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.