wallet uses the ERC1967 proxy pattern. If new features are introduced in the future, users may choose whether to upgrade their wallet logic.

Upgrading the logic requires the user’s signature. You may review the open-source upgrade on-chain first, and decide whether to provide your signature.

Because a signature is required, even if the execution EOA’s private key is leaked, a hacker cannot upgrade your wallet to malicious logic without your approval—your funds remain fully protected.

2. Adding a new prediction factory contract

Prediction contracts are able to deduct funds from the wallet contract (see the “Deduction” section for details). But how does the wallet contract determine whether a prediction contract is authorized?

The wallet contract does not track each prediction contract directly, as the number of prediction contracts can be very large. Instead, it only stores **the addresses of factory prediction contracts**. Whenever a factory deploys a prediction contract, the factory records the new contract’s address in the permission contract. Before processing any deduction request, the wallet contract queries the permission contract to verify whether the prediction contract was created by an officially authorized factory. Only prediction contracts generated by legitimate factories are allowed to deduct funds from the wallet.

When a wallet contract is deployed, it comes with one built-in factory prediction contract. Through the upgradeWallet function, you can add additional factory prediction contracts to your authorization list. The code of every new factory is available on-chain for your review. Adding a new factory requires both your signature and validation by the permission contract. This ensures that neither a compromised EOA nor an attacker can arbitrarily add unauthorized factories to your list, providing complete protection over your assets.

3. Adding a new oracle contract

After you authorize it, the oracle contract can deduct funds from the wallet (see the deduction section).

Unlike prediction contracts, only one oracle contract operates at any given time, so a factory pattern is not used. The wallet directly stores the oracle's address.

Through the upgradeWallet function, your wallet will recognize the new oracle contract, whose code will also be publicly available on-chain.

Authorizing a new oracle also requires your signature, ensuring that neither the platform nor attackers can add a malicious oracle—fully protecting user assets.

Reference Code:

function name: upgradeWallet

input:
1. uint256 requestId (Server ID for this transaction)
2. address newImplementation (The address of the new logic of wallet contract. If no update is required for the logic, input address(0))
3. address newPredictionFactory (The address of the new Prediction Factory contract. If no update is required for the Prediction Factory, input address(0))
4. address newOracle (The address of the new Oracle contract. If no update is required for the Oracle, input address(0))
5. bytes signature (The user's signature. If a wallet is bound to an EOA, a signature is required for upgrades)

output:
no return value

PREDICTION FACTORY CONTRACT

Before detailing prediction contracts, it's crucial to understand the prediction factory contract:

1. Each call to the prediction factory contract deploys a new prediction contract at a new address. All prediction contracts are generated by the factory contract, ensuring identical code and security.
2. The platform calls the factory contract whenever a new prediction market needs to be created. It is highly recommended that you verify the prediction contract you interact with was deployed by the official factory contract before trading to ensure fund security.

Please visit https://pickyesno.com/docs/contract/contract-address for the official prediction factory contract address to ensure the safety of your funds.

PREDICTION CONTRACT

Our prediction contract code supports multiple types of prediction markets. You can determine the type of the current prediction market based on the result returned by the getSetting function (see the section below for details).

Different prediction contracts may use different oracle contracts, but all of them are designed to produce results that fully reflect real-world outcomes. You can learn how the various oracles operate in the following sections.For each option, you can only choose "**Yes**" or "**No**." If a market has N options, there will be N "**Yes**" and N "**No**" choices available.

When trading, users don't need to select all options; they can choose only those they believe offer the highest expected returns. You can even select both "Yes" and "No" on the same option, depending on your strategy.

We wish those who develop excellent strategies to achieve greater returns!

PREDICTION CONTRACT RESULT

Prediction contract results are categorized into two types:

- **Single Outcome**

  In the same round, different options are mutually exclusive, and only one result can occur.

  For example: In the 2049 X country presidential election with candidates A, B, and C, only one can win; multiple winners are impossible.

- **Independent Outcome**

  In the same round, different options can occur simultaneously and are not mutually exclusive.

For example: In the 2049 X country election, the outcomes for candidate Y's party in states A, B, and C (win or lose) can vary, resulting in zero, one, two, or three wins

ORDER MATCHING

The transaction process is as follows:

1. User submits an order via the platform.
2. Backend matches counter-orders.
3. Backend calls the broker method to settle the match on-chain, orders are officially established, and assets are transferred.

Let's verify **the 4 major security mechanisms of the transaction:**

- **Can the platform access your assets if matching fails?**

  No! The broker function only executes correctly upon successful matching. If the platform matches inappropriate takers (buyer) and makers (seller), the on-chain transaction will revert.

- **After successful matching, is the on-chain data consistent with the trade data?**

  Yes, and even if there's a discrepancy, it must result in a more favorable price for the user.

- **Are the transaction amounts correct during matching and asset transfer?**

  Yes, transferred amounts are strictly calculated based on the order and user signature information

- **Are assets still secure after transfer?**

  Yes, assets are held securely within the prediction contract.

A crucial point to understand is that the amount deducted from your pre-signed allowance will often be more than the USDC that is actually transferred. For example, if you place a market order in a prediction market and the trade executes at a real price of 50 cents, the platform will only transfer 50 cents of USDC. However, we will deduct 99.9 cents from your pre-signed allowance.

