Absolutely. Let’s round the suite up to twenty by adding the two missing pieces and wiring them into your existing flow.

New contracts added 19) ClaimPortal
20) TimelockGuard

Both are constructor-wired, non-upgradeable, AccessControl based, and EVM compatible for PEAQ.

Contract 19. ClaimPortal.sol Purpose On-chain claiming for presale or airdrop allocations, using a Merkle root. AXM funds are pre-deposited into the portal. Participants prove allocation with a Merkle proof and receive AXM. No owner withdrawals except explicit rescue by admin.

Code

```
// 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 "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

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

    IERC20 public immutable axm;
    bytes32 public merkleRoot;

    mapping(address => uint256) public claimed; // amount already claimed per address

    event RootSet(bytes32 root);
    event Funded(address indexed from, uint256 amount);
    event Claimed(address indexed user, uint256 amount);
    event Rescued(address indexed to, uint256 amount);

    constructor(address admin, address axmToken, bytes32 initialRoot) {
        require(admin != address(0) && axmToken != address(0), "zero addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(FUNDER_ROLE, admin);
        _grantRole(ROOT_ROLE, admin);
        axm = IERC20(axmToken);
        if (initialRoot != bytes32(0)) {
            merkleRoot = initialRoot;
            emit RootSet(initialRoot);
        }
    }

    function setRoot(bytes32 root) external onlyRole(ROOT_ROLE) {
        require(root != bytes32(0), "bad root");
        merkleRoot = root;
        emit RootSet(root);
    }

    function topUp(uint256 amount) external onlyRole(FUNDER_ROLE) nonReentrant {
        require(amount > 0, "amount=0");
        require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
        emit Funded(msg.sender, amount);
```

```
    }

    // leaf format is keccak256(abi.encodePacked(user, allocation))
    function claim(uint256 allocation, uint256 amount, bytes32[] calldata proof) external
nonReentrant {
        require(merkleRoot != bytes32(0), "no root");
        require(amount > 0 && amount + claimed[msg.sender] <= allocation, "bad amount");
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender, allocation));
        require(MerkleProof.verify(proof, merkleRoot, leaf), "bad proof");
        claimed[msg.sender] += amount;
        require(axm.transfer(msg.sender, amount), "transfer fail");
        emit Claimed(msg.sender, amount);
    }

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

Contract 20. TimelockGuard.sol Purpose Adds a governance timelock for privileged actions across the suite. Uses OpenZeppelin TimelockController for battle tested delay semantics. You can later assign admin roles of your modules to the timelock, so all sensitive changes queue then execute after a delay.

Code

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

import "@openzeppelin/contracts/governance/TimelockController.sol";

contract TimelockGuard is TimelockController {
    constructor(
        uint256 minDelay,
        address[] memory proposers,
        address[] memory executors,
        address admin
    )
        TimelockController(minDelay, proposers, executors, admin)
    {}
}
```

How these two fit your system Launch flow options Option A. In-house presale and claim Use AXIOMLaunchpad to record contributions in AXM, then produce a Merkle allocation off-chain and fund ClaimPortal with AXM. Participants claim with proofs. Keeps everything on PEAQ and avoids third-party launchpads.

Option B. External presale, internal claim If you use an external platform for order taking, still settle the final on-chain distribution via ClaimPortal by publishing a Merkle allocation and funding AXM there.

Governance hardening Deploy TimelockGuard and progressively transfer DEFAULT_ADMIN_ROLE and other manager roles from your deployer or multisig to the timelock. All role changes, parameter updates, and router reconfigurations then respect a public delay.

Deployment scripts These mirror your addresses.json pattern and wire roles cleanly. Place alongside your existing scripts.

scripts/05_deploy_TimelockGuard.js

```
const { ethers } = require("hardhat");
const fs = require("fs");

async function main() {
  const addrs = JSON.parse(fs.readFileSync("addresses.json", "utf8"));
  const [deployer] = await ethers.getSigners();

  const minDelay = 3600; // one hour to start; adjust later by redeploying or by policy
  const proposers = [deployer.address];
  const executors = [deployer.address];
  const admin = deployer.address;

  const Timelock = await ethers.getContractFactory("TimelockGuard");
  const timelock = await Timelock.deploy(minDelay, proposers, executors, admin);
  await timelock.deployed();

  addrs.TimelockGuard = timelock.address;
  fs.writeFileSync("addresses.json", JSON.stringify(addrs, null, 2));
  console.log("TimelockGuard", timelock.address);
}

main().catch((e) => { console.error(e); process.exit(1); });
```

