# Step-by-Step Smart Contract Build Plan

Here's a logical order that builds from foundation to full system:

## Phase 1: Foundation & Core Infrastructure (Week 1)

### Step 1: ErrorsEvents.sol + AccessRoles.sol

**Why first:** Every contract will import these.

```
// ErrorsEvents.sol - Centralized errors & events
error Unauthorized();
error Paused();
error InvalidAmount();
error CapExceeded();
event Deposited(address indexed user, uint256 usdc, uint256 shares);
event YieldHarvested(uint256 epoch, uint256 usdc);

// AccessRoles.sol - Role management
contract AccessRoles {
    address public governor;
    address public guardian;
    address public keeper;
    address public treasury;

    modifier onlyGovernor() { require(msg.sender == governor); _; }
    modifier onlyGuardian() { require(msg.sender == guardian); _; }
    modifier onlyKeeper() { require(msg.sender == keeper); _; }
}
```

**Usage:** Imported by all contracts for consistent errors/events and access control.

### Step 2: PerpBondToken.sol (ERC-20 Receipt)

**Why now:** It's simple and needed by the vault.

```
contract PerpBondToken is ERC20, AccessRoles {
    // Only vault can mint/burn
    address public vault;

    function mint(address to, uint256 amount) external {
        require(msg.sender == vault, "Only vault");
        _mint(to, amount);
    }

    // Transferable but non-redeemable (no burn function
for users)
}
```

**Usage:**

- Minted when users deposit USDC
- Represents proportional share of entire vault
- Transferable on secondary markets
- Does NOT entitle to withdraw principal (perpetual bond)

### Step 3: SafeTransferLib.sol + MathLib.sol

**Why now:** Needed for vault operations.

```
// SafeTransferLib.sol
library SafeTransferLib {
    function safeTransfer(IERC20 token, address to, uint256
amount) internal {
        // Handle non-standard ERC20s (USDT, etc.)
    }
    function safeTransferFrom(...) internal {}
}

// MathLib.sol
library MathLib {
    uint256 constant BPS = 10_000;

    function mulBps(uint256 amount, uint256 bps) internal
pure returns (uint256) {
        return amount * bps / BPS;
    }
    function convertToShares(uint256 assets, uint256
totalAssets, uint256 totalShares)
        internal pure returns (uint256) {
        if (totalShares == 0) return assets; // Bootstrap
        return assets * totalShares / totalAssets;
    }
}
```

**Usage:** Safe token operations and share math throughout the system.

## Phase 2: Core Vault & Registry (Week 1-2)

### Step 4: IStrategyAdapter.sol Interface

**Why now:** Vault needs to know how to talk to adapters.

```
interface IStrategyAdapter {
    // Deploy USDC into strategy
    function deposit(uint256 usdcAmount) external returns
(uint256 deployed);

    // Harvest rewards, convert to USDC
    function harvest() external returns (uint256 usdcOut);

    // Current USDC value of position
    function tvl() external view returns (uint256
usdcValue);

    // Emergency withdraw (if possible)
    function emergencyWithdraw() external returns (uint256
recovered);

    // Metadata
    function name() external view returns (string memory);
    function underlyingToken() external view returns
(address); // AERO, PENDLE, CVX
}

// Optional feature interfaces
interface IVotingAdapter {
    function vote(address[] calldata gauges, uint256[]
calldata weights) external;
}

interface ILockingAdapter {
    function lockedUntil() external view returns (uint256
timestamp);
}
```

**Usage:** All adapters must implement this. Vault calls these methods.

### Step 5: AdapterRegistry.sol

**Why now:** Vault needs to know which adapters are valid before accepting deposits.

```
contract AdapterRegistry is AccessRoles {
    struct AdapterConfig {
        bool active;
        uint256 tvlCap;        // Max USDC this adapter can
hold
        uint256 maxAllocBps;   // Max % of vault (e.g.,
4000 = 40%)
        uint256 slippageBps;   // Max slippage for swaps
        address oracle;        // Price feed
    }

    mapping(address => AdapterConfig) public adapters;
    address[] public adapterList;

    function registerAdapter(address adapter, AdapterConfig
calldata config)
        external onlyGovernor {
        adapters[adapter] = config;
        adapterList.push(adapter);
        emit AdapterRegistered(adapter);
    }

    function pauseAdapter(address adapter) external
onlyGuardian {
        adapters[adapter].active = false;
    }
    function getActiveAdapters() external view returns
(address[] memory) {
        // Filter active only
    }
}
```

**Usage:**

- Governor registers new protocols (veAERO, vePENDLE, vlCVX)
- Vault checks registry before depositing
- Guardian can pause risky adapters
- Frontend reads this for UI