You can rest assured that this will not cause any loss of assets. The amount of USDC transferred will be exactly equal to the actual execution price. Your pre-signed allowance is simply consumed at a faster rate. The reason for this is that when you submit a market order, the final execution amount isn't known yet. By deducting the maximum possible execution price for that market, which is 99.9 cents, we prevent the transaction from reverting due to an insufficient pre-signed allowance.

broker Method Details

Reference code:

function name: broker

input:
1. uint256 requestId (server ID for this transaction)
2. uint256 optionId (ID of the chosen option)
3. Order taker (taker's order information)
4. Order[] maker (array of maker's order information)

output:
no return value

Order Structure Definition

- **uint256 mode**: 1 = Market Order, 2 = Limit Order
- **uint256 buysell**: 1 = Buy Yes, 2 = Buy No, 3 = Sell Yes, 4 = Sell No
- **uint256 expectedPrice**: When mode=1, it represents the expected total order amount for a market order (6 decimal places, in USDC); when mode=2, it represents the expected buy/sell price for a limit order (6 decimal places, in USDC)
- **uint256 expectedShares**: When mode=1, it represents the total amount of the actual transaction amount of the market order (6 decimal places, in USDC); when mode=2, it represents the expected number of shares of the limit order (6 decimal places).
- **uint256 matchedPrice**: Actual execution price (6 decimal places, in USDC)
- **uint256 matchedShares**: Actual number of shares (6 decimal places)
- **uint256 fee: Fee rate** (4 decimal places)
- **address wallet**: User's wallet contract address

- **bytes32 uuid**: Unique identifier of the order
- **bytes signature**: User's signature authorization for the trade

Matching Process

- The order matching is determined by the off-chain system, which matches transactions in the following order:

  1. **Price Priority**

     Highest buy price prioritized to match lowest sell price (i.e., most favorable price for the buyer).

  2. **Time Priority**

     If prices are the same and directions are the same, prioritize by order submission time

- Process taker Order:
- Query each maker, process order matching:
  - Compare if taker and maker match.
  - Process fund transfer and share updates upon success

Order Verification Details:

1. **Orders without signature**: Skip verification
2. **Orders with signature:**

- **Market Order:** Actual execution amount ≤ Expected execution amount
- **Limit Order:** Actual shares ≤ Expected shares, and execution price is favorable to the user (Buy Orders: Execution price ≤ Limit; Sell Orders: Execution price ≥ Limit).

Buying and selling shares both require signature verification to confirm that the action was performed by you. Signature verification uses encodedData, which includes the following fields:

- Contract identifier (bytes32)
- optionId
- mode
- buysell
- expectedPrice
- expectedShares
- fee
- uuid
- chainId
- prediction contract address

These signature parameters ensure that any order submitted by the platform must match your original order data exactly, preventing both replay attacks and cross-chain replay scenarios.

When deducting funds from your wallet contract, the wallet will also verify that the oracle used by the prediction contract is one you have approved. For details on how to add approved oracles, please refer to the upgradeWallet section.

Like other parameters, the fee is also provided off-chain and included in the signed payload. Therefore, **if the fee differs in any way from the value you signed, the entire transaction will revert, ensuring that the platform cannot charge even a single unit more than what you authorized**.

Taker and Maker Matching Rules

1. **Taker buy Yes**
   - Matchable: Maker buy No or sell Yes
   - Matching Conditions:
     - Buy Yes vs Buy No: Prices add up to 1 USDC
     - Buy Yes vs Sell Yes: Prices must be equal
2. **Taker buy No**
   - Matchable: Maker buy Yes or sell No
   - Matching Conditions:
     - Buy No vs Buy Yes: Prices add up to 1 USDC
     - Buy No vs Sell No: Prices must be equal
3. **Taker sell Yes**
   - Matchable: Maker buy Yes or sell No
   - **Matching Conditions:**
     - Sell Yes vs. Buy Yes: Prices must be equal
     - Sell Yes vs. Sell No: Prices add up to 1 USDC

       (Note: Sell Yes with Sell No results in cancelling shares)

4. Taker sell No

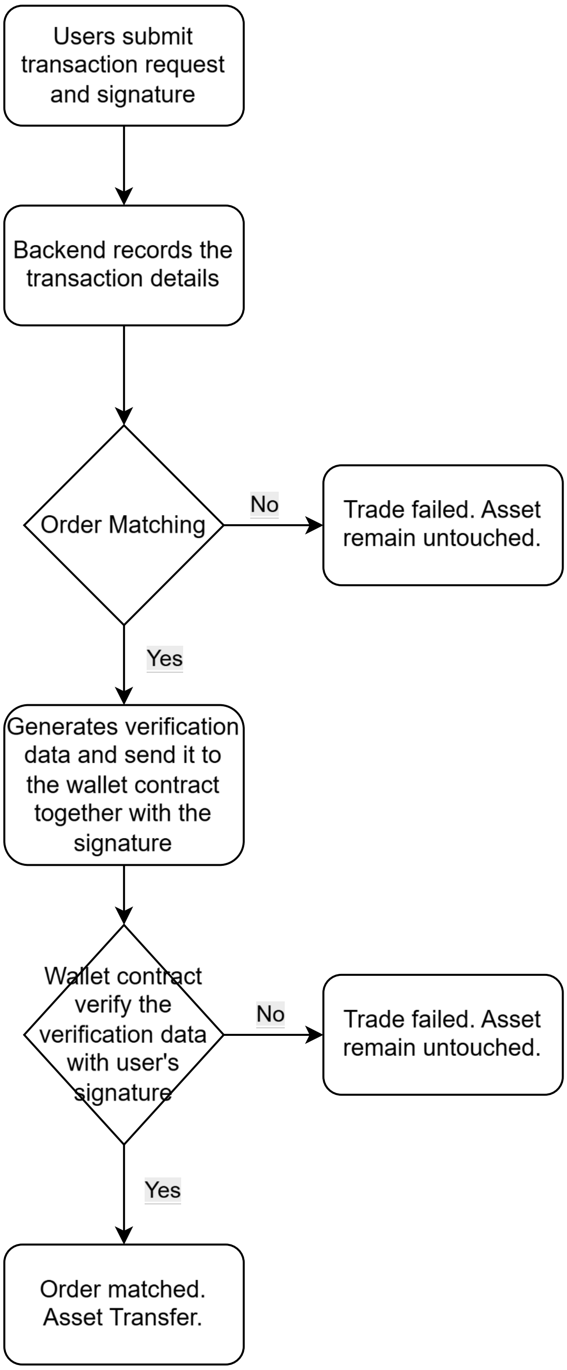
- **Matchable:** Maker buy No or sell Yes
- **Matching Conditions:**
  - Sell No vs. Buy No: Prices must be equal
  - Sell No vs. Sell Yes: Prices add up to 1 USDC

