Nice—let's round out the suite with the remaining modules. All are PEAQ-ready, constructor-wired, non-upgradeable, and low-risk by design.

NodeLicenseNFT.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

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

    uint256 public nextId = 1;

    // Optional: encode metadata in IPFS CID (bytes32) to keep on-chain light
    mapping(uint256 => bytes32) public metadataCid;

    // Soulbound behavior toggle; default true for non-transferable licenses
    bool public soulbound = true;

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

    constructor(address admin) ERC721("Axiom Node License", "AXN") {
        _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 cid) external onlyRole(ISSUER_ROLE) returns (uint256 id)
{
        require(to != address(0), "zero addr");
        id = nextId++;
        _safeMint(to, id);
        if (cid != bytes32(0)) {
            metadataCid[id] = cid;
            emit MetadataSet(id, cid);
        }
        emit Minted(id, to, 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");
    }
}
```

AXIOMLaunchpad.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";

// Minimal AXM-denominated sale escrow. Tracks allocations; no external token custody.
// Organizer later distributes sale token off-chain or via a separate claim contract.
contract AXIOMLaunchpad is AccessControl, ReentrancyGuard {
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    IERC20 public immutable axm;

    uint64  public start;
    uint64  public end;
    uint256 public softCap;      // in AXM wei
    uint256 public hardCap;      // in AXM wei
    uint256 public price;        // quote: saleUnits per 1 AXM (fixed point 1e18)

    bool    public cancelled;
    uint256 public raised;

    mapping(address => uint256) public contributed; // AXM contributed
    mapping(address => uint256) public allocation;  // sale units purchased

    event ConfigSet(uint64 start, uint64 end, uint256 softCap, uint256 hardCap, uint256 price);
    event Contributed(address indexed user, uint256 axmAmount, uint256 unitsOut);
    event Cancelled();
    event Withdrawn(address indexed to, uint256 amount);
    event Refunded(address indexed user, uint256 amount);

    constructor(address admin, address axmToken, uint64 _start, uint64 _end, uint256 _softCap,
uint256 _hardCap, uint256 _price) {
        require(admin != address(0) && axmToken != address(0), "zero addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
        axm = IERC20(axmToken);
        setConfig(_start, _end, _softCap, _hardCap, _price);
    }

    function setConfig(uint64 _start, uint64 _end, uint256 _softCap, uint256 _hardCap, uint256
_price) public onlyRole(MANAGER_ROLE) {
        require(_start < _end && _price > 0 && _hardCap >= _softCap, "bad params");
        start = _start; end = _end; softCap = _softCap; hardCap = _hardCap; price = _price;
        emit ConfigSet(start, end, softCap, hardCap, price);
    }
```

```
    function contribute(uint256 axmAmount) external nonReentrant {
        require(!cancelled, "cancelled");
        require(block.timestamp >= start && block.timestamp <= end, "not active");
        require(axmAmount > 0 && raised + axmAmount <= hardCap, "cap reached");
        require(axm.transferFrom(msg.sender, address(this), axmAmount), "transferFrom fail");
        raised += axmAmount;
        contributed[msg.sender] += axmAmount;
        uint256 units = (axmAmount * price) / 1e18;
        allocation[msg.sender] += units;
        emit Contributed(msg.sender, axmAmount, units);
    }

    function cancel() external onlyRole(MANAGER_ROLE) {
        cancelled = true;
        emit Cancelled();
    }

    function withdrawProceeds(address to) external onlyRole(MANAGER_ROLE) nonReentrant {
        require(!cancelled, "cancelled");
        require(block.timestamp > end && raised >= softCap, "not finalized");
        uint256 bal = axm.balanceOf(address(this));
        require(axm.transfer(to, bal), "transfer fail");
        emit Withdrawn(to, bal);
    }

    function refund() external nonReentrant {
        require(cancelled || (block.timestamp > end && raised < softCap), "no refund");
        uint256 amt = contributed[msg.sender];
        require(amt > 0, "none");
        contributed[msg.sender] = 0;
        allocation[msg.sender] = 0;
        require(axm.transfer(msg.sender, amt), "transfer fail");
        emit Refunded(msg.sender, amt);
    }
}
```

