PeerHire
A Decentralized Work Protocol for Trusted Collaboration and Enforcement

White Paper

July 2025

Sumit Nirmal
Co-Founder & CEO

Chanchal Kuntal
Co-Founder & COO

PeerHire is a decentralized protocol for project-based remote collaboration. It enforces trust through smart contract-based escrow, synthetic payment logic, and a decentralized jury protocol for fair dispute resolution.

Abstract

PeerHire is a decentralized protocol designed to solve the long-standing problems of trust, fairness, and enforcement in remote and freelance work. Existing marketplaces such as Upwork and Fiverr maintain opaque, centralized control over user identities, payment flows, and dispute processes that often skew outcomes in favor of higher volume users or established profiles. This setup disproportionately impacts students and early-career professionals, who lack institutional credentials and bargaining power, yet remain the most eager contributors to the digital labor economy.

This paper presents PeerHire as a modular, smart contract-powered protocol built on Polygon. At its foundation lies a programmable framework for creating on-chain agreements, milestone-locked escrow, and decentralized conflict resolution. Users never touch wallets, tokens, or gas; instead, fiat payments (via Razorpay or similar gateways) are mapped to synthetic pUSDC, a protocol-native unit of account pegged 1:1 to USD. This synthetic token is used for all on-chain value transfers while avoiding regulatory and UX complexity. ERC-4337 account abstraction handles wallet creation, gasless interactions, and social login, enabling invisible, custodial-lite access for non-crypto users.

At the heart of enforcement is pJustice: a decentralized dispute resolution protocol where verified jurors are selected based on skill-tags and historical accuracy. Disputes are adjudicated via on-chain voting, backed by peer staking, AI mediation, and quadratic weighting to ensure both fairness and scalability. Verdicts directly unlock or redirect escrowed funds, no platform admin or middle layer required. All reviews, ratings, and outcomes are recorded on-chain to form a transparent and immutable reputation ledger.

Beyond its initial focus on students and entry-level talent, PeerHire is designed as a composable layer for broader protocol integrations. DAOs can plug into the system to outsource work with built-in dispute mechanisms. Projects can spin up bounties or grants using native smart contract flows. The protocol is flexible enough to accommodate external token incentives, cross-chain extensions, and DAO-to-DAO labor coordination, forming the basis of a decentralized HR infrastructure for the open internet.

Our long-term ambition isn't to compete with freelance platforms, it's to render them obsolete. PeerHire doesn't just shift gig work onto a blockchain. It replaces platform trust with programmable enforcement. It replaces biased moderators with skill-aware juries. And it makes every contract verifiable, every transaction accountable, and every dispute resolvable without centralized permission. This paper lays the foundation for that future where coordination is trustless, global, and student-first by design.

Contents

1. Introduction

The global freelance economy has grown rapidly, with millions of people working across borders, platforms, and time zones. But beneath the surface, the system is fragile. Platforms hold too much power, workers have too little leverage, and trust is often enforced by reputation scores that are easy to game or impossible to earn if you're new.

Freelancers routinely face problems like delayed payments, scope creep, fake job postings, or disputes that get "resolved" with canned replies and opaque rulings. The stakes are even higher for students and early-career professionals, the very people trying to build portfolios, credibility, and income. They're often asked to prove themselves without institutional credentials or verified histories, and they usually can't afford legal recourse when things go wrong.

The problem isn't just bad actors. It's that the rules of trust and accountability are centralized, inconsistent, and largely invisible.

PeerHire is an attempt to fix that. It's not a platform, it's a protocol. One that uses smart contracts to define the rules upfront, lock payments in escrow, and settle conflicts through jurors selected for their skill and past reputation. No moderators, no middlemen. Just code, community, and accountability.

We start with students and early professionals because that's where the pain is sharpest and where the need for credibility is greatest. But the protocol is designed to scale: to pseudonymous builders, DAO contributors, and remote teams who want trust baked into the stack, not bolted on after the fact.

2. Related Work

Most freelance marketplaces including Upwork and Fiverr rely on centralized moderation, platform-controlled escrow, and opaque dispute processes. They provide structure, but that structure comes with steep platform fees, biased support decisions, and no recourse when things go wrong. Reputation is siloed inside the platform, and even a minor dispute can ruin a freelancer's visibility or account. The system is efficient for the platform, not the user.

There have been decentralized attempts to fix parts of this. Kleros introduced on-chain arbitration, using game theory and token incentives to align jurors around fair dispute resolution. Aragon Court explored similar ideas, emphasizing community governance and transparency in rulings. Both are valuable precedents but they weren't designed with the specific needs of freelancers, students, or fiat-first users in mind. Their integration paths into real workflows remain limited.

On the wallet side, ERC-4337 (account abstraction) has opened the door to crypto-native systems that don't feel like crypto. Users can sign in with social accounts, use session keys, and never touch seed phrases. PeerHire builds on this to create smart contract wallets that are invisible to the user because most freelancers shouldn't need to know what a wallet is to use trustless systems.

Other relevant work includes Soulbound Tokens (SBTs), Proof of Humanity, and other identity or reputation protocols that tie on-chain data to individual history or credentials. These show promise but haven't yet seen mass adoption, mostly because trust systems alone don't enforce fairness. PeerHire treats reputation as just one part of the trust stack, not the whole story.

Finally, Web2 trust mechanisms, reviews, KYC, community flags work until they don't. They're prone to manipulation, asymmetry, and moderator fatigue. When enforcement relies on a closed platform, it only works as long as you trust the platform. PeerHire flips that model: it's trust without gatekeepers, and enforcement without permission.

3. System Overview

PeerHire is a modular, trust-minimized protocol for managing freelance work agreements, payments, and disputes on-chain with the simplicity of Web2 and the enforceability of smart contracts. The core idea is that trust should be programmable, not assumed. Instead of relying on a centralized platform to handle identity, moderation, and payments, PeerHire delegates these responsibilities to smart contracts, DAO-based jurors, and transparent logic.

The workflow begins when a client creates a project and selects a freelancer. The terms including milestones, deadlines, and budget are written into a smart contract (‘ProjectAgreementContract‘). The client funds the project in local fiat currency through a supported payment gateway (e.g., Razorpay, Stripe). The backend logs this transaction and mints a synthetic stablecoin (pUSDC) equivalent to the received fiat. The backend logs this fiat deposit and issues a synthetic on-chain token (‘pUSDC‘) 1:1 to the value received.

This ‘pUSDC‘ is stored in the client’s smart wallet, which is abstracted using ERC-4337 (no private keys or browser extensions required). As milestones are completed and approved, funds are released from the ‘EscrowContract‘ directly to the freelancer. If there’s disagreement, either party can trigger a dispute, which activates the ‘pJusticeProtocol‘.