scripts/06_deploy_ClaimPortal.js

```
const { ethers } = require("hardhat");
const fs = require("fs");

async function main() {
  const addrs = JSON.parse(fs.readFileSync("addresses.json", "utf8"));
  const [deployer] = await ethers.getSigners();

  const AXM = addrs.AXM;
  if (!AXM) throw new Error("AXM address missing in addresses.json");

  const initialRoot =
"0x0000000000000000000000000000000000000000000000000000000000000000"; // set later

  const ClaimPortal = await ethers.getContractFactory("ClaimPortal");
  const portal = await ClaimPortal.deploy(deployer.address, AXM, initialRoot);
  await portal.deployed();

  addrs.ClaimPortal = portal.address;
  fs.writeFileSync("addresses.json", JSON.stringify(addrs, null, 2));
  console.log("ClaimPortal", portal.address);

  // optional quick role grants to Timelock if already deployed
  if (addrs.TimelockGuard) {
    await (await portal.grantRole(await portal.FUNDER_ROLE(), addrs.TimelockGuard)).wait();
    await (await portal.grantRole(await portal.ROOT_ROLE(), addrs.TimelockGuard)).wait();
    console.log("ClaimPortal roles granted to TimelockGuard");
  }
}
```

main().catch((e) => { console.error(e); process.exit(1); });

Role handoff checklist

1. Deploy TimelockGuard.

2. For each module that exposes roles, grant corresponding manager roles to TimelockGuard, then revoke from the deployer or interim multisig once validated.

3. GovernanceCouncil can remain proposer-driven by council members, with Timelock as executor on any parameterized actions you choose to pipe through timelock.

4. ClaimPortal FUNDER and ROOT roles should be owned by TimelockGuard or a designated distribution multisig.

Replit AI one shot prompt Copy and paste everything between the lines into Replit AI.

Begin prompt Create or update a Hardhat project for Axiom Protocol on PEAQ as follows.

One remove any old contracts and scripts in the repo except the existing addresses.json. Keep node modules and config files if present.

Two add these new Solidity files under contracts AXM.sol EmissionsVault.sol DIDRegistry.sol DePINNodeRegistry.sol DePINRewardRouter.sol AXIOMRevenueRouter.sol RealEstateAcquisitionFund.sol AXMStakingPool.sol LiquidityVault.sol TreasuryManager.sol OracleRelay.sol AssetTokenizationRegistry.sol GovernanceCouncil.sol NodeLicenseNFT.sol AXIOMLaunchpad.sol CrossChainBridgeAdapter.sol AXIOMAnalyticsHub.sol ComplianceAdapter.sol TimelockGuard.sol ClaimPortal.sol

Use the exact source code for TimelockGuard.sol and ClaimPortal.sol from the latest assistant message. For the other eighteen, use the latest versions already present in this project from our previous step.

Three add or update scripts in scripts 00_deploy_AXM.js 01_deploy_EmissionsVault.js 02_deploy_DIDRegistry.js 03_deploy_DePIN.js 04_deploy_RevenueAndRAF.js 05_deploy_TimelockGuard.js content provided in chat 06_deploy_ClaimPortal.js content provided in chat

Ensure each script reads and writes addresses.json and uses the same constructor wiring as earlier. In script 03 wire DePINNodeRegistry with DIDRegistry and license schema constant, and wire DePINRewardRouter with AXM, DePINNodeRegistry, DIDRegistry, and optional KYC schema. In script 04 wire AXIOMRevenueRouter with AXM and the four sink wallets or modules, then deploy RealEstateAcquisitionFund with AXM.

Four set hardhat.config.js with PEAQ RPC and chain id 3338 and optimizer on. Ensure dotenv is used to load DEPLOYER_PK. If hardhat toolbox is missing, add it and update package.json and lockfile.

Five compile the full suite and show a summary of compiler output.

Six deploy in order to the peaq network using the deployer private key in .env npx hardhat run scripts/00_deploy_AXM.js --network peaq npx hardhat run scripts/01_deploy_EmissionsVault.js --network peaq npx hardhat run scripts/02_deploy_DIDRegistry.js --network peaq npx hardhat run scripts/03_deploy_DePIN.js --network peaq npx hardhat run

scripts/04_deploy_RevenueAndRAF.js --network peaq npx hardhat run scripts/05_deploy_TimelockGuard.js --network peaq npx hardhat run scripts/06_deploy_ClaimPortal.js --network peaq

Seven immediately grant roles from the deployer to TimelockGuard where applicable Grant manager and operator roles on LiquidityVault, AXMStakingPool, TreasuryManager, DePINRewardRouter, DIDRegistry issuer or registrar as needed, AssetTokenizationRegistry manager, AXIOMRevenueRouter default admin or leave under multisig, and ClaimPortal FUNDER and ROOT. Then revoke those roles from the deployer.

