ChatGPT

Innovative Blockchain Project Ideas for a Sui Hackathon

Below are **15 creative project ideas** across DeFi, AI, and CityTech/Education tracks. Each idea centers on Move smart contracts (core on-chain logic) and includes an overview and key implementation steps. The concepts emphasize **Sui's object model, ownership types (address-owned, shared, etc.)**, **capability-based access control**, and **rich data structures (vectors, tables)** in Move.

1. Decentralized Parametric Insurance (DeFi/Civic Tech)

Description: A blockchain-based insurance pool that pays out automatically when real-world conditions are met. For example, farmers could insure crops against drought or flood, with payouts triggered by oracle data (e.g. weather metrics). Unlike traditional insurance, claims are **automated via smart contracts** – if the predefined trigger (like rainfall below a threshold) occurs, the contract releases compensation instantly [1]. This removes paperwork and builds trust through transparency. The project fits DeFi (financial insurance) and Civic Tech (helping communities with risk management).

Implementation Steps:
1. **Smart Contract Modeling:** Create a Move module for an **InsurancePool** (shared object) holding pooled funds. Define a struct for **Policy** (could be an NFT-like object) with fields like coverage amount, premium, insured address, and trigger conditions. Policies might be *address-owned* by the buyer, while the pool is a *shared* object accessible to all participants.
2. **Oracle Integration:** Use an oracle (e.g. Pyth or custom feed) to provide external data (weather or price). The Move contract can include a function `trigger_payout(policy_id, data)` that anyone can call when a valid oracle data object is passed. If conditions are met, the contract transfers payout to the policy holder. (In tests, you can simulate the oracle input).
3. **Access Control:** Introduce an **AdminCap** for pool managers. Only the admin (insurer) can create new policies or withdraw excess funds, by requiring the caller to present the `AdminCap` object in those entry functions. This **capability object** gating ensures unauthorized users cannot mint policies arbitrarily [2] [3].
4. **Data Structures & Logic:** Maintain a vector or table of active policies within the InsurancePool. For example, use a vector to store policy IDs or a Table mapping policy ID to details for quick lookup on claim. Use Move's *Table* if needing to map policy IDs (keys) to Policy structs (values).
5. **Testing Scenarios:** Write Move unit tests covering scenarios: policy purchase (funds go into pool), trigger condition met (payout transfers to user and policy marked paid), trigger not met (no payout), only admin can create or withdraw (others should fail). Edge cases like invalid oracle data or double payout attempts should be tested.
6. **Frontend:** Build a React/TypeScript dApp where users connect their Sui wallet, view available insurance policies, purchase a policy (submitting a transaction to call the Move function that creates a Policy object), and later see if a payout was triggered. The front end could also display oracle data feed status (for demo, perhaps a button to simulate the event).

2. Community Crowdfunding with Quadratic Funding (DeFi/City Tech)

Description: A decentralized crowdfunding or grants platform that matches community contributions using a **quadratic funding** mechanism. Users can propose projects (charity, public goods, startups) and others donate. A matching fund (from a sponsor or city budget) is distributed not just by amounts donated, but by number of distinct contributors – broad support is rewarded with larger matches [4]. This encourages many small donations by the community, rather than relying on a few large donors [5]. The blockchain ensures transparency: funds are held in escrow and only released if the funding goal or conditions are met. This project lies at the intersection of DeFi (managing pooled funds, payments) and City Tech (public goods funding).

Implementation Steps:
1. **Smart Contract Modeling:** Create a **Project** struct (likely an object with `has key`) to represent each crowdfunding campaign. This could be a *shared object* so multiple users can contribute. Fields might include target amount, deadline, creator address, total raised, and a vector of contributions (or a Table mapping contributor -> amount). There will also be a global **MatchingPool** (shared object) holding the sponsor's matching funds.
2. **Contribution Logic:** Implement an entry function `contribute(project_id, amount)` where users can send SUI or a custom coin. The function will record the contribution (e.g., push to the contributions vector and increment total). The use of a **Table** can help tally unique contributors by checking if an address exists as a key [4]. Ensure the contributed coins are moved into the Project's escrow (the Project object could own a balance or coin object).
3. **Quadratic Match Calculation:** After the funding round, a function `finalize_project(project_id)` is called (maybe by anyone or by an admin) which computes the matching funds for the project. The formula uses the number of contributors squared relative to donation sums (you can simplify in Move for hackathon purposes). Based on the quadratic funding formula, calculate match from the MatchingPool [6]. Transfer the matched amount and raised funds to the project creator if success criteria are met (e.g., at least X contributors or reached minimum goal). If the project failed (e.g., didn't reach a minimum threshold), allow contributors to withdraw their donations (refund).
4. **Access Control:** Use capability objects to secure certain actions. For example, an **OrganizerCap** could be given to project creators allowing them to call `finalize_project` or withdraw funds. Alternatively, make `finalize_project` public but enforce that funds can only go to the creator's address (stored in the Project). The MatchingPool's withdrawal (to refill or move leftover funds) could be gated by an admin capability held by the funding organization.
5. **Testing:** Write tests for multiple scenarios: multiple contributors donate (ensure contributions accumulate correctly), finalize calculates a higher match for broader support (e.g., 5 people giving 1 SUI each yields more match than 1 person giving 5 SUI) [7]. Test failure case where project doesn't meet criteria (contributors can get refund via a function `refund_contribution`). Also test capability enforcement (only rightful owner can withdraw funds, etc.).
6. **Frontend:** Develop a React app to list active projects with details (from on-chain Project objects). Users connect wallet, choose a project, and enter an amount to donate (transaction to `contribute`). Show live total raised and maybe an *estimated* quadratic match (computed off-chain for display). After deadline, if user is project owner, provide a button to finalize and trigger payout. If a project failed, provide contributors a refund button (calls refund function). The interface thus covers reading on-chain data (project state, contributions) and sending transactions (donate, finalize, refund).

3. Multi-Signature DAO Wallet (DeFi Infrastructure)

Description: Implement a on-chain **multi-signature wallet** for a group (like a DAO treasury). A multi-sig wallet requires M out of N owners to approve a transaction before it executes. This increases security for managing funds or assets by requiring consensus. The project would allow creating a wallet object with a set of owners, proposing transactions, and collecting approvals on-chain. Only when enough approvals are recorded will the contract execute the transaction (e.g., transfer funds). This idea demonstrates core blockchain functionality for security and governance in DeFi DAOs, and can also fit CityTech for managing community funds securely.

