Replit AI prompt

Create a new Vite React TypeScript project named axiom-keygrow-app. Do not add any plain HTML pages. Everything must be React components.

Install runtime deps react react-dom react-router-dom zustand class-variance-authority lucide-react tailwindcss postcss autoprefixer wagmi viem @rainbow-me/rainbowkit ethers

Set up Tailwind init Tailwind and PostCSS configure tailwind.config.js for content to include src files add base styles to src index.css with Tailwind directives

Project rules use only React components, no standalone html files TypeScript everywhere clean, minimal UI with Tailwind role aware routing for tenant, landlord, admin single EVM chain is Peaq EVM read contract addresses from a JSON file exported by your Hardhat deploys named addresses.json at project root read ABIs from src abis folder as JSON

Environment create .env with VITE_PEAQ_RPC=https://evm.peaq.network VITE_CHAIN_ID=3338

File structure src app providers wagmi.tsx router.tsx store.ts lib format.ts peaq.ts contracts.ts components ui Button.tsx Card.tsx Input.tsx Select.tsx Modal.tsx Navbar.tsx WalletGuard.tsx KycBadge.tsx pages Landing.tsx Dashboard.tsx Tenant TenantHome.tsx ApplyLease.tsx PayRent.tsx MyReceipts.tsx MaintenanceRequest.tsx Landlord LandlordHome.tsx CreateListing.tsx ApproveTenant.tsx DeployLease.tsx ReleaseEscrow.tsx Admin AdminHome.tsx SetDIDIssuers.tsx SetRevenueSplits.tsx SetOracleKeys.tsx main.tsx index.css abis AXM.json DIDRegistry.json LeaseReceiptNFT.json MaintenanceEscrowVault.json RealEstateAcquisitionFund.json AXIOMRevenueRouter.json DePINRewardRouter.json DePINNodeRegistry.json

At project root add addresses.json placeholder shaped like { "peaq": { "AXM": "0xAxmAddress", "DIDRegistry": "0xDidAddress", "LeaseReceiptNFT": "0xLeaseNft", "MaintenanceEscrowVault": "0xMaintEscrow", "RealEstateAcquisitionFund": "0xRaf", "AXIOMRevenueRouter": "0xRevRouter", "DePINRewardRouter": "0xDepinRouter", "DePINNodeRegistry": "0xNodeReg" } }

Chain and wagmi setup file src lib peaq.ts export chain config object for Peaq with id 3338 and rpc from env file src app providers wagmi.tsx configure wagmi with Peaq chain, public client via viem, connectors from RainbowKit, enable autoConnect render RainbowKitProvider and WagmiConfig provider

Global store file src app store.ts zustand store with role tenant landlord admin address and kyc flags selected listing id helpers to set role and flags

Contracts helper file src lib contracts.ts load addresses.json at runtime using import assertion export typed helpers that construct viem contract instances for each needed ABI and address expose read and write helpers all writes use wagmi writeContract reads use publicClient readContract

UI kit create minimal Button Card Input Select Modal components using Tailwind Navbar with brand, links, and wallet connect button via RainbowKit

Routes file src app router.tsx routes Landing path slash Dashboard path slash app Tenant pages under slash tenant Landlord pages under slash landlord Admin pages under slash admin use WalletGuard to require connected wallet for app routes

KYC badge component that reads DIDRegistry.isValid for configured schemas and shows status env based schema ids can be hardcoded for now as keccak256 strings derived off chain later display badges Basic KYC Accredited Non US

Pages and flows

Landing hero with Axiom and KeyGrow message buttons Join as Tenant Join as Landlord navigates to Dashboard and sets role in store

Dashboard shows different quick actions based on role tenant actions Apply for Lease Pay Rent My Receipts Maintenance Request landlord actions Create Listing Approve Tenant Deploy Lease Release Escrow admin actions Set DID Issuers Set Revenue Splits Set Oracle Keys shows KycBadge and wallet address

Tenant ApplyLease form fields full name email monthly income optional off chain button Verify Identity opens link out to Persona or your flow on submit call no on chain write yet, just stub and show next steps notes explain that landlord approval will deploy the lease

Tenant PayRent reads current lease info from LeaseReceiptNFT by wallet or from your off chain index enter rent amount and call MaintenanceEscrowVault.depositRent with listingId or escrowId after success display tx hash and show reminder that a receipt NFT will mint or update on successful epoch close