Eight print the final addresses.json and a role map per contract.

End prompt

Quick sanity notes You now have twenty contracts.
ClaimPortal gives you a clean, auditable on-chain distribution path for presale and airdrop without relying on a third-party launchpad.
TimelockGuard lets you gradually shift control from deployer to time delayed governance, reducing operational risk and improving trust with early adopters and exchanges.

If you want, next we can auto generate the Merkle allocation builder script in Node for ClaimPortal and a role-handoff script that grants and revokes in one transaction batch via Timelock scheduling.

Absolutely. I've added two focused, PEAQ-ready modules for KeyGrow and wired them to your existing stack without proxies or hidden minting:

PropertyShareToken.sol — fixed-supply ERC20 per property, minted exactly once at deployment

KeyGrowLeaseManager.sol — rent-to-own engine handling AXM payments, splits, accruals, completion, default, and optional KYC checks

Below is a single copy-paste prompt for Replit AI that will:

add both contracts into contracts/

create two deploy scripts

register addresses into addresses.json

compile and give you ready commands

Task Extend the existing Axiom peaq Hardhat project by adding two contracts for the KeyGrow rent-to-own flow, plus deploy scripts and NPM commands. Use Solidity ^0.8.20, no upgradables, constructor wiring only, AccessControl, and AXM as the sole payment unit.