Disputes are resolved by a panel of jurors selected based on skill-tag alignment and reputation. Jurors stake pUSDC, vote on evidence submitted off-chain (hashed on-chain), and are rewarded or penalized based on alignment with the final outcome. An optional AI assistant helps summarize disputes but doesn’t override human voting. Final judgments impact not only payments but also on-chain reputation.

This architecture is designed to support:

- Fiat-to-smart contract conversion via synthetic pUSDC
- Walletless access and login abstraction
- Modular contracts for every step of work enforcement
- Decentralized, skill-based dispute resolution (pJustice)
- Soulbound, tamper-resistant reputation scores

3.1. Actors

- **Client** — initiates and funds the project using fiat. Clients interact with project creation and milestone approval logic, and can raise disputes if delivery is unsatisfactory.

- **Freelancer** — fulfills the project requirements. Freelancers undergo optional identity verification via the `IdentityContract` and build a public, skill-specific reputation over time.

- **Juror** — a trusted PeerHire user selected from a pool based on skill tags and past accuracy. Jurors stake `pUSDC` to vote on disputes and are subject to slashing for malicious or irrelevant voting.

- **Admin (off-chain backend)** — handles fiat payment logs, mints/burns synthetic `pUSDC`, and triggers wallet deployment. Does not hold funds or override on-chain logic.

- **PJDAO (PeerHire Justice DAO)** — governs the dispute protocol, juror selection logic, and staking incentives. Over time, it can evolve to govern other platform parameters such as reputation decay, minimum stake amounts, or eligible juror pools.

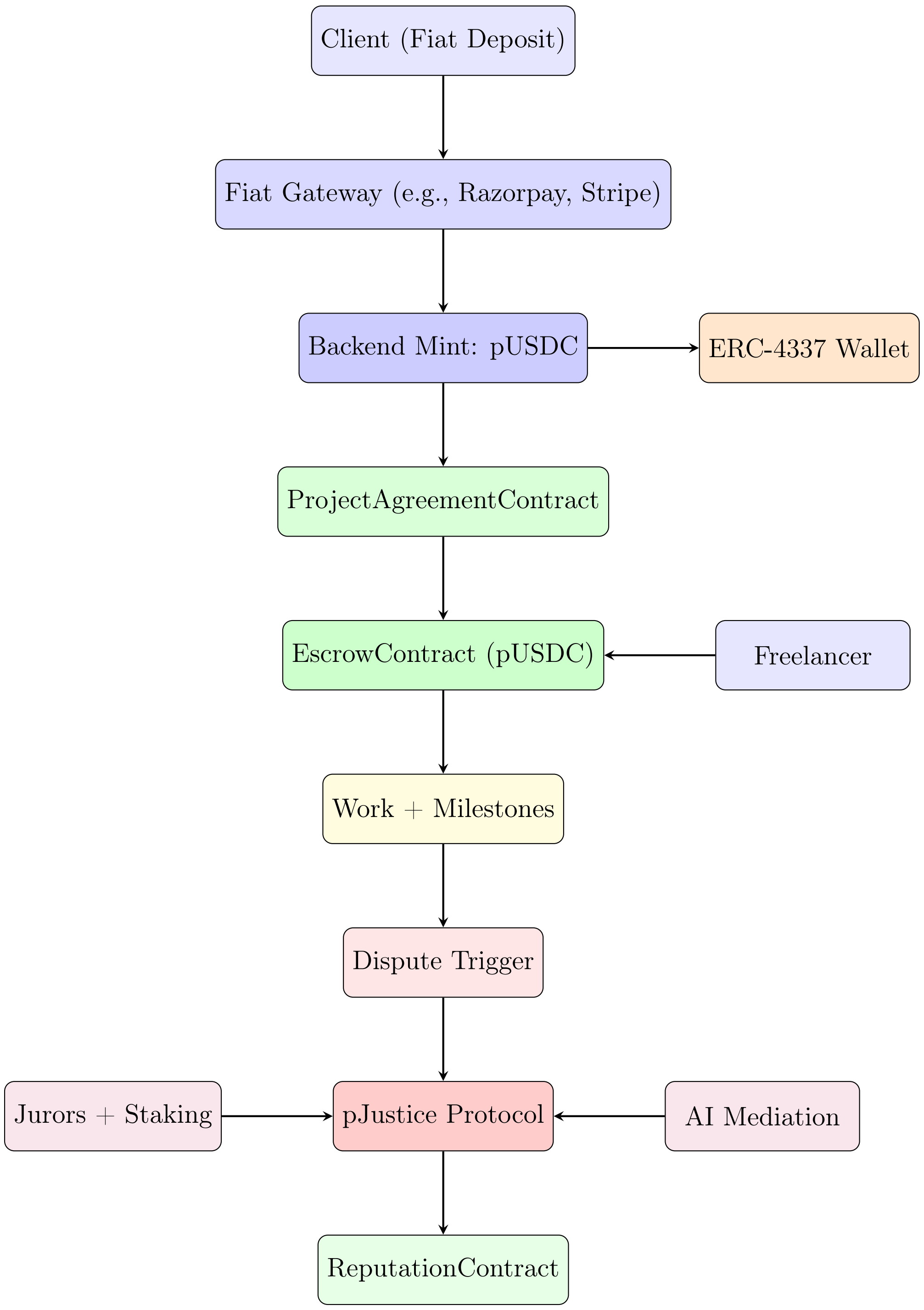

Figure 1: PeerHire Protocol Flow: From Fiat Onboarding to DAO Resolution

3.2. User Experience Goals

PeerHire is designed to eliminate crypto friction entirely from the user journey. No wallets. No seed phrases. No gas fees. Users sign in with familiar credentials (Google, GitHub, etc.), and the platform silently deploys an ERC-4337 smart wallet in the background. Fiat payments are processed through traditional gateways, with synthetic tokens minted by the backend and locked in smart contracts on Polygon.

Everything users see balances, tasks, milestones, juror feedback feels like Web2. But behind the scenes, every rule is enforced by code. This is what makes PeerHire different: Web2 usability, Web3 guarantees.

4. Smart Contract Architecture

PeerHire is composed of nine core contracts, each handling a specific part of the trust, enforcement, and coordination stack. These contracts interact to ensure project agreements, payments, identity, and dispute resolution all function on-chain with minimal off-chain dependencies.

Each contract is modular, upgradeable via DAO proposals, and collectively forms the backbone of PeerHire's programmable trust layer.

4.1. IdentityContract

Purpose:
The `IdentityContract` manages the assignment, verification, and on-chain representation of user identities within the PeerHire ecosystem. It provides a decentralized trust primitive to link users (clients, freelancers, jurors) to verifiable off-chain metadata while preserving privacy. Its primary goals are to mitigate Sybil attacks, ensure fair juror selection, and provide credibility without relying on centralized KYC services.