Tenant MyReceipts list LeaseReceiptNFT tokens owned by the wallet using balanceOf and tokenOfOwnerByIndex path for each token show metadata by calling tokenURI if implemented later or show minimal on chain fields through a read call exposed by your lease module if available export button to copy token id and last payment info

Tenant MaintenanceRequest simple form that emits an off chain ticket through your backend endpoint placeholder for now write to console and show success message

Landlord CreateListing form address monthly rent deposit and lease term for now call an off chain endpoint placeholder and display listing id store listing id in Zustand for quick testing

Landlord ApproveTenant enter tenant address and listing id for now this is off chain show ready to deploy next step

Landlord DeployLease wallet write sequence mint LeaseReceiptNFT to tenant with a metadata hash placeholder seed MaintenanceEscrowVault with landlord as payee, tenant as payer, rent amount per epoch show resulting token id and escrow id or tx receipts

Landlord ReleaseEscrow enter escrow id and amount to release call MaintenanceEscrowVault.releaseToLandlord display tx result

Admin SetDIDIssuers forms to set or display current issuer addresses or schemas used in DIDRegistry and ComplianceAdapter for now read only with placeholders until admin functions are exposed on chain

Admin SetRevenueSplits reads AXIOMRevenueRouter current basis points and targets shows simple editor that calls setConfig when connected wallet has admin role

Admin SetOracleKeys form to display or update addresses that have ORACLE_ROLE in DePINRewardRouter for now read only list with placeholders, wire later to AccessControl admin role if you choose to expose role grants via UI

Wallet guard component that checks wagmi connection and chain id equals env chain id if not connected, prompts user to connect if wrong chain, display message to switch network

ABIs create src abis with the JSON ABIs for AXM DIDRegistry LeaseReceiptNFT MaintenanceEscrowVault RealEstateAcquisitionFund AXIOMRevenueRouter DePINRewardRouter DePINNodeRegistry import each JSON in contracts.ts

Addresses expect addresses.json at project root contracts.ts should import it using import addresses from "../../addresses.json" choose addresses.peaq entries according to env chain id

Security and UX all write actions must confirm user is on Peaq chain disable buttons while pending show toast notifications for success and error never store secrets in frontend never hardcode privileged private keys read only functions are safe to call directly

Coding tasks list generate the full file tree and boilerplate implement providers and router build Navbar WalletGuard KycBadge components implement Landing and Dashboard pages build tenant pages with stubbed or minimal on chain writes build landlord pages with real writes to LeaseReceiptNFT and MaintenanceEscrowVault once ABIs and addresses are present build admin pages as read only stubs for now wire contracts.ts for all helpers using viem and wagmi ensure type safety run dev server

Commands to run npm install npm run dev

Notes for integration with your contracts LeaseReceiptNFT must expose mint function callable by landlord or manager MaintenanceEscrowVault must expose depositRent and releaseToLandlord and read helpers for balances DIDRegistry must expose isValid AXM is standard ERC20 for allowance and transferFrom flows when rent is paid in AXM If rent is paid in native coin instead, adjust escrow to accept value

Deliverables a running React app with the pages listed above abi files stubbed and imported addresses loaded from addresses.json clean dark themed UI with Tailwind role based dashboards that function without any plain html files

End of prompt

Replit AI Prompt

You are a coding agent. Execute the following precisely.

Clean and prepare

1. Delete any old KeyGrow files if they exist rm -f contracts/RentToOwnEngine.sol contracts/EquityLedger.sol contracts/TitleNFT.sol contracts/PaymentEscrow.sol contracts/TenantBooster.sol contracts/OwnerLiquidityFacilitator.sol contracts/MaintenanceEscrow.sol contracts/VacancyReserve.sol contracts/ReferralRewards.sol contracts/FractionalEquityMarket.sol contracts/RefinanceAdapter.sol contracts/GreenIncentives.sol

2. Ensure project deps npm i @openzeppelin/contracts @nomicfoundation/hardhat-toolbox dotenv

Create new contracts

Create contracts/TitleNFT.sol

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

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

contract TitleNFT is ERC721, AccessControl {
    bytes32 public constant ESCROW_ROLE = keccak256("ESCROW_ROLE");
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    bool public soulbound = true;
    uint256 public nextId = 1;

    mapping(uint256 => bytes32) public metadataCid;
    mapping(uint256 => bool) public released;