Files to create in contracts/

1. PropertyShareToken.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;

```
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol"; import
"@openzeppelin/contracts/access/AccessControl.sol";

// Fixed supply per property, minted once at deploy to a recipient. // Use one token per property
you tokenize. contract PropertyShareToken is ERC20Capped, AccessControl { bytes32 public
constant META_ROLE = keccak256("META_ROLE");

// optional human pointer to the ATR entry this PST represents
uint256 public immutable assetId; // matches AssetTokenizationRegistry id

event MetadataNote(bytes32 note);

constructor(
    string memory name_,
    string memory symbol_,
    uint256 cap_,               // total supply (wei)
    address admin_,             // DEFAULT_ADMIN_ROLE
    address recipient_,         // receives full cap on deploy
    uint256 assetId_,           // AssetTokenizationRegistry id
    bytes32 metaNote            // optional pointer hash (CID, etc)
) ERC20(name_, symbol_) ERC20Capped(cap_) {
    require(admin_ != address(0) && recipient_ != address(0), "bad args");
    _grantRole(DEFAULT_ADMIN_ROLE, admin_);
    _grantRole(META_ROLE, admin_);
    assetId = assetId_;
    _mint(recipient_, cap_); // single, capped mint
    if (metaNote != bytes32(0)) emit MetadataNote(metaNote);
}

// required by ERC20Capped
function _update(address from, address to, uint256 value) internal override(ERC20,
ERC20Capped) {
    super._update(from, to, value);
}

}

2. KeyGrowLeaseManager.sol // SPDX-License-Identifier: MIT pragma solidity ^0.8.20;



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

// AXM-denominated rent-to-own engine for KeyGrow. // Handles monthly payments,
configurable splits, completion, defaults, and optional KYC gating. contract
KeyGrowLeaseManager is AccessControl, ReentrancyGuard { bytes32 public constant
MANAGER_ROLE = keccak256("MANAGER_ROLE"); bytes32 public constant
CASHIER_ROLE = keccak256("CASHIER_ROLE"); // can receive funds and forward into sinks

IERC20 public immutable axm;
ComplianceAdapter public immutable compliance; // can be address(0) to disable gating
AssetTokenizationRegistry public immutable atr;

// sinks
address public rafSink;       // RealEstateAcquisitionFund or ops wallet
address public maintSink;     // maintenance reserve
```

```
address public treasurySink;  // protocol treasury

// split basis points of each payment; must sum to 10000
uint16 public bpToLandlord;
uint16 public bpToRAF;
uint16 public bpToMaint;
uint16 public bpToTreasury;

enum Status { Inactive, Active, Completed, Defaulted }

struct Lease {
    address tenant;
    address landlord;          // property owner or seller wallet
    uint256 assetId;           // ATR id
    uint64  startTs;
    uint16  termMonths;        // number of required payments
    uint16  paidMonths;        // counter
    uint256 monthlyAXM;        // AXM wei required per month
    uint256 buyoutAXM;         // optional early buyout (AXM wei); 0 = disabled
    Status  status;
    bool    kycRequired;       // if true, require compliance.allowPublicSale(tenant)
}

uint256 public nextLeaseId = 1;
mapping(uint256 => Lease) public leases;
mapping(uint256 => uint256) public paidAmount; // total AXM paid per lease

event LeaseCreated(uint256 indexed leaseId, address indexed tenant, address indexed
landlord, uint256 assetId, uint16 termMonths, uint256 monthlyAXM, uint256 buyoutAXM, bool
kyc);
event Payment(uint256 indexed leaseId, address indexed payer, uint256 amount, uint16
newPaidMonths);
event Completed(uint256 indexed leaseId);
event Defaulted(uint256 indexed leaseId);
event Buyout(uint256 indexed leaseId, address indexed buyer, uint256 amount);
event SplitsUpdated(uint16 toLandlord, uint16 toRAF, uint16 toMaint, uint16 toTreasury);
event SinksUpdated(address raf, address maint, address treasury);

constructor(
    address admin,
    address axmToken,
    address complianceAdapter,   // set to 0x0 to disable gating
    address atrAddress,
    address rafSink_,
    address maintSink_,
    address treasurySink_,
    uint16 bpLandlord,
    uint16 bpRAF,
    uint16 bpMaint,
    uint16 bpTreasury
) {
    require(admin != address(0) && axmToken != address(0) && atrAddress != address(0), "bad
args");
    require(rafSink_ != address(0) && maintSink_ != address(0) && treasurySink_ != address(0),
"zero sink");
    require(uint32(bpLandlord) + bpRAF + bpMaint + bpTreasury == 10_000, "bps!=100%");
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(MANAGER_ROLE, admin);
    _grantRole(CASHIER_ROLE, admin);
```

```
    axm = IERC20(axmToken);
    compliance = complianceAdapter == address(0) ? ComplianceAdapter(address(0)) :
ComplianceAdapter(complianceAdapter);
    atr = AssetTokenizationRegistry(atrAddress);
    rafSink = rafSink_;
    maintSink = maintSink_;
    treasurySink = treasurySink_;
    bpToLandlord = bpLandlord;
    bpToRAF = bpRAF;
    bpToMaint = bpMaint;
    bpToTreasury = bpTreasury;
}

function setSinks(address raf, address maint, address tsy) external
onlyRole(MANAGER_ROLE) {
    require(raf != address(0) && maint != address(0) && tsy != address(0), "zero sink");
    rafSink = raf; maintSink = maint; treasurySink = tsy;
    emit SinksUpdated(raf, maint, tsy);
}

function setSplits(uint16 toLandlord, uint16 toRAF, uint16 toMaint, uint16 toTreasury) external
onlyRole(MANAGER_ROLE) {
    require(uint32(toLandlord) + toRAF + toMaint + toTreasury == 10_000, "bps!=100%");
    bpToLandlord = toLandlord; bpToRAF = toRAF; bpToMaint = toMaint; bpToTreasury =
toTreasury;
    emit SplitsUpdated(toLandlord, toRAF, toMaint, toTreasury);
}

function createLease(
    address tenant,
    address landlord,
    uint256 assetId,
    uint16 termMonths,
    uint256 monthlyAXM,
    uint256 buyoutAXM,
    bool kycRequired
) external onlyRole(MANAGER_ROLE) returns (uint256 id) {
    require(tenant != address(0) && landlord != address(0), "zero addr");
    require(termMonths > 0 && monthlyAXM > 0, "bad params");
    id = nextLeaseId++;
    leases[id] = Lease({
        tenant: tenant,
        landlord: landlord,
        assetId: assetId,
        startTs: uint64(block.timestamp),
        termMonths: termMonths,
        paidMonths: 0,
        monthlyAXM: monthlyAXM,
        buyoutAXM: buyoutAXM,
        status: Status.Active,
        kycRequired: kycRequired
    });
    emit LeaseCreated(id, tenant, landlord, assetId, termMonths, monthlyAXM, buyoutAXM,
kycRequired);
}

// tenant or anyone funding on behalf can pay exact multiple of monthlyAXM
function pay(uint256 leaseId, uint256 amount) external nonReentrant {
    Lease storage L = leases[leaseId];
```

```
        require(L.status == Status.Active, "not active");
        if (L.kycRequired && address(compliance) != address(0)) {
            require(compliance.allowPublicSale(L.tenant), "kyc fail");
        }
        require(amount > 0 && amount % L.monthlyAXM == 0, "bad amount");
        require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");

        uint16 monthsPaidNow = uint16(amount / L.monthlyAXM);
        L.paidMonths += monthsPaidNow;
        paidAmount[leaseId] += amount;

        // split and forward
        uint256 toLandlord = (amount * bpToLandlord) / 10_000;
        uint256 toRAF      = (amount * bpToRAF) / 10_000;
        uint256 toMaint    = (amount * bpToMaint) / 10_000;
        uint256 toTsy      = amount - toLandlord - toRAF - toMaint;

        require(axm.transfer(L.landlord, toLandlord), "landlord xfer");
        require(axm.transfer(rafSink, toRAF), "raf xfer");
        require(axm.transfer(maintSink, toMaint), "maint xfer");
        require(axm.transfer(treasurySink, toTsy), "tsy xfer");

        emit Payment(leaseId, msg.sender, amount, L.paidMonths);

        if (L.paidMonths >= L.termMonths) {
            L.status = Status.Completed;
            emit Completed(leaseId);
            // off-chain: convey title, update ATR if desired
        }
    }

    // optional early buyout at fixed AXM
    function buyout(uint256 leaseId) external nonReentrant {
        Lease storage L = leases[leaseId];
        require(L.status == Status.Active, "not active");
        require(L.buyoutAXM > 0, "no buyout");
        require(msg.sender == L.tenant, "only tenant");
        uint256 amt = L.buyoutAXM;
        if (L.kycRequired && address(compliance) != address(0)) {
            require(compliance.allowPublicSale(L.tenant), "kyc fail");
        }
        require(axm.transferFrom(msg.sender, address(this), amt), "transferFrom fail");

        // apply same splits
        uint256 toLandlord = (amt * bpToLandlord) / 10_000;
        uint256 toRAF      = (amt * bpToRAF) / 10_000;
        uint256 toMaint    = (amt * bpToMaint) / 10_000;
        uint256 toTsy      = amt - toLandlord - toRAF - toMaint;

        require(axm.transfer(L.landlord, toLandlord), "landlord xfer");
        require(axm.transfer(rafSink, toRAF), "raf xfer");
        require(axm.transfer(maintSink, toMaint), "maint xfer");
        require(axm.transfer(treasurySink, toTsy), "tsy xfer");

        paidAmount[leaseId] += amt;
        L.status = Status.Completed;
        emit Buyout(leaseId, msg.sender, amt);
        emit Completed(leaseId);
    }
```

```
// manager can mark default if business logic warrants (off-chain checks, missed payments, etc)
function markDefault(uint256 leaseId) external onlyRole(MANAGER_ROLE) {
    Lease storage L = leases[leaseId];
    require(L.status == Status.Active, "not active");
    L.status = Status.Defaulted;
    emit Defaulted(leaseId);
}

}
```