Design Goals

- Prevent duplicate or fake identities (Sybil resistance).
- Allow pseudonymous users to establish verifiable credibility.
- Enable skill-tagged verification (e.g., student, alumnus, certified designer).
- Integrate cleanly with ERC-4337 wallet abstraction.
- Optionally extend to ZK-based proofs of identity without revealing raw data.

Workflow Summary

The identity lifecycle progresses through four key stages:

1. **Unregistered:** The user has no presence on-chain.
2. **Registered:** User submits hashed metadata via the frontend (e.g., college, email, GitHub ID).
3. **Verified:** Admin backend validates metadata (e.g., API call to a college registry) and verifies the user.

4. **Augmented:** Verified users may mint a non-transferable badge (Soulbound NFT) or earn domain credentials (e.g., “Top Python Contributor”).

State Machine Diagram

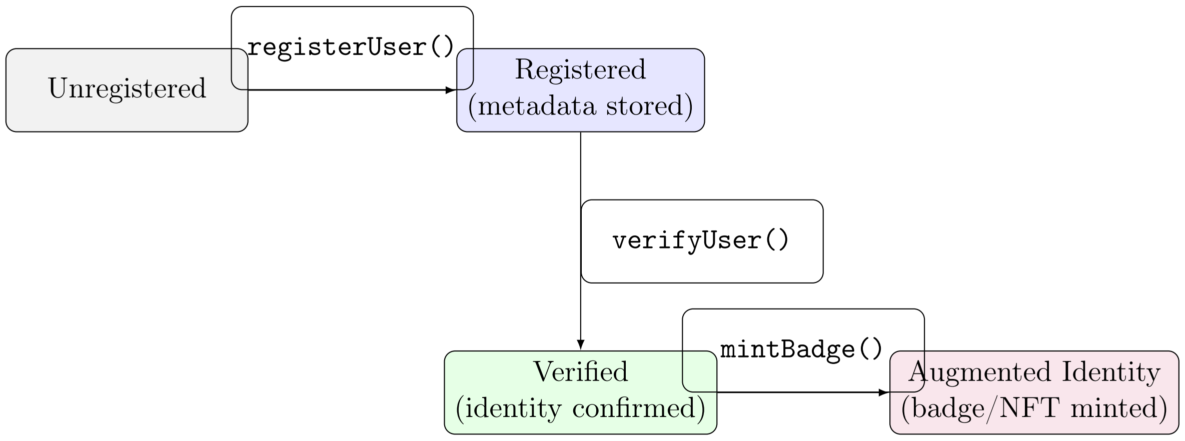
Unregistered
registerUser()
Registered (metadata stored)
verifyUser()
Verified (identity confirmed)
mintBadge()
Augmented Identity (badge/NFT minted)

Figure 2: Identity lifecycle and state transitions

Interface Specification

- `registerUser(address user, string metadataHash)`
  Registers a new user and stores the hashed metadata string (e.g., off-chain IPFS pointer). Emits `UserRegistered` event.

- `verifyUser(address user, bytes calldata verificationProof)`
  Called by backend/API-integrated verifier. On success, flags the identity as verified. Can optionally emit a badge request trigger.

- `isVerified(address user) external view returns (bool)`
  Returns verification status of the user.

- `getUserMetadata(address user) external view returns (string)`
  Returns IPFS or Arweave content hash of the user’s metadata blob.

- `mintBadge(address user, string badgeType)`
  Optionally mints a soulbound NFT or ERC-1155-based non-transferable proof of verified identity or contribution.

Integration with Other Modules

- **JurorPoolContract** — Only verified users can become jurors. Skill tags from `IdentityContract` are used for domain-matching.

- **ReputationContract** — Links project ratings and achievements to on-chain identities for transparency.

- **GovernanceContract** — Ensures voting power in DAO proposals is limited to unique verified participants.

Privacy and Extensibility

All metadata (email, student ID, skills, etc.) is hashed off-chain before being committed. PeerHire may later integrate zero-knowledge proof systems (e.g., Semaphore, Sismo) to allow privacy-preserving attestations such as "verified student at IIT" without revealing raw data.

Real-World Flow Example

1. A student from IIT Kharagpur signs up via frontend with college email and enrollment number.
2. The backend verifies enrollment via an academic registry API or email-based handshake.
3. Upon successful verification, the backend calls `verifyUser()` on-chain.
4. The user can now mint a non-transferable "Verified Student" NFT as part of their identity.
5. This NFT later acts as a requirement for joining high-stakes jury pools or applying for verified client gigs.

4.2. ProjectAgreementContract

Purpose:
The `ProjectAgreementContract` defines and records the immutable terms of a freelance engagement between a client and a freelancer. It ensures both parties are cryptographically bound to the agreed scope, milestones, deadlines, and payment structure. Once a project is accepted, neither party can alter terms unilaterally, eliminating ambiguity and minimizing scope creep.

Design Goals

- Ensure all project metadata is verifiable and tamper-proof.
- Disallow any edits post-acceptance (enforce immutability).
- Bind multiple stakeholders (client, freelancer, escrow contract) to shared logic.
- Enable deterministic triggers for milestone release, timeout logic, or dispute raising.

Contract Lifecycle

The project follows a 4-phase lifecycle:

1. **Drafted:** Client creates a project with all required details. It is not yet active.
2. **Accepted:** A freelancer agrees to the terms, locking the contract.
3. **In Progress:** Freelancer starts work on milestones.
4. **Closed:** All milestones completed and funds released, or project terminated via dispute.

State Diagram

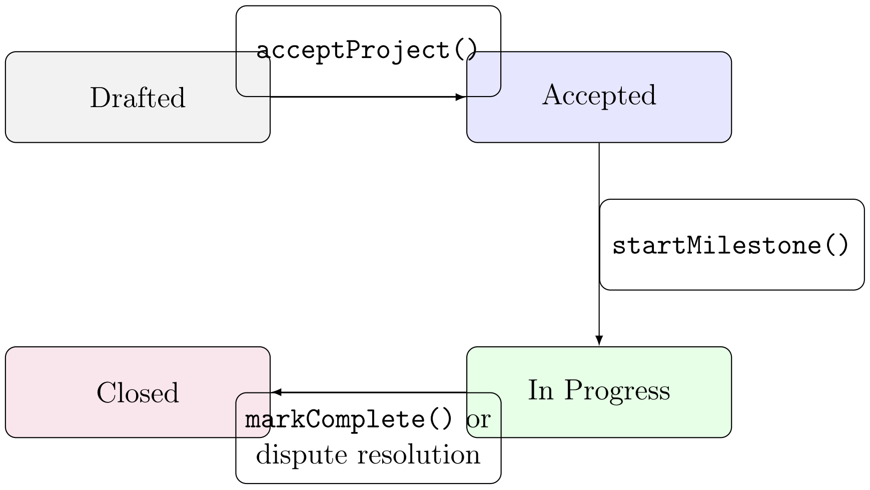

Figure 3: ProjectAgreement state transitions

Interface Specification

- `createProject(bytes32 projectId, address client, address freelancer, uint256[] milestones)`
  Initializes the project with a unique ID, defining the milestone count and payout structure. Only the client may call this.