    (Note: Sell Yes with Sell No results in cancelling shares)

Summary

- **Matching failure:** No deduction.
- **Matching success:** Signature verification, trade is favorable to the user.
- **Transfer amount**: Complies with signature authorization.
- **Security after transfer**: Shares and assets are settled synchronously, ensuring risk-free reward redemption.

Order Matching Flowchart


Users submit transaction request and signature
Backend records the transaction details
Order Matching
No
Trade failed. Asset remain untouched.
Yes
Generates verification data and send it to the wallet contract together with the signature
Wallet contract verify the verification data with user's signature
No
Trade failed. Asset remain untouched.
Yes
Order matched. Asset Transfer.

SETTLEMENT (settle)

Once the oracle announces the result, the platform calls the settle method to settle the shares held by users. After settlement, the user's deserved amount is directly transferred from the prediction contract to the user's wallet contract.

Reference code:

function name: settle

input:
1. uint256 requestId (server ID for this transaction)
2. uint256 optionId (prediction option to settle)
3. address[] wallets (array of user wallet addresses to receive profits)

output:
no return value

Settlement Logic

When the settle method is called, the prediction contract gets the result by calling the getOutcome method of the oracle contract. The possible results from the oracle are as follows:

- **Result = 0 (Result not yet available):**

  Result not yet available, the transaction reverts.

- **Result = 1 (Yes)**

  Users receive 1 USDC for every 1 "Yes" share held.

- **Result = 2 (No)**

  Users receive 1 USDC for every 1 "No" share held.

- **Result = 3 (Tie)**

  Both "Yes" or "No" shares receive 0.5 USDC per share.

  (Note: Regardless of the prices at the time, if it's 3 or 5, each share always pays 0.5 USDC)

- **Result = 4 (Pending Arbitration)**

  The transaction reverts.

- **Result = 5 (Prediction Market Canceled)**

  Both "Yes" or "No" shares receive 0.5 USDC per share.

  (Note: Regardless of the prices at the time, if it's 3 or 5, each share always pays 0.5 USDC)

User Self-Settlement

To prevent users from being unable to retrieve their assets in extreme circumstances (e.g., platform service termination), PickYesNo allows users to call the settlement function themselves to retrieve their assets.

This method is identical to the official settlement rules but requires the user to bear the gas fees. Thus, even if the platform terminates service, users can still claim their deserved profits through this method.

Reference code:

function name: settle

input:
1. uint256 optionId (Prediction option to settle)
2. address wallet (Target address for profit distribution, must hold shares for this optionId)

output:
no return value

Platform's Usable Amount

From the above explanation, we see that after settlement, the user's deserved amount is transferred from the prediction contract to the user's wallet contract. These amounts were transferred into the prediction contract from the user's wallet contract during order matching (broker).

Is it possible for the project team to withdraw USDC from the prediction contract before settlement, causing rewards to be undeliverable? PickYesNo’s contract code completely prevents this from happening.

The transferToFee function is the only way to transfer USDC out of the prediction contract besides settlement. From the transferToFee code, you can clearly see that the USDC we transfer cannot exceed the fees threshold. The fees are the service fees explicitly reserved for the project during trade matching.

Therefore, PickYesNo can indeed transfer USDC out of the prediction contract, but only limited to the explicitly defined service fees during matching.

Reference code:

function name: transferToFee

input:
1. uint256 requestId (Server ID for this transaction)
2. uint256 amount (Amount of USDC to transfer from the prediction contract)

output:
no return value

Getting Prediction Market Information

Information for each option in a prediction market is stored on the blockchain, and you can query it by calling the getSetting method.

This is a view method and does not consume any gas fees. It returns the content of a custom data structure called PredictionSetting.

For different types of prediction markets, the inputs and outputs of this function have different meanings. Below, we explain it in two scenarios.

Reference code:

function name: getSetting

Cryptocurrency Prediction Markets

Input:
1. uint256 optionId (In cryptocurrency prediction markets, optionId represents the round index)

Output:
Returns a custom data structure PredictionSetting. In cryptocurrency prediction markets, each field has the following meaning:

1. uint64 optionId (The prediction round index. Each cryptocurrency prediction market differs only in its end time. To avoid deploying redundant contracts, we use optionIds to represent prediction markets with different end times)
2. uint64 roundNo (Not applicable in cryptocurrency prediction markets)
3. uint32 startTime (The base timestamp used to calculate the market end time. The actual end time is calculated as:
startTime + interval * optionId.
After the end time, users can no longer buy or sell shares)
4. uint32 endTime (Not applicable in cryptocurrency prediction markets. If this value is 0, it indicates a cryptocurrency prediction market)
5. uint32 interval (The time interval between each round)
6. uint16 aggregator (Different values represent different cryptocurrencies. In cryptocurrency markets, this value is non-zero)
7. uint16 maxVotes (Not applicable in cryptocurrency prediction markets)
8. uint64 stakingAmount (Not applicable in cryptocurrency prediction markets)
9. uint64 challengeStaking (Not applicable in cryptocurrency prediction markets)