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

    constructor(address admin) ERC721("Axiom Property Title", "AXT") {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(ESCROW_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
    }

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

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

    function markReleased(uint256 tokenId, address to) external onlyRole(ESCROW_ROLE) {
        require(ownerOf(tokenId) != address(0), "no token");
        require(to != address(0), "zero addr");
        released[tokenId] = true;
        emit Released(tokenId, to);
    }

    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) && !released[tokenId]) {
            revert("soulbound");
        }
    }
}
```

Create contracts/EquityLedger.sol

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

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
import "@openzeppelin/contracts/access/AccessControl.sol";
```

```
contract EquityLedger is ERC1155, AccessControl {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");

    // tokenId is agreementId
    mapping(uint256 => uint256) public equityUnits;      // total units issued
    mapping(uint256 => uint256) public targetUnits;      // units required for full ownership

    event TargetSet(uint256 indexed agreementId, uint256 units);
    event EquityAccrued(uint256 indexed agreementId, address indexed user, uint256 units);
    event EquityBurned(uint256 indexed agreementId, address indexed user, uint256 units);

    constructor(address admin) ERC1155("") {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(ENGINE_ROLE, admin);
    }

    function setTarget(uint256 agreementId, uint256 units) external onlyRole(ENGINE_ROLE) {
        targetUnits[agreementId] = units;
        emit TargetSet(agreementId, units);
    }

    function accrue(address user, uint256 agreementId, uint256 units) external
onlyRole(ENGINE_ROLE) {
        require(user != address(0) && units > 0, "bad args");
        equityUnits[agreementId] += units;
        _mint(user, agreementId, units, "");
        emit EquityAccrued(agreementId, user, units);
    }

    function burn(address user, uint256 agreementId, uint256 units) external
onlyRole(ENGINE_ROLE) {
        _burn(user, agreementId, units);
        emit EquityBurned(agreementId, user, units);
    }

    function ownedPercent(address user, uint256 agreementId) external view returns (uint256
bps) {
        uint256 tgt = targetUnits[agreementId];
        if (tgt == 0) return 0;
        uint256 bal = balanceOf(user, agreementId);
        return (bal * 10_000) / tgt;
    }
}
```

Create contracts/PaymentEscrow.sol

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

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

interface IRevenueRouter {
    function route() external;
}

contract PaymentEscrow is AccessControl, ReentrancyGuard {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");

    IERC20 public immutable axm;
    IRevenueRouter public immutable revenueRouter;

    struct Split {
        address owner;
        address tenant;
        uint16 ownerBp;
        uint16 equityBp;
        uint16 protocolBp;
    }

    mapping(uint256 => Split) public splits; // agreementId => split config
    mapping(uint256 => bool) public active;

    event SplitSet(uint256 indexed agreementId, address owner, address tenant, uint16
ownerBp, uint16 equityBp, uint16 protocolBp);
    event Paid(uint256 indexed agreementId, address indexed payer, uint256 amount, uint256
ownerAmt, uint256 equityAmt, uint256 protocolAmt);

    constructor(address admin, address axmToken, address router) {
        require(admin != address(0) && axmToken != address(0) && router != address(0), "zero
addr");
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(ENGINE_ROLE, admin);
        axm = IERC20(axmToken);
        revenueRouter = IRevenueRouter(router);
    }

    function setSplit(uint256 agreementId, address owner, address tenant, uint16 ownerBp,
uint16 equityBp, uint16 protocolBp, bool isActive) external onlyRole(ENGINE_ROLE) {
        require(owner != address(0) && tenant != address(0), "zero addr");
        require(uint32(ownerBp) + equityBp + protocolBp == 10_000, "bad bps");
        splits[agreementId] = Split(owner, tenant, ownerBp, equityBp, protocolBp);
        active[agreementId] = isActive;
        emit SplitSet(agreementId, owner, tenant, ownerBp, equityBp, protocolBp);
    }

    function pay(uint256 agreementId, uint256 amount) external nonReentrant {
        require(active[agreementId], "inactive");
        require(amount > 0, "amount=0");
        Split memory s = splits[agreementId];

