Tokenized SPV & Prediction Market

Marc Aliaga
Blockchain Infrastructure Design

February 2026

1 Contract: SPV_vault.sol

This contract acts as the primary liquidity provider and manages the structured payout hierarchy through a waterfall distribution.

```
function invest(uint256 amount, uint256 tranche) external {
    require(tranche <= 2, "Invalid tranche");
    usdc.transferFrom(msg.sender, address(this), amount);
    if(tranche == 0) principalSr += amount;
    _mint(msg.sender, tranche, amount, "");
}

function totalCollateral() public view returns (uint256) {
    return principalSr + principalMz + principalEq;
}
```

Listing 1: Investment and Collateral Logic

1.1 Function Explanations

- **invest(amount, tranche):** Enables investors to deposit USDC into the vault. The contract handles the token transfer, updates the accounting for the specific risk tranche, and mints an ERC-1155 token as proof of participation.
- **totalCollateral():** A view function that aggregates the principal from all tranches. This is a critical metric used by the market to determine maximum risk exposure.
- **getPayouts():** Calculates the "Waterfall" settlement. It distributes the final pool (capital minus losses plus premiums) by prioritizing the Senior tranche first, followed by Mezzanine, and finally the Equity tranche.

2 Contract: PredictionMarket.sol

The core application engine that manages user bets, outcome pricing, and ensures the continuous solvency of the underlying SPV.

```
function buyYes(uint256 quantity) external {
    uint256 cost = calculatePrice(quantity);
```

```
    require(vault.totalCollateral() >= quantity, "Vault: Insufficient
        collateral");
    usdc.transferFrom(msg.sender, address(vault), cost);
    yesToken.mint(msg.sender, quantity);
}

function resolve(bool _winningOutcome) external onlyOwner {
    require(block.timestamp >= marketEndTime, "Market not ended");
    resolved = true;
    winningOutcome = _winningOutcome;
}
```

Listing 2: Betting and Resolution Guardrails

2.1 Function Explanations

- **buyYes / buyNo(quantity):** Executes ticket purchases for users. It first verifies that the Vault holds sufficient collateral to cover the potential max payout. If solvent, it transfers the premium to the Vault and mints the outcome tokens.
- **calculatePrice(quantity):** Determines the premium cost for the user. This function is designed to feed yield directly back to the SPV investors.
- **resolve(outcome):** Finalizes the event. It validates that the expiration time has passed and defines the winning outcome, allowing the Vault to begin the liquidation process.

3 Contract: MarketToken.sol

A standardized ERC-20 implementation used to represent both the settlement currency (Mock USDC) and the speculative outcome tokens (YES/NO).

```
function mint(address to, uint256 amount) external onlyOwner {
    _mint(to, amount);
}
```

Listing 3: Minting and Access Control

3.1 Function Explanations

- **mint(to, amount):** Allows the contract owner (typically the PredictionMarket contract) to create new tokens. This represents a user's speculative position in a given event.
- **approve(spender, amount):** Standard ERC-20 function allowing the Vault or Market contracts to move funds on behalf of the user during investments or bets.

4 Summary Function Matrix

The following table summarizes the core operational functions of the ecosystem:

Function	Location	Role	Key Dependency
`invest`	SPV_vault	Capital Influx	ERC-20 Approval
`totalCollateral`	SPV_vault	Solvency Check	Accounting Logic
`getPayouts`	SPV_vault	Settlement	Market Resolution
`buyYes / buyNo`	Market	Risk Execution	Vault Collateral
`resolve`	Market	Oracle/Admin	Block Timestamp
`mint`	MarketToken	Asset Issuance	Ownable Access

Table 1: Operational mapping of the SPV-Prediction Market architecture.

[IMAGE: Sequence diagram showing interaction between invest, buyYes, and resolve functions]

Figure 1: Interaction Flow between Vault, Market, and Tokens.