CrossChainBridgeAdapter.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";

// Canonical locker for AXM on PEAQ. Locks AXM and emits events for an external bridge to
mint on a remote chain.
// When tokens return from the remote chain, an authorized relayer unlocks to a recipient.
contract CrossChainBridgeAdapter is AccessControl, ReentrancyGuard {
    bytes32 public constant RELAYER_ROLE = keccak256("RELAYER_ROLE");
    IERC20 public immutable axm;

    event Locked(address indexed from, bytes32 indexed dstChain, bytes32 indexed dstAddress,
uint256 amount, bytes32 lockId);
    event Unlocked(address indexed to, uint256 amount, bytes32 srcChain, bytes32 burnTx);

    mapping(bytes32 => bool) public processed; // prevent double unlocks per source event
```

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

    function lock(bytes32 dstChain, bytes32 dstAddress, uint256 amount, bytes32 lockId)
external nonReentrant {
        require(amount > 0, "amount=0");
        require(!processed[lockId], "dup lockId");
        processed[lockId] = true; // avoid user reusing same id
        require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
        emit Locked(msg.sender, dstChain, dstAddress, amount, lockId);
    }

    function unlock(address to, uint256 amount, bytes32 srcChain, bytes32 burnTx) external
nonReentrant onlyRole(RELAYER_ROLE) {
        require(to != address(0) && amount > 0, "bad args");
        require(!processed[burnTx], "already unlocked");
        processed[burnTx] = true;
        require(axm.transfer(to, amount), "transfer fail");
        emit Unlocked(to, amount, srcChain, burnTx);
    }
}
```

AXIOMAnalyticsHub.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

// On-chain index of per-epoch analytics references for nodes: store CIDs, checksums, or signed
digests.
contract AXIOMAnalyticsHub is AccessControl {
    bytes32 public constant PUBLISHER_ROLE = keccak256("PUBLISHER_ROLE");

    // nodeId => epoch => digest
    mapping(uint256 => mapping(uint256 => bytes32)) public reportDigest;
    // optional human-readable pointer: nodeId => epoch => cid
    mapping(uint256 => mapping(uint256 => bytes32)) public reportCid;

    event Published(uint256 indexed nodeId, uint256 indexed epoch, bytes32 digest, bytes32
cid);

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

    function publish(uint256 nodeId, uint256 epoch, bytes32 digest, bytes32 cid) external
onlyRole(PUBLISHER_ROLE) {
        require(nodeId != 0 && epoch != 0 && digest != bytes32(0), "bad args");
        reportDigest[nodeId][epoch] = digest;
        if (cid != bytes32(0)) reportCid[nodeId][epoch] = cid;
        emit Published(nodeId, epoch, digest, cid);
    }
```

