# Report: Decentralized Inheritance Protocol

Noah Klaholz, Vincent Schall, Max Mendes Carvalho

November 2025

## Contents

# 1 Introduction

## 1.1 Motivation

No one can escape death - but what happens to your crypto when you die? According to [4], it is estimated that around 3.7 million Bitcoin are lost and unrecoverable. One of the top reasons is death: crypto holders that passed away and failed to share access information with heirs will be responsible for inaccessible funds.
Traditional inheritance systems are flawed: they take very long, are expensive and more often than not lead to conflict between the heirs. We want to solve these problems by introducing a decentralized inheritance protocol.

## 1.2 The Decentralized Inheritance Protocol

The idea is as follows: anyone can create a will by deploying the inheritance protocol contract. After that, depositing coins, tokens and assets, as well as defining beneficiaries or heirs by adding their wallet addresses, is quick and easy with function calls to the contract. For each beneficiary, the owner can define a payout amount as a percentage of the total deposited assets.
Furthermore, deposited assets are invested using Aave[1]. This allows the balance to grow instead of laying dry.
The owner has to check in at least every 90 days to verify that he's still alive. As long as these check-ins occur, there will be no payout. When a check-in is missed, there is a 30-day grace period during which a check-in can still be made to reset the timer. If no check-in occurs within this grace period, the contract enters a verification phase where a trusted notary can upload death verification.

In case of death, trusted oracles (in most cases a notary) are used to verify the death via death certificates before initiating the payout.

# 2 Tool usage / tech stack

This project spans two domains: on-chain smart contract development and an off-chain web client for interaction.

## 2.1 Core Languages

**Solidity** (v0.8.28) is used for smart contract logic, leveraging built-in overflow checks and modern language features [22]. **TypeScript** is used across both the contract testing layer and the frontend for static typing and improved tooling [24].

## 2.2 Smart Contract Framework and Tooling

**Hardhat** serves as the primary development and testing framework [12]. It provides a local Ethereum network, deterministic deployments, stack traces, and plugin extensibility. Scripts like `scripts/deploy.js` and `scripts/auto-deploy.js` automate contract deployment, while `start.js` boots a local node and performs an initial deployment for the frontend to consume.

**Hardhat Ignition** is adopted for more declarative deployment pipelines [13]. Ignition modules (e.g., in `ignition/modules/`) describe deployment intent, helping reduce manual sequencing errors and making deployments reproducible.

[1] Aave — a decentralized lending protocol: supply crypto to earn interest via liquidity pools. `https://aave.com/docs/developers/liquidity-pool`

**OpenZeppelin Contracts** supplies audited base contracts (`Ownable`, `ReentrancyGuard`, `IERC20`) to reduce implementation risk and accelerate development [19]. Using well-established libraries mitigates common vulnerabilities and increases readability for reviewers.

## 2.3 Testing Stack

Unit and integration tests use **Mocha** as the test runner and **Chai** with the `chai-as-promised` pattern for expressive assertions [15, 6]. **Hardhat Network Helpers** assist with time skips to keep tests isolated and deterministic. The testing approach validates critical flows: beneficiary management, state machine transitions, death oracle integration, and payout distribution.

## 2.4 Blockchain Interaction Library

**Ethers.js** (v6) is used both in tests and the frontend for provider abstraction, contract bindings, and wallet interaction [11]. It offers a concise API, rich TypeScript definitions, and strong support for modern Ethereum features (signing, ENS, event queries).

## 2.5 Frontend Framework and UI Stack

The web client is built with **Next.js** (v15) [16] for file-based routing, server-side rendering (SSR), and asset optimization. **React 18** provides the component model and concurrent rendering features [21]. **Tailwind CSS** supplies utility-first styling for rapid UI iteration and consistent spacing/color scales [23]. Post-processing is handled by **PostCSS** and **Autoprefixer** to normalize styles across browsers [20, 3]. Icons come from **Lucide React** for a lightweight, customizable icon set [14].

## 2.6 Wallet and Web3 UX

**Web3Modal** simplifies multi-wallet connection flows on the frontend, abstracting provider selection and improving user onboarding [25]. This reduces friction versus building custom wallet connectors manually, while maintaining extensibility for future wallet providers.

## 2.7 Linting and Code Quality

**ESLint** with the Next.js configuration enforces consistent style and catches common mistakes in the React/TypeScript codebase [10]. Static analysis complements TypeScript's type checking by addressing stylistic and best-practice concerns (unused variables, unsafe React patterns). For Solidity, reliance on established OpenZeppelin components and compiler warnings keeps the contract surface maintainable; future work could add a dedicated static analysis pass (e.g., Slither) to further enhance assurance.