10. uint32 votingDuration (Not applicable in cryptocurrency prediction markets)
11. uint32 challengeDuration (Not applicable in cryptocurrency prediction markets)
12. uint32 totalRewards (Not applicable in cryptocurrency prediction markets)
13. uint8 rewardRanking (Not applicable in cryptocurrency prediction markets)
14. uint8 challengePercent (Not applicable in cryptocurrency prediction markets)
15. bool independent (Not applicable in cryptocurrency prediction markets)

General Prediction Markets

Input:
1. uint256 optionId (prediction option to view information for)

Output:
Returns a custom data structure PredictionSetting. In general prediction markets, each field has the following meaning:

1. uint64 optionId (The prediction option)
2. uint64 roundNo (The prediction round. For example, in a lottery, each draw has the same winning conditions and numbers. To avoid repeatedly deploying highly similar contracts, we introduce the concept of "rounds" within a single contract to represent different lottery periods.
Not all prediction contracts have multiple rounds. For events such as elections, where the interval between events is long and the candidates differ each time, a single contract usually contains only one round, corresponding to a single prediction market)
3. uint32 startTime (Not applicable in general prediction markets. If this value is 0, it indicates a general prediction market)
4. uint32 endTime (The Unix timestamp at which the prediction market ends)
5. uint32 interval (Not applicable in general prediction markets. If this value is 0, it indicates a general prediction market)
6. uint16 aggregator (Not applicable in general prediction markets. If this value is 0, it indicates a general prediction market)
7. uint16 maxVotes (The maximum number of votes allowed per option)8. uint64 stakingAmount (Staked amount for voting. Since USDC has 6 decimals, the number you get needs to be divided by 10^6 to get the actual USDC amount. For example, if the contract returns 10^8, it means 100 USDC)

9. uint64 challengeStaking (Challenge staking amount. Since USDC has 6 decimals, the number you get needs to be divided by 10^6 to get the actual USDC amount. For example, if the contract returns 10^8, it means 100 USDC)
10. uint32 votingDuration (Voting duration (in seconds))
11. uint32 challengeDuration (Challenge duration (in seconds))
12. uint32 totalRewards (Total prize money. Since USDC has 6 decimals, the number you get needs to be divided by 10^6 to get the actual USDC amount. For example, if the contract returns 10^8, it means 100 USDC)
13. uint8 rewardRanking (Ranking threshold for receiving rewards)
14. uint8 challengePercent (Percentage of prize money taken by challenger: 0~100%)
15. bool independent (Indicates whether the outcomes are independent: true means it's an independent outcome, while false means it's a single outcome)

Setting Prediction Market Information

When the platform needs to add a new option or correct a mistake in an existing option's parameters, we use the setSetting function.

The platform cannot just change settings arbitrarily. There are strict limitations:

1. The optionId of any new option must be greater than 0.
2. If the option does not already exist, a new optionId may be added. Its attributes—roundNo, endTime, votingDuration, and challengeDuration—will be set according to the input, while all other parameters will use default values.
3. If the option already exists, it can only be modified when the oracle's getOutcome function returns 0, meaning no result has been finalized. In this case, only three attributes may be updated: endTime, votingDuration, and challengeDuration

Reference code:

function name: setSetting

input:
PredictionSetting is our custom data structure. The meaning of each item in this structure can be found in the explanation for the getSetting function above.

output:
no return value

ORACLE CHAINLINK CONTRACT

The Chainlink oracle contract is specifically designed for cryptocurrency predictions.

Using the savePrice function, we directly store Chainlink's fair, transparent, and tamper-proof price data on-chain. The getOutcome function then determines the final result based on this stored data.

SAVE PRICE

The savePrice function retrieves the price of a specific round from Chainlink and stores it on-chain.

To explain this clearly, we define two key terms:

1. Round

In cryptocurrency prediction markets, all prediction questions follow the same format:
Whether the price of a given cryptocurrency in round n has increased or decreased compared to round n−1.

Each cryptocurrency prediction market differs only in its end time. To avoid deploying redundant contracts, we use round to represent prediction markets with different end times.

This round is entirely different from the roundId defined in Chainlink's AggregatorV3Interface.

2. Price of the n-th Round

To determine the price of the n-th round, we must first determine the market end time.

The end time is calculated as:

startTime + interval × optionId (see startTime in the prediction contract's getSetting function).

This value is a Unix timestamp. We then iterate through all price records already written to Chainlink and select the price whose timestamp is **the closest one after (≥) this end time**. That price is defined as the price of the n-th round.

If no such price is found, the search can be retried later with different input parameters, without affecting the final outcome.

Our contract is able to determine whether Chainlink has already recorded price data after the target timestamp, and only prices that have already been written to Chainlink will be stored.

Reference Code

Function name: savePrice

Input:

1. uint256 requestId (Used by the server to identify the transaction ID)

2. address prediction (Address of the prediction contract)

3. uint256 optionId (Specifies which round's price to store)

4. uint80 startRoundId (If this value is 0, the search starts from the most recently updated Chainlink data. Otherwise, this value is treated as the starting roundId for the search)

5. uint256 length (Maximum number of iterations. If the target price is not found within this limit, the transaction reverts)

Output:

None.

Getting Prediction Results (getOutcome)

To retrieve the result of the n-th round using getOutcome, the prices for both round n and round n−1 must first be stored by calling savePrice.

Reference code:

function name: getOutcome

Input:

1. address prediction (The address of the prediction contract)

2. uint256 optionId (The round number to query)

Output:

1. uint256 outcome (One of the following values:

0: Still waiting for the savePrice function to store the prices for round optionId and round optionId - 1.

1: The price of round optionId is greater than or equal to the price of round optionId - 1.

2: The price of round optionId is less than the price of round optionId - 1.

5: If the required prices are not recorded within 7 days after the end time, the round is marked as canceled, allowing users holding shares in the prediction market to settle.)