```
    function get(uint256 nodeId, uint256 epoch) external view returns (bytes32 digest, bytes32
cid) {
        return (reportDigest[nodeId][epoch], reportCid[nodeId][epoch]);
    }
}
```

ComplianceAdapter.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

// Stateless gatekeeper helpers that other modules can consult.
contract ComplianceAdapter is AccessControl {
    DIDRegistry public immutable did;
    bytes32 public immutable KYC_SCHEMA;
    bytes32 public immutable ACCREDITED_SCHEMA;
    bytes32 public immutable NON_US_SCHEMA;

    event Decision(bytes32 indexed rule, address indexed subject, bool allowed);

    constructor(address admin, address didRegistry, bytes32 kycSchema, bytes32
accreditedSchema, bytes32 nonUsSchema) {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        did = DIDRegistry(didRegistry);
        KYC_SCHEMA = kycSchema;
        ACCREDITED_SCHEMA = accreditedSchema;
        NON_US_SCHEMA = nonUsSchema;
    }

    function checkBasicKYC(address user) public view returns (bool) {
        return KYC_SCHEMA == bytes32(0) ? true : did.isValid(user, KYC_SCHEMA);
    }

    function checkAccredited(address user) public view returns (bool) {
        return ACCREDITED_SCHEMA == bytes32(0) ? true : did.isValid(user,
ACCREDITED_SCHEMA);
    }

    function checkNonUS(address user) public view returns (bool) {
        return NON_US_SCHEMA == bytes32(0) ? true : did.isValid(user, NON_US_SCHEMA);
    }

    function allowPublicSale(address user) external view returns (bool ok) {
        ok = checkBasicKYC(user);
    }

    function allowRegD(address user) external view returns (bool ok) {
        ok = checkBasicKYC(user) && checkAccredited(user);
    }

    function allowRegS(address user) external view returns (bool ok) {
        ok = checkBasicKYC(user) && checkNonUS(user);
    }
}
```

Want deploy scripts for these too? I can mirror your existing pattern so they drop addresses into the same addresses.json and wire role grants cleanly.

Got it. Here is the full Operational Layer for PEAQ. All contracts are constructor-wired, no minting, AccessControl based, and compatible with AXM as the single reward and treasury token. Comments use only line slashes to respect your formatting rule.

LiquidityVault.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 LiquidityVault is AccessControl, ReentrancyGuard {
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    bytes32 public constant INCENTIVE_ROLE = keccak256("INCENTIVE_ROLE");

    IERC20 public immutable axm;          // reward and treasury token
    IERC20 public immutable lpToken;      // LP token custody

    address public gauge;                 // external rewards distributor, optional
    bool    public lpWithdrawLocked;      // optional lock for LP tokens

    event LPDeposited(address indexed from, uint256 amount);
    event LPWithdrawn(address indexed to, uint256 amount);
    event IncentivesFunded(address indexed to, uint256 amount);
    event GaugeSet(address indexed gauge);
    event LPWithdrawLockSet(bool locked);

    constructor(address admin, address axmToken, address lpTokenAddress) {
        require(admin != address(0) && axmToken != address(0) && lpTokenAddress !=
address(0), "zero addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
        _grantRole(INCENTIVE_ROLE, admin);
        axm = IERC20(axmToken);
        lpToken = IERC20(lpTokenAddress);
    }

    function depositLP(uint256 amount) external nonReentrant onlyRole(MANAGER_ROLE) {
        require(amount > 0, "amount=0");
        require(lpToken.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
        emit LPDeposited(msg.sender, amount);
    }

    function withdrawLP(address to, uint256 amount) external nonReentrant
onlyRole(MANAGER_ROLE) {
        require(!lpWithdrawLocked, "lp locked");
        require(to != address(0) && amount > 0, "bad args");
        require(lpToken.transfer(to, amount), "transfer fail");
        emit LPWithdrawn(to, amount);
    }
```

```
    function setLPWithdrawLocked(bool locked) external onlyRole(MANAGER_ROLE) {
        lpWithdrawLocked = locked;
        emit LPWithdrawLockSet(locked);
    }

    function setGauge(address newGauge) external onlyRole(MANAGER_ROLE) {
        gauge = newGauge;
        emit GaugeSet(newGauge);
    }

    function fundIncentives(uint256 amount, address recipient) external nonReentrant
onlyRole(INCENTIVE_ROLE) {
        require(recipient != address(0) && amount > 0, "bad args");
        require(axm.transferFrom(msg.sender, address(this), amount), "fund fail");
        require(axm.transfer(recipient, amount), "transfer fail");
        emit IncentivesFunded(recipient, amount);
    }