Implementation Steps:
1. **Data Modeling:** Design a **MultisigWallet** Move struct (probably *shared*) containing a list (vector) of owner addresses or public keys. Include a threshold `M` (number of required signatures). Also define a struct for **TransactionProposal** with details (e.g., target address, amount, maybe an action type) and a vector of approvals (or a bitmask) from owners. Each proposal could be an object or just a struct inside the wallet's state stored in a table (keyed by proposal id).
2. **Proposal Lifecycle:** Implement entry functions in the Move module: `propose_tx(wallet_id, to: address, amount: u64, ctx)` creates a new proposal (ensuring the caller is an owner). It could create a **Proposal** object (with `has key`) that is *owned by the Wallet object* (object ownership type), or simply add to a vector in the wallet. Return an ID for the proposal. Then `approve_tx(wallet_id, proposal_id)` can be called by owners to record their approval (e.g., add their address to the approvals list). Use checks to prevent duplicate approval by same owner.
3. **Execution:** Provide `execute_tx(wallet_id, proposal_id)` that can be called once enough approvals are collected. It should check that the approvals count >= M (threshold). If yes, it performs the intended action – e.g., transfers coins from the wallet's balance to the target. In Sui, the **MultisigWallet** could itself hold assets (like it might own some SUI coins or other objects). The execution function would likely need to have the Wallet object borrow the coin and use `transfer::transfer` to send out funds. After execution, mark the proposal as executed (or delete it to prevent reuse).
4. **Ownership & Security:** The wallet's **owner list** is set at creation. Creation function `create_wallet(owners: vector<address>, M, ctx)` will publish a new Wallet object. Since multiple people need to call, you might have one creator specify all owners for simplicity. Alternatively, allow adding/removing owners via proposals as well. Ensure only an owner can propose or approve by checking `tx_context::sender()` against the stored owners list (this is basic access control via logic). For added safety, you could issue each owner an **OwnerCap** (capability object) when the wallet is created, and require that capability in approve/execute calls – this explicitly ties function access to possession of the cap [2].
5. **Testing:** Simulate a scenario with 3 owners (A, B, C) and threshold 2. In tests, have A propose a payment, A and B approve (should succeed), then try execution (should transfer funds). Test that execution fails if only A approved (not enough). Test non-owner cannot propose or approve (should be rejected). If using capability objects for owners, test that the function indeed requires the cap (calls without it fail). Also test adding a new owner via a multi-sig decision if you implement that.
6. **Frontend:** Build a dashboard for the DAO wallet. The React app lets an owner connect their wallet, view the wallet's balance and pending proposals. Provide forms to propose a new transaction (enter recipient and amount). Show a list of proposals with their approval status (who has approved, who hasn't). For each proposal, if the user is an owner, allow them to click "Approve" (sends transaction to call `approve_tx`). If enough approvals are reached, allow execution (perhaps automatically by the first user who notices threshold reached and clicks "Execute"). This front end helps visualize multi-sig governance in action and reads on-chain state of proposals and approvals.

4. NFT Rental Marketplace (Gaming/DeFi)

Description: A platform to **rent out NFTs** (such as in-game assets, virtual land, or digital collectibles) for a short period without permanent transfer of ownership. NFT rentals unlock extra utility: lenders earn passive income on assets they own but aren't using, and borrowers get temporary access to expensive or high-level items at low cost [8] [9]. For example, a player could rent a rare game item NFT for a week instead of buying it. The smart contract handles the rental terms and ensures the NFT returns to the owner after the rental duration or that the NFT can't be misused. This project highlights creative use of smart contracts for asset control (a form of "digital lease"), relevant in gaming and metaverse economies.

Implementation Steps:
1. **NFT Representation:** On Sui, NFTs are just Move objects with `has key`. You can integrate with an existing NFT standard or define a simple **GameItem** struct. The rental protocol might wrap the original NFT or hold it in escrow. Two approaches exist [10] [9]: (a) **Collateralized** – borrower puts up collateral which is held until NFT is returned; (b) **Non-collateralized** – the contract mints a **wrapped NFT** for the borrower while the original is locked, then burns the wrapped token on return [11] [12]. For simplicity, approach (b) is appealing: create a **WrappedNFT** struct that references the original NFT ID.
2. **Smart Contract Workflow:** Implement `rent_out(nft_id, renter: address, duration, fee, collateral)` function. The owner calls this to initiate a rental to a specific renter (or you can have an open listing and then renter accepts it). The contract will transfer the original NFT into a **escrow** (e.g., an object held by the contract or mark it as shared with restricted access). Then either require the renter to pay a fee and possibly collateral. Issue the renter a **WrappedNFT** (or a capability) that gives them temporary usage rights. Store the rental terms (expiry time, etc.) in a global *Rentals* table mapping NFT ID -> rental info.
3. **Access Control & Capability:** Ensure that only the original owner can call `rent_out` for their NFT. This can be done by requiring the NFT as a mutable reference in the function (only the owner can send an owned object into a function call). The renter's rights could be enforced by a **BorrowerCap** – e.g., the wrapped NFT or a capability token given to the renter signifies their right to use the asset in-game or on another platform. The contract might have a function `use_item(nft_id, action)` that can only be called by someone who presents the valid BorrowerCap for that NFT. This way, on-chain usage of the NFT is gated. Off-chain, the game could simply trust the on-chain record that this address is the current renter.
4. **Return or Expiry:** Implement `end_rental(nft_id)` which can be called by the owner or automatically after time. If the rental period is over (check current timestamp against stored expiry), the contract burns the Borrower's wrapped NFT/capability and returns the original NFT to the owner (transfer back to owner's address) [13]. If using collateral, it would return the collateral to renter at this point (assuming NFT was returned intact). If renter fails to return and collateral was used, the owner could keep the collateral (this logic can be included for collateralized version). Utilize Sui's clock or timestamp in transactions to enforce time-based conditions (if available; if not, simulate by requiring an explicit call to end the rental which anyone can trigger after a certain block time or epoch).
5. **Testing:** Write tests for both happy path and malicious scenarios: a successful rental where NFT goes to escrow and back to owner after expiry, fee transfer is correct; renter trying to call `end_rental` early (should either not return NFT until due time); an unauthorized third-party trying to manipulate the NFT (should fail because they lack the BorrowerCap or NFT object). Also test that owner cannot re-rent an NFT that is already rented out (contract should mark it unavailable). If collateral is involved, test that failing to return NFT results in owner keeping collateral.
6. **Frontend:** The UI would list NFTs available for rent (or allow owner to list one). For an NFT, show details like rental fee and duration. A user with the asset can set parameters and submit a "List for Rent" transaction. Interested renters connect wallet and click "Rent" to trigger the rental agreement transaction.

The app should show rented NFTs in a user's dashboard with a countdown timer for expiry. If the rental expired, allow the owner to reclaim (calling end_rental). If using a wrapped NFT, the renter's wallet will show an NFT token that could be displayed as "Borrowed [OriginalName]". This front end will utilize wallet integration to sign rental transactions and display on-chain data such as current rentals and ownership statuses.

5. Real Estate Asset Tokenization & Rental Contracts (DeFi/City Tech)

Description: A platform to represent **real-world properties as NFTs (or tokens)** on Sui and facilitate **rental agreements via smart contracts**. Each property (house, apartment, land title) is tokenized as a unique object on-chain. The project has two parts: **(a) Fractional ownership** – allow a property NFT to be split among multiple investors or represented by fungible tokens for shared ownership (like REIT shares); **(b) Smart lease contracts** – tenants and landlords sign on-chain rental agreements where rent payments are automated and security deposits are held in escrow. Using blockchain for real estate can increase transparency in ownership records and automate enforcement of lease terms. Real estate tokenization opens new liquidity and ownership models [14], while smart leases ensure trust (e.g., deposit returns or rent payment enforcement) without relying on centralized parties.