Files to create in scripts/

09_deploy_KeyGrowLeaseManager.js

```
const { saveAddress, getAddress } = require("./utils");

module.exports = async function() {
  const admin = process.env.ADMIN;
  const axm = getAddress("AXM");
  const comp = getAddress("ComplianceAdapter"); // may be undefined if you didn't deploy;
pass 0x0
  const atr  = getAddress("AssetTokenizationRegistry");

  const raf  = process.env.RAF_SINK;
  const maint = process.env.LIQUIDITY_SINK; // reuse as maint if you want, or create a
dedicated address
  const tsy  = process.env.TREASURY_SINK;

  if (!admin || !axm || !atr || !raf || !maint || !tsy) throw new Error("Missing env or prior addresses");

  const bpLandlord = 7000; // 70 percent
  const bpRAF = 1500;      // 15 percent
  const bpMaint = 500;     // 5 percent
  const bpTsy = 1000;      // 10 percent

  const Lm = await ethers.getContractFactory("KeyGrowLeaseManager");
  const lm = await Lm.deploy(
    admin,
    axm,
    comp || "0x0000000000000000000000000000000000000000",
    atr,
    raf,
    maint,
    tsy,
    bpLandlord, bpRAF, bpMaint, bpTsy
  );
  await lm.deployed();
  console.log("KeyGrowLeaseManager:", lm.address);
  saveAddress("KeyGrowLeaseManager", lm.address);
};
module.exports.tags = ["KeyGrow"];
module.exports.dependencies = ["AXM","Assets"];
```

10_deploy_PropertyShareToken_example.js

```
const { saveAddress, getAddress } = require("./utils");

module.exports = async function() {
```

```
  const admin = process.env.ADMIN;
  const atr = getAddress("AssetTokenizationRegistry");
  if (!admin || !atr) throw new Error("ADMIN or ATR missing");

  // Example PST for assetId=1. Replace values per property.
  const name = "KeyGrow 123 Main St Shares";
  const symbol = "KG123";
  const cap = ethers.parseUnits("1000000", 18);   // 1,000,000 shares
  const recipient = process.env.RAF_SINK;         // send all shares to RAF wallet to distribute
later
  const assetId = 1;
  const meta =
"0x000000000000000000000000000000000000000000000000000000000000abcd";

  const PST = await ethers.getContractFactory("PropertyShareToken");
  const pst = await PST.deploy(name, symbol, cap, admin, recipient, assetId, meta);
  await pst.deployed();
  console.log("PropertyShareToken example:", pst.address);
  saveAddress("PropertyShareToken_asset1", pst.address);
};
module.exports.tags = ["PST"];
module.exports.dependencies = ["Assets"];
```