    function rescueToken(address token, address to, uint256 amount) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        require(to != address(0) && amount > 0, "bad args");
        require(IERC20(token).transfer(to, amount), "rescue fail");
    }
}
```

AXMStakingPool.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

contract AXMStakingPool is AccessControl, ReentrancyGuard {
    bytes32 public constant FUNDER_ROLE = keccak256("FUNDER_ROLE");

    IERC20 public immutable axm;

    uint256 public rewardRate;
    uint256 public lastUpdate;
    uint256 public rewardPerTokenStored;

    uint256 public totalStaked;
    mapping(address => uint256) public balances;
    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event RewardAdded(uint256 amount, uint256 newRate, uint256 duration);
    event RewardClaimed(address indexed user, uint256 amount);

    constructor(address admin, address axmToken) {
        require(admin != address(0) && axmToken != address(0), "zero addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(FUNDER_ROLE, admin);
        axm = IERC20(axmToken);
        lastUpdate = block.timestamp;
```

```
    }

    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdate = block.timestamp;
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }

    function rewardPerToken() public view returns (uint256) {
        if (totalStaked == 0) return rewardPerTokenStored;
        return rewardPerTokenStored + ((rewardRate * (block.timestamp - lastUpdate) * 1e18) /
totalStaked);
    }

    function earned(address account) public view returns (uint256) {
        return (balances[account] * (rewardPerToken() - userRewardPerTokenPaid[account]) /
1e18) + rewards[account];
    }

    function stake(uint256 amount) external nonReentrant updateReward(msg.sender) {
        require(amount > 0, "amount=0");
        totalStaked += amount;
        balances[msg.sender] += amount;
        require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
        emit Staked(msg.sender, amount);
    }

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

    function getReward() external nonReentrant updateReward(msg.sender) {
        uint256 r = rewards[msg.sender];
        require(r > 0, "nothing");
        rewards[msg.sender] = 0;
        require(axm.transfer(msg.sender, r), "transfer fail");
        emit RewardClaimed(msg.sender, r);
    }

    function notifyRewardAmount(uint256 amount, uint256 duration) external nonReentrant
onlyRole(FUNDER_ROLE) updateReward(address(0)) {
        require(amount > 0 && duration > 0, "bad params");
        require(axm.transferFrom(msg.sender, address(this), amount), "fund fail");

        if (block.timestamp >= lastUpdate) {
            rewardRate = amount / duration;
        } else {
            uint256 remaining = lastUpdate - block.timestamp;
            uint256 leftover = remaining * rewardRate;
            rewardRate = (amount + leftover) / duration;
        }
```

```
        lastUpdate = block.timestamp;
        emit RewardAdded(amount, rewardRate, duration);
    }
}
```

GovernanceCouncil.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

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

    ERC20Votes public immutable axm;
    uint256 public quorumBps;            // basis points of total supply
    uint256 public votingDelayBlocks;    // delay before voting starts
    uint256 public votingPeriodBlocks;   // voting duration

    enum Vote { None, For, Against, Abstain }

    struct Proposal {
        address proposer;
        bytes32 descriptionHash;
        uint256 snapshotBlock;
        uint256 endBlock;
        uint256 forVotes;
        uint256 againstVotes;
        uint256 abstainVotes;
        bool executed; // signaling, off chain execution or timelock handled elsewhere
    }

    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 axmToken, uint256 _quorumBps, uint256 _delay, uint256
_period) {
        require(admin != address(0) && axmToken != address(0), "zero addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(PROPOSER_ROLE, admin);
        axm = ERC20Votes(axmToken);
        quorumBps = _quorumBps;
        votingDelayBlocks = _delay;
        votingPeriodBlocks = _period;
    }

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

```
        votingPeriodBlocks = _period;
    }

    function propose(bytes32 descriptionHash) external onlyRole(PROPOSER_ROLE) 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 {
        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 = axm.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 = axm.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);
    }
```

```
}
```