Implementation Steps:
1. **Property Token Model:** Define a Move struct **PropertyNFT** with `has key` to represent a property title. Fields might include an ID, address/metadata, and an optional record of current owner or tenant. Use Sui **object ownership** features: initially, the PropertyNFT is address-owned by the property owner (landlord). To support fractional ownership, you could create a separate **PropertyShare** coin type (with `has store` to allow fungible supply) representing shares in the property. Alternatively, simply allow the PropertyNFT to be owned by a shared custody object if multiple owners – but a simpler approach is a **fungible token** for shares. One owner can "fractionalize" an NFT by locking it and minting share tokens in return.
2. **Rental Agreement Contract:** Create a struct **LeaseContract** (has key) to manage a rental. This could be a *shared object* or owned by landlord but accessible to tenant via capabilities. It contains terms: property ID, landlord, tenant, rent amount, deposit amount, rental period (start/end timestamps). When a tenant and landlord come to an agreement (off-chain or via the app), deploy a new LeaseContract object by a function `create_lease(property_id, tenant, rent, deposit, duration)`. This function should ensure it's called by the property owner (landlord) and perhaps also takes payment/deposit from tenant upfront. On creation, transfer the tenant's deposit (coins) into the LeaseContract's custody. Possibly mark the PropertyNFT as "encumbered" or transfer it to the LeaseContract object (so that the tenant has a form of guarantee of occupancy).
3. **Payment and Enforcement:** Implement a function `pay_rent(lease_id)` that the tenant can call to pay the periodic rent. The LeaseContract holds a record of next due date and required amount. Each call transfers the rent amount from tenant to landlord (perhaps via holding in contract then immediately forwarding to landlord's address). If rent is not paid by due date (could be checked by anyone calling a `check_overdue` function after the date), define an outcome: for example, mark contract as terminated due to breach. The deposit could be partially or fully forfeited to landlord in that case or some penalty logic. If the full term is completed successfully, implement `terminate_lease(lease_id)` which can be called by landlord at end date – it should return the property's possession back to landlord (if it was locked) and return the deposit to tenant (assuming no damages). This termination function might require both parties' agreement or some arbitration if early termination – for hackathon scope, assume happy path or simple

rules.
4. **Access Control:** Use Move's **capability objects** to restrict functions. For example, when creating the LeaseContract, give the tenant a **TenantCap** (or simply use the fact that the lease object lists the tenant's address and check against `tx_context::sender()` for tenant-only actions like pay_rent). A **LandlordCap** could similarly restrict that only the landlord can evict or terminate early. This ensures that only authorized parties can trigger state changes in the lease (e.g., tenant can't reclaim deposit without landlord's approval and vice versa). The underlying PropertyNFT might remain owned by the landlord throughout but if you want to prevent double-renting, you might **wrap** the property into the LeaseContract (similar to NFT rental above) so it cannot be transferred until lease ends.
5. **Testing:** Simulate a full lease lifecycle: creation (assert deposit moved and stored, tenant and landlord recorded), tenant pays rent (assert balances updated correctly, next due date moves forward), tenant misses a payment (simulate by advancing a logical time or calling overdue check function, ensure contract marks default and maybe transfers deposit to owner as penalty), normal lease end (landlord calls terminate after end date, deposit returns to tenant, contract closed). Also test unauthorized actions: e.g., a third party tries to terminate someone else's lease (should fail due to checks or missing capability), tenant tries to call landlord-only function, etc. Include tests for fractional ownership if implemented: e.g., splitting a PropertyNFT into share tokens and then aggregating them back to regain full ownership.
6. **Frontend:** A web interface could allow users to browse tokenized properties available (especially if fractional investing is implemented, users could buy "shares" of a property via a token purchase). For the rental part, if a user is a landlord, they can initiate a new lease by selecting one of their property tokens and inputting tenant address and terms, then send a transaction. The tenant would then confirm by sending the deposit (perhaps the contract creation could be done in one step if tenant's wallet is used, or two steps: landlord offers, tenant accepts). Show the current lease status: next payment due date, a button for tenant to "Pay Rent" (triggers transaction), and at lease end a "End Lease" for the landlord. Also display on-chain verification of payments (e.g., a history of rent transactions stored as events or in the contract state). This UI ensures all interactions (view property NFT data, pay with wallet, etc.) are accessible and user-friendly.

6. Supply Chain Provenance Tracker (City Tech / Enterprise)

Description: A blockchain application that tracks products through a supply chain, enhancing transparency and trust. Each time a product moves from one stage (manufacturer, distributor, retailer) to the next, a record is added on-chain. This creates an immutable **provenance log** that anyone (including consumers or regulators) can verify [15]. For example, consider a pharmaceutical supply chain: the drug's batch is represented by an object on Sui; as it passes quality checks, shipping, warehouse storage, etc., each handler updates the object's state or adds a new entry. Blockchain's tamper-proof ledger makes it easier to identify counterfeit or diverted products [15]. This project is primarily CityTech (public safety and trust), but also enterprise-focused.

Implementation Steps:
1. **Object Model:** Model a **Product** as a Move struct (with `has key`) that represents an item or batch. It can start with the manufacturer as the owner. To allow multiple parties to update it, you might convert it to a *shared object* when the tracking begins (so that it can be mutated by various actors). The Product could have fields like `origin`, `current_holder`, and a vector of **Checkpoints** (each checkpoint struct might contain location, timestamp, holder, and status info). Alternatively, maintain a separate **TrackingLog** object that is shared and appends entries. Keeping it all in the Product object for simplicity is fine, just ensure it can grow a list of updates.
2. **Capability-based Roles:** Assign each supply chain participant a role with specific permissions. For

instance, **ManufacturerCap**, **DistributorCap**, **RetailerCap** could be resource objects given to addresses representing those companies. Certain Move functions like `record_checkpoint(product_id, new_holder, info)` will require the caller to present the appropriate capability to ensure only authorized parties can update the product record at a given stage. For example, only someone with a DistributorCap should call the function that marks a product as received at distribution center. Alternatively, store in the product an allowed `next_role` and check the caller's role matches. Capabilities are a straightforward way to gate these updates [2].

3. **Transfer of Ownership vs Shared Updates:** Decide whether the Product object itself changes ownership as it moves (address-owned by whoever currently holds it) or remains shared with an internal state field denoting current holder. Sui allows *object-to-object* ownership as well, so a pattern could be to nest the Product inside a **Holder** object (like a warehouse object owns all products it currently holds). However, that might complicate queries. For hackathon, a simpler approach: leave Product as shared and just update a `current_holder` field each time. Use a Table for quick lookup of products by holder if needed for querying (mapping holder -> list of product IDs).

4. **Immutable Audit Trail:** Ensure that each checkpoint added is append-only. You might append to a vector of checkpoints. Because blockchain is immutable in history, even if the object state changes, one could query past versions or use event emit (if Sui Move supports events) on each update. If events are available, emit an event with checkpoint details so an off-chain system or explorer can reconstruct the entire chain of custody easily [15].

5. **Testing:** Simulate a full chain: Manufacturer creates a Product (with initial data). Then call functions in sequence for each stage: e.g., manufacturer -> distributor -> retailer. At each step, verify the function rejects calls from an entity that doesn't have the proper capability or if steps are out of order. Check that the Product's history vector length increases and data matches the inputs at each step. Also test tampering: attempt to skip a step or modify an old checkpoint (should be impossible because either object fields are not directly mutable except via allowed functions, and old vector entries remain as history). If using events, you can test that events are emitted.

6. **Frontend:** Develop a web dashboard for supply chain tracking. Each participant (company) would connect with their wallet (which holds a role capability NFT proving they are a manufacturer, etc.). The UI allows them to scan or enter a Product ID and update its status ("Mark as shipped", "Receive at Warehouse", "Out for Delivery", etc.). These actions correspond to calling the Move entry functions with their capability. For the public or consumer side, provide a search by Product ID (or maybe a QR code scan that resolves to an ID) to view the provenance log – the app would read the Product object's checkpoints vector or fetch past events, and display an ordered timeline of where the product has been. This instills confidence by letting anyone verify the chain of custody recorded on Sui blockchain (as noted, blockchain provides a trustworthy record of each stage [15]).