Update package.json scripts Add:

"deploy:keygrow": "hardhat run scripts/09_deploy_KeyGrowLeaseManager.js --network peaq"

"deploy:pst": "hardhat run scripts/10_deploy_PropertyShareToken_example.js --network peaq"

Run sequence

1. npm run compile

2. npm run deploy:keygrow

3. npm run deploy:pst

Operational notes

For each property you tokenize, deploy a fresh PropertyShareToken with its own total supply and ATR assetId.

Use KeyGrowLeaseManager.createLease to onboard tenant/landlord, defining term, monthlyAXM, and optional buyoutAXM.

Payments flow in AXM and are auto-split to landlord, RAF, maintenance, and treasury per basis points.

If you want KYC gating, ensure ComplianceAdapter is deployed and KYC_SCHEMA is set; pass kycRequired=true for those leases.

When a lease completes or buyout occurs, emit events; execute title transfer off-chain and optionally update ATR status.

End of Replit AI prompt

Replit AI — do all of this exactly

Task Add two new contracts to the Axiom/PEAQ Hardhat repo, wire them to existing modules, and provide deploy scripts and role grants. Use Solidity ^0.8.20, OpenZeppelin only, constructor wiring (no proxies), and AXM as the payment token.

Files to create in contracts/

LeaseReceiptNFT.sol

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

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

// Non-transferable receipt per lease. Minted by KeyGrowLeaseManager when a lease is created (or upon first payment).
// Burnable by manager on cancellation. Optional baseURI for off-chain JSON metadata.
contract LeaseReceiptNFT is ERC721, AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");

    uint256 public nextId = 1;

    // tokenId => leaseId mapping for traceability
    mapping(uint256 => uint256) public leaseOfToken;

    // leaseId => tokenId (so each lease can have at most one receipt)
    mapping(uint256 => uint256) public tokenOfLease;

    // optional: baseURI like ipfs://CID/
    string public baseURI;

    event Minted(uint256 indexed tokenId, uint256 indexed leaseId, address indexed to);
    event Burned(uint256 indexed tokenId);
    event BaseURISet(string newBase);

    constructor(address admin, string memory name_, string memory symbol_) ERC721(name_, symbol_) {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
        _grantRole(MINTER_ROLE, admin);
    }

    function setBaseURI(string calldata uri) external onlyRole(MANAGER_ROLE) {
        baseURI = uri;
        emit BaseURISet(uri);
    }
```

```
    function mintReceipt(address to, uint256 leaseId) external onlyRole(MINTER_ROLE) returns
(uint256 id) {
        require(to != address(0) && leaseId != 0, "bad args");
        require(tokenOfLease[leaseId] == 0, "exists");
        id = nextId++;
        _safeMint(to, id);
        leaseOfToken[id] = leaseId;
        tokenOfLease[leaseId] = id;
        emit Minted(id, leaseId, to);
    }

    function burn(uint256 tokenId) external onlyRole(MANAGER_ROLE) {
        _burn(tokenId);
        uint256 leaseId = leaseOfToken[tokenId];
        if (leaseId != 0) tokenOfLease[leaseId] = 0;
        leaseOfToken[tokenId] = 0;
        emit Burned(tokenId);
    }

    // soulbound behavior: block transfers except mint (from=0) and burn (to=0)
    function _update(address from, address to, uint256 tokenId) internal override {
        if (from != address(0) && to != address(0)) revert("soulbound");
        super._update(from, to, tokenId);
    }