AssetTokenizationRegistry.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

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

    enum Status { Inactive, Active, Archived }

    struct Asset {
        string propertyId;        // external ID or slug
        string location;          // human readable location
        string metadataCid;       // IPFS or similar content ID
        uint256 valuation;        // arbitrary unit, usually USD scaled by 1e2 or 1e6 off chain
        uint256 backingAmount;    // AXM or gold grams count per your policy, off chain verified
        Status status;
    }

    uint256 public nextId = 1;
    mapping(uint256 => Asset) public assets;

    event Registered(uint256 indexed id, string propertyId, string location);
    event Updated(uint256 indexed id, string metadataCid, uint256 valuation, uint256
backingAmount, Status status);

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

    function registerAsset(string calldata propertyId, string calldata location, string calldata
metadataCid, uint256 valuation, uint256 backingAmount) external onlyRole(MANAGER_ROLE)
returns (uint256 id) {
        id = nextId++;
        assets[id] = Asset({
            propertyId: propertyId,
            location: location,
            metadataCid: metadataCid,
            valuation: valuation,
            backingAmount: backingAmount,
            status: Status.Active
        });
        emit Registered(id, propertyId, location);
        emit Updated(id, metadataCid, valuation, backingAmount, Status.Active);
    }

    function updateAsset(uint256 id, string calldata metadataCid, uint256 valuation, uint256
backingAmount, Status status) external onlyRole(MANAGER_ROLE) {
        Asset storage a = assets[id];
        a.metadataCid = metadataCid;
        a.valuation = valuation;
        a.backingAmount = backingAmount;
        a.status = status;
        emit Updated(id, metadataCid, valuation, backingAmount, status);
```

```
    }
}
```

TreasuryManager.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 TreasuryManager is AccessControl, ReentrancyGuard {
    bytes32 public constant OPERATOR_ROLE = keccak256("OPERATOR_ROLE");

    event Pay(address indexed token, address indexed to, uint256 amount, bytes32 refId);
    event Sweep(address indexed token, address indexed to, uint256 amount);

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

    function pay(address token, address to, uint256 amount, bytes32 refId) external
nonReentrant onlyRole(OPERATOR_ROLE) {
        require(token != address(0) && to != address(0) && amount > 0, "bad args");
        require(IERC20(token).transfer(to, amount), "transfer fail");
        emit Pay(token, to, amount, refId);
    }

    function sweep(address token, address to, uint256 amount) external nonReentrant
onlyRole(DEFAULT_ADMIN_ROLE) {
        require(token != address(0) && to != address(0) && amount > 0, "bad args");
        require(IERC20(token).transfer(to, amount), "transfer fail");
        emit Sweep(token, to, amount);
    }
}
```

OracleRelay.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

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

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

    struct Feed {
        int256 value;
        uint64 updatedAt;
        uint64 ttl;
    }

    mapping(bytes32 => Feed) public feeds;

    event FeedSet(bytes32 indexed key, int256 value, uint64 updatedAt, uint64 ttl);

    constructor(address admin) {
```

```
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(ORACLE_ROLE, admin);
    }

    function setFeed(bytes32 key, int256 value, uint64 ttl) external onlyRole(ORACLE_ROLE) {
        feeds[key] = Feed({ value: value, updatedAt: uint64(block.timestamp), ttl: ttl });
        emit FeedSet(key, value, uint64(block.timestamp), ttl);
    }

    function get(bytes32 key) external view returns (int256 value, bool valid) {
        Feed memory f = feeds[key];
        if (f.updatedAt == 0) return (0, false);
        bool fresh = f.ttl == 0 || block.timestamp <= f.updatedAt + f.ttl;
        return (f.value, fresh);
    }
}
```

Next steps

1. Drop these six files into contracts and compile with your existing Hardhat config.

2. Add deploy scripts similar to the prior ones, wiring constructors with admin, AXM address, and any LP token or schema keys.

3. Fund AXMStakingPool and DePINRewardRouter from EmissionsVault or treasury, and grant roles through your multisig or admin account.

4. If you want, I can generate ready to run deploy scripts for these six modules using your current addresses.json pattern.