## 2.8 Runtime and Scripting

**Node.js** underpins Hardhat scripts, local tooling, and the custom automation scripts (`start.js`, `cleanup.js`) [18]. These scripts orchestrate development workflow: spinning up a local chain, deploying contracts, and cleaning artifacts to reset state between test sessions.

## 2.9 Design Rationale

The chosen stack prioritizes:

- **Auditability**: Using audited libraries (OpenZeppelin) and type-safe code (TypeScript) reduces risk.

- **Developer Velocity**: Hardhat + Next.js + Tailwind enable iterative development with fast feedback loops.
- **Maintainability**: Consistent tooling (Ethers.js across backend/frontend) minimizes integration complexity.
- **Extensibility**: Ignition and modular deployment scripts allow easy evolution (multi-asset support, new oracles).
- **User Experience**: Web3Modal and Tailwind produce a smoother interaction surface for non-technical beneficiaries.

### 2.10 Integration Flow

The typical local flow is:

1. Run `node start.js` to launch a Hardhat node and automatically deploy contracts (via Ignition or scripts).
2. The deployment script writes a `deployment-info.json` artifact consumed by the frontend for contract addresses and ABI metadata.
3. Frontend connects to the local network through Ethers.js and Web3Modal, pulling contract state (beneficiaries, balances, state machine phase).
4. Mocha/Chai test suite validates contract invariants in isolation; manual UI interactions can then be performed to cross-check behavior.
5. `cleanup.js` purges artifacts and resets caches when a fresh environment is needed.

## 3 Smart Contract architecture

### 3.1 General Design and Flow

The inheritance protocol is implemented in a single smart contract and composes well-known primitives from *OpenZeppelin* for access control and safety [19]. In particular, the contract inherits from `Ownable` to grant the will's creator administrative privileges, uses `ReentrancyGuard` to protect sensitive functions, and interacts with funds through the `IERC20` interface of the ERC-20 standard [5]. For external integration, the contract talks to a lending pool (Aave-compatible mock) to invest idle balances and to a death oracle for verification.

The constructor wires all dependencies (token, death oracle, notary, pool) and initializes the state machine that models the life cycle of a will. The state machine transitions among four phases:

```
1    enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }
```

Listing 1: Contract state machine

*ACTIVE* is the normal operating phase where the owner can manage beneficiaries and funds. If the owner misses a check-in for more than 90 days, the state moves to *WARNING*. After a 30-day grace period without check-in, the state advances to *VERIFICATION*. Once the death oracle confirms the owner's passing, the state becomes *DISTRIBUTION*, which triggers payout.

The `updateState()` function is public so that the notary, family members, or any third party can progress the state machine when the objective conditions are met. This creates an incentive-aligned mechanism: beneficiaries want the state to be up to date to receive their funds and a trusted notary can be instructed to call this function regularly. When the state reaches *DISTRIBUTION*, the contract immediately invokes `distributePayout()` and emits a `StateChanged` event.

## 3.2 Roles and Access Control

We use `Ownable` for a single privileged owner (the testator), and a dedicated *notary* address for external verification tasks [19]. Access is enforced by modifiers:

- `onlyOwner`: administrative actions (check-in, adding/removing beneficiaries, deposits/withdrawals) are restricted to the owner.
- `onlyNotary`: only the notary can upload death verification proofs.
- `onlyPreDistribution`: prevents fund mutations once the system is in the distribution phase.
- `onlyDistribution`: guards payout functions so they are callable only in the final phase.
- `onlyActiveWarning`: Prevents Administrative changes like adding beneficiaries from being executed unless in the ACTIVE or WARNING phase.

Functions that transfer value also use the `nonReentrant` modifier from `ReentrancyGuard` to mitigate reentrancy (SWC-107) [1].

## 3.3 Beneficiaries and Payout Logic

Beneficiaries are kept in a fixed-size array of at most ten entries to keep gas costs predictable and iteration bounded. Each entry stores a payout address and a percentage amount. The contract enforces that:

- No duplicate beneficiary addresses exist.
- The total determined payout never exceeds 100%.
- Add/remove operations are only allowed before distribution and require a fresh owner check-in.
- All administrative changes can only be made by the contract's owner.

On distribution, the contract retrieves the pool balance from Aave, computes each beneficiary's share by percentage, and transfers tokens accordingly by iterating through the list of beneficiaries. If the sum of percentages is below 100%, the residual is sent to a donation address to prevent funds from being stranded forever.

## 3.4 Funds Management and Aave Integration

The protocol accepts an ERC-20 token (MockUSDC in our deployment) via `deposit`. The owner first approves the contract to spend tokens, then the contract supplies tokens into an Aave-compatible pool (MockAavePool in our deployment) to accrue yield [5, 2]. Withdrawals reverse the flow: tokens are pulled from the pool and transferred back to the owner. Critical operations are protected with `nonReentrant` and disallowed after distribution.