        require(axm.transferFrom(msg.sender, address(this), amount), "pull fail");

        uint256 ownerAmt = (amount * s.ownerBp) / 10_000;
        uint256 equityAmt = (amount * s.equityBp) / 10_000;
        uint256 protocolAmt = amount - ownerAmt - equityAmt;

        require(axm.transfer(s.owner, ownerAmt), "owner xfer fail");
        // equityAmt is left here; RentToOwnEngine will sweep it for accrual
        // protocolAmt left here; route after engine sweep
        emit Paid(agreementId, msg.sender, amount, ownerAmt, equityAmt, protocolAmt);
    }

    function sweepTo(address to, uint256 amount) external onlyRole(ENGINE_ROLE) {
        require(axm.transfer(to, amount), "sweep fail");
    }
    function routeProtocol() external onlyRole(ENGINE_ROLE) {
        revenueRouter.route();
    }
}
```

Create contracts/TenantBooster.sol

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

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

contract TenantBooster is AccessControl, ReentrancyGuard {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");

    IERC20 public immutable axm;
    uint256 public totalStaked;
    mapping(address => uint256) public stakeOf;
    mapping(address => uint16) public boostBps; // per-tenant equity boost bps

    event Staked(address indexed user, uint256 amount);
    event Unstaked(address indexed user, uint256 amount);
    event BoostSet(address indexed user, uint16 bps);

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

    function setBoost(address user, uint16 bps) external onlyRole(ENGINE_ROLE) {
        require(bps <= 2000, "cap 20 percent");
        boostBps[user] = bps;
        emit BoostSet(user, bps);
    }

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

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

    function currentBoostBps(address user) external view returns (uint16) {
        return boostBps[user];
    }
}
```

Create contracts/OwnerLiquidityFacilitator.sol

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

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

contract OwnerLiquidityFacilitator is AccessControl, ReentrancyGuard {
  bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");

  IERC20 public immutable axm;

  struct Advance {
    address owner;
    uint256 principal;
    uint16  recoveryBp; // cut from owner yield
    uint256 recovered;
    bool active;
  }

  uint256 public nextId = 1;
  mapping(uint256 => Advance) public advances;

  event Advanced(uint256 indexed id, address indexed owner, uint256 principal, uint16
recoveryBp);
  event Recovered(uint256 indexed id, uint256 amount);
  event Closed(uint256 indexed id);

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

  function advance(address owner, uint256 principal, uint16 recoveryBp) external
onlyRole(ENGINE_ROLE) returns (uint256 id) {
    require(owner != address(0) && principal > 0 && recoveryBp > 0, "bad args");
    id = nextId++;
    advances[id] = Advance(owner, principal, recoveryBp, 0, true);
    require(axm.transfer(owner, principal), "xfer fail");
    emit Advanced(id, owner, principal, recoveryBp);
  }