OPTIMISTIC ORACLE CONTRACT

The optimistic oracle contract is critical to the system. Except for cryptocurrency markets, the outcomes of all prediction markets are determined by this contract.

Unlike cryptocurrency prediction markets, general prediction markets do not have on-chain data sources like Chainlink. To ensure that the oracle can still produce fair, impartial, and transparent results, PickYesNo has carefully designed an optimistic oracle that ensures every outcome aligns with real-world facts.

PickYesNo’s optimistic oracle adopts a decentralized architecture, where users vote for the market outcome they believe to be correct. In addition, our innovative challenge mechanism allows users to dispute the voting result and trigger platform arbitration when they have doubts. The platform rewards users who select the correct outcome during the voting and challenge phases, recognizing their contributions to maintaining fairness and integrity.

We believe this approach enables the oracle to deliver the most objective results possible and prevents financially powerful individuals or organizations from distorting the truth through capital advantages in financial markets.

Prediction Contract Results

The oracle contract handles different prediction contract types differently. Below, we first introduce single outcomes and independent outcomes:

- **Single Outcome:** In the same round, different options are mutually exclusive; only one option can be true.
  Example: In the 2049 X country presidential election, among candidates A, B, and C, only one can win; multiple winners are impossible.
- **Independent Outcome:** In the same round, different options can occur simultaneously and are not mutually exclusive.

> Example: In the 2049 X country election, candidate Y's party winning or losing in states A, B, and C respectively could result in zero wins, single wins, multiple wins, or all wins.

VOTING

During the voting phase of the oracle contract, users may cast votes within the oracle to determine the outcome of a prediction market. The definition and timing of the voting phase will be explained in detail in later sections.

Unlike prediction contracts, which only allow “Yes” or “No,” the oracle contract introduces an additional possible outcome: Ambiguous 50-50 Settlement.

The purpose of this outcome is to handle unexpected scenarios in which selecting “Yes” or “No” would be inappropriate. In such cases, “Ambiguous 50-50 Settlement” can be used as the final outcome.

For independent-outcome prediction markets, users may choose from three possible outcomes when voting on each option: Yes, No, and Ambiguous 50-50 Settlement.

Each option in the same round has its own lifecycle and its own vote count. The result of an option is whichever of the three outcomes receives the most votes.

For single-outcome prediction markets, users may choose only between Yes and Ambiguous 50-50 Settlement when voting on an option.

In a single-outcome market, at most one option may end with the result “Yes.” In this mode, the vote counts for “Ambiguous 50-50 Settlement” are shared across all options. This means that regardless of which option a user assigns their “Ambiguous” vote to, all such votes are aggregated and compared against the “Yes” votes of each option.

If “Ambiguous 50-50 Settlement” receives the highest number of votes, then every option in that round will return “Ambiguous 50-50 Settlement” as its result.

If an option’s “Yes” votes exceed the total “Ambiguous” votes and exceed all other options’ “Yes” votes, querying that option will return “Yes,” while all other options in the same round will return “No.”

If the final result is Ambiguous 50-50 Settlement, any vote cast for “Ambiguous” (for any option) is considered correct.

If the final result is Yes for a particular option, then only users who voted “Yes” for that specific option are considered to have voted correctly.

The above descriptions apply only in cases where arbitration is not required. Details about arbitration outcomes and the conditions under which arbitration is triggered will be covered in later chapters.To ensure fair voting, PickYesNo requires users to stake a certain amount of USDC when voting. If the user's voted result is ultimately confirmed to align with the facts, the staked funds will be fully refunded, and additional rewards will be granted; otherwise, the staked funds will be forfeited.

Reference code:

function name: vote

input:
1. uint256 requestId (Server ID for this transaction)

2. address prediction (Prediction contract address to vote on)
3. uint256 optionId (Option ID within the prediction contract)
4. uint256 outcome (Voted result: 1=Yes, 2=No, 3=Draw)
5. address wallet (User's wallet contract address)
6. bytes signature (Delegated signature for this transaction, used for identity verification. The signature covers the prediction, optionId, outcome, and the staked amount. If any part of the transaction submitted by the platform differs from the signed data, the entire transaction will revert. This ensures the platform cannot modify your voting parameters or charge even a single unit more than the amount you approved.)

output:
no return value

Voting Restrictions

- Must be during the Voting Round.
- **Single Outcome**: Each user can only vote once per prediction market.
- **Independent Outcome**: Each user can only vote once per option.
- Wallet contract must have sufficient USDC staked to the oracle contract.
- If the wallet contract is bound to an EOA, the platform cannot unilaterally use the assets of a user to vote. This is because it requires either a signature authorization from the user or a pre-signed allowance.

Getting Prediction Results (getOutcome)

The oracle contract can query outcome of the current option through the getOutcome function.

The oracle returns an integer between [0, 5], where 1, 2, 3, and 5 represent final outcomes. Once any of these values is produced, it becomes permanent, and both the oracle and the prediction contract can use it to distribute rewards and settle results.

A returned value of 0 or 4 is not final and will transition into a final outcome as time progresses.

Same returned value affects the prediction market and oracle users differently, as outlined below.

	For prediction market users	For oracle users
0： Pending for Final Result	Settlement cannot proceed until a final outcome is produced.	Settlement cannot proceed until a final outcome is produced.
1：Yes	For every Yes share a user holds, they will receive 1 USDC.	Only users who voted or challenged with “Yes” are eligible for rewards.
2：No	For every No share a user	Only users who voted or

	holds, they will receive 1 USDC.	challenged with “No” are eligible for rewards.
3：Unclear (50/50 Settlement)	For every Yes share a user holds, they will receive 0.5 USDC. For every No share a user holds, they will receive 0.5 USDC.	Only users who voted or challenged with “Unclear (50/50 Settlement)” are eligible for rewards.
4：Pending for Arbitration	Settlement cannot proceed until a final outcome is produced.	Rewards cannot be distributed until a final outcome is produced. When a validator initiates a challenge or when the top votes result in a tie, the process enters this stage and awaits platform arbitration.
5：Cancelled	For every Yes share a user holds, they will receive 0.5 USDC. For every No share a user holds, they will receive 0.5 USDC.	No rewards are given regardless of the voting or challenge option chosen, but all staked amounts will be fully refunded.