## 3.5 Death Verification and Oracles

Death verification is abstracted behind the `IDeathOracle` interface. The notary calls `uploadDeathVerification` with a boolean and proof bytes; the oracle persists the attestation and can still be called upon by beneficiaries to verify the death. The state machine polls the oracle by calling `isDeceased(owner())` and, if true, transitions to *DISTRIBUTION*. In our test setup we use a mock oracle to enable deterministic unit tests. For a production deployment, this component could be backed by a notarized registry, a government API gateway, or decentralized oracle networks. Additionally,

`updateState()` could be automated using off-chain keepers (e.g., Chainlink Automation) to guarantee timely transitions without relying on manual calls [7]. However, since beneficiaries already have an incentive to update the state regularly, the decision was made to avoid the extra cost for off-chain automation.

### 3.6 Security Considerations

Our design follows standard Solidity best practices [22, 19]:

- Reentrancy protection on functions that transfer tokens [1].
- Access control via explicit roles and clear phase guards (`onlyPreDistribution`, `onlyDistribution`, ...).
- Use of `immutable` and `constant` for critical configuration to reduce runtime risk and gas cost.
- Bounded iteration over at most ten beneficiaries to avoid unbounded gas usage.
- Overflow/underflow safety from the Solidity 0.8.x checked arithmetic [22].

Threats and mitigations:

- **Oracle risk**: a compromised notary/oracle could wrongfully trigger distribution. This is mitigated organizationally (trusted notaries) and could be strengthened with multi-sig attestations or time delays.
- **Griefing/liveness**: anyone can call `updateState()`, but transitions are conditional and idempotent; no value is at risk.
- **External calls**: interactions with the pool and token are performed after state updates and protected by `nonReentrant`. The donation transfer happens last to simplify reasoning.

### 3.7 Gas and Scalability

The fixed-size array avoids storage resizes and bounds loops to a maximum of ten iterations. Getter functions such as `getActiveBeneficiaries()` build a compact memory array for off-chain consumers, trading a small amount of gas for simpler client logic. State checks in `updateState()` are in constant time. While the design targets personal wills (low on-chain scale), it remains economical for typical usage.

### 3.8 Known Limitations and Future Work

- **Single-asset support**: the current implementation handles one ERC-20 token instance. Extending to multiple assets would require per-asset accounting and distribution.
- **Maximum of ten beneficiaries**: chosen for simplicity and predictable gas; a dynamic structure[8] could be introduced. However this would require careful consideration of gas efficiency and time complexity, while larger will structures could also be realized by having multiple wills and distributing funds accordingly.
- **Oracle centralization**: production setups should consider decentralized attestations or multi-party notaries.
- **Automation**: integrating keepers would remove the need for manual `updateState()` calls [7].
- **UX improvements**: support for EIP-2612 `permit` and richer events to make indexing easier [9].

# 4 Design Decisions and Scope Refinement

The initial project proposal outlined an ambitious feature set including vesting schedules, age-gated releases, multi-asset support, and automated state transitions. During development, we refined the scope based on technical feasibility, cost-benefit analysis, and project timeline constraints. This section documents the key decisions and trade-offs.

## 4.1 Removal of Vesting Schedules

The original design included sophisticated vesting mechanisms to distribute inheritances gradually over time (e.g., 10% per year for 10 years) to prevent sudden wealth syndrome and protect young beneficiaries from mismanaging lump sums. However, we ultimately removed vesting from the final implementation.

**Technical Challenge: Time-Based Execution.** Smart contracts on Ethereum are fundamentally passive: they cannot self-execute based on time. A vesting schedule requires periodic unlocking (e.g., monthly or yearly), but no on-chain primitive can trigger a function call at a future timestamp without external intervention. This creates three potential solutions, each with significant drawbacks:

1. **Off-chain automation (Chainlink Keepers):** Use a keeper network to call `releaseVestedFunds()` at scheduled intervals [7]. This introduces recurring costs (keeper fees), centralizes liveness assumptions (keeper availability), and adds complexity to the deployment and maintenance workflow.
2. **Pull-based vesting:** Beneficiaries call `withdraw()` whenever they want funds, and the contract calculates how much has vested since the last withdrawal. While this eliminates automation costs, it creates poor UX (beneficiaries must remember to claim) and removes the protective aspect of enforced gradual distribution—beneficiaries can simply wait and withdraw the full amount later.
3. **Incentivized third-party calls:** Reward external actors for calling unlock functions by giving them a small fee. This adds game-theoretic complexity, potential griefing vectors (claiming fees without benefiting beneficiaries), and still requires someone to monitor and trigger transactions.