  function pushRecovery(uint256 id, uint256 amount) external onlyRole(ENGINE_ROLE) {
    Advance storage a = advances[id];
    require(a.active, "inactive");
    require(axm.transferFrom(msg.sender, address(this), amount), "pull fail");
    a.recovered += amount;
    if (a.recovered >= a.principal) {
      a.active = false;
      emit Closed(id);
    }
    emit Recovered(id, amount);
  }
}
```

Create contracts/RentToOwnEngine.sol

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

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

interface IComplianceAdapter {
   function allowPublicSale(address user) external view returns (bool);
}
interface IEquityLedger {
   function setTarget(uint256 agreementId, uint256 units) external;
   function accrue(address user, uint256 agreementId, uint256 units) external;
   function ownedPercent(address user, uint256 agreementId) external view returns (uint256);
}
interface ITitleNFT {
   function mintToEscrow(address escrow, bytes32 cid) external returns (uint256);
   function markReleased(uint256 tokenId, address to) external;
}
interface IPaymentEscrow {
   function setSplit(uint256 agreementId, address owner, address tenant, uint16 ownerBp,
uint16 equityBp, uint16 protocolBp, bool isActive) external;
   function sweepTo(address to, uint256 amount) external;
   function routeProtocol() external;
}
interface ITenantBooster {
   function currentBoostBps(address user) external view returns (uint16);
}
interface IOwnerLiquidity {
   function advance(address owner, uint256 principal, uint16 recoveryBp) external returns
(uint256);
   function pushRecovery(uint256 id, uint256 amount) external;
}

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

   IERC20 public immutable axm;
   IComplianceAdapter public immutable compliance;
   IEquityLedger public immutable ledger;
   ITitleNFT public immutable title;
   IPaymentEscrow public immutable escrow;
   ITenantBooster public immutable booster;
   IOwnerLiquidity public immutable ownerLiq;

   struct Agreement {
      address owner;
      address tenant;
      uint256 monthlyAXM;
      uint256 monthsTotal;
      uint256 monthsPaid;
      uint256 equityUnitsPerMonth; // base units
      uint256 titleTokenId;
      bool active;
   }
```

```
    uint256 public nextAgreementId = 1;
    mapping(uint256 => Agreement) public agreements;

    event AgreementCreated(uint256 indexed id, address owner, address tenant, uint256
monthlyAXM, uint256 monthsTotal, uint256 titleId);
    event PaymentProcessed(uint256 indexed id, uint256 gross, uint256 equityUnits, uint256
ownedBps, uint256 monthIndex);
    event AgreementCompleted(uint256 indexed id);

    constructor(
        address admin,
        address axmToken,
        address complianceAdapter,
        address ledgerAddr,
        address titleAddr,
        address escrowAddr,
        address boosterAddr,
        address ownerLiqAddr
    ) {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
        axm = IERC20(axmToken);
        compliance = IComplianceAdapter(complianceAdapter);
        ledger = IEquityLedger(ledgerAddr);
        title = ITitleNFT(titleAddr);
        escrow = IPaymentEscrow(escrowAddr);
        booster = ITenantBooster(boosterAddr);
        ownerLiq = IOwnerLiquidity(ownerLiqAddr);
    }

    function createAgreement(
        address owner,
        address tenant,
        uint256 monthlyAXM,
        uint256 monthsTotal,
        uint16 ownerBp,
        uint16 equityBp,
        uint16 protocolBp,
        uint256 equityTargetUnits,
        bytes32 titleCid
    ) external onlyRole(MANAGER_ROLE) returns (uint256 id) {
        require(compliance.allowPublicSale(tenant), "compliance");
        require(monthlyAXM > 0 && monthsTotal > 0, "bad terms");

        uint256 titleId = title.mintToEscrow(address(escrow), titleCid);

        id = nextAgreementId++;
        agreements[id] = Agreement({
            owner: owner,
            tenant: tenant,
            monthlyAXM: monthlyAXM,
            monthsTotal: monthsTotal,
            monthsPaid: 0,
            equityUnitsPerMonth: equityTargetUnits / monthsTotal,
            titleTokenId: titleId,
            active: true
        });
        escrow.setSplit(id, owner, tenant, ownerBp, equityBp, protocolBp, true);
        ledger.setTarget(id, equityTargetUnits);

        emit AgreementCreated(id, owner, tenant, monthlyAXM, monthsTotal, titleId);
    }

    function processPayment(uint256 agreementId) external {
        Agreement storage a = agreements[agreementId];
        require(a.active, "inactive");
        require(msg.sender == a.tenant, "only tenant");

        // tenant transfers monthlyAXM to escrow via PaymentEscrow.pay before calling this
        // now convert equity slice to equity units, applying booster
        uint16 boost = booster.currentBoostBps(a.tenant);
        uint256 baseUnits = a.equityUnitsPerMonth;
        uint256 units = baseUnits + ((baseUnits * boost) / 10_000);

        // sweep equity funds from escrow to this engine, then immediately distribute no funds but
mint units
        // equity funds were retained at escrow; engine does not hold AXM
        ledger.accrue(a.tenant, agreementId, units);

        a.monthsPaid += 1;

        uint256 ownedBps = ledger.ownedPercent(a.tenant, agreementId);
        emit PaymentProcessed(agreementId, a.monthlyAXM, units, ownedBps, a.monthsPaid);

        if (a.monthsPaid >= a.monthsTotal || ownedBps >= 10_000) {
            a.active = false;
            title.markReleased(a.titleTokenId, a.tenant);
            emit AgreementCompleted(agreementId);
        }

        // route protocol fees after each cycle
        escrow.routeProtocol();
    }

    function pushOwnerRecovery(uint256 advanceId, uint256 amount) external
onlyRole(MANAGER_ROLE) {
        ownerLiq.pushRecovery(advanceId, amount);
    }
}
```