- `acceptProject(bytes32 projectId)`
  Called by the freelancer to accept project terms. Emits `ProjectAccepted` and transitions to `Accepted` state.

- `startMilestone(bytes32 projectId, uint256 milestoneIndex)`
  Signals work start. Sets a deadline countdown and logs milestone metadata.

- `markMilestoneComplete(bytes32 projectId, uint256 milestoneIndex)`
  Signals submission by freelancer. Escrow may now be released (or client disputes).

- `getProjectDetails(projectId)`
  Returns metadata (parties, budget, milestone status) for frontend or external systems.

Security Considerations

- Once accepted, no function exists to modify scope or budget.
- If a client fails to act post-submission, a time-lock fallback triggers escrow release after grace period.
- All transitions emit events that can be indexed off-chain by TheGraph or similar.

Integration with Other Modules

- **EscrowContract** — milestone triggers map to `releaseFunds()` or `freezeFunds()`.

- **DisputeResolutionContract** — milestone status is used to verify if a dispute trigger is valid.
- **ReputationContract** — triggers rating window after final milestone.

Example Flow: Freelance Graphic Design Project

1. Client Alice drafts a 3-milestone project: Logo → Branding Guide → Social Templates.
2. Freelancer Bob accepts via frontend; project becomes immutable.
3. Bob begins work. For each milestone:
   - He calls `startMilestone()` to begin timer.
   - On delivery, he calls `markMilestoneComplete()`.
   - Client either approves release or raises dispute.
4. Once final milestone is completed and funds are released, the project transitions to `Closed`.
5. Bob receives ratings, and his skill score for "Graphic Design" increases on-chain.

4.3. EscrowContract

Purpose:
The `EscrowContract` is responsible for the custody, conditional release, and final settlement of milestone-linked project funds. It holds synthetic tokens denominated in `pUSDC`, which are minted off-chain at a 1:1 equivalence with USD based on fiat currency received from clients (e.g., INR via Razorpay). Each milestone within a project is associated with a distinct locked amount. These funds remain in escrow until either successful delivery and approval, a dispute-triggered arbitration, or time-based fallback logic.

Design Goals

- Enforce programmable, milestone-based fund locking without centralized custody.
- Enable dispute-triggered freezing of funds pending tribunal resolution.
- Allow controlled refunds and enforce immutable fund flow auditability.
- Prevent race conditions, double spending, or privilege escalation.

Fiat Onboarding to pUSDC

- Clients fund projects through fiat (e.g., INR 24,000 via Razorpay).
- The off-chain backend fetches the prevailing exchange rate (e.g., 1 USD = 80 INR) and computes the USD equivalent (e.g., $300).
- 300 `pUSDC` tokens are minted to the client's ERC-4337 wallet and allocated milestone-wise via `depositFunds()` calls.

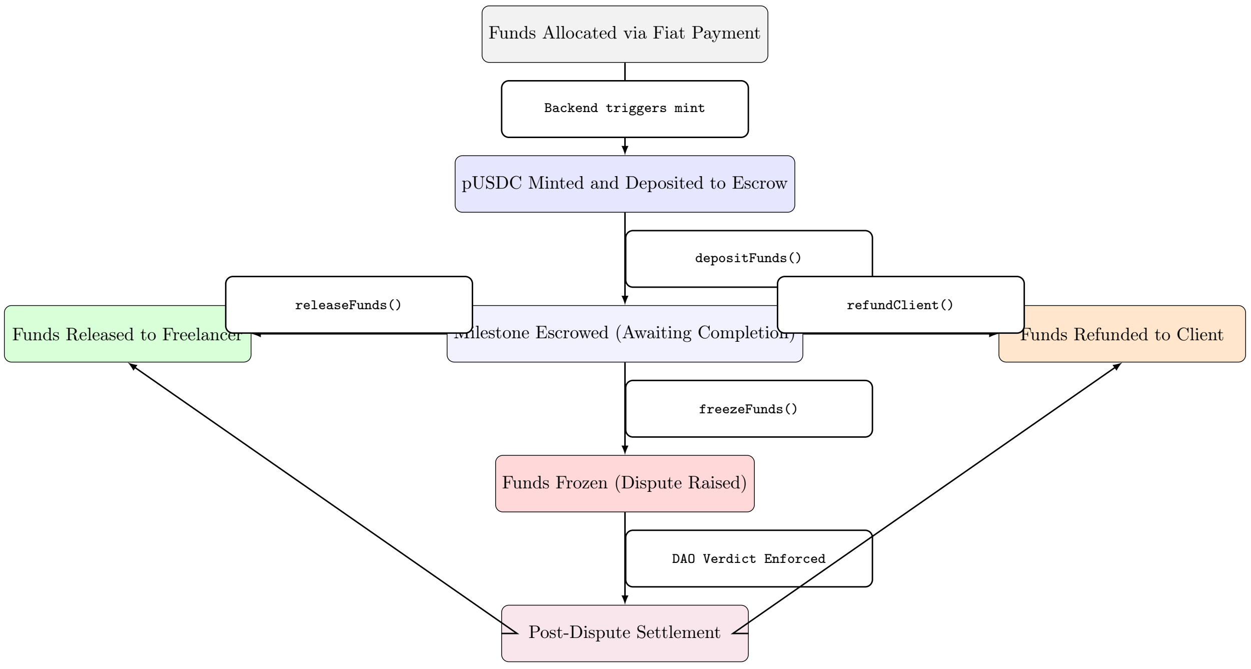
Funds Allocated via Fiat Payment
Backend triggers mint
pUSDC Minted and Deposited to Escrow
depositFunds()
releaseFunds()
refundClient()
Funds Released to Freelancer
Milestone Escrowed (Awaiting Completion)
Funds Refunded to Client
freezeFunds()
Funds Frozen (Dispute Raised)
DAO Verdict Enforced
Post-Dispute Settlement

Figure 4: State transitions for milestone-bound funds in the `EscrowContract`

State Diagram

Interface Overview

- `depositFunds(bytes32 projectId, uint256 milestoneIndex, uint256 amount)`
  Called by the backend after minting `pUSDC`. Deposits funds into escrow, linked to the specified milestone.
- `releaseFunds(bytes32 projectId, uint256 milestoneIndex)`
  Releases escrowed funds to the freelancer upon client approval or post-dispute resolution.
- `refundClient(bytes32 projectId, uint256 milestoneIndex)`
  Returns funds to the client if the milestone is canceled before submission or the freelancer defaults.
- `freezeFunds(bytes32 projectId, uint256 milestoneIndex)`
  Locks funds temporarily when a dispute is raised. Only the DAO or arbitration logic can move funds post-freeze.