**Scope Prioritization.** Given a fixed development timeline, we chose to focus on the core inheritance mechanism (state machine, death verification, percentage-based distribution) rather than vesting. The current design delivers immediate value for the primary use case—preventing total loss of crypto assets upon death—while vesting primarily addresses a secondary concern (beneficiary financial responsibility). Future iterations could introduce optional vesting as a modular extension for users who specifically need it.

**Alternative Mitigations.** Users concerned about lump-sum distributions can approximate vesting by creating multiple inheritance contracts with staggered check-in periods or by allocating funds to custodial services that provide off-chain vesting. Additionally, the protocol's percentage-based allocation allows splitting inheritances among trustees or financial advisors who can manage gradual distributions off-chain.

## 4.2 Manual State Transitions Over Automated Keepers

The contract's state machine (`ACTIVE` → `WARNING` → `VERIFICATION` → `DISTRIBUTION`) relies on anyone calling `updateState()` to check conditions and advance phases. We considered integrating Chainlink Automation to guarantee timely transitions but decided against it.

**Cost vs. Benefit.** Chainlink Automation charges per execution (currently $0.10–$1.00 depending on gas prices and network) [7]. For a contract that might remain in `ACTIVE` state for years or decades, regular keeper pings would accumulate significant costs with no user benefit. Even monthly checks over 10 years would cost $12–$120, and the contract would need to be pre-funded to cover these fees.

**Incentive Alignment.** Beneficiaries have a direct financial incentive to call `updateState()` after the owner's death: it is the only way to trigger distribution and receive their inheritance. Similarly, the notary (often a trusted family member or legal professional) can be instructed to monitor the contract. Since `updateState()` is public and gas costs are negligible compared to inheritance values, relying on interested parties is economically rational and removes recurring operational expenses.

**Liveness Assumption.** This design assumes at least one beneficiary or the notary will act when needed. For high-value estates, this is a reasonable assumption. Users requiring absolute automation (e.g., no living beneficiaries, all recipients are minors) could separately fund a keeper service, but we judged this edge case insufficient to justify baking automation costs into every deployment.

### 4.3 Aave Integration for Yield Generation

Funds deposited into the inheritance contract are supplied to an Aave-compatible lending pool to earn yield while awaiting distribution [2]. This decision stems from several factors:

**Capital Efficiency.** Inheritance timelines are inherently long — assets might sit in the contract for years or decades. Leaving funds idle represents a significant opportunity cost. By supplying assets to Aave, the contract earns interest (typically 2–6% APY on stablecoins like USDC), growing the inheritance value over time. For a $100,000 deposit earning 4% over 10 years, this adds $48.024 to the final distribution.

**Security and Maturity.** Aave V3 is one of the most battle-tested DeFi protocols, with over $10 billion in total value locked and extensive security audits [2]. Its lending pools are non-custodial (the contract retains withdrawal rights) and have a strong track record. Using a proven primitive reduces implementation risk compared to building a custom yield strategy.

**Liquidity and Composability.** Aave supplies remain liquid: the contract can withdraw funds at any time for owner withdrawals or final distribution. This flexibility is critical—vesting or locked staking would prevent the owner from accessing their own funds. Aave's `aToken` model (interest-bearing tokens) integrates cleanly with ERC-20 workflows.

**Stablecoin Focus.** The current implementation uses USDC (or a mock equivalent) to avoid volatility risk. Yield on stablecoins is lower than on volatile assets but ensures predictable distributions. Extending to ETH or other tokens would require additional risk disclosures and potentially dynamic allocation strategies, which we deferred to future work.

### 4.4 Single Notary vs. Multi-Signature Verification

The original proposal mentioned 2-of-3 trusted contacts for death verification. The implemented design uses a single notary address. This simplification reduces contract complexity (no threshold signature logic) and reflects a pragmatic trust model: users select one highly trusted entity (e.g., a lawyer, family member, or professional executor service). The notary's role is narrowly

scoped to uploading death verification; they cannot withdraw funds or change beneficiaries, limiting abuse potential. Future versions could introduce multi-sig verification via OpenZeppelin's `AccessControl` or a separate oracle aggregator for users requiring decentralized attestation.

### 4.5 Scope Reductions and Future Work

Several features from the initial proposal were deferred:

### 4.6 Summary of Trade-offs

The implemented protocol represents a pragmatic subset of the initial vision, optimized for:

- **Core value delivery:** Solving the primary problem (crypto inheritance loss) without feature bloat.
- **Gas efficiency:** Bounded operations, minimal storage, no recurring costs.
- **Security surface:** Fewer features mean fewer attack vectors and simpler auditing.
- **User autonomy:** No mandatory off-chain dependencies or subscription fees.

Future work can layer additional features (vesting, multi-asset, decentralized oracles) as optional modules or through a factory pattern that deploys specialized contract variants based on user needs.

