Replit AI Prompt

Goal Remove any older variants of these modules and ship the new Realtor ecosystem on top of the existing Axiom stack. Keep AXM as-is. Add six new contracts, compile, deploy to peaq, write addresses.json, and grant roles. Keep code style consistent with your repo. Use Solidity 0.8.20. Use OpenZeppelin. All constructors are wired. No block comments, no unchecked arithmetic, no unsafe casts.

Project prep

1. Ensure these dependencies exist npm i @openzeppelin/contracts @nomicfoundation/hardhat-toolbox dotenv

2. Ensure hardhat.config.js has peaq network url https colon slash slash evm.peaq.network chainId 3338 accounts from DEPLOYER_PK in .env

3. Create or keep scripts/util writeAddresses.js helper that merges and writes addresses.json at project root.

writeAddresses.js example const fs = require("fs"); const path = require("path"); function writeAddresses(net, obj) { const p = path.join(process.cwd(), "addresses.json"); let all = {}; if (fs.existsSync(p)) all = JSON.parse(fs.readFileSync(p, "utf8")); all[net] = { ...(all[net] || {}), ...obj }; fs.writeFileSync(p, JSON.stringify(all, null, 2)); console.log("addresses.json updated for", net); } module.exports = { writeAddresses };

Contracts Create these files in contracts folder.

contracts/RealtorProfileRegistry.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

```
import "@openzeppelin/contracts/access/AccessControl.sol";

contract RealtorProfileRegistry is AccessControl { bytes32 public constant MANAGER_ROLE =
keccak256("MANAGER_ROLE");

struct Profile {
    address owner;
    bytes32 region;          // region or market hash
    bytes32 licenseHash;     // hash of license credential payload
    bytes32 profileCid;      // IPFS style CID stored as bytes32 reference or gateway key
    bool active;
}

mapping(address => Profile) public profiles;
mapping(address => mapping(uint256 => bool)) public linkedAsset; // addr -> assetId -> linked

event Registered(address indexed user, bytes32 region, bytes32 licenseHash, bytes32
profileCid);
event Updated(address indexed user, bytes32 region, bytes32 licenseHash, bytes32 profileCid,
bool active);
event AssetLinked(address indexed user, uint256 assetId, bool linked);

constructor(address admin) {
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(MANAGER_ROLE, admin);
}

function register(bytes32 region, bytes32 licenseHash, bytes32 profileCid) external {
    require(profiles[msg.sender].owner == address(0), "exists");
    profiles[msg.sender] = Profile({
        owner: msg.sender,
        region: region,
        licenseHash: licenseHash,
        profileCid: profileCid,
        active: true
    });
    emit Registered(msg.sender, region, licenseHash, profileCid);
}

function update(bytes32 region, bytes32 licenseHash, bytes32 profileCid, bool active) external {