Dispute-Aware Fund Control

The contract integrates directly with the `DisputeResolutionContract` via cross-contract calls and interface bindings. Once `freezeFunds()` is executed, any action on the disputed milestone is paused until a binding verdict is returned. Upon resolution:

- A full release may be ordered to the freelancer.
- A partial refund or full refund may be ordered to the client.

- In future versions, multi-party or split settlements can be supported.

Example: USD-Denominated Escrow Flow

1. A client funds a three-milestone project by paying 24,000.
2. The backend computes the USD equivalent ($300) and mints 300 `pUSDC`.
3. 100 `pUSDC` is deposited into milestone 1's escrow.
4. Milestone 1 is marked complete and approved. `releaseFunds()` is called. Freelancer receives 100 `pUSDC`.
5. Milestone 2 leads to a dispute. Funds are frozen until DAO resolution. Final payout is made as per the DAO's ruling.

4.4. DisputeResolutionContract

Purpose:
The `DisputeResolutionContract` is the enforcement and coordination layer for PeerHire's decentralized arbitration mechanism, known as the pJustice Protocol. It receives dispute triggers from either project participant (client or freelancer), collects cryptographic evidence proofs (via IPFS hashes), selects a skill-matched jury pool, and enforces the verdict on-chain. This contract ensures that dispute resolution is transparent, reputation-aware, and tamper-proof, replacing centralized arbitration with trustless logic.

Design Goals

- Route all disputes through a public, DAO-verifiable mechanism.
- Prevent unilateral action during unresolved milestones.
- Weight juror influence based on domain expertise and past accuracy.
- Maintain immutability of submitted evidence and votes.
- Allow optional AI-summarized context for jurors (off-chain augmentation).

Dispute Lifecycle

1. A client or freelancer calls `raiseDispute()`, specifying the disputed milestone.
2. `freezeFunds()` is triggered in the `EscrowContract` to lock funds.
3. Both parties submit relevant materials (IPFS hashes) via `submitEvidence()`.
4. The contract calls `triggerJuryVote()`, initiating juror selection.
5. Jurors vote within a fixed window. Votes are tallied on-chain.
6. On majority consensus, `resolveDispute()` is called, enforcing the verdict.

State Diagram

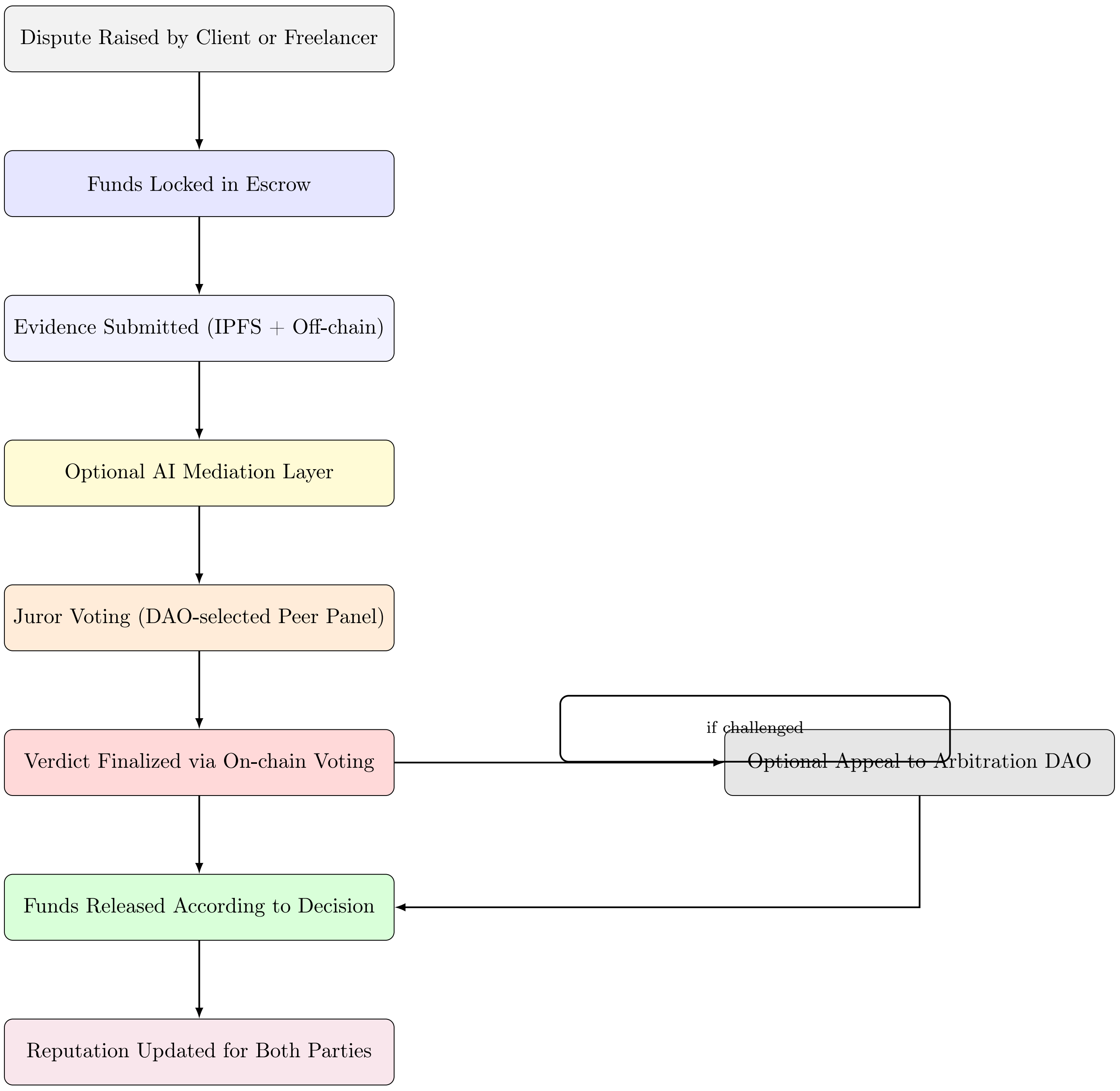
Dispute Raised by Client or Freelancer
Funds Locked in Escrow
Evidence Submitted (IPFS + Off-chain)
Optional AI Mediation Layer
Juror Voting (DAO-selected Peer Panel)
Verdict Finalized via On-chain Voting
if challenged
Optional Appeal to Arbitration DAO
Funds Released According to Decision
Reputation Updated for Both Parties

Figure 5: Dispute resolution lifecycle in the pJustice Protocol

Interface Specification

- `raiseDispute(bytes32 projectId, uint256 milestoneIndex)`
  Called by a party to initiate a dispute. Automatically freezes funds in escrow.