Reference code:

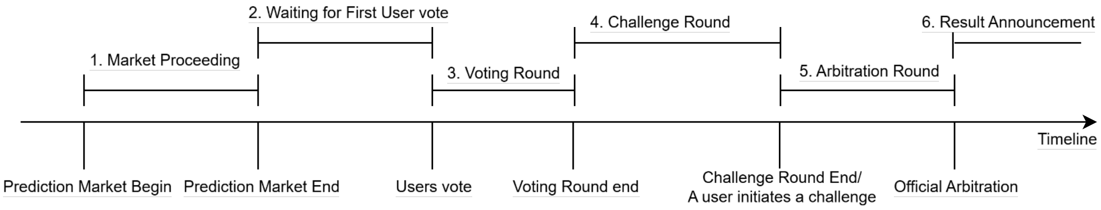
function name: getOutcome

input:
1. address prediction (The prediction contract address)
2. unit256 optionId (The option ID within the prediction contract)

output:
1. uint256: The returned prediction result. The possible values are integers between [0, 5].

Oracle Contract Decision Process

For each prediction contract, the oracle contract timeline can be divided into:



Stage Descriptions

Different options within different Prediction Contracts are handled by the oracle with different processes.

1. **Market Proceeding**

    - During this stage, the corresponding prediction market has not yet ended, and neither users nor the platform can take any action.
    - If the prediction market type is "**Single Outcome**”, all options share the same end time; if "**Independent Outcome**”, each option has its own end time.
    - After the prediction market ends, oracle enters Stage 2.

2. **Waiting for First User Vote:**

    - After the corresponding prediction market ends, the oracle automatically enters this stage until the first user vote
    - This stage can last for a maximum of 120 days. As long as no users vote within this 120-day period, the stage will continue, and all subsequent stages will be postponed.
    - If there are no user votes for more than 120 days, the oracle skips to stage 6 and sets the returned result to 5: Canceled. This is done so that even if no one participates in oracle voting, users who hold prediction contract shares can still get their USDC back.
    - If the prediction market type is "**Single Outcome**”, all options collectively leave this stage as soon as one option receives a user vote; if "**Independent Outcome**”, each option must receive a user vote to leave this stage.

3. **Voting Round**

- Immediately after the first vote is casted, the voting round begins. The duration of the voting round is fixed, starting from the time the first vote is cast and ending after a set period.
- The votingDuration parameter determines how long the voting round lasts and can be queried using the getSetting function in the corresponding prediction contract.
- If the prediction market type is "**Single Outcome**”, all options share the votingDuration; if "**Independent Outcome**”, each option has its own votingDuration.

4. **Challenge Round**
    - The challenge round automatically begins after the voting round ends. Two scenarios can lead to the end of the challenge round:
        1. A user initiates a challenge during the challenge round.
        2. The challenge round automatically ends after challengeDuration. challengeDuration is also a parameter of the prediction contract and can be queried using the getSetting function in the corresponding prediction contract.
5. **Arbitration Round**
    - Not all prediction markets enter the arbitration round. An arbitration round is entered if one of the following two conditions occurs:
        1. A user initiates a challenge during the challenge round, immediately ending the challenge round and entering the arbitration round.
        2. After challengeDuration, at least two options have identical

vote counts, requiring platform arbitration to determine the final result.

- Conversely, if no user initiates a challenge during the Challenge Round, and there are no tied votes after the Challenge Round ends, then there will be no Arbitration Round, and the result will be announced directly.
- If a market enters the Arbitration Round and the platform doesn't arbitrate within 120 days, the oracle will move to Stage 6 and set the result to 5 (Canceled). This is a crucial failsafe designed to ensure that if the platform's service terminates and arbitration becomes impossible, users who participated in the oracle and the prediction market can still get their USDC back.

6. **Result Announcement**

   - The final result is announced, and the prediction market can use this result to settle.
   - If no arbitration round was entered, the final result is the option with the highest user votes.
   - If an arbitration round was entered, the final result is either the platform Arbitration or an automatic return of 5 after 120 days.

Logic for Determining Returned Predction Results

Calling getOutcome at different stages will return different results.

1. **Returns 0**

   Indicating no result yet**.** Calling getOutcome during "Waiting for Prediction Market to End," "Waiting for First User Vote," "Voting Round," and "Challenge Round" stages will all return 0.

2. **Returns 4**

   Indicating currently in Arbitration Round.

3. **Returns 1, 2, 3, or 5**

   This signifies the final result provided by the oracle, and the prediction market can use this result for settlement. Users who voted or challenged in the oracle can also receive rewards based on this result.

CHALLENGE

During the challenge round of a prediction contract, users can dispute voting results.

Unlike prediction contracts, which only allow “Yes” or “No,” the oracle contract introduces an additional possible outcome: Ambiguous 50-50 Settlement.

This option exists because prediction markets may occasionally encounter unforeseen circumstances in which selecting either “Yes” or “No” is inappropriate. In such cases, “Ambiguous 50-50 Settlement” can serve as the final result.

For independent-outcome prediction markets, users have three choices during the oracle’s challenge phase for each option: Yes, No, and Ambiguous 50-50 Settlement.