    Profile storage p = profiles[msg.sender];
    require(p.owner == msg.sender, "not owner");
    p.region = region;
    p.licenseHash = licenseHash;
    p.profileCid = profileCid;
    p.active = active;
    emit Updated(msg.sender, region, licenseHash, profileCid, active);
}

function managerSetActive(address user, bool active) external onlyRole(MANAGER_ROLE) {
    require(profiles[user].owner == user, "no profile");
    profiles[user].active = active;
    emit Updated(user, profiles[user].region, profiles[user].licenseHash, profiles[user].profileCid,
active);
}

function setAssetLink(uint256 assetId, bool on) external {
    require(profiles[msg.sender].owner == msg.sender, "no profile");
    linkedAsset[msg.sender][assetId] = on;
    emit AssetLinked(msg.sender, assetId, on);
}

function hasProfile(address user) external view returns (bool) {
    return profiles[user].owner == user && profiles[user].active;
}

}

contracts/RealtorBadgeNFT.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol"; import
"@openzeppelin/contracts/access/AccessControl.sol";

contract RealtorBadgeNFT is ERC721, AccessControl { bytes32 public constant ISSUER_ROLE
= keccak256("ISSUER_ROLE"); bytes32 public constant MANAGER_ROLE =
keccak256("MANAGER_ROLE");

uint256 public nextId = 1;
bool public soulbound = true;

mapping(uint256 => bytes32) public badgeType;  // type key
mapping(uint256 => bytes32) public metadataCid;

event Minted(uint256 indexed tokenId, address indexed to, bytes32 badge, bytes32 cid);
event Burned(uint256 indexed tokenId);
event SoulboundSet(bool enabled);
event MetadataSet(uint256 indexed tokenId, bytes32 cid);

constructor(address admin) ERC721("Axiom Realtor Badge", "ARB") {
  _grantRole(DEFAULT_ADMIN_ROLE, admin);
  _grantRole(ISSUER_ROLE, admin);
  _grantRole(MANAGER_ROLE, admin);
}

function setSoulbound(bool enabled) external onlyRole(MANAGER_ROLE) {
  soulbound = enabled;
  emit SoulboundSet(enabled);
}

function mint(address to, bytes32 badge, bytes32 cid) external onlyRole(ISSUER_ROLE)
returns (uint256 id) {
  require(to != address(0), "zero addr");
  id = nextId++;
  _safeMint(to, id);
  badgeType[id] = badge;
  if (cid != bytes32(0)) {
    metadataCid[id] = cid;
    emit MetadataSet(id, cid);
  }
  emit Minted(id, to, badge, cid);
}

function burn(uint256 tokenId) external onlyRole(MANAGER_ROLE) {
  _burn(tokenId);
  emit Burned(tokenId);
}

function setMetadata(uint256 tokenId, bytes32 cid) external onlyRole(MANAGER_ROLE) {
  require(_exists(tokenId), "no token");
  metadataCid[tokenId] = cid;
  emit MetadataSet(tokenId, cid);
}

function _beforeTokenTransfer(address from, address to, uint256 tokenId, uint256 batchSize)
internal override {
  super._beforeTokenTransfer(from, to, tokenId, batchSize);
  if (soulbound && from != address(0) && to != address(0)) revert("soulbound");
}

}
```