7. Carbon Credit Tracking & Trading System (DeFi/City Tech)

Description: An environmental DApp for managing **carbon credits** on-chain. Companies or individuals who reduce emissions get carbon credit tokens, which they can sell to those who need offsets. The blockchain ensures **no double spending of credits** and transparent tracking from issuance to retirement (when a credit is used to offset emissions) [16]. Core features include: tokenization of carbon credits (each representing, say, 1 ton of CO_2), a marketplace to trade them, and a mechanism to "retire" a credit (mark it as used for offset) with public proof. This fits DeFi (trading of tokens) and CityTech, as it can aid city or corporate sustainability programs with open accounting.

Implementation Steps:

1. **Token Design:** Decide between using a **fungible coin type** or NFTs for credits. Likely, carbon credits are fungible in units (though with different projects, they might be non-fungible by source). For simplicity, create a Move **Coin** type called `CarbonCredit` with a certain supply. Sui Move provides a framework for coin types (implementing the `Transfer` and `store` abilities). This would allow standard wallet usage for the credits. Each credit coin could have metadata of its origin project, or different categories can be separate coin types if needed (e.g., renewable vs forestry credits).
2. **Issuance (Minting):** Only authorized registries or authorities should create new carbon credits (to ensure they correspond to real emissions reductions). Implement an **IssuerCap** object held by approved issuers. The Move module can have a function `mint_credits(amount, recipient)` that requires the caller to present an IssuerCap. This function mints the specified amount of `CarbonCredit` coins to the recipient's address. All mint events or the total supply change is tracked on chain. By limiting mint to those with the cap, you prevent arbitrary creation of credits.
3. **Trading Marketplace:** Since credits are coins, they can be transferred P2P easily. You might build a simple marketplace in the Move module or just rely on users trading via a DEX. A simple approach: implement a **listing object**: users who want to sell credits can lock a certain amount in a `Listing` (with price info), and buyers can purchase by sending payment (like SUI) to an entry function that transfers the credits to buyer and SUI to seller. However, a full order book may be overkill for hackathon – even a basic escrow sale or direct transfer mechanism is fine to demonstrate.
4. **Retirement (Burning):** Key feature: when a credit is used to offset carbon, it should be **retired** so it cannot be reused or resold. Implement `retire_credit(amount, reason)` which essentially burns the given amount of CarbonCredit coins (removing them from circulation) and logs an event or record that these credits were retired by a certain account for a certain purpose. The immutability of blockchain records here provides verifiable proof that those credits are claimed and not double-counted [16]. For transparency, you might maintain a global counter or object that tracks total retired credits, or even a mapping of addresses to how many they've retired (for reputation or scoring purposes).
5. **Testing:** Test minting by authorized vs unauthorized (only IssuerCap owners can mint; others should fail). Test transfers of credits between users (ensure balances update). Test the retire function actually burns coins (e.g., try retiring more than you have should fail, retiring should reduce total supply). Test that once retired, those credits are not spendable – essentially, once burned they're gone, but you might also store a *RetiredCredit* NFT as a certificate in the user's account to symbolize their action (non-transferable, just for record). If so, test that such certificate is issued correctly. Also test marketplace listing if implemented (list, buy, ensure correct transfers).
6. **Frontend:** Build a user-friendly interface for both issuers and regular users. Issuers (with a special wallet key or address flagged) can access an admin panel to mint new credits to project owners. Regular users see their balance of CarbonCredit (the wallet integration should treat it like any coin). Provide a marketplace view where sellers can post offers (if doing in-app trades) and buyers can purchase. Also crucial is a **Retire Credits** feature: user enters amount of credits to retire, maybe tags it with a project or reason, then confirms. After transaction, update the UI to show their new balance and perhaps display a "Certificate of Retirement" (which could be an NFT stored, or just a transaction receipt). Also, display aggregate stats like total retired credits (from on-chain data) to reinforce the transparency benefit (blockchain records prevent double counting of retired credits [16]). This app not only demonstrates token handling but also a socially impactful blockchain use case.

8. On-Chain Lottery & Verifiable Random Draw (DeFi/Gaming)

Description: A decentralized lottery dApp where users buy tickets (with cryptocurrency) and a winner is chosen randomly at a set interval, with prize money paid out automatically via the smart contract. The twist is to make the randomness verifiable (using an oracle or cryptographic technique) so participants can trust that the outcome isn't rigged. Lotteries are a classic blockchain showcase because of transparency: all ticket purchases are recorded, and anyone can inspect the contract's logic for selecting winners. This project demonstrates managing a **pool of funds**, NFTs or receipts for tickets, and possibly integration with randomness oracles. It fits DeFi (as it deals with pooled funds distribution) and can be gamified (Gaming).

Implementation Steps:
1. **Ticket Sale:** Implement a Move module for the lottery. A lottery round could be represented by a **LotteryRound** object (shared) that holds state: ticket price, list or count of tickets sold, and maybe a randomness result or winner field (empty until draw). Create an entry function `buy_ticket(round_id)` that accepts payment (e.g., moves a certain amount of SUI or a coin into the contract) and then issues the buyer a **Ticket** (this could be a simple struct or just a number). Perhaps mint an NFT Ticket object for the buyer as proof of entry. At minimum, record their entry in the LotteryRound's storage: e.g., push the buyer's address or ticket number into a vector.
2. **Random Winner Selection:** Since true randomness is tricky on-chain, integrate an oracle or a pseudo-random method. One approach: use an off-chain service that observes the lottery round and calls a `draw_winner(round_id, random_seed)` function with a randomness source (the contract would verify it, if possible). Another approach is to use the last few digits of a future block's hash as random (not perfect but could suffice for demo). The Move function `draw_winner` (probably restricted to an authorized role or the oracle address) will pick a winner index from the tickets vector. For fairness, you might use the modulo of a random seed with the number of tickets to select the index. If using Chainlink VRF or similar is not possible directly on Sui at this time, simply simulating an oracle by calling from a privileged account with a random number is okay.
3. **Payout Logic:** When a winner is determined, the contract should transfer the prize. The prize is basically the pooled funds from ticket sales (minus maybe a fee). Ensure the funds from `buy_ticket` were stored – perhaps the LotteryRound object holds a balance of coin. After drawing, call `transfer::transfer` to send the coin balance to the winner's address. Mark the round as completed (and maybe store the winner and winning ticket for record). If you issue NFT tickets, you could even mark the winning ticket NFT's metadata as "Winner".
4. **Security & Fairness:** Use a **DrawAdminCap** that only a trusted party or oracle node possesses to call `draw_winner`, to prevent arbitrary draws. All other functions (like buying tickets) are open to anyone. You might also prevent buying tickets after a certain time or after a maximum count, by checking within `buy_ticket` (e.g., the round could store an `end_time` or status). If a user tries to buy after the round ended, reject the transaction. To increase user trust, emphasize that once the round is over, even the admin cannot change outcome except via the random draw function which follows the public logic (everyone can see the code that winner = hash % N, etc., providing **verifiability** when given a legitimate random hash).
5. **Testing:** Write tests where multiple simulated users buy tickets (you can call `buy_ticket` from different test accounts). Ensure the contract accumulates the fund correctly. Then simulate the draw: either call `draw_winner` with a fixed seed and verify the winner is chosen as expected. For example, if 3 tickets and you pass a seed that mod 3 equals 1, check the winner corresponds to ticket index 1 in your vector. Test edge cases: no tickets (should not draw), one ticket (that one should win obviously), trying to buy after draw (should fail). If possible, test that only the admin (e.g., test account with DrawAdminCap) can call draw and others are rejected.