- `submitEvidence(bytes32 projectId, string calldata ipfsHash)`
  Accepts immutable references to off-chain data such as deliverables, communication logs, or agreements.

- `triggerJuryVote(bytes32 projectId)`
  Begins the juror selection and voting phase. Jurors are filtered based on skill tag, DAO reputation, and stake.

- `resolveDispute(bytes32 projectId, address winner)`
  Executes the final decision — releasing funds, adjusting reputation, and logging results.

DAO and Juror Integration

- Jurors are selected from the `JurorPoolContract` using weighted random sampling.
- Quadratic voting may be enforced to prevent stake dominance.
- Verdicts can be binary (winner takes all) or multi-weighted (e.g., partial split).
- After verdict, the `ReputationContract` is updated to reflect outcome.

Example Use Case: UI Design Dispute

1. Client disputes the final UI submission, citing incomplete Figma delivery.
2. Freelancer uploads zipped source files and GitHub links via IPFS.
3. Jury of 7 verified designers is selected based on past design projects.
4. Majority rules in favor of freelancer. Funds released. Client reputation downgraded.

4.5. ReputationContract

Purpose:
The `ReputationContract` is PeerHire's immutable ledger for storing and updating skill-specific performance history of users. It enables clients and freelancers to build portable, verifiable work reputations over time. Every review is tied to a project milestone, a skill domain, and a verified actor ensuring reviews cannot be gamed, deleted, or forged.

Motivation and Role in the Protocol

Traditional freelancing platforms suffer from opaque reputation systems where reviews can be:

- Removed or altered by the platform itself
- Padded with fake clients or paid reviews
- Aggregated into vague scores with little skill-specific insight

PeerHire's approach is different:

- All ratings are cryptographically linked to completed, paid milestones
- Reviews must include a domain-specific skill tag (e.g., `React`, `Solidity`, `Figma`)

- Ratings are final once submitted; no edits or deletion are allowed
- Reputation impact from disputes is directly recorded on-chain

Review Lifecycle

1. A project milestone is marked complete and payment is released from escrow.
2. The client is prompted to rate the freelancer on 1–3 relevant skills.
3. The review is recorded as a permanent on-chain entry.
4. The rating is visible to future clients, jurors, and DAO voters.
5. If a dispute occurs, the juror verdict may trigger a `updateReputationFromDispute()` call, increasing or decreasing the participant's score.

Interface Specification

- `submitRating(address freelancer, string skillTag, uint8 rating)`
  Allows clients to submit a score (1–5) tagged to a specific skill. Can only be called once per milestone after payment release.
- `getRatings(address user)`
  Returns the list of all skill tags and associated average scores, allowing frontend apps to display rich user profiles.
- `updateReputationFromDispute(address user, int8 delta)`
  Invoked by the `DisputeResolutionContract` to penalize or reward users based on juror consensus.

On-Chain Features

- **Skill-Tagged Ratings**: Enables multidimensional review (e.g., 4.8 in Python, 3.9 in Communication).
- **Soulbound Badges**: Freelancers with high reputation in a skill category may be issued SBTs that serve as on-chain credentials.
- **Juror View**: During a dispute, jurors can access both parties' full rating history, dispute history, and most relevant skill scores.
- **No Double Rating**: Only one rating per milestone per skill can be submitted.

Example Use Case

A backend developer completes a milestone.

- The client releases the funds.
- They submit the following review:
  - Skill: `Node.js` → 4.7
  - Skill: `Communication` → 5.0

- These ratings are permanently recorded and can be queried by future clients.
- The developer receives a soulbound badge for achieving 10+ projects with `Node.js` rating above 4.5.

Future Enhancements

- **Decay Logic**: Reputation scores could optionally decay over time if inactivity is detected.
- **DAO-Proposal Driven Modifiers**: PJDAO can propose changes to weights, cooldowns, and badge logic.
- **External Verification**: Integration with GitHub, Figma, or LinkedIn to validate external portfolios.

4.6. pUSDCContract

Purpose:
The `pUSDCContract` governs PeerHire's synthetic stablecoin system. This contract issues and manages `pUSDC`, an internal token that reflects fiat payments made by clients off-chain. It enables blockchain enforcement of payment terms without requiring users to hold crypto or interact with public stablecoins.

Design Rationale

Most users on PeerHire pay using fiat (e.g., INR via UPI or Razorpay). However, escrow, milestone logic, and juror staking operate on-chain. The solution: issue a synthetic token that represents real-world value but stays within the bounds of PeerHire's smart contract ecosystem.

`pUSDC` allows us to:

- Mirror fiat payments on-chain with 1:1 accuracy
- Enable escrow, staking, and rewards logic in Solidity
- Avoid the user onboarding burden of crypto wallets or USDC custody

Token Characteristics

- **Value Peg:** 1 `pUSDC` equals 1 USD worth of fiat. (For example, INR 83.3 = 1 pUSDC.)
- **Synthetic by Design:** It does not exist outside PeerHire and cannot be transferred to external wallets.
- **Lifecycle Bound:** Minted when a user pays fiat, burned when refunds are processed or funds are settled.
- **Scoped Utility:** Used internally across escrow, dispute, and governance contracts.

Flow Overview

1. A client initiates a project and pays INR 10,000 via Razorpay.
2. Backend confirms the payment and mints 120.00 `pUSDC` (if USD = INR 83.3).
3. Funds are moved into milestone escrow via the `EscrowContract`.
4. On completion, the freelancer receives the pUSDC payout.
5. pUSDC is burned once fiat withdrawal is processed off-chain.

Interface Functions

- `mint(address to, uint256 amount)`
  Called by the backend after fiat confirmation. Only whitelisted accounts can mint.
- `burn(address from, uint256 amount)`
  Removes tokens when funds are withdrawn or refunded.
- `transfer(address to, uint256 amount)`
  Allows internal transfers between smart wallets. Tokens cannot leave the PeerHire system.
- `balanceOf(address user)`
  Returns the pUSDC balance of any address used for UI display and staking checks.