contracts/RealtorReferralRouter.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/AccessControl.sol"; import "@openzeppelin/contracts/security/ReentrancyGuard.sol"; import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract RealtorReferralRouter is AccessControl, ReentrancyGuard { bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");

IERC20 public immutable axm;

mapping(address => address) public referrerOf;

```
mapping(address => uint256) public accrued;

event ReferrerSet(address indexed user, address indexed referrer);
event Credited(address indexed referrer, uint256 amount, bytes32 refType, bytes32 refId);
event Claimed(address indexed referrer, uint256 amount);

constructor(address admin, address axmToken) {
   _grantRole(DEFAULT_ADMIN_ROLE, admin);
   _grantRole(MANAGER_ROLE, admin);
   axm = IERC20(axmToken);
}

function setReferrer(address referrer) external {
   require(referrer != msg.sender, "self");
   require(referrerOf[msg.sender] == address(0), "already set");
   referrerOf[msg.sender] = referrer;
   emit ReferrerSet(msg.sender, referrer);
}

function credit(address user, uint256 amount, bytes32 refType, bytes32 refId) external
onlyRole(MANAGER_ROLE) {
   address r = referrerOf[user];
   require(r != address(0), "no referrer");
   require(amount > 0, "amount=0");
   accrued[r] += amount;
   emit Credited(r, amount, refType, refId);
}

function claim(address to) external nonReentrant {
   uint256 amt = accrued[msg.sender];
   require(amt > 0, "nothing");
   accrued[msg.sender] = 0;
   require(axm.transfer(to, amt), "transfer fail");
   emit Claimed(msg.sender, amt);
}

}

contracts/VisibilityBoostStaking.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/AccessControl.sol"; import
"@openzeppelin/contracts/security/ReentrancyGuard.sol"; import
"@openzeppelin/contracts/token/ERC20/IERC20.sol"; import "./RealtorProfileRegistry.sol";

contract VisibilityBoostStaking is AccessControl, ReentrancyGuard { bytes32 public constant
MANAGER_ROLE = keccak256("MANAGER_ROLE");

IERC20 public immutable axm;
RealtorProfileRegistry public immutable profiles;

uint256 public totalStaked;
mapping(address => uint256) public staked;

// boost parameters
uint256 public k;          // curvature numerator
uint256 public kDen;       // curvature denominator
uint256 public maxBoost;   // hard cap as 1e18 fixed point

event Staked(address indexed user, uint256 amount);
event Withdrawn(address indexed user, uint256 amount);
event ParamsSet(uint256 k, uint256 kDen, uint256 maxBoost);

constructor(address admin, address axmToken, address profileRegistry, uint256 _k, uint256
_kDen, uint256 _maxBoost) {
   _grantRole(DEFAULT_ADMIN_ROLE, admin);
   _grantRole(MANAGER_ROLE, admin);
   axm = IERC20(axmToken);
   profiles = RealtorProfileRegistry(profileRegistry);
   k = _k;
   kDen = _kDen;
   maxBoost = _maxBoost;
   emit ParamsSet(k, kDen, maxBoost);
}

function setParams(uint256 _k, uint256 _kDen, uint256 _maxBoost) external
onlyRole(MANAGER_ROLE) {
   k = _k;
   kDen = _kDen;
   maxBoost = _maxBoost;
   emit ParamsSet(k, kDen, maxBoost);
}

function stake(uint256 amount) external nonReentrant {
   require(amount > 0, "amount=0");
   require(profiles.hasProfile(msg.sender), "no profile");
   staked[msg.sender] += amount;
   totalStaked += amount;
   require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
   emit Staked(msg.sender, amount);
}

function withdraw(uint256 amount) external nonReentrant {
   require(amount > 0 && staked[msg.sender] >= amount, "bad amount");
   staked[msg.sender] -= amount;
   totalStaked -= amount;
   require(axm.transfer(msg.sender, amount), "transfer fail");
   emit Withdrawn(msg.sender, amount);
}

function boostPowerOf(address user) external view returns (uint256) {
   if (staked[user] == 0) return 0;
   uint256 b = (staked[user] * k) / kDen;
   if (b > maxBoost) return maxBoost;
   return b;
}

}

contracts/RealtorReputationOracle.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/AccessControl.sol"; import
"./RealtorProfileRegistry.sol";

contract RealtorReputationOracle is AccessControl { bytes32 public constant ORACLE_ROLE =
keccak256("ORACLE_ROLE");

RealtorProfileRegistry public immutable profiles;
```

```
// score is 0 to 10000 fixed point meaning up to 100.00
mapping(address => uint256) public scoreOfUser;
mapping(address => uint256) public lastEpoch;