6. **Frontend:** The user interface would show the current lottery round info: ticket price, total tickets sold (or odds), time left until draw. Users connect their wallet and can press "Buy Ticket" to purchase (which triggers the transaction). After purchase, they might see their ticket number or an NFT in their wallet. Once the round is over and a winner drawn, the UI should display the winner (maybe their address or ticket number) and the prize amount. If the connected user won, it should prominently congratulate them and show the prize in their wallet balance. Optionally, display transparency info such as the random seed or hash used for the draw. A nice addition: a history of past rounds with winners (the contract could store past results or one could index events off-chain and display it). This project thus involves wallet connections for payment and reading contract state for live updates.

9. Recurring Payments & Subscription Manager (DeFi/ Infrastructure)

Description: A smart contract system to support **recurring payments** (subscriptions) on Sui. Currently, most blockchain payments are one-off; this project would enable a user to authorize a subscription so that a service can be paid at regular intervals automatically or with minimal user intervention [17] [18]. For example, a content platform could charge 5 SUI per month. The core idea is to create an on-chain agreement that either locks funds and releases them periodically or at least records permission for recurring charges. This demonstrates a **decentralized subscription** model, eliminating the need for traditional payment processors and allowing trustless enforcement of subscription terms (no forgetting to cancel free trials—smart contracts can handle it!).

Implementation Steps:
1. **Subscription Object:** Define a **Subscription** struct (has key) that represents an active subscription agreement. Fields include: subscriber (payer address), merchant (payee address), interval (e.g., seconds or days between payments), amount per interval, next_payment_time, and maybe a count or end date if it's not indefinite. This object can be *owned by the subscriber's address* or be a shared object indexed by both parties. Owning it by subscriber might make sense (so they can easily find/cancel it), but we might allow the payee to trigger charges, so shared could work too.
2. **Establishing a Subscription:** Provide an entry function `create_subscription(merchant: address, amount: Coin<SUI>, interval: u64, ctx)` that a user (subscriber) calls to initiate. They would specify the interval and attach the first payment (perhaps as a coin object). The function will create the Subscription object storing these terms. It should also perhaps hold the initial payment ready to transfer to merchant. One approach: directly transfer the first payment to the merchant immediately on creation (since user provided it). Or hold it and require merchant to claim. To ensure future payments, one idea is to also lock some deposit or pre-fund, but that puts burden on user. Instead, we might simply trust that when time comes, the subscriber or merchant will trigger the next payment manually.
3. **Automating Payments:** Because smart contracts can't by themselves initiate transactions on Sui (someone must send a transaction), you can implement a function `process_payment(sub_id)` that when called (by anyone, or by merchant) will check if the current time is past `next_payment_time`. If so, it transfers the `amount` from subscriber to merchant. This requires the subscriber *pre-approval* or locking of funds. We have a few options: (a) On subscription creation, lock a large pool of the subscriber's funds in the contract (like funding an account from which payments will draw) and authorize the contract to transfer to merchant on schedule. (b) Or do not lock, but expect the subscriber to keep enough balance and simply attempt transfer from their address (which would require their signature each time – meaning truly

automatic pull is not possible without their key). Since we want a more autonomous approach, option (a) is closer. We can have the subscriber deposit, say, 6 months worth of SUI upfront into the Subscription object's escrow. Then each month, the contract moves one installment to merchant and decrements the escrow balance. If funds run out, the subscription can pause or cancel.

4. **Cancellation and Control:** The subscriber should be able to cancel anytime to stop future charges. Implement `cancel_subscription(sub_id)` that only the subscriber (owner) can call. This would mark it inactive and refund any remaining escrow balance back to the subscriber. The merchant might also have a function to cancel (in case they discontinue service; that should also refund subscriber's remaining balance). These operations are sensitive, so use the stored addresses to check caller, or give each party a **Capability**: e.g., a `SubscriptionCap` given to merchant to allow them to call `process_payment` or cancel from their side. But checking `tx_context::sender()` against the stored merchant in Subscription might suffice for gating merchant actions.

5. **Testing:** Create a subscription with a test account, deposit some amount. Simulate the passage of time by calling `process_payment` when due (you may need to manipulate `Clock` if Sui provides one, or simply override `next_payment_time` for testing). Verify that payments transfer correctly to the merchant (you can check merchant's coin balance increased). Test cancellation: if subscriber cancels early, ensure no further payments can be processed and leftover funds return. Test that no one except subscriber or merchant can cancel or trigger payments (unless you allow anyone to trigger payment, but if so, test that too – it could be safe to allow anyone to call `process_payment` since the logic will only move funds if it's due and authorized). Also test edge cases like creating a subscription with not enough initial funding (maybe reject or handle via partial periods).

6. **Frontend:** The app should allow a user to manage their subscriptions like a **Web3 subscription dashboard**. A subscriber can connect wallet, see a form "Create Subscription" where they input merchant address (could be an ID from a list), amount, interval, and pre-fund amount. On submission, it triggers the transaction. After that, the user's dashboard lists active subscriptions with details (next payment date, merchant name if known, remaining balance). They should have a "Cancel" button next to each. For merchants, if they connect their wallet, they see a list of subscribers who have active subs with them. The merchant could click "Charge Now" if a payment is due (or perhaps this could be automatic via a backend agent, but UI might allow manual trigger for demo). This frontend emphasizes reading on-chain subscription objects and enabling user transactions to manage them, showcasing how blockchain can handle subscription-style agreements without centralized payment processors [17].

10. Decentralized Identity & Reputation (Soulbound Tokens) (AI & Education & DeFi)

Description: Establish a **Web3 identity system** using **Soulbound Tokens (SBTs)** – non-transferable NFTs that represent personal credentials or reputation [19]. The idea is to give users a way to accumulate verifiable credentials (like a degree, a course certificate, a government ID, or a reputation score for contributions) on their Sui wallet without fearing them being sold or separated from their identity. For example, an educational institution could issue a diploma SBT to a student's address, which the student can showcase but not sell. Employers or DAOs could check these SBTs to verify qualifications. Over time, users build a **trust profile** of achievements and reputation on-chain [20]. This project blends Education (credentials), CityTech (digital ID), and even AI (if we add an AI that provides scoring or verification based on data). The smart contract aspect focuses on managing non-transferable tokens and perhaps a **reputation ledger** that can be incremented by community votes or contributions.

Implementation Steps:

1. **Soulbound Token Implementation:** Create a Move module for an SBT. This could be as simple as a struct `SoulboundToken` with `has key` and some metadata fields (e.g., name, issuer, description). The key part is to **restrict transfers**. In Move, if you do not implement any `transfer` or `public_transfer` for the object, then by default it cannot be transferred between addresses. Alternatively, you mark it as **non-transferable** explicitly by not giving it the `store` ability or by ensuring any transfer function aborts. You might register the type so that wallets recognize it as an NFT representing a credential. Each type of credential might be a different struct or you have a generic one with a category field.
2. **Issuance Control:** Only authorized entities (like universities, government, or employer addresses) should issue certain SBTs. Implement capability objects like **IssuerCap** for each category of credential (EducationIssuerCap for diplomas, EmployerIssuerCap for work badges, etc.). The contract functions `mint_sbt(receiver: address, data: …)` will require the issuer to present the appropriate cap. For example, University A holds an EducationIssuerCap that allows them to call `mint_diploma(address student, string degree)` which creates a SoulboundToken in the student's account representing their degree. Because SBTs are non-transferable, the student can't send it elsewhere – it's bound to their "soul" (their address) [19].
3. **Reputation Scoring:** For a general reputation token, you could maintain a **ReputationScore** mapping or token that counts points. For instance, in a DAO context, every time a user contributes, an admin can call `add_reputation(user, points)` to increment a score stored in a table or in an SBT (you could have one SBT per user that holds a score field updated over time). This part might require careful design to avoid abuse – maybe use a community voting or an AI (off-chain) to recommend reputation changes. But to keep on-chain, a simple approach: a reputation SBT that an authorized contract or set of addresses can update (by capability).
4. **Verification and Usage:** Provide read-only helper functions or events such that external dApps can easily verify a user's credentials. For example, a function `has_credential(addr: address, cred_type: string): bool` that checks the user's inventory for a specific SBT type. Or just rely on queries off-chain to the objects. One could also implement a **Zero-knowledge proof** approach for privacy (where user proves they have an SBT without revealing all details), but that's advanced and probably not needed in a basic hackathon project. Focus on the transparent use-case: e.g., a job marketplace dApp could filter candidates by requiring they hold a particular SBT (the job license or degree).
5. **Testing:** Mint various SBTs in tests: ensure that when trying to transfer an SBT object via normal transfer call it fails or is impossible. Test that only the rightful issuer can mint (others calling mint should abort). If there's an update function (like add_reputation), test that unauthorized cannot call it. Test querying logic if any. Also test a scenario where a user loses a key – since SBT is bound to the address, you might consider a **recovery** mechanism (perhaps an admin can reissue a new token to a new address if proof of identity off-chain). This could be a stretch feature: implement an `invalidate_sbt(old_id)` by issuer and `reissue_sbt(new_address)` to handle lost wallets (with careful checks).
6. **Frontend:** This would manifest as a **profile page** for users. After connecting their wallet, they can see their collection of SBT badges (diplomas, certificates, memberships, etc.). The UI can show each credential's details (since they are NFTs, metadata can be displayed). If the user is an issuer (e.g., logged in as a university admin), the UI would present a form to issue a new credential (enter recipient address, details, then call `mint`). For reputation, maybe display their score or badges of achievements. Additionally, demonstrate usage: for instance, an "exclusive access" section where the app checks for a specific SBT. If the user lacks it, show a message "you need X credential to access this feature." If they have it, grant access. This shows integration of on-chain identity into app logic. Such identity tokens, being non-transferable, **represent a person's achievements and reputation** which cannot be sold [19], aligning with the concept of personal identity on Web3.

11. Decentralized Freelance Marketplace with Escrow (City Tech/ DeFi)

Description: A peer-to-peer freelance jobs platform where payments are held in **escrow smart contracts** until work is completed. This eliminates the need for a centralized platform like Upwork, and builds trust between clients and freelancers through **trustless escrow** [21]. A client creates a job listing with payment; a freelancer delivers work; if the client approves, the escrow releases funds to the freelancer. If there's a dispute, an arbitrator or predefined rules can resolve it (for hackathon, maybe let client or both agree). The system can also record **reputation** for each party after each job (tying in with idea #10 possibly). The core blockchain functionality here is managing conditional payments and multi-party approval flows with smart contracts, fitting DeFi (escrow of payments) and CityTech (empowering gig economy without intermediaries).

Implementation Steps:
1. **Job and Escrow Structs:** Define a **JobContract** struct (has key) to represent an escrow for a freelance job. Fields: client address, freelancer address (once hired/assigned), payment amount (maybe a Coin object or balance), a description or IPFS hash for job details, status (Posted, InProgress, Completed, Disputed, etc.). Initially, a client creates a JobContract with status Posted and with the payment locked inside it. This means the client would call `create_job(freelancer: address, description: ..., payment: Coin<SUI>)`. The function will take the payment coin and store it in the JobContract object (the object now *owns* these funds, ensuring neither party can access them until conditions met). It will record the chosen freelancer (or you could have a mechanism where freelancer is not pre-selected, but for simplicity the client might directly choose or this is a one-to-one contract).
2. **Workflow Functions:** The freelancer, after doing the work, calls `submit_work(job_id, proof_link)` – this could update the job status to Completed and perhaps store a link or hash of the delivered work (like an IPFS hash of a file or just mark it as delivered). Then the client calls `approve_work(job_id)` which will transfer the payment to the freelancer and mark job as approved/closed. If the client is unhappy, they could call `reject_work(job_id)` to mark dispute (and maybe then both parties or an arbitrator need to resolve manually, which is beyond straightforward logic – for hackathon, you might simply allow the client to cancel and refund if work not approved, but that's open to abuse by client; a better approach is to involve a third-party or require both to agree to cancel). For simplicity: include a `cancel_job(job_id)` that client can call only before freelancer submits work (to get refund), and maybe a `dispute_job(job_id)` that flags an issue (but without an arbitration mechanism, dispute could just freeze the funds requiring off-chain resolution).
3. **Access Control:** Ensure only the designated freelancer can submit work on that contract (check `tx_context::sender() == freelancer` in `submit_work`). Only the client should approve or cancel, etc. You could formalize this with capabilities – e.g., give the freelancer a **JobFreelancerCap** at job creation (transfer a tiny object representing their right to finalize the job) and similarly a **JobClientCap** to the client. However, since their addresses are known and fixed, a simpler check by address is fine. Capabilities would matter if you wanted to enforce that they must present the cap to act, adding an extra security layer (but risk losing the cap means losing control, so address check might be safer in this context).
4. **Reputation Recording:** Optionally, after completion, allow each party to rate the other. For example, `rate_client(job_id, score)` and `rate_freelancer(job_id, score)` functions that each party can call once, which would write a rating into a table mapping the user to an accumulated score or store a feedback object. This can integrate with the SBT reputation (#10) by minting a feedback token or updating a score in a reputation object for that user. If including this, ensure one party cannot rate themselves or

multiple times (tie it to the job and require that job is completed and the rater is one of the parties).
5. **Testing:** Simulate a full happy-path flow: client creates job (funds escrow, verify JobContract now holds the fund and status is Posted), freelancer submits (status -> Completed), client approves (fund moves to freelancer and status -> Closed). Check balances to confirm escrow released exactly the payment amount. Test cancellation: client cancels before freelancer submits (funds refunded to client). Test unauthorized: e.g., someone other than freelancer trying to submit, or a third party trying to approve (should fail due to checks). If dispute flow included, test that dispute stops payout (i.e., after dispute, approve function maybe cannot be called unless some condition toggled by arbitrator). Also test rating: ensure each can only rate once and the score updates correctly.
6. **Frontend:** The dApp interface would have two views: **Client view** and **Freelancer view**. Clients can post a new job by filling a form (description and freelancer's wallet address or selecting from a list of available freelancers) and specifying payment amount. On creation, their wallet will send the payment to the contract. The job then appears in their dashboard with status. The freelancer, when logged in, will see jobs assigned to them with status. They should have an option to "Submit work" (perhaps upload a file to IPFS via the app and then call `submit_work` with the hash). The client then sees the job marked as ready for review and can click "Approve" or "Reject". If approved, the UI indicates the payment released (maybe show transaction hash or updated balances). If rejected/disputed, you might simply mark it and show a message to contact support (since full dispute mediation is complex). Also show past jobs with whether they were completed successfully. If implementing reputation, display each user's rating (maybe as stars) in their profile. This decentralized marketplace showcases how **escrow smart contracts enable trustless payments** where funds are only released upon mutual agreement [21].