### Step 6: PerpBondVault.sol (The Heart)

**Why now:** We have tokens, adapters interface, and registry. Time for the main vault.

```
contract PerpBondVault is AccessRoles {
    using SafeTransferLib for IERC20;
    using MathLib for uint256;

    IERC20 public immutable usdc;
    PerpBondToken public immutable perpToken;
    AdapterRegistry public registry;

    // Allocation weights (basis points, sum to 10000)
    mapping(address => uint256) public targetAllocationBps;

    // Idle USDC not yet deployed
    uint256 public idleUsdc;

    // ERC-4626-style but NO withdrawals
    function deposit(uint256 usdcAmount) external returns
(uint256 shares) {
        // 1. Transfer USDC from user
        usdc.safeTransferFrom(msg.sender, address(this),
usdcAmount);

        // 2. Calculate shares based on current NAV
        shares = convertToShares(usdcAmount);
        // 3. Mint receipt tokens
        perpToken.mint(msg.sender, shares);

        // 4. Add to idle pool (deploy in batches via
rebalance)
        idleUsdc += usdcAmount;

        emit Deposited(msg.sender, usdcAmount, shares);
    }

    function totalAssets() public view returns (uint256) {
        uint256 total = idleUsdc;
        address[] memory adapters =
registry.getActiveAdapters();
        for (uint i = 0; i < adapters.length; i++) {
            total += IStrategyAdapter(adapters[i]).tvl();
        }
        return total;
    }

    function convertToShares(uint256 assets) public view
returns (uint256) {
        uint256 supply = perpToken.totalSupply();
        if (supply == 0) return assets; // Bootstrap 1:1
        return assets.mulDiv(supply, totalAssets());
    }

    // Deploy idle USDC to adapters based on target weights
    function rebalance() external onlyKeeper {
        address[] memory adapters =
registry.getActiveAdapters();
        uint256 totalBps = 0;

        // Calculate total weight
        for (uint i = 0; i < adapters.length; i++) {
            totalBps += targetAllocationBps[adapters[i]];
        }

        // Deploy proportionally
        for (uint i = 0; i < adapters.length; i++) {
            address adapter = adapters[i];
            uint256 allocation =
idleUsdc.mulBps(targetAllocationBps[adapter]) / totalBps;

            if (allocation > 0) {
                usdc.safeTransfer(adapter, allocation);

IStrategyAdapter(adapter).deposit(allocation);
            }
        }

        idleUsdc = 0;
    }

    // Governor sets allocation strategy
    function setTargetAllocations(address[] calldata
adapters, uint256[] calldata bps)
        external onlyGovernor {
        uint256 total = 0;
        for (uint i = 0; i < adapters.length; i++) {
            require(registry.adapters(adapters[i]).active,
"Not active");
            targetAllocationBps[adapters[i]] = bps[i];
            total += bps[i];
        }
        require(total == 10000, "Must sum to 100%");
    }

    // NO withdraw() or redeem() - principal is locked
forever
}
```

**Usage:**

- Users call `deposit()` with USDC → get PerpBond tokens
- Keeper calls `rebalance()` weekly to deploy idle USDC
- Governor adjusts `targetAllocationBps` based on APY forecasts
- Vault aggregates TVL across all adapters

## Phase 3: Yield Operations (Week 2)

### Step 7: RouterGuard.sol + OracleLib.sol

**Why now:** Needed before building Harvester.

```
// RouterGuard.sol - Whitelist DEX routes
contract RouterGuard is AccessRoles {
    mapping(address => bool) public allowedRouters; //
Uniswap, Curve, etc.
    mapping(address => mapping(address => uint256)) public
maxSlippageBps; // tokenIn => tokenOut

    function validateSwap(
        address router,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 minOut
    ) external view {
        require(allowedRouters[router], "Router not
allowed");

        uint256 expectedOut =
OracleLib.getExpectedAmount(tokenIn, tokenOut, amountIn);
        uint256 maxSlippage = maxSlippageBps[tokenIn]
[tokenOut];
        uint256 minAllowed = expectedOut.mulBps(10000 -
maxSlippage);

        require(minOut >= minAllowed, "Slippage too high");
    }
}

// OracleLib.sol - Chainlink + TWAP
library OracleLib {
    function getExpectedAmount(address tokenIn, address
tokenOut, uint256 amountIn)
        internal view returns (uint256) {
        // Chainlink price feeds with TWAP fallback
        // Return expected output for sanity check
    }
}
```