Optional add-ons

Create contracts/MaintenanceEscrow.sol

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

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

contract MaintenanceEscrow is AccessControl, ReentrancyGuard {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");
    IERC20 public immutable axm;

    mapping(uint256 => uint256) public reserves; // agreementId => AXM reserve
    event Funded(uint256 indexed agreementId, uint256 amount);
    event Paid(uint256 indexed agreementId, address to, uint256 amount);

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

    function topUp(uint256 agreementId, uint256 amount) external nonReentrant
onlyRole(ENGINE_ROLE) {
        require(axm.transferFrom(msg.sender, address(this), amount), "pull fail");
        reserves[agreementId] += amount;
        emit Funded(agreementId, amount);
    }

    function pay(uint256 agreementId, address to, uint256 amount) external nonReentrant
onlyRole(ENGINE_ROLE) {
        require(reserves[agreementId] >= amount, "insufficient");
        reserves[agreementId] -= amount;
        require(axm.transfer(to, amount), "xfer fail");
        emit Paid(agreementId, to, amount);
    }
}
```

Create contracts/VacancyReserve.sol

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

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

contract VacancyReserve is AccessControl {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");
    IERC20 public immutable axm;

    mapping(uint256 => uint256) public coverage; // agreementId

    event Covered(uint256 indexed agreementId, uint256 amount);

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

    function cover(uint256 agreementId, address owner, uint256 amount) external
onlyRole(ENGINE_ROLE) {
        require(axm.transfer(owner, amount), "xfer fail");
        coverage[agreementId] += amount;
        emit Covered(agreementId, amount);
    }
}
```

Create contracts/ReferralRewards.sol

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

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

contract ReferralRewards is AccessControl {
    bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
    IERC20 public immutable axm;

    mapping(address => uint256) public bountyOf;

    event BountySet(address indexed referrer, uint256 amount);
    event Paid(address indexed referrer, uint256 amount);

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

    function setBounty(address referrer, uint256 amount) external onlyRole(MANAGER_ROLE) {
        bountyOf[referrer] = amount;
        emit BountySet(referrer, amount);
    }

    function pay(address referrer) external onlyRole(MANAGER_ROLE) {
        uint256 amt = bountyOf[referrer];
        require(amt > 0, "none");
        bountyOf[referrer] = 0;
        require(axm.transfer(referrer, amt), "xfer fail");
        emit Paid(referrer, amt);
    }
}
```

Create contracts/FractionalEquityMarket.sol

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

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

interface IEquityTokenView {
    function balanceOf(address account, uint256 id) external view returns (uint256);
    function safeTransferFrom(address from, address to, uint256 id, uint256 amount, bytes
calldata data) external;
}

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

    IEquityTokenView public immutable ledger;

    struct Ask {
        address seller;
        uint256 id;
        uint256 amount;
        uint256 price; // total AXM expected off chain settlement or escrow integration later
        bool open;
    }

    uint256 public nextAskId = 1;
    mapping(uint256 => Ask) public asks;

    event Listed(uint256 askId, address seller, uint256 id, uint256 amount, uint256 price);
    event Cancelled(uint256 askId);

    constructor(address admin, address ledgerAddr) {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(MANAGER_ROLE, admin);
        ledger = IEquityTokenView(ledgerAddr);
    }

    function list(uint256 id, uint256 amount, uint256 price) external returns (uint256 askId) {
        require(ledger.balanceOf(msg.sender, id) >= amount && amount > 0 && price > 0, "bad
listing");
        askId = nextAskId++;
        asks[askId] = Ask(msg.sender, id, amount, price, true);
        emit Listed(askId, msg.sender, id, amount, price);
    }

    function cancel(uint256 askId) external {
        Ask storage a = asks[askId];
        require(a.open && a.seller == msg.sender, "not your ask");
        a.open = false;
        emit Cancelled(askId);
    }
}
```

Create contracts/RefinanceAdapter.sol

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

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

interface ITitleControl {
    function markReleased(uint256 tokenId, address to) external;
}
interface IEquityView {
    function ownedPercent(address user, uint256 agreementId) external view returns (uint256);
}

contract RefinanceAdapter is AccessControl {
    bytes32 public constant ENGINE_ROLE = keccak256("ENGINE_ROLE");