12. NFT Ticketing Platform for Events (City Tech/DeFi)

Description: A blockchain-based event ticketing system using NFTs as tickets. Each ticket is a unique token tied to a specific event and seat, issued by the event organizer to the buyer. **Blockchain ticketing** increases transparency and security: it prevents counterfeit tickets and allows control over resale (e.g., setting price caps or royalties on secondary sales) [22] [23]. After the event, the NFT ticket could even serve as a collectible or proof-of-attendance badge. This project emphasizes on-chain asset management (minting NFTs, transferring, validating ownership at entry) and can incorporate rules like "tickets un-transferable close to event date" or "organizer gets 10% of any resale". It aligns with CityTech (improving public event management) and DeFi/NFT tech.

Implementation Steps:
1. **Event and Ticket Structs:** Define an **Event** struct (has key) for each event, with details like name, date, organizer (address), and maybe a limit on tickets or a pricing structure. Then a **Ticket NFT** struct (has key) that contains an `event_id` reference, a seat or ticket number, and perhaps a boolean field `used` (to mark if it's already checked-in). Each Ticket is an object that will be owned by the buyer (address ownership). Mark the Ticket such that it can be transferred (to allow resale) but you will control that via functions. Possibly implement the transferable with a custom function to enforce rules (instead of letting free transfer).
2. **Minting Tickets:** The organizer (who holds an OrganizerCap for that event or simply the address that created the Event object) can call `mint_ticket(event_id, buyer_address, seat_info)` to create a new Ticket and transfer it to the buyer. Alternatively, an initial sale function `buy_ticket(event_id)` could be open to public which when called with payment will mint a ticket to `tx_context.sender`. For simplicity, you might pre-mint a batch of Ticket objects and let people claim/purchase them. Payment handling: incorporate payment in ticket purchase if needed (like require a certain coin amount and transfer

it to organizer or an event treasury).
3. **Transfer/Resale Control:** To implement policies like preventing scalping, you can restrict how transfers occur. For example, you might override or not expose a direct `transfer<Ticket>` function. Instead, have a function `resell_ticket(ticket_id, to: address, price)` that does: require that the current time is more than X hours before event (to disallow last-minute transfer that can facilitate scalping bots), or enforce that the price is <= some cap. If a resale occurs, you could have the function split the payment – e.g., enforce a royalty to the organizer (say 10%). This means the buyer would call `resell_ticket` providing the buyer's address and the buyer would actually send the payment which the contract divides. This gets complicated to orchestrate on-chain; a simpler approach: just allow free transfer but encourage using the contract for sale to enforce rules. For hack demo, you can simplify to free transfers but track them, or implement a minimal marketplace where the ticket owner can list it and a new buyer purchases through the contract so it can enforce a fee.
4. **Verification (Check-In):** Provide a function `use_ticket(ticket_id)` that marks the ticket as used (set the `used` field to true). This would be called at the event gate by a staff member. To ensure only event staff can do this, give the event organizer or a staff address a **CheckerCap** (capability) that must be presented to use the ticket. Then, when a ticket is used, it cannot be reused or resold (you might even burn or lock it after use). This on-chain check-in could be accompanied by an off-chain scanner that reads the NFT's QR code, then calls this function via a backend.
5. **Testing:** Mint some tickets and simulate transfers: ensure that only the organizer can mint. Test normal transfer if allowed, or use the `resell_ticket` function if that's how transfers are done – check that rules like time or price cap are enforced (you can simulate current time by perhaps reading a Time object if available, or have an event field for cutoff and just not test time logic strictly). Test that `use_ticket` cannot be called by just anyone (should require the CheckerCap or at least the organizer's authority). Mark a ticket used and then try transferring it (should either fail or if transferred, at least the used status is carried so it won't be valid for entry). If payment is involved in any sale, test that the payment distribution (to seller and organizer royalty) sums up correctly.
6. **Frontend:** The application would allow event organizers to log in (connect with their wallet) and create a new Event (input details, call create_event function). Then they can either issue tickets or configure a sale. For users, show available events and remaining tickets. A user can purchase a ticket by clicking an event and confirming the purchase transaction. After purchase, the ticket NFT should appear in their wallet (the app can show a "My Tickets" section reading their owned Ticket objects). For resales, implement a section where a user can list their ticket: input price, and then another user can see listed tickets for an event and buy one (the app would coordinate calling the resell function appropriately). At the venue, a staff interface (maybe a simplified mobile view) can scan a ticket QR (the QR would contain ticket ID) and call the check-in function – the UI then shows success or if already used. This demonstrates how **NFT tickets solve issues like counterfeiting and scalping by leveraging blockchain verification** [22]. The organizer could also see a dashboard of all tickets (sold, transferred, used) for their event in real-time.

13. Gamified Learning Platform with On-Chain Rewards (Education/AI)

Description: An educational dApp that uses blockchain to **reward learners with tokens or NFTs** as they complete quizzes or courses. This idea merges Education with Blockchain (and potentially AI if you generate questions or personalize content). For instance, a platform could have coding challenges or quizzes; when a student passes a quiz, a smart contract mints them an NFT certificate or issues tokens as a reward. These credentials are recorded on-chain (which employers or teachers can verify), and rewards can be used to

unlock further content or traded. The blockchain ensures the achievements are tamper-proof and owned by the student. Optionally, incorporate an AI tutor: e.g., an AI chatbot that gives hints or generates personalized quizzes, and still record achievements on-chain for transparency.

Implementation Steps:

1. **Course and Quiz Contracts:** Define a **Course** object with details about a module or topic. Each course can have one or more quizzes or tasks. Instead of putting questions on-chain, which is impractical, keep questions off-chain or pre-defined, and focus on recording completion. One approach: for each quiz, have a Move function `submit_answer(course_id, quiz_id, answer_hash)` that a user calls when they finish a quiz. The contract can't grade a free-form answer (that's where an AI or server would come in), but it could accept an already graded result. So perhaps the flow is: user takes quiz off-chain, the system or AI grades it, and if passed, the dApp calls the contract function to record the pass. To prevent cheating, you might require a signature or code from the grader. But for hackathon, a simple simulation: the user calls `complete_quiz` and we trust them (or have an "oracle" account that calls it to confirm).
2. **Reward Minting:** Once a quiz is passed, the contract mints a reward. This could be an **NFT badge** (e.g., "Blockchain Basics Certificate") or a certain amount of an ERC-20-like **token** (like EDU tokens). Use Move to mint an NFT for the specific achievement – a struct AchievementNFT with course info and student address. Mark it non-transferable if it's meant as a personal certificate, or transferable if it's a collectible. Alternatively or in addition, mint fungible reward tokens (like points) that could be spent in the platform for perks. Showcasing NFT mint is more visible as a credential.
3. **Tracking Progress:** The contract can maintain a **record** of which user completed which course. Perhaps each Course object has a Table of (address -> bool) for completion, or each user address has a profile object listing completed courses. If using NFTs as proof, that alone can indicate completion when queried. But having a mapping or event for completion might be useful for quick queries or to prevent duplicate rewards (only allow one NFT per user per course). Implement checks in `submit_answer` (or `complete_quiz`) that if the user already completed it, don't double-issue rewards.
4. **AI Integration (optional):** If including AI, you might not do that in Move (since AI models run off-chain). Instead, you could have an off-chain component: e.g., an AI tutor chatbot that uses OpenAI to provide dynamic questions or explanations to the student. This is outside the blockchain, but your platform can still log when the user accomplishes something. Possibly, for an "explain this code" task, the AI verifies the explanation and then calls the smart contract to record success. Since the user asked for AI & Blockchain, at least mention how AI can be used for personalization while blockchain handles credentialing.
5. **Testing:** If focusing on on-chain part, test the reward issuance: call the complete function for a user, ensure the NFT is minted to them and the completion status is recorded. Test that calling it again doesn't double mint. Test that an unauthorized user (not the oracle/grader) cannot call the function if you intended only platform to call it (or if open, ensure it at least doesn't mint twice or that some validation is in place). If tokens are awarded, test the token balance increments. If some courses have prerequisites, you could enforce that by checking if the user holds a certain NFT before allowing another (test that gating if implemented).
6. **Frontend:** The user interface is like a learning dashboard. Users sign in with wallet, enroll in a course (maybe just choose from list). They go through content (which could be just text/video outside the blockchain). Then they take a quiz. The quiz could be served by the web app or an AI chatbot. Once they finish, if they passed, the app will trigger a blockchain transaction to claim their reward (perhaps the app calls a backend to verify answers then that backend uses the user's wallet or an oracle key to call the contract). After that, the user sees a new NFT badge in their profile on the blockchain (the app can show it, maybe even allow them to share a link to prove completion). If using tokens, show their token balance and maybe a small marketplace to redeem tokens for something (could be simple like "100 tokens for a