**Usage:** Harvester uses this before every swap to prevent sandwich attacks.

## Step 8: Harvester.sol

**Why now:** We can now safely harvest and swap rewards.

```
contract Harvester is AccessRoles {
    PerpBondVault public vault;
    AdapterRegistry public registry;
    RouterGuard public routerGuard;

    struct SwapRoute {
        address router;
        bytes path; // Encoded route
    }

    // tokenAddress => route to USDC
    mapping(address => SwapRoute) public rewardRoutes;

    function harvestAll() external onlyKeeper returns
(uint256 totalUsdc) {
        address[] memory adapters =
registry.getActiveAdapters();

        for (uint i = 0; i < adapters.length; i++) {
            uint256 usdcOut = _harvestAdapter(adapters[i]);
            totalUsdc += usdcOut;
        }

        emit HarvestedAll(totalUsdc);
    }

    function _harvestAdapter(address adapter) internal
returns (uint256 usdcOut) {
        // 1. Call adapter.harvest() - gets reward tokens
        IStrategyAdapter(adapter).harvest();

        // 2. Identify reward tokens (AERO, PENDLE, CVX,
bribes, etc.)
```

```
        address[] memory rewards =
_getRewardTokens(adapter);

        // 3. Swap each to USDC
        for (uint j = 0; j < rewards.length; j++) {
            usdcOut += _swapToUsdc(rewards[j]);
        }

        // 4. USDC stays in Harvester, ready for
Distributor
        return usdcOut;
    }

    function _swapToUsdc(address token) internal returns
(uint256 usdcOut) {
        uint256 balance =
IERC20(token).balanceOf(address(this));
        if (balance == 0) return 0;

        SwapRoute memory route = rewardRoutes[token];

        // Validate with RouterGuard
        uint256 minOut = _calculateMinOut(token, balance);
        routerGuard.validateSwap(route.router, token, usdc,
balance, minOut);

        // Execute swap
        // ... router.swap(route.path, balance, minOut)

        return usdcOut;
    }
}
```

**Usage:**

- Keeper calls `harvestAll()` weekly
- Harvester calls each adapter's `harvest()`
- Swaps all rewards to USDC
- USDC accumulated in Harvester, ready for distribution

## Step 9: Distributor.sol

**Why now:** We have harvested USDC, now distribute to users.

```
contract Distributor is AccessRoles {
    PerpBondVault public vault;
    Harvester public harvester;
    PerpBondToken public perpToken;
    IERC20 public usdc;

    uint256 public currentEpoch;

    struct EpochData {
        uint256 timestamp;
        uint256 totalUsdc;       // Harvested this epoch
        uint256 totalShares;     // Snapshot at close
        uint256 usdcPerShare;    // totalUsdc / totalShares
    }

    mapping(uint256 => EpochData) public epochs;
    mapping(address => uint256) public lastClaimedEpoch;
    mapping(address => bool) public autoCompound;

    function closeEpoch() external onlyKeeper {
        // 1. Pull USDC from Harvester
        uint256 harvestedUsdc =
usdc.balanceOf(address(harvester));
        usdc.transferFrom(address(harvester),
address(this), harvestedUsdc);

        // 2. Take performance fee (e.g., 10%)
        uint256 fee = harvestedUsdc.mulBps(1000); // 10%
        usdc.transfer(treasury, fee);
        uint256 netUsdc = harvestedUsdc - fee;

        // 3. Record epoch
        uint256 totalShares = perpToken.totalSupply();
        epochs[currentEpoch] = EpochData({
            timestamp: block.timestamp,
            totalUsdc: netUsdc,
            totalShares: totalShares,
            usdcPerShare: netUsdc * 1e18 / totalShares //
Scaled for precision
        });

        currentEpoch++;
        emit EpochClosed(currentEpoch - 1, netUsdc,
totalShares);
    }

    function claim() external returns (uint256 claimed) {
        uint256 userShares =
perpToken.balanceOf(msg.sender);

        for (uint256 i = lastClaimedEpoch[msg.sender]; i <
currentEpoch; i++) {
            claimed += userShares *
epochs[i].usdcPerShare / 1e18;
        }

        lastClaimedEpoch[msg.sender] = currentEpoch;

        if (autoCompound[msg.sender]) {
            // Re-deposit into vault
            usdc.approve(address(vault), claimed);
            vault.deposit(claimed);
        } else {
            usdc.transfer(msg.sender, claimed);
        }

        emit Claimed(msg.sender, claimed,
autoCompound[msg.sender]);
    }

    function getClaimable(address user) external view
returns (uint256) {
        uint256 userShares =
perpToken.balanceOf(msg.sender);
        uint256 total = 0;
        for (uint256 i = lastClaimedEpoch[user]; i <
currentEpoch; i++) {
            total += userShares * epochs[i].usdcPerShare /
1e18;
        }

        return total;
    }

    function toggleAutoCompound() external {
        autoCompound[msg.sender] = !
autoCompound[msg.sender];
    }
}
```