event ScoreSet(address indexed user, uint256 score, uint256 epoch);

constructor(address admin, address profileRegistry) {
   _grantRole(DEFAULT_ADMIN_ROLE, admin);
   _grantRole(ORACLE_ROLE, admin);
   profiles = RealtorProfileRegistry(profileRegistry);
}

function setScore(address user, uint256 score, uint256 epoch) external
onlyRole(ORACLE_ROLE) {
   require(profiles.hasProfile(user), "no profile");
   require(score <= 10000, "too high");
   scoreOfUser[user] = score;
   lastEpoch[user] = epoch;
   emit ScoreSet(user, score, epoch);
}

}

contracts/RealtorCouncil.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/AccessControl.sol"; import
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Votes.sol"; import
"./RealtorProfileRegistry.sol";

contract RealtorCouncil is AccessControl { bytes32 public constant PROPOSER_ROLE =
keccak256("PROPOSER_ROLE");

ERC20Votes public immutable axmVotes;
RealtorProfileRegistry public immutable profiles;

uint256 public quorumBps;
uint256 public votingDelayBlocks;
uint256 public votingPeriodBlocks;

enum Vote { None, For, Against, Abstain }

struct Proposal {
   address proposer;
   bytes32 descriptionHash;
   uint256 snapshotBlock;
   uint256 endBlock;
   uint256 forVotes;
   uint256 againstVotes;
   uint256 abstainVotes;
   bool executed;
}

uint256 public nextId = 1;
mapping(uint256 => Proposal) public proposals;
mapping(uint256 => mapping(address => Vote)) public receipts;

event ProposalCreated(uint256 id, address proposer, bytes32 descriptionHash, uint256
snapshotBlock, uint256 endBlock);
event VoteCast(uint256 id, address voter, Vote vote, uint256 weight);
event MarkExecuted(uint256 id);

constructor(address admin, address axmTokenVotes, address profileRegistry, uint256
_quorumBps, uint256 _delay, uint256 _period) {
   _grantRole(DEFAULT_ADMIN_ROLE, admin);
   _grantRole(PROPOSER_ROLE, admin);
   axmVotes = ERC20Votes(axmTokenVotes);
   profiles = RealtorProfileRegistry(profileRegistry);
   quorumBps = _quorumBps;
   votingDelayBlocks = _delay;
   votingPeriodBlocks = _period;
}

modifier onlyVerified(address user) {
   require(profiles.hasProfile(user), "not verified");
   _;
}

function setParams(uint256 _quorumBps, uint256 _delay, uint256 _period) external
onlyRole(DEFAULT_ADMIN_ROLE) {
   quorumBps = _quorumBps;
   votingDelayBlocks = _delay;
   votingPeriodBlocks = _period;
}

function propose(bytes32 descriptionHash) external onlyVerified(msg.sender) returns (uint256
id) {
   uint256 start = block.number + votingDelayBlocks;
   uint256 end = start + votingPeriodBlocks;
   id = nextId++;
   proposals[id] = Proposal({
      proposer: msg.sender,
      descriptionHash: descriptionHash,
      snapshotBlock: start,
      endBlock: end,
      forVotes: 0,
      againstVotes: 0,
      abstainVotes: 0,
      executed: false
   });
   emit ProposalCreated(id, msg.sender, descriptionHash, start, end);
}

function castVote(uint256 id, uint8 voteType) external onlyVerified(msg.sender) {
   Proposal storage p = proposals[id];
   require(block.number >= p.snapshotBlock && block.number <= p.endBlock, "not active");
   require(receipts[id][msg.sender] == Vote.None, "voted");

   Vote v = Vote(voteType);
   require(v == Vote.For || v == Vote.Against || v == Vote.Abstain, "bad vote");

   uint256 weight = axmVotes.getPastVotes(msg.sender, p.snapshotBlock);
   require(weight > 0, "no voting power");
   receipts[id][msg.sender] = v;

   if (v == Vote.For) p.forVotes += weight;
   else if (v == Vote.Against) p.againstVotes += weight;
   else p.abstainVotes += weight;
```

```
    emit VoteCast(id, msg.sender, v, weight);
}

function quorumReached(uint256 id) public view returns (bool) {
    Proposal memory p = proposals[id];
    uint256 total = axmVotes.getPastTotalSupply(p.snapshotBlock);
    uint256 needed = (total * quorumBps) / 10000;
    return p.forVotes + p.againstVotes + p.abstainVotes >= needed;
}

function isSucceeded(uint256 id) public view returns (bool) {
    Proposal memory p = proposals[id];
    if (block.number <= p.endBlock) return false;
    if (!quorumReached(id)) return false;
    return p.forVotes > p.againstVotes;
}

function markExecuted(uint256 id) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(isSucceeded(id), "not succeeded");
    proposals[id].executed = true;
    emit MarkExecuted(id);
}

}
```