# Appendices

```
1
2       // SPDX-License-Identifier: MIT
3       pragma solidity ^0.8.28;
4
5       import "@openzeppelin/contracts/access/Ownable.sol";
6       import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
7       import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
8       import {IDeathOracle} from "./mocks/IDeathOracle.sol";
9       import {MockAavePool} from "./mocks/MockAavePool.sol";
10
11      contract InheritanceProtocol is Ownable, ReentrancyGuard {
12
13          IERC20 public immutable usdc;
14          IDeathOracle public immutable deathOracle;
15          address private notaryAddress;
16          MockAavePool public aavePool;
17
18          // address for donations (underdetermined payout)
19          address private ourAddress;
20
21          /**
22           * Stores address and payout percentage amount (0-100) of
                a beneficiary.
23           */
24          struct Beneficiary {
25              address payoutAddress;
26              uint256 amount;
27          }
```

```
28
29          Beneficiary[10] private _beneficiaries;
30
31          State private _currentState;
32
33          uint256 private _lastCheckIn;
34          bool private _called = false;
35
36          uint256 private constant NOT_FOUND = type(uint256).max;
37          uint256 private constant MAX_BENEFICIARIES = 10;
38          uint256 private constant MAX_PERCENTAGE = 100;
39          uint256 private constant CHECK_IN_PERIOD = 90 * 1 days;
40          uint256 private constant GRACE_PERIOD = 30 * 1 days;
41
42          event BeneficiaryAdded(address indexed payoutAddress,
                uint256 amount, uint256 index);
43          event BeneficiaryRemoved(address indexed payoutAddress,
                uint256 index);
44          event Deposited(uint256 amount);
45          event Withdrawn(uint256 amount);
46          event CheckedIn(uint256 timestamp);
47          event StateChanged(uint256 timestamp, State from, State
                to);
48          event PayoutMade(uint256 amount, address payoutAddress);
49          event TestEvent(string s);
50          event TestEventNum(uint s);
51
52          /**
53           * Initializes a new InheritanceProtocol.
54           * @param _usdcAddress address of the currency used
                 (non-zero).
55           */
56          constructor(address _usdcAddress, address
                _deathOracleAddress, address _notaryAddress, address
                _aavePoolAddress) Ownable(msg.sender) {
57              require(_usdcAddress != address(0), "USDC address
                    zero");
58              require(_deathOracleAddress != address(0), "Death
                    Oracle address zero");
59              ourAddress =
                    0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266;
60              usdc = IERC20(_usdcAddress);
61              deathOracle = IDeathOracle(_deathOracleAddress);
62              notaryAddress = _notaryAddress;
63              aavePool = MockAavePool(_aavePoolAddress);
64              _currentState = State.ACTIVE;
65              _lastCheckIn = block.timestamp;
66          }
67
68          /// ---------- MODIFIERS ----------
69
70          /**
71           * This modifier requires the function call to be made
                 before distribution.
72           */
73          modifier onlyPreDistribution() {
74              require(_currentState < State.DISTRIBUTION, "Cannot
                    modify funds post-distribution");
```

```
        _;
    }

    /**
     * This modifier requires the function call to be made in the ACTIVE or WARNING phase
     */
    modifier onlyActiveWarning() {
        require(_currentState < State.VERIFICATION, "Cannot make administrative changes without Owner check-In");
        _;
    }

    /**
     * This modifier requires the function call to be made in the DISTRIBUTION phase
     */
    modifier onlyDistribution() {
        require(_currentState == State.DISTRIBUTION, "Can only make payouts in distribution phase");
        _;
    }

    /**
     * This modifier requires the function call to be made by the notary
     */
    modifier onlyNotary() {
        require(msg.sender == notaryAddress, "Only notary can call this function");
        _;
    }

    /// ---------- STATE MACHINE & CHECK-INS ----------

    /**
     * Defines the state of the contract.
     *   - Active: mutable state, owner check-ins required.
     *   - Warning: Missed check-in, notification sent at 90 days,
     *     verification phase starts at 120 days.
     *   - Verification: submission of death certificate (30 days).
     *   - Distribution: distribute assets based on defined conditions.
     */
    enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }

    /**
     * Updates the State in the State-Machine
     * Should always be possible and accessible by anyone
     * @return currentState after execution
     */
    function updateState() public returns (State) {
        uint256 elapsed = uint256(block.timestamp) - _lastCheckIn;
        State oldState = _currentState;
```

```
122
123                 // --- Phase transitions in logical order ---
124
125                 // If in ACTIVE and check-in expired      WARNING
126                 if (_currentState == State.ACTIVE && elapsed >
                        CHECK_IN_PERIOD) {
127                     _currentState = State.WARNING;
128                 }
129
130                 // If in WARNING and grace period expired
                        VERIFICATION
131                 if (_currentState == State.WARNING && elapsed >
                        CHECK_IN_PERIOD + GRACE_PERIOD) {
132                     _currentState = State.VERIFICATION;
133                 }
134
135                 // If in VERIFICATION and death confirmed
                        DISTRIBUTION
136                 if (_currentState == State.VERIFICATION &&
                        deathOracle.isDeceased(owner())) {
137                     _currentState = State.DISTRIBUTION;
138                 }
139
140                 emit StateChanged(block.timestamp, oldState,
                        _currentState);
141
142                 // Trigger payout if we reached DISTRIBUTION
143                 if (_currentState == State.DISTRIBUTION) {
144                     distributePayout();
145                 }
146
147                 return _currentState;
148             }
149
150             /**
151              * Changes the state of the contract to a given state.
152              * @param to the state to change to.
153              */
154             function changeState (State to) public {
155                 require(to != _currentState, "Already in requested
                        state");
156                 emit StateChanged(block.timestamp, _currentState, to);
157                 _currentState = to;
158             }
159
160             /**
161              * The owner checks in to verify that he's alive.
162              * Should be possible in active and warning state.
163              */
164             function checkIn() public onlyOwner {
165                 require(_currentState == State.ACTIVE || _currentState
                        == State.WARNING, "Need to be in active or warning
                        state");
166                 emit CheckedIn(block.timestamp);
167                 _lastCheckIn = block.timestamp;
168             }
169
170             /// ---------- BENEFICIARY HANDLING ----------
```

```
171
172             /**
173              * Finds the index of a beneficiary in the beneficiaries
                     list.
174              * @param _address the address whose index to find.
175              * @return the index if the address is in the list,
                     'NOT_FOUND' otherwise.
176              */
177             function findBeneficiaryIndex(address _address) public
                    view returns (uint256) {
178                 if (_address == address(0)) {
179                     return NOT_FOUND;
180                 }
181                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
182                     if (_beneficiaries[i].payoutAddress == _address) {
183                         return i;
184                     }
185                 }
186                 return NOT_FOUND;
187             }
188
189             /**
190              * Removes a beneficiary with a given address.
191              * Only the owner can perform this action.
192              * @param _address the address to remove.
193              * Fails if the provided address is zero OR not in the
                     list of beneficiaries.
194              * @return true if the deletion was successful, false
                     otherwise.
195              */
196             function removeBeneficiary(address _address) public
                    onlyOwner onlyActiveWarning returns (bool) {
197                 checkIn();
198                 uint256 index = findBeneficiaryIndex(_address);
199                 if (index == NOT_FOUND) {
200                     return false;
201                 }
202                 delete _beneficiaries[index];
203                 emit BeneficiaryRemoved(_address, index);
204                 return true;
205             }
206
207             /**
208              * Adds a beneficiary to the list.
209              * Only the owner can perform this action.
210              * Requirements:
211              *  - List not full
212              *  - Payout after adding <= 100
213              * @param _address the address to add to the list.
214              * @param _amount the payout amount related to this
                     address.
215              * @return true if the addition was successful, false
                     otherwise.
216              */
217             function addBeneficiary(address _address, uint256 _amount)
                    public onlyOwner onlyActiveWarning returns (bool) {
218                 checkIn();
219                 require(_address != address(0), "Invalid address");
```

```
220                 require(_amount > 0 && _amount <= MAX_PERCENTAGE,
                        "Invalid amount");
221
222                 // Check for duplicate
223                 if (findBeneficiaryIndex(_address) != NOT_FOUND) {
224                     return false;
225                 }
226
227                 uint256 currentSum = getDeterminedPayoutPercentage();
228                 if (currentSum + _amount > MAX_PERCENTAGE) {
229                     // it should not be possible to payout more than
                            100%
230                     return false;
231                 }
232
233                 // Find empty slot
234                 uint256 emptyIndex = NOT_FOUND;
235                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
236                     if (_beneficiaries[i].payoutAddress == address(0))
                            {
237                         emptyIndex = i;
238                         break;
239                     }
240                 }
241
242                 if (emptyIndex == NOT_FOUND) {
243                     return false; // Max beneficiaries reached
244                 }
245
246                 _beneficiaries[emptyIndex] = Beneficiary({
                        payoutAddress: _address, amount: _amount });
247                 emit BeneficiaryAdded(_address, _amount, emptyIndex);
248                 return true;
249             }
250
251             /// ---------- BALANCE HANDLING ----------
252
253             /**
254              * Deposits a given amount of USDC.
255              * @param _amount the amount to deposit.
256              */
257             function deposit(uint256 _amount) external onlyOwner
                    nonReentrant onlyPreDistribution {
258                 checkIn();
259                 require(_amount > 0, "Amount has to be greater than
                        zero.");
260
261                 usdc.transferFrom(msg.sender, address(this), _amount);
262
263                 usdc.approve(address(aavePool), _amount);
264
265                 aavePool.supply(address(usdc), _amount, address(this));
266
267                 emit Deposited(_amount);
268             }
269
270             /**
271              * Withdraws a given amount of USDC.
```

```
272             * @param _amount the amount to withdraw.
273             */
274            function withdraw(uint256 _amount) external onlyOwner
                   nonReentrant onlyPreDistribution {
275                checkIn();
276                require(_amount > 0, "Amount has to be greater than
                       zero.");
277                require(getBalance() >= _amount, "Insufficient
                       balance");
278
279                aavePool.withdraw(address(usdc), _amount,
                       address(this));
280
281                usdc.transfer(msg.sender, _amount);
282                emit Withdrawn(_amount);
283            }
284
285            /// ---------- DEATH CERTIFICATION ----------
286
287            /**
288             * Upload the death verification to the chain
289             * Only callable by the notary
290             */
291            function uploadDeathVerification(bool _deceased, bytes
                   calldata _proof) external onlyNotary{
292                deathOracle.setDeathStatus(owner(), _deceased, _proof);
293            }
294
295            /**
296             * Checks if the owner died by calling death certificate
                    oracle.
297             * @return true if the owner died, else otherwise.
298             */
299            function checkIfOwnerDied() public view returns (bool) {
300                return deathOracle.isDeceased(owner());
301            }
302
303            /// ---------- DISTRIBUTION METHODS ----------
304
305            /**
306             * Distributes the payout based on definitions given by
                    owner.
307             * Is only called in the updateState() Function, after
                    death verification
308             */
309            function distributePayout() public {
310                require(!_called, "Payout can only be called once.");
311                _called = true;
312                bool donation = !isPayoutFullyDetermined();
313                uint256 count = getActiveCount();
314                Beneficiary[] memory activeBeneficiaries =
                       getActiveBeneficiaries();
315                uint256 balanceRemainingInPool = getBalance();
316                uint256 originalBalance =
                       aavePool.withdraw(address(usdc),
                       balanceRemainingInPool, address(this));
317                for (uint256 i=0; i<count; i++) {
```

```
                    Beneficiary memory beneficiary =
                        activeBeneficiaries[i];
                    uint256 amount = beneficiary.amount;
                    address payoutAddress = beneficiary.payoutAddress;

                    uint actualAmount = (originalBalance * amount) /
                        MAX_PERCENTAGE;

                    usdc.transfer( payoutAddress, actualAmount);
                    emit PayoutMade(actualAmount, payoutAddress);
                }
                if (donation) {
                    // If the payout is not fully determined, the rest
                        of the balance will be sent to the developer
                        team.
                    // For now this is hardcoded as the first address
                        generated by hardhat when running a local node.
                    uint256 donatedAmount =
                        aavePool.withdraw(address(usdc), getBalance(),
                        address(this));
                    usdc.transfer(ourAddress, donatedAmount);
                    emit PayoutMade(donatedAmount, ourAddress);
                }
            }

            /// ---------- VIEW METHODS ----------

            /**
             * Checks if the currently defined payout is fully
                 determined, meaning
             * 100% of the balance is being spent.
             * @return true if the full balance will be spent, false
                 otherwise.
             */
            function isPayoutFullyDetermined() public view returns
                (bool) {
                uint256 sum = getDeterminedPayoutPercentage();
                return sum == MAX_PERCENTAGE;
            }

            /**
             * Calculates the percentage amount of currently
                 determined payout.
             * @return a number between 0 and 100, equivalent to the
                 combined relative payout.
             */
            function getDeterminedPayoutPercentage() public view
                returns (uint256) {
                uint256 sum;
                for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
                    if (_beneficiaries[i].payoutAddress != address(0))
                        {
                        sum += _beneficiaries[i].amount;
                    }
                }
                return sum;
            }

```

```
362             /**
363              * Gets the current balance.
364              * @return the balance of the combined deposited funds.
365              */
366             function getBalance() public view returns (uint256) {
367                 return aavePool.getBalance(address(this));
368             }
369
370             /**
371              * Getter for the beneficiaries list.
372              * @return the list of 10 beneficiaries (might contain
                     empty slots).
373              */
374             function getBeneficiaries() public view returns
                    (Beneficiary[10] memory) {
375                 return _beneficiaries;
376             }
377
378             /**
379              * Counts the number of active beneficiaries.
380              * @return the number of active beneficiaries.
381              */
382             function getActiveCount() public view returns (uint256) {
383                 uint256 count;
384                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
385                     if (_beneficiaries[i].payoutAddress != address(0))
                            {
386                         count++;
387                     }
388                 }
389                 return count;
390             }
391
392             /**
393              * Gets only the active beneficiaries.
394              * @return an array of beneficiaries.
395              */
396             function getActiveBeneficiaries() public view returns
                    (Beneficiary[] memory) {
397                 uint256 activeCount = getActiveCount();
398                 Beneficiary[] memory active = new
                        Beneficiary[](activeCount);
399                 uint256 count = 0;
400                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
401                     if (_beneficiaries[i].payoutAddress != address(0))
                            {
402                         active[count] = _beneficiaries[i];
403                         count++;
404                     }
405                 }
406                 return active;
407             }
408
409             /**
410              * Gets the current state of the contract.
411              * @return the current state.
412              */
413             function getState() public view returns (State) {
```

```
414                 return _currentState;
415             }
416 
417             /**
418              * Gets the last check-in time.
419              * @return the last check-in time.
420              */
421             function getLastCheckIn() public view returns (uint256) {
422                 return _lastCheckIn;
423             }
424 
425         }
```

Listing 2: smart contract

# A References

The entire project can be found at the [17] at https://github.com/vincentschall/decentralized_inheritance_proto

# References


[1] *A Broad Overview of Reentrancy Attacks in Solidity Contracts*. Accessed 2025-11-16. 2025. URL: https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-broad-overview-of-reentrancy-attacks-in-solidity-contracts.

[2] *Aave V3: Pool Contract and Supplying Liquidity*. Accessed 2025-11-16. 2025. URL: https://docs.aave.com/developers/core-contracts/pool.

[3] *Autoprefixer Documentation*. Accessed 2025-11-16. 2025. URL: https://github.com/postcss/autoprefixer.

[4] Bidget. *How Many Bitcoin Have Been Lost?* Accessed 2025-11-06. 2025. URL: https://www.bitget.com/wiki/how-many-bitcoin-have-been-lost.

[5] V. Buterin and F. Vogelsteller. *ERC-20: Token Standard*. Accessed 2025-11-16. 2015. URL: https://eips.ethereum.org/EIPS/eip-20.

[6] *Chai Assertion Library*. Accessed 2025-11-16. 2025. URL: https://www.chaijs.com/.

[7] *Chainlink Automation Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.chain.link/chainlink-automation/introduction.

[8] *Dynamic Arrays and its Operations in Solidity*. Accessed 2025-11-16. 2025. URL: https://www.geeksforgeeks.org/solidity/dynamic-arrays-and-its-operations-in-solidity/.

[9] *EIP-2612: Permit — 712-signed approvals*. Accessed 2025-11-16. 2020. URL: https://eips.ethereum.org/EIPS/eip-2612.

[10] *ESLint Documentation*. Accessed 2025-11-16. 2025. URL: https://eslint.org/docs/latest/.

[11] *Ethers.js v6 Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.ethers.org/v6/.

[12] *Hardhat Documentation*. Accessed 2025-11-16. 2025. URL: https://hardhat.org/docs.

[13] *Hardhat Ignition Deployment Framework*. Accessed 2025-11-16. 2025. URL: https://hardhat.org/ignition.

[14] *Lucide React Icons*. Accessed 2025-11-16. 2025. URL: https://lucide.dev/guide/packages/lucide-react.

[15] *Mocha Test Framework Documentation*. Accessed 2025-11-16. 2025. URL: https://mochajs.org/.

[16] *Next.js 15 Documentation*. Accessed 2025-11-16. 2025. URL: https://nextjs.org/docs.

[17] Vincent Schall Noah Klaholz and Max Mendes Carvalho. *Decentralized Inheritance Protocol*. Repository. https://github.com/vincentschall/decentralized$_{i}nheritance_{p}rotocol$. 2025.

[18] *Node.js Documentation v20+*. Accessed 2025-11-16. 2025. URL: https://nodejs.org/en/docs.

[19] *OpenZeppelin Contracts Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.openzeppelin.com/contracts/5.x/.

[20] *PostCSS Documentation*. Accessed 2025-11-16. 2025. URL: https://postcss.org/.

[21] *React 18 Documentation*. Accessed 2025-11-16. 2025. URL: https://react.dev/.

[22] *Solidity Documentation v0.8.28*. Accessed 2025-11-16. 2025. URL: https://docs.soliditylang.org/en/v0.8.28/.

[23] *Tailwind CSS Documentation*. Accessed 2025-11-16. 2025. URL: https://tailwindcss.com/docs.

[24] *TypeScript Handbook*. Accessed 2025-11-16. 2025. URL: https://www.typescriptlang.org/docs/handbook/intro.html.

[25] *Web3Modal Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.walletconnect.com/web3modal.