**Usage:**

- Keeper calls `closeEpoch()` weekly after harvest
- Users call `claim()` anytime to get USDC
- Auto-compound users automatically re-deposit
- Frontend reads `getClaimable()` for UI

## Phase 4: Voting & Adapters (Week 3)

### Step 10: VoterRouter.sol

**Why now:** Optional but needed for governance participation.

```
contract VoterRouter is AccessRoles {
    AdapterRegistry public registry;

    struct VoteIntent {
        address adapter;
        address[] gauges;
        uint256[] weights;
    }

    function executeVotes(VoteIntent[] calldata intents)
external onlyKeeper {
        for (uint i = 0; i < intents.length; i++) {
            VoteIntent memory intent = intents[i];
require(registry.adapters(intent.adapter).active,
"Inactive");

            // Call adapter's vote function

IVotingAdapter(intent.adapter).vote(intent.gauges,
intent.weights);
        }

        emit VotesExecuted(intents.length);
    }
}
```

**Usage:**

- Keeper submits weekly voting intents
- Router delegates to each adapter's vote function
- Maximizes bribes/fees from governance participation

### Step 11: First Adapter - AerodromeVeAdapter.sol

**Why now:** We need at least one real adapter to test the system.

```
contract AerodromeVeAdapter is IStrategyAdapter,
IVotingAdapter, AccessRoles {
    IERC20 public usdc;
    IERC20 public aero;
    IVeAero public veAero;
    IVoter public voter;

    uint256 public lockedAero;

    function deposit(uint256 usdcAmount) external override
returns (uint256) {
        require(msg.sender == vault, "Only vault");

        // 1. Swap USDC → AERO
        uint256 aeroAmount = _swapUsdcToAero(usdcAmount);

        // 2. Lock AERO → veAERO (max lock: 4 years)
        aero.approve(address(veAero), aeroAmount);
        veAero.createLock(aeroAmount, block.timestamp + 4 *
365 days);

        lockedAero += aeroAmount;
        return usdcAmount;
    }

    function harvest() external override returns (uint256
usdcOut) {
        // 1. Claim rewards (emissions, bribes, fees)
        voter.claimRewards();

        // 2. Collect all reward tokens
        address[] memory rewards = voter.getRewardTokens();

        // 3. Swap to USDC (done by Harvester for
consistency)
        // This adapter just transfers rewards to Harvester
        for (uint i = 0; i < rewards.length; i++) {
            uint256 bal =
IERC20(rewards[i]).balanceOf(address(this));
            IERC20(rewards[i]).transfer(harvester, bal);
        }

        // Return estimated USDC value (Harvester does
actual swap)
        return _estimateUsdcValue(rewards);
    }

    function tvl() external view override returns (uint256)
{
        // Get current AERO price in USDC
        uint256 aeroPrice = _getAeroPrice();
        return lockedAero * aeroPrice / 1e18;
    }

    function vote(address[] calldata gauges, uint256[]
calldata weights)
        external override {
        require(msg.sender == voterRouter, "Only router");
        voter.vote(gauges, weights);
    }

    function name() external pure override returns (string
memory) {
        return "Aerodrome veAERO";
    }
}
```

**Usage:**

- Vault calls `deposit()` to lock USDC as veAERO
- Keeper triggers `harvest()` weekly to claim rewards
- Router calls `vote()` to maximize bribes
- Reports `tvl()` for vault NAV calculation

### Step 12: Repeat for VePendleAdapter.sol & VlCvxAdapter.sol

Same pattern as Aerodrome adapter, just different protocol integrations.

## Phase 5: Testing & Deployment (Week 4)

### Step 13: Unit Tests

```
// test/unit/Vault.t.sol
// test/unit/Distributor.t.sol
// test/unit/Harvester.t.sol
// test/unit/AerodromeVeAdapter.t.sol
```

### Step 14: Fork Tests

```
// test/fork/WeeklyCycleFork.t.sol
// Simulate: deposit → rebalance → harvest → distribute →
claim
```

### Step 15: Deployment Scripts

```
// script/DeployCore.s.sol
// script/DeployAdapters.s.sol
// script/ConfigurePolicy.s.sol
```

## Summary Build Order

| Phase | Contracts | Why |
|---|---|---|
| **1. Foundation** | ErrorsEvents, AccessRoles, PerpBondToken, Libs | Everyone imports these |
| **2. Core** | IStrategyAdapter, AdapterRegistry, PerpBondVault | Heart of the system |
| **3. Yield** | RouterGuard, OracleLib, Harvester, Distributor | Reward processing |
| **4. Protocols** | VoterRouter, AerodromeVeAdapter, etc. | Actual yield generation |
| **5. Testing** | Unit, Fork, Integration | Ensure everything works |

**Want to start with Phase 1? I can help you write ErrorsEvents.sol and AccessRoles.sol first.**