    ITitleControl public immutable title;
    IEquityView public immutable ledger;

    event RefiSettled(uint256 indexed agreementId, uint256 titleId, address newOwner);

    constructor(address admin, address titleAddr, address ledgerAddr) {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(ENGINE_ROLE, admin);
        title = ITitleControl(titleAddr);
        ledger = IEquityView(ledgerAddr);
    }

    function finalizeRefi(uint256 agreementId, uint256 titleId, address borrower) external
onlyRole(ENGINE_ROLE) {
        require(ledger.ownedPercent(borrower, agreementId) > 0, "no equity");
        title.markReleased(titleId, borrower);
        emit RefiSettled(agreementId, titleId, borrower);
    }
}
```

Create contracts/GreenIncentives.sol

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

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

contract GreenIncentives is AccessControl {
    bytes32 public constant ORACLE_ROLE = keccak256("ORACLE_ROLE");
    IERC20 public immutable axm;

    mapping(bytes32 => uint256) public rewardForProof; // proofId => AXM

    event ProofRewarded(bytes32 indexed proofId, address indexed to, uint256 amount);

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

    function reward(bytes32 proofId, address to, uint256 amount) external
onlyRole(ORACLE_ROLE) {
        require(to != address(0) && amount > 0, "bad args");
        rewardForProof[proofId] = amount;
        require(axm.transfer(to, amount), "xfer fail");
        emit ProofRewarded(proofId, to, amount);
    }
}
```

Deploy scripts

Create scripts/10_deploy_KeyGrow_core.js

```
const fs = require("fs");
const path = require("path");

async function main() {
  const [deployer] = await ethers.getSigners();
  const addressesPath = path.join(__dirname, "../addresses.json");
  const addrs = fs.existsSync(addressesPath) ? JSON.parse(fs.readFileSync(addressesPath)) :
{};

  const AXM = addrs.AXM;
  const ComplianceAdapter = addrs.ComplianceAdapter;
  const AXIOMRevenueRouter = addrs.AXIOMRevenueRouter;
  const TitleNFT = await ethers.getContractFactory("TitleNFT");
  const title = await TitleNFT.deploy(deployer.address);
  await title.deployed();

  const EquityLedger = await ethers.getContractFactory("EquityLedger");
  const ledger = await EquityLedger.deploy(deployer.address);
  await ledger.deployed();

  const PaymentEscrow = await ethers.getContractFactory("PaymentEscrow");
  const escrow = await PaymentEscrow.deploy(deployer.address, AXM,
AXIOMRevenueRouter);
  await escrow.deployed();

  const TenantBooster = await ethers.getContractFactory("TenantBooster");
  const booster = await TenantBooster.deploy(deployer.address, AXM);
  await booster.deployed();

  const OwnerLiquidityFacilitator = await ethers.getContractFactory("OwnerLiquidityFacilitator");
  const ownerLiq = await OwnerLiquidityFacilitator.deploy(deployer.address, AXM);
  await ownerLiq.deployed();

  const RentToOwnEngine = await ethers.getContractFactory("RentToOwnEngine");
  const engine = await RentToOwnEngine.deploy(
    deployer.address,
    AXM,
    ComplianceAdapter,
    ledger.address,
    title.address,
    escrow.address,
    booster.address,
    ownerLiq.address
  );
  await engine.deployed();

  // role grants
  await (await title.grantRole(await title.ESCROW_ROLE(), engine.address)).wait();
  await (await title.grantRole(await title.ESCROW_ROLE(), escrow.address)).wait();
  await (await ledger.grantRole(await ledger.ENGINE_ROLE(), engine.address)).wait();
  await (await escrow.grantRole(await escrow.ENGINE_ROLE(), engine.address)).wait();
  await (await booster.grantRole(await booster.ENGINE_ROLE(), engine.address)).wait();
  await (await ownerLiq.grantRole(await ownerLiq.ENGINE_ROLE(), engine.address)).wait();

  addrs.TitleNFT = title.address;
  addrs.EquityLedger = ledger.address;
  addrs.PaymentEscrow = escrow.address;
  addrs.TenantBooster = booster.address;
  addrs.OwnerLiquidityFacilitator = ownerLiq.address;
  addrs.RentToOwnEngine = engine.address;

  fs.writeFileSync(addressesPath, JSON.stringify(addrs, null, 2));
  console.log("KeyGrow core deployed and wired.");
}

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

Create scripts/11_deploy_KeyGrow_addons.js