mentorship session coupon" to illustrate use). Additionally, show a public leaderboard or profile page where each user's on-chain achievements are visible, highlighting how blockchain provides an **immutable record of educational achievements** that can't be easily falsified.

14. AI-Powered DAO Governance Assistant (AI/Blockchain)

Description: A novel idea combining AI and blockchain: an **AI Assistant for a DAO** that analyzes proposals and on-chain data to provide recommendations or summaries to governance token holders. The AI could read the text of proposals, check relevant metrics (like treasury status, past votes), and then the **blockchain part** is to integrate this assistant into the governance process. For example, the AI's summary and risk assessment of each proposal could be **written on-chain** (as an immutable record or as an advice object attached to the proposal). Or the AI could even be authorized to **initiate certain routine proposals** if conditions are met (acting as an autonomous agent for the DAO). This project emphasizes smart contracts for governance (voting, proposals) and uses AI for decision support. The core smart contract could be similar to a governance module (managing proposals, votes, execution), but extended to accept input from an AI oracle. It fits AI & Blockchain track and DeFi (DAO treasury management).

Implementation Steps:
1. **DAO Governance Contract:** Implement a basic governance system: a **Proposal** struct with `id, description, yes_votes, no_votes, status, etc.` and functions to create proposals, vote, and execute proposals if passed. This is standard – token-weighted voting or one-address-one-vote depending on design. Focus on including fields or hooks for AI input. For instance, add a field in Proposal like `ai_advice: String` or `risk_score: u8` which the AI can fill. The proposal creation could be open to any member or restricted to whitelisted (or an AI itself).
2. **AI Assistant Input:** Decide how the AI communicates on-chain. One way: treat the AI as a special oracle that can call a function `provide_analysis(proposal_id, summary: String, score: u8)` on the contract. You would give the AI agent's on-chain address a special capability (e.g., **AdvisorCap**). Only the AI (or rather, an off-chain service acting on AI's behalf with that cap) can call this. When a new proposal is created, the contract could emit an event or simply be polled by the AI service. The service generates a summary, then sends a transaction with `AdvisorCap` to record its analysis on the chain (updating the Proposal's `ai_advice` and `risk_score`). This makes the analysis part of the proposal record that voters can see on-chain.
3. **Automated Proposals:** You can also allow the AI to create certain proposals directly. For example, if on-chain metrics indicate treasury is low, the AI might propose "mint more tokens" or "adjust budget". To enable this, you can grant the AI's address the right to call `create_proposal` with `AdvisorCap` for specific types. You'd have logic like: if caller is AI and they attached AdvisorCap, allow creation even if they don't have normal proposal rights (or the cap itself is the right). This is potentially powerful, so emphasize it's for predetermined scenarios or to augment humans, not fully control.
4. **Voting and Execution:** The rest of governance contract runs normally: users vote (perhaps weight by a governance token they hold, which could be an object with a balance). Once voting period over, an `execute_proposal` can be called to enact changes if it passed. Execution might just emit an event (in a real system it could call other contract functions to change parameters, etc., but that might be beyond scope to implement fully). For demonstration, maybe integrate with another part of the system – e.g., a passed proposal could trigger a coin transfer from a treasury account (which you manage as an object) to some target (like spending funds).
5. **Testing:** Simulate a cycle: create a proposal (by a user), ensure AI (simulated by a specific test account with AdvisorCap) can call `provide_analysis` to add a summary. Check that if someone else tries to call

that function, it aborts. Have several accounts vote and tally. Possibly simulate an AI-created proposal: directly call create_proposal with the AI account and ensure it succeeds whereas a normal user might need to hold tokens or meet a threshold. Test execution does expected state changes (maybe if proposal was to transfer funds, check funds moved, if just a state toggle, check that).
6. **Frontend:** The interface is a **DAO Dashboard** with an AI twist. It would list proposals with their details, including any AI-provided summary or risk score. For instance: "Proposal #5: Upgrade Protocol – AI Analysis: *This proposal has low risk and potentially improves efficiency.* (Score: 8/10)". Voters can read the full text and the AI notes, then cast their vote using their connected wallet (the app will call the vote function, signing with their address and possibly including how many tokens they stake, etc.). If the AI itself is allowed to propose, perhaps surface "AI suggested" proposals distinctly in the UI. You could even have a chatbot UI element where users type a question and an AI (off-chain) responds based on the DAO's data, but that's more AI side. At least, ensure the front end displays the on-chain recorded AI advice. This project showcases a forward-looking concept: **AI assistance in decentralized governance** – with blockchain recording the AI's inputs for transparency and letting the community decide with that extra context.

Each of these ideas leverages Move smart contracts as the core, demonstrating handling of **ownership models** (shared vs. owned objects) [24], **capability-based access** for privileged actions [2], and complex data management with **vectors, tables, and custom resource types**. They also all include a front-end component for user interaction (wallet connections, data display, transaction submission) to fulfill a complete dApp implementation. By exploring any of these project ideas, developers can showcase a thorough understanding of Sui Move's features and deliver a functional prototype aligning with the hackathon tracks and technical requirements.

---

[1] [14] [17] [18] [21] 5 Surprising Ways to Use Smart Contracts | Avax.network
https://www.avax.network/about/blog/5-surprising-ways-to-use-smart-contracts

[2] [3] Lesson 3 - Objects and Capabilities | Scaf
https://scafsui.com/learn/lesson_3/

[4] [5] [6] [7] WTF is Quadratic Funding?
https://www.wtfisqf.com/

[8] [9] [10] [11] [12] [13] Everything You Need to Know About NFT Rentals - NFT Plazas
https://nftplazas.com/nft-rentals-everything/

[15] Blockchain for Supply Chain: Uses and Benefits
https://www.oracle.com/blockchain/what-is-blockchain/blockchain-for-supply-chain/

[16] The Rise of Tokenized Carbon Credits | Why Blockchain Is Transforming Carbon Markets
https://www.carbonmark.com/post/the-rise-of-tokenized-carbon-credits-why-blockchain-is-changing-everything

[19] [20] What are Soulbound Tokens (SBT)? | Coinbase
https://www.coinbase.com/learn/crypto-glossary/what-are-soulbound-tokens-sbt

[22] [23] What is NFT ticketing and how could it disrupt entertainment? | Coinbase
https://www.coinbase.com/learn/crypto-glossary/what-is-nft-ticketing-and-how-could-it-disrupt-entertainment

[24] Learn Move for Sui: 11 Key Concepts Explained Simply | by Filip Jerga | Eincode | Medium
https://medium.com/eincode/learn-move-for-sui-11-key-concepts-explained-simply-39299763959c