Deploy scripts Create these files in scripts folder. They follow your pattern and write addresses.json. Replace placeholders where noted.

scripts/21_deploy_RealtorProfileRegistry.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const Admin = deployer.address; const F = await ethers.getContractFactory("RealtorProfileRegistry"); const c = await F.deploy(Admin); await c.deployed(); writeAddresses(network.name, { RealtorProfileRegistry: c.address }); console.log("RealtorProfileRegistry", c.address); }; module.exports.tags = ["realtor-profile"];

scripts/22_deploy_RealtorBadgeNFT.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const Admin = deployer.address; const F = await ethers.getContractFactory("RealtorBadgeNFT"); const c = await F.deploy(Admin); await c.deployed(); writeAddresses(network.name, { RealtorBadgeNFT: c.address }); console.log("RealtorBadgeNFT", c.address); }; module.exports.tags = ["realtor-badge"];

scripts/23_deploy_RealtorReferralRouter.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const addrs = require("../addresses.json")[network.name]; const axm = addrs.AXM; const Admin = deployer.address; const F = await ethers.getContractFactory("RealtorReferralRouter"); const c = await F.deploy(Admin, axm); await c.deployed(); writeAddresses(network.name, { RealtorReferralRouter: c.address }); console.log("RealtorReferralRouter", c.address); }; module.exports.tags = ["realtor-referral"];

scripts/24_deploy_VisibilityBoostStaking.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const addrs = require("../addresses.json")[network.name]; const axm = addrs.AXM; const profile = addrs.RealtorProfileRegistry; const Admin = deployer.address; const k = ethers.utils.parseUnits("1", 18); const kDen = ethers.utils.parseUnits("10000", 18); const maxBoost = ethers.utils.parseUnits("2", 18); const F = await ethers.getContractFactory("VisibilityBoostStaking"); const c = await F.deploy(Admin, axm, profile, k, kDen, maxBoost); await c.deployed(); writeAddresses(network.name, { VisibilityBoostStaking: c.address }); console.log("VisibilityBoostStaking", c.address); }; module.exports.tags = ["realtor-boost"];

scripts/25_deploy_RealtorReputationOracle.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const addrs = require("../addresses.json")[network.name]; const profile = addrs.RealtorProfileRegistry; const Admin = deployer.address; const F = await ethers.getContractFactory("RealtorReputationOracle"); const c = await F.deploy(Admin, profile); await c.deployed(); writeAddresses(network.name, { RealtorReputationOracle: c.address }); console.log("RealtorReputationOracle", c.address); }; module.exports.tags = ["realtor-rep"];

scripts/26_deploy_RealtorCouncil.js const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const addrs = require("../addresses.json")[network.name]; const axmVotes = addrs.AXM; const profile = addrs.RealtorProfileRegistry; const Admin = deployer.address; const quorumBps = 500;   // 5 percent const delay = 5;   // 5 blocks const period = 20000;   // about an hour depending on final chain params const F = await ethers.getContractFactory("RealtorCouncil"); const c = await F.deploy(Admin, axmVotes, profile, quorumBps, delay, period); await c.deployed(); writeAddresses(network.name, { RealtorCouncil: c.address }); console.log("RealtorCouncil", c.address); }; module.exports.tags = ["realtor-council"];

Post deploy role grants Create scripts/27_grant_roles_realtor_stack.js

const { writeAddresses } = require("./util/writeAddresses"); module.exports = async function({ ethers, network }) { const [deployer] = await ethers.getSigners(); const addrs = require("../addresses.json")[network.name];

const profile = await ethers.getContractAt("RealtorProfileRegistry", addrs.RealtorProfileRegistry); const badge = await ethers.getContractAt("RealtorBadgeNFT", addrs.RealtorBadgeNFT); const referral = await ethers.getContractAt("RealtorReferralRouter", addrs.RealtorReferralRouter); const boost = await ethers.getContractAt("VisibilityBoostStaking", addrs.VisibilityBoostStaking); const rep = await ethers.getContractAt("RealtorReputationOracle", addrs.RealtorReputationOracle); const council = await ethers.getContractAt("Real