For single-outcome prediction markets, users have only two choices during the challenge phase for each option: Yes and Ambiguous 50-50 Settlement.

If the final result is Ambiguous 50-50 Settlement, any challenge cast for “Ambiguous” on any option is considered correct.

If the final result is Yes for a particular option, then only users who challenged with “Yes” for that specific option are considered to have provided the correct challenge result.

To prevent malicious challenges, users are also required to stake a certain amount of USDC when challenging. If the user's challenged result is ultimately confirmed to align with the facts, the staked funds will be fully refunded, and additional, greater rewards will be granted; otherwise, the staked funds will be forfeited.

<u>Reference code</u>

function name: challenge

input:
1. uint256 requestId (Server ID for this transaction)
2. address prediction (Prediction contract address to challenge)

3. uint256 optionId (Option ID within the prediction contract)
4. uint256 outcome (User's perceived correct result: 1=Yes, 2=No, 3=Draw)
5. address wallet (User's wallet contract address)
6. bytes signature (Delegated signature for this transaction. The signature covers the prediction, optionId, outcome, and the staked amount. If any part of the transaction submitted by the platform differs from the signed data, the entire transaction will revert. This ensures the platform cannot modify your challenging parameters or charge even a single unit more than the amount you approved.)

output:
No return value

Challenge Restriction

- Must be during the Challenge Round.
- Challenge attempts are limited.
  - **Single Outcome:** Only one challenge allowed across all options.
  - **Independent Outcome:** Each option can be challenged once.
- Cannot challenge the option with the current highest votes, unless there's a tie.
- Wallet contract must have sufficient USDC staked to the oracle contract.
- The platform cannot unilaterally use the voting user's assets, as it requires user signature authorization or a pre-signed allowance

ARBITRATION

The platform can conduct arbitration in the following two scenarios:

Conditions	Descriptions
User initiated a challenge	**Single Outcome:** The platform arbitrates All Options. **Independent Outcome:** The platform only arbitrates the Challenged Option.
A tie in Voting Round	**Single Outcome:** If there are two or more options with the **same highest number of votes**, the platform arbitrates All Options. **Independent Outcome:** For the same option, if at least two outcomes have the **same highest number of votes**, the platform arbitrates the option.

Arbitration is initiated by the Executor EOA, but it only becomes valid when accompanied by two signatures from the designated Arbitrator EOAs. This ensures that even if an attacker obtains the Executor EOA's private key, they still cannot arbitrarily issue arbitration results, keeping all outcomes fully aligned with the facts.

The arbitrator EOA only provides signatures and does not connect to the network, minimizing the risk of private-key leakage and offering higher security than the executor EOA. Each arbitration also requires signatures from two independent arbitrator EOAs, further enhancing the security of PickYesNo.

Reference code

function name: arbitrate

input:
1. uint256 requestId (Server ID for this transaction)
2. address prediction (Prediction contract address to arbitrate)
3. uint256 optionId (Option ID to arbitrate)
4. uint256 outcome (Official ruled result: 1=Yes, 2=No, 3=Draw, 5=Canceled)

output:
no return value

REWARD

Anyone can call the reward function, but most of the time, the platform will do it to save you on gas fees. All rewards are provided by the platform.

Even if the platform goes bust, you can still call the function yourself to get your staked funds back.

Reference code

function name: reward

input:
1. uint256 requestId (Server ID for this transaction)
2. address prediction (Prediction contract address to issue rewards for)
3. uint256 optionId (The option ID)
4. address[] wallets (Array of user wallet contract addresses to receive rewards)

output:
No return value

Reward Distribution Flow

1. **The reward function calls getOutcome internally to get th**e result parameter outcome, which will be used for reward distribution.

   - If the outcome is 0 or 4, the transaction will revert.
   - Otherwise the platform will check if a reward should be distributed.

2. **Rewards are only distributed** when the outcome is 1, 2, or 3. If the outcome is 5, only the staked funds are returned.
3. **If there's a challenge record**
   - **Challenge Success:**
     - The challenger receives rewards challengeRewards calculated based on the challengePercent set in the prediction contract.
     - The challenger can get:
       - challengeRewards,
       - and their staked USDC back.