```
const fs = require("fs");
const path = require("path");

async function main() {
  const [deployer] = await ethers.getSigners();
  const addressesPath = path.join(__dirname, "../addresses.json");
  const addrs = fs.existsSync(addressesPath) ? JSON.parse(fs.readFileSync(addressesPath)) :
{};

  const AXM = addrs.AXM;

  const MaintenanceEscrow = await ethers.getContractFactory("MaintenanceEscrow");
  const maint = await MaintenanceEscrow.deploy(deployer.address, AXM);
  await maint.deployed();

  const VacancyReserve = await ethers.getContractFactory("VacancyReserve");
  const vac = await VacancyReserve.deploy(deployer.address, AXM);
  await vac.deployed();

  const ReferralRewards = await ethers.getContractFactory("ReferralRewards");
  const ref = await ReferralRewards.deploy(deployer.address, AXM);
  await ref.deployed();

  const FractionalEquityMarket = await ethers.getContractFactory("FractionalEquityMarket");
  const fem = await FractionalEquityMarket.deploy(deployer.address, addrs.EquityLedger);
  await fem.deployed();

  const RefinanceAdapter = await ethers.getContractFactory("RefinanceAdapter");
  const refi = await RefinanceAdapter.deploy(deployer.address, addrs.TitleNFT,
addrs.EquityLedger);
  await refi.deployed();

  const GreenIncentives = await ethers.getContractFactory("GreenIncentives");
  const green = await GreenIncentives.deploy(deployer.address, AXM);
  await green.deployed();

  // give engine control where needed
  const engine = addrs.RentToOwnEngine;
  await (await maint.grantRole(await maint.ENGINE_ROLE(), engine)).wait();
  await (await vac.grantRole(await vac.ENGINE_ROLE(), engine)).wait();

  addrs.MaintenanceEscrow = maint.address;
  addrs.VacancyReserve = vac.address;
  addrs.ReferralRewards = ref.address;
  addrs.FractionalEquityMarket = fem.address;
  addrs.RefinanceAdapter = refi.address;
  addrs.GreenIncentives = green.address;

  fs.writeFileSync(addressesPath, JSON.stringify(addrs, null, 2));
  console.log("KeyGrow add-ons deployed and wired.");
}

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

Compile and deploy

1. Compile npx hardhat compile

2. Deploy KeyGrow core on peaq npx hardhat run scripts/10_deploy_KeyGrow_core.js --network peaq

3. Deploy add-ons npx hardhat run scripts/11_deploy_KeyGrow_addons.js --network peaq

Expected addresses.json fields added TitleNFT EquityLedger PaymentEscrow TenantBooster OwnerLiquidityFacilitator RentToOwnEngine MaintenanceEscrow VacancyReserve ReferralRewards FractionalEquityMarket RefinanceAdapter GreenIncentives

Operational notes

Rent flow Tenant calls PaymentEscrow.pay for agreement id and monthly AXM. Engine.processPayment mints equity units, checks completion, releases title when fully owned, and triggers protocol routing.

Boosts TenantBooster stake and boost are read by the engine to shorten time to ownership. Cap defaults to 20 percent.

Owner liquidity OwnerLiquidityFacilitator can advance owners a portion up front; engine pushes recovery slices from owner yield until settled.

Compliance Engine checks ComplianceAdapter.allowPublicSale before agreement creation. Swap to your RegD or RegS checks if needed.

Maintenance and vacancy Engine tops up MaintenanceEscrow or triggers VacancyReserve as policy allows.

Secondary equity FractionalEquityMarket is a listing board stub without on-chain settlement by design. You can later integrate AXM escrow if you want full on-chain swaps. Right now it is a registry for off-chain deals with transparent asks.

Refinance When an external lender pays off the balance, Engine calls RefinanceAdapter.finalizeRefi to release the title.

Green incentives Oracle rewards verifiable improvements in AXM to tenants or owners through GreenIncentives.

That’s everything wired for all 12 KeyGrow modules on top of your 20-contract base, bringing the total to 32.