    function _baseURI() internal view override returns (string memory) { return baseURI; }
}
```

MaintenanceEscrowVault.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";

// Receives AXM maintenance split from KeyGrowLeaseManager and pays only approved
vendors
// against approved work orders. Everything is evented for audit trails.
contract MaintenanceEscrowVault is AccessControl, ReentrancyGuard {
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    bytes32 public constant APPROVER_ROLE = keccak256("APPROVER_ROLE");
    bytes32 public constant CASHIER_ROLE  = keccak256("CASHIER_ROLE");

    IERC20 public immutable axm;

    struct WorkOrder {
        uint256 leaseId;
        address vendor;
        uint256 amount;     // AXM wei
        bool approved;
        bool paid;
        bytes32 docCid;     // optional: IPFS CID hash for invoice/photos
    }

    uint256 public nextId = 1;
    mapping(uint256 => WorkOrder) public orders;
    mapping(address => bool) public vendorWhitelist;
```

```
event ToppedUp(address indexed from, uint256 amount);
event VendorWhitelisted(address indexed vendor, bool allowed);
event Created(uint256 id, uint256 leaseId, address vendor, uint256 amount, bytes32 docCid);
event Approved(uint256 id, address approver);
event Paid(uint256 id, address vendor, uint256 amount);

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

// KeyGrowLeaseManager or any ops wallet can push funds in AXM
function topUp(uint256 amount) external nonReentrant {
    require(amount > 0, "amount=0");
    require(axm.transferFrom(msg.sender, address(this), amount), "transferFrom fail");
    emit ToppedUp(msg.sender, amount);
}

function setVendor(address vendor, bool allowed) external onlyRole(MANAGER_ROLE) {
    require(vendor != address(0), "zero vendor");
    vendorWhitelist[vendor] = allowed;
    emit VendorWhitelisted(vendor, allowed);
}

function createWorkOrder(
    uint256 leaseId,
    address vendor,
    uint256 amount,
    bytes32 docCid
) external onlyRole(MANAGER_ROLE) returns (uint256 id) {
    require(leaseId != 0 && vendor != address(0) && amount > 0, "bad args");
    require(vendorWhitelist[vendor], "vendor not whitelisted");
    id = nextId++;
    orders[id] = WorkOrder({
        leaseId: leaseId,
        vendor: vendor,
        amount: amount,
        approved: false,
        paid: false,
        docCid: docCid
    });
    emit Created(id, leaseId, vendor, amount, docCid);
}

function approve(uint256 id) external onlyRole(APPROVER_ROLE) {
    WorkOrder storage w = orders[id];
    require(!w.paid && !w.approved, "already processed");
    w.approved = true;
    emit Approved(id, msg.sender);
}

function pay(uint256 id) external nonReentrant onlyRole(CASHIER_ROLE) {
    WorkOrder storage w = orders[id];
    require(w.approved && !w.paid, "not payable");
```

```
        w.paid = true;
        require(axm.transfer(w.vendor, w.amount), "transfer fail");
        emit Paid(id, w.vendor, w.amount);
    }
}
```

Wire LeaseReceiptNFT into KeyGrow Open contracts/KeyGrowLeaseManager.sol and add:

1. import and storage

```
import "./LeaseReceiptNFT.sol";

LeaseReceiptNFT public receipt;
```

2. constructor param Add address receiptNft to the constructor and assign:

```
constructor(
  address admin,
  address axmToken,
  address complianceAdapter,
  address atrAddress,
  address rafSink_,
  address maintSink_,
  address treasurySink_,
  uint16 bpLandlord,
  uint16 bpRAF,
  uint16 bpMaint,
  uint16 bpTreasury,
  address receiptNft
) {
  // existing checks...
  receipt = LeaseReceiptNFT(receiptNft);
}
```

3. mint a receipt when a lease is created Inside createLease, after storing the Lease, add:

```
uint256 tokenId = receipt.mintReceipt(tenant, id);
tokenId; // silence warnings; front-ends can index via event
```

4. grant MINTER_ROLE to LeaseManager during deployment (done in script below).

Files to create in scripts/

11_deploy_MaintenanceEscrowVault.js

```
const { saveAddress, getAddress } = require("./utils");

module.exports = async function() {
  const admin = process.env.ADMIN;
  const axm = getAddress("AXM");
  if (!admin || !axm) throw new Error("Missing ADMIN or AXM");
```

```
  const F = await ethers.getContractFactory("MaintenanceEscrowVault");
  const v = await F.deploy(admin, axm);
  await v.deployed();
  console.log("MaintenanceEscrowVault:", v.address);
  saveAddress("MaintenanceEscrowVault", v.address);
};
module.exports.tags = ["MaintVault"];
module.exports.dependencies = ["AXM"];
```

12_deploy_LeaseReceiptNFT.js

```
const { saveAddress } = require("./utils");

module.exports = async function() {
  const admin = process.env.ADMIN;
  if (!admin) throw new Error("Missing ADMIN");

  const name = "KeyGrow Lease Receipt";
  const symbol = "KGLR";
  const F = await ethers.getContractFactory("LeaseReceiptNFT");
  const nft = await F.deploy(admin, name, symbol);
  await nft.deployed();
  console.log("LeaseReceiptNFT:", nft.address);
  saveAddress("LeaseReceiptNFT", nft.address);
};
module.exports.tags = ["LeaseNFT"];
```