Security Design

- **Backend Verification:** Mint and burn are strictly backend-controlled, tied to fiat transaction hashes or order IDs.
- **No Arbitrage Risk:** pUSDC is non-transferable to exchanges and has no real-world market value.
- **Audit-Ready Supply:** Every minted pUSDC is traceable to a real fiat inflow.

Future Extensions

- Deploy pUSDC to L2s (zkEVM, Base) for cheaper execution.
- Add Chainlink proof-of-reserve or fiat oracle integration.
- Enable programmable refunds through DAO-controlled modules.

4.7. JurorPoolContract

Purpose:
The `JurorPoolContract` manages PeerHire's decentralized jury system. It maintains a list of eligible jurors, enforces staking requirements, calculates eligibility based on skill and reputation, and handles economic incentives. This contract ensures that dispute decisions are made by trusted, domain-relevant community members who are financially aligned with the platform's integrity.

Design Objectives

- Filter jurors by domain-specific expertise and trust score.
- Require jurors to stake pUSDC before participating in any dispute.
- Enable slashing for malicious votes or inactivity.
- Offer monetary rewards for aligned, majority-based voting.
- Track historical jury performance on-chain.

Juror Lifecycle

1. A user opts into jury participation by calling `stake(amount)` and becomes part of the juror pool.
2. When a dispute arises, the system queries `isEligible(juror, skillTag)` to filter candidates.
3. A set of 5–11 jurors is selected based on a weighted algorithm (reputation + randomness).
4. After voting, jurors are either slashed or rewarded, depending on whether they align with the majority verdict.

Eligibility Logic

To maintain the quality of decisions, jurors must meet the following criteria:

- Verified identity through `IdentityContract`
- Reputation score above a threshold (e.g., Level 2+)
- Matching skillTag with the project domain (e.g., "UI/UX", "Solidity", "Writing")
- No active slashing penalties or unresolved vote failures

Interface Specification

- `stake(uint256 amount)`
  Adds the caller to the juror pool and locks pUSDC. Required before being selected.
- `unstake(uint256 amount)`
  Withdraws staked pUSDC, provided the juror is not active in a current dispute.
- `isEligible(address juror, string calldata skillTag)`
  Returns `true` if the juror satisfies all requirements for a given skill-tagged dispute.
- `slashJuror(address juror)`
  Deducts staked pUSDC and flags the juror if they vote irresponsibly, violate neutrality, or time out.
- `rewardJuror(address juror, uint256 reward)`
  Allocates rewards to jurors whose vote aligns with the final verdict.

Economic Model

- Jurors are required to stake a minimum of 5–10 `pUSDC`.
- Rewards are distributed proportionally to jurors who vote with majority consensus.
- Malicious, abstaining, or outlier jurors may be penalized up to 100% of their stake.
- Jury performance data (accuracy, vote history, dispute logs) is stored on-chain for transparency.

Example Flow

1. Alice (a verified developer) stakes 15 `pUSDC` and joins the juror pool for "Solidity" disputes.
2. A smart contract project enters dispute over failed audit delivery.
3. Alice is selected as part of a 7-member jury.
4. She reviews the evidence, votes responsibly, and aligns with the consensus.
5. She receives a 2 `pUSDC` reward, boosting her on-chain juror trust score.

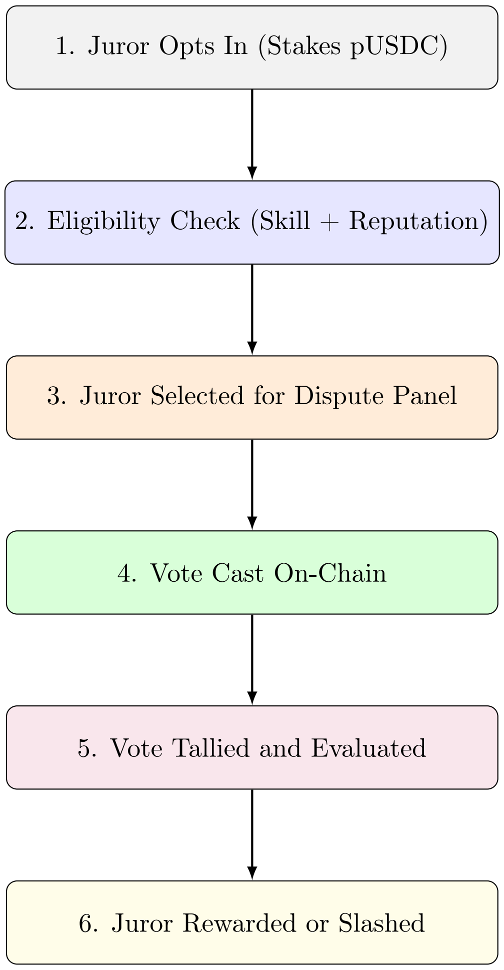
1. Juror Opts In (Stakes pUSDC)
2. Eligibility Check (Skill + Reputation)
3. Juror Selected for Dispute Panel
4. Vote Cast On-Chain
5. Vote Tallied and Evaluated
6. Juror Rewarded or Slashed

Figure 6: Juror lifecycle in the PeerHire dispute resolution process

4.8. VotingContract

Purpose:
The `VotingContract` governs how juror panels are formed, how votes are cast, and how verdicts are determined during a dispute. It ensures that decisions are made collectively, securely, and fairly using mechanisms like skill-based filtering, quadratic weighting, and cryptographic vote privacy.

Design Objectives

- Select domain-relevant jurors from the pool fairly and transparently.
- Prevent collusion by keeping votes private until revealed.
- Support both binary and nuanced verdict models (e.g., partial payouts).
- Ensure that only eligible jurors can vote in active disputes.
- Minimize sybil risk and vote dominance using quadratic voting (optional).

Voting Lifecycle

1. **Start:** The `DisputeResolutionContract` triggers `startVote()` when evidence submission ends.
2. **Selection:** Jurors are randomly drawn from the eligible pool, filtered by skill tags and reputation scores.
3. **Voting:** Each juror submits a private vote via `submitVote()`. Votes are cryptographically hashed or stored off-chain with on-chain verification.
4. **Tallying:** After the voting window closes, `tallyVotes()` calculates the final outcome and triggers downstream actions.