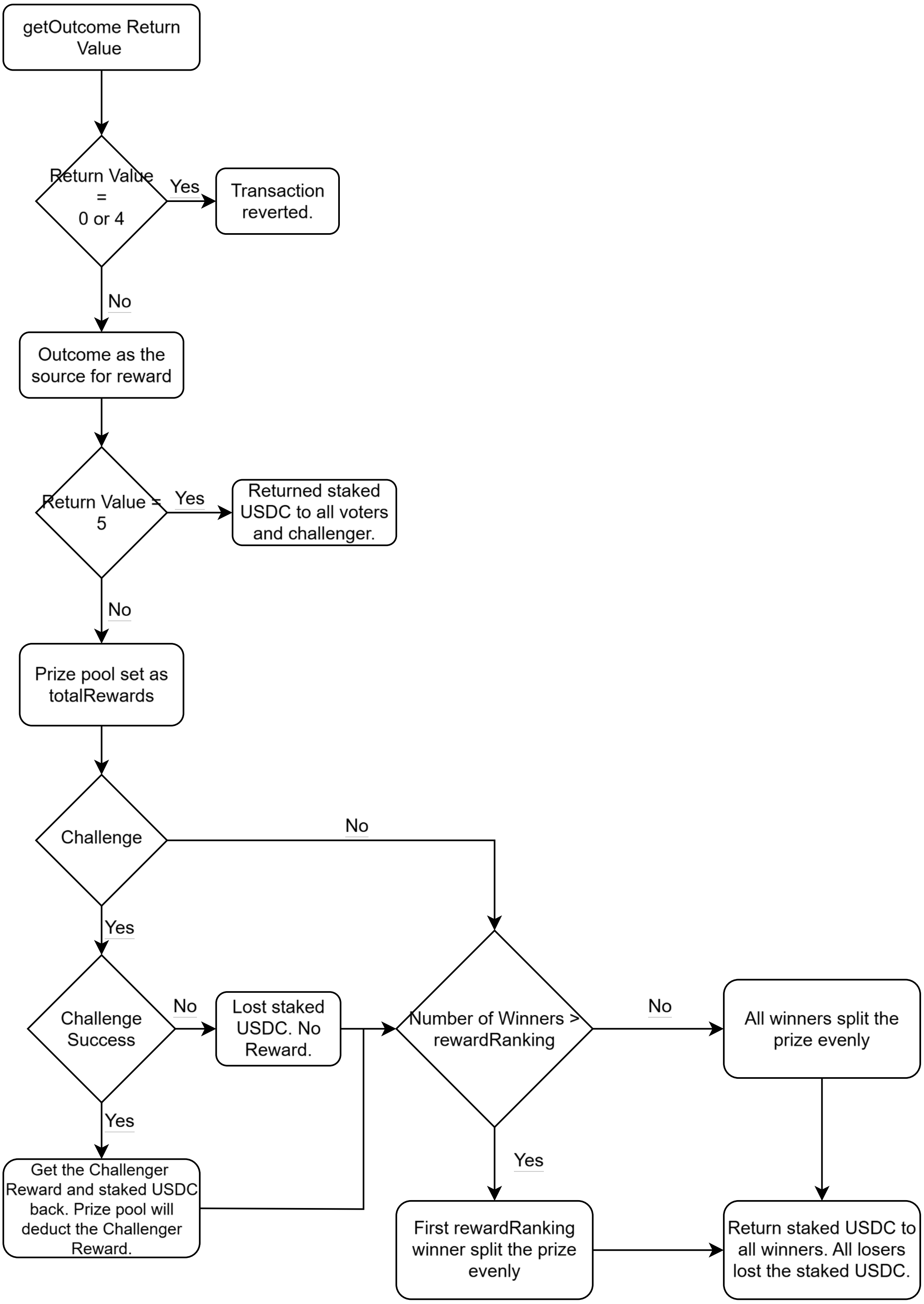
- **Challenge Failed:**
  - The platform takes the challenger's staked USDC, and no reward is issued.
- outcome=5：
  - No rewards will be distributed, but the staked amount will be fully refunded.

4. **Voting Rewards**

- **Correct Vote:** If a user's vote is correct, their staked funds are returned.
  - If the number of correct voters is greater than rewardRanking, the top rewardRanking users split the rewards evenly.
  - If the number of correct voters is less than or equal to rewardRanking, all users who voted correctly split the rewards evenly.
- **Incorrect Vote:** The user's staked USDC is taken by the platform.
- outcome=5：
  - No rewards will be distributed, but the staked amount will be fully refunded

5. The prize money and returned staked funds are transferred directly from the oracle contract to the user's wallet contract.

Reward Distribution Flowchart


getOutcome Return Value
Return Value = 0 or 4
Yes
Transaction reverted.
No
Outcome as the source for reward
Return Value = 5
Yes
Returned staked USDC to all voters and challenger.
No
Prize pool set as totalRewards
Challenge
No
Yes
Challenge Success
No
Lost staked USDC. No Reward.
Yes
Get the Challenger Reward and staked USDC back. Prize pool will deduct the Challenger Reward.
Number of Winners > rewardRanking
No
All winners split the prize evenly
Yes
First rewardRanking winner split the prize evenly
Return staked USDC to all winners. All losers lost the staked USDC.

Platform's Usable Funds: SAFU, Always

After rewards are distributed, the deserved amount is transferred from the oracle contract to the user's wallet contract. The funds in the oracle consist of prize money from the project and user stakes (for voting and challenging).

Is it possible for the project to drain the oracle contract of USDC before rewards are paid out? No. The oracle contract code completely prevents this.

A variable called stakingAmount controls how much the platform can transfer out of the oracle contract. This design ensures that the transferToFee function, the only way the platform can transfer USDC out of the contract, **cannot touch user stakes.**

We can transfer USDC from the oracle contract, but it's only limited to the prize money provided by the platform. This means that if the platform's service is terminated, any prize money may not be distributed. However, users will still be able to fully withdraw their staked funds from voting and challenges.

Reference code

:

function name: transferToFee

input:
1. uint256 requestId (Server ID for this transaction)
2. uint256 amount (Amount of USDC to transfer from the oracle contract)

output:
no return value

PERMISSION CONTRACT

In PickYesNo's smart contract ecosystem, most functions are restricted to specific EOAs (Externally Owned Accounts). These permissions are centrally managed by our Permission Contracts and define the following roles:

1. **Fee EOA (feeEOA):** Each smart contract can only have one fee address at any given time. All fees, penalties, and other revenue generated in the contract are automatically sent to this address.
2. **Manager EOA (managerEOA):** All smart contracts share one manager address. This address is used for administrative tasks.
3. **Operator EOA (managerEOA):** All smart contracts share one operator address. This address is used for operation tasks.
4. **Arbitrator EOA (arbitratorEOA):** The address dedicated to oracle arbitration.
5. **Executor EOA (executorEOA):** A smart contract can have multiple executor addresses. The vast majority of user interactions with a contract must be initiated through an executor address. To keep user funds safe, an executor must have the user's signature or a pre-signed allowance before calling a contract; otherwise, the transaction will automatically revert.

REFERENCE

Contract Address: https://pickyesno.com/docs/contract/contract-address

Contract Source Code: https://pickyesno.com/docs/contract/contract-sourcecode

Contract Audit Report: https://pickyesno.com/docs/contract/contract-audit-report