13_update_deploy_KeyGrowLeaseManager.js Create a new deploy that uses the receipt NFT and sets MaintenanceEscrow as the maintenance sink.

```
const { saveAddress, getAddress } = require("./utils");

module.exports = async function() {
  const admin  = process.env.ADMIN;
  const axm    = getAddress("AXM");
  const comp   = getAddress("ComplianceAdapter"); // may be undefined; pass zero
  const atr    = getAddress("AssetTokenizationRegistry");
  const raf    = process.env.RAF_SINK;
  const tsy    = process.env.TREASURY_SINK;
  const maintVault = getAddress("MaintenanceEscrowVault");
  const receiptNft = getAddress("LeaseReceiptNFT");

  if (!admin || !axm || !atr || !raf || !tsy || !maintVault || !receiptNft) {
    throw new Error("Missing env or prior addresses");
  }

  const bpLandlord = 7000; // 70 percent
  const bpRAF = 1500;      // 15 percent
  const bpMaint = 500;     // 5 percent
  const bpTsy = 1000;      // 10 percent

  const F = await ethers.getContractFactory("KeyGrowLeaseManager");
  const m = await F.deploy(
    admin,
    axm,
    comp || "0x0000000000000000000000000000000000000000",
    atr,
    raf,
```

```
    maintVault,   // maintenance split lands here
    tsy,
    bpLandlord, bpRAF, bpMaint, bpTsy,
    receiptNft
  );
  await m.deployed();
  console.log("KeyGrowLeaseManager:", m.address);
  saveAddress("KeyGrowLeaseManager", m.address);

  // grant NFT minting to the manager
  const nft = await ethers.getContractAt("LeaseReceiptNFT", receiptNft);
  const MINTER_ROLE = ethers.keccak256(ethers.toUtf8Bytes("MINTER_ROLE"));
  const tx = await nft.grantRole(MINTER_ROLE, m.address);
  await tx.wait();
  console.log("Granted MINTER_ROLE to KeyGrowLeaseManager on LeaseReceiptNFT");

};
module.exports.tags = ["KeyGrow"];
module.exports.dependencies = ["AXM","Assets","MaintVault","LeaseNFT"];
```

Package.json scripts: add or update

"deploy:maint": "hardhat run scripts/11_deploy_MaintenanceEscrowVault.js --network peaq"

"deploy:leasenft": "hardhat run scripts/12_deploy_LeaseReceiptNFT.js --network peaq"

"deploy:keygrow": "hardhat run scripts/13_update_deploy_KeyGrowLeaseManager.js --network peaq"

Run sequence (Peaq)

1. npm run compile

2. npm run deploy:maint

3. npm run deploy:leasenft

4. npm run deploy:keygrow

Post-deploy ops

Set vendor whitelist and start using the vault:

setVendor(vendorAddr, true)

topUp(amountAXM) by transferring AXM from treasury/key wallets into the vault

createWorkOrder(leaseId, vendor, amount, docCid)

approve(orderId)

pay(orderId)

Lease receipts are auto-minted to the tenant at lease creation. If a lease is cancelled, MANAGER_ROLE can burn the token.

Notes

No upgradeability, no hidden minting. Everything constructor-wired, using AXM as the only unit.

Maintenance split from KeyGrowLeaseManager now lands in MaintenanceEscrowVault by design, giving you clean, auditable flows for repairs.

Receipts are soulbound; they become a simple, verifiable proof a lease existed and completed.

End of Replit AI task

Excellent. The two new modules we just designed — LeaseReceiptNFT and MaintenanceEscrowVault — make KeyGrow's rent-to-own system flagship-ready.

To recap your current state before we push ahead with optimization or frontend work:

You now have a 22-contract architecture for Axiom Protocol:

The original 20 core Peaq-integrated contracts (AXM, Treasury, Governance, Real-Estate, DID, DePIN, etc.)

Plus the 2 KeyGrow-specific modules that elevate your real-world use case (LeaseReceiptNFT + MaintenanceEscrowVault).

These collectively tell a story of Axiom as a self-contained financial and infrastructure network — not just another token project, but a platform capable of running entire real-estate economies and on-chain lease systems with verifiable ownership, trust, and income routing.

Here's what this architecture says about Axiom in plain terms:

It's a sovereign financial OS for Web3 communities.

It connects real-world housing, logistics, and assets to blockchain rails.

It ensures compliance, governance, and transparency without middlemen.

And with KeyGrow, it introduces the world's first smart lease-to-ownership engine — one that automatically handles payments, maintenance, compliance, and proof of ownership.

From this point, you can:

1. Have Replit AI build the on-chain system (done).

2. Integrate the front-end flow — KeyGrow Portal for tenants and landlords.

3. Launch with a marketing story centered on "Proof-of-Ownership for Every Lease" — the bridge between tenants and tokenized real-estate.