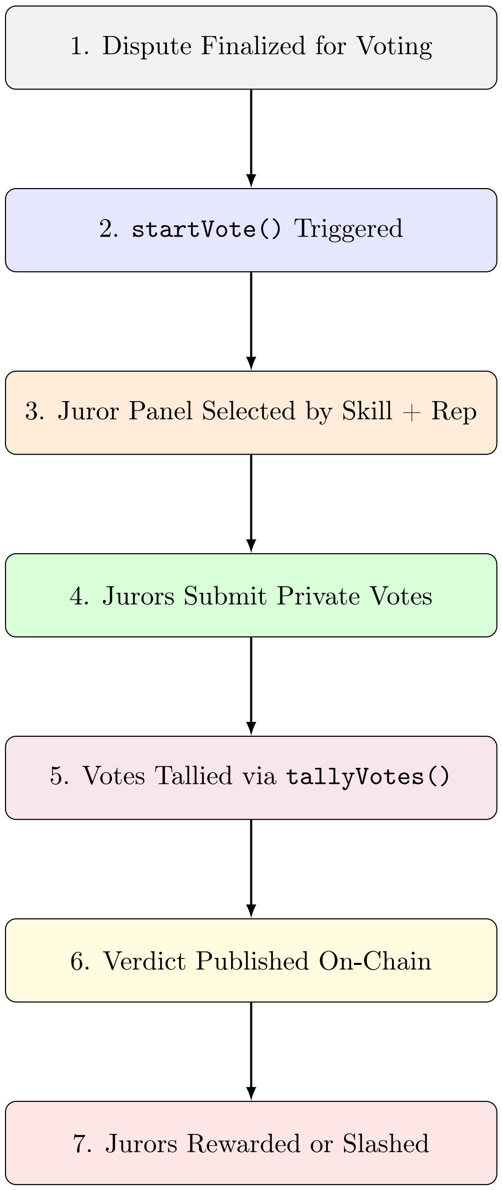
1. Dispute Finalized for Voting
2. startVote() Triggered
3. Juror Panel Selected by Skill + Rep
4. Jurors Submit Private Votes
5. Votes Tallied via tallyVotes()
6. Verdict Published On-Chain
7. Jurors Rewarded or Slashed

Figure 7: Voting lifecycle for juror-led dispute resolution

Interface Specification

- `startVote(bytes32 disputeId, address[] calldata jurors)`
  Initializes a new vote session. Can only be called by the `DisputeResolutionContract`.
- `submitVote(bytes32 disputeId, address juror, bool vote)`
  Called by jurors during the voting window. Only eligible jurors for the given dispute can participate.
- `tallyVotes(bytes32 disputeId)`
  Aggregates votes and outputs a verdict. This function also triggers reward/slashing logic in the `JurorPoolContract`.

Vote Models and Extensions

- **Binary Voting:** Most disputes are resolved with simple yes/no outcomes — e.g., "release payment" or "refund client".
- **Weighted Voting:** In complex or subjective cases, the DAO may enable partial allocation votes (e.g., 70/30 verdicts).
- **Quadratic Logic (Optional):** Vote weights can be modulated by juror reputation scores to avoid majority abuse.

Example Use Case

1. A UI design project is disputed.
2. The system selects 7 jurors with "UI/UX" tags and high reputation.
3. Each juror votes privately over 48 hours.
4. 5 of 7 vote in favor of the freelancer.
5. `tallyVotes()` finalizes the verdict. Payment is released. Jurors are rewarded or slashed accordingly.

4.9. GovernanceContract

Purpose:
The `GovernanceContract` implements the on-chain upgrade and decision-making mechanism for PeerHire's decentralized protocol. It provides a tamper-resistant coordination framework through which verified DAO members (i.e., jurors with sufficient on-chain reputation) can submit, vote on, and execute protocol changes. This includes updates to staking requirements, dispute resolution logic, contract parameters, and even governance itself.

Unlike traditional platforms that rely on opaque admin panels or centralized discretion, PeerHire uses this contract to enforce protocol-level decisions that are transparent, auditable, and community-driven.

Design Objectives

- Enforce parameter changes and upgrade logic without centralized intervention.
- Empower contributors and jurors to shape protocol evolution via proposal-submission rights.
- Ensure fair voting through quadratic-weighted logic and DAO reputation thresholds.
- Avoid stagnation by enabling secure smart contract call execution through encoded `callData`.

Governance Lifecycle Overview

1. A juror meeting eligibility (reputation + stake) submits a governance proposal using `submitProposal()`, including the description and proposed bytecode logic (`callData`).
2. A voting window (default: 7 days) begins. All eligible DAO members may vote using `voteOnProposal()`, with votes weighted by historical jury accuracy.

3. If quorum and majority thresholds are met, the proposal is marked as "Passed."
4. After an optional time delay (used for safety and emergency overrides), any participant may trigger `executeProposal()`, finalizing the upgrade or rule change.

Governance Flow

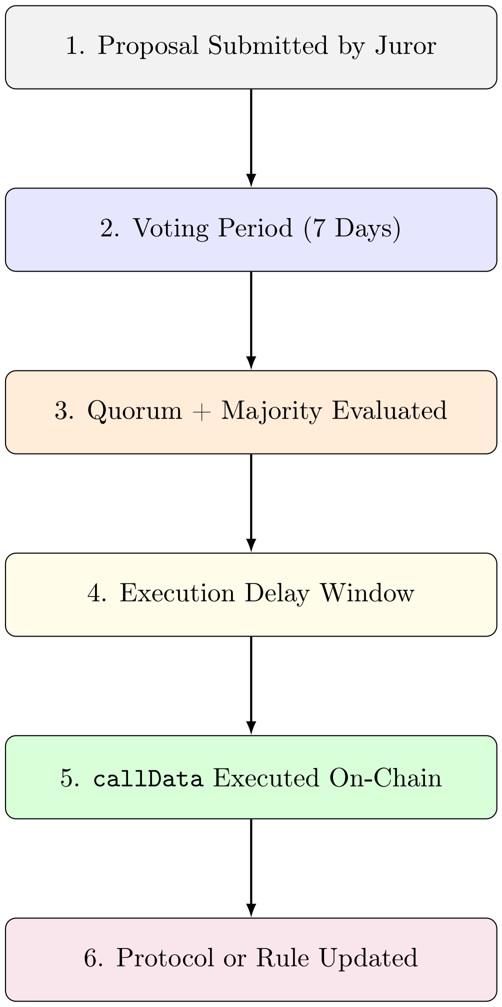
1. Proposal Submitted by Juror
2. Voting Period (7 Days)
3. Quorum + Majority Evaluated
4. Execution Delay Window
5. callData Executed On-Chain
6. Protocol or Rule Updated

Figure 8: Governance proposal lifecycle in PJDAO

Interface-Level Specification

- `submitProposal(string title, string description, bytes callData)`
  Submits a governance proposal with encoded function logic to be executed upon approval.
- `voteOnProposal(uint256 proposalId, bool support)`
  Allows a DAO member to vote on an active proposal. Reputation and staking status may influence voting power.
- `executeProposal(uint256 proposalId)`
  Executes the callData if the proposal passed all voting conditions and the minimum delay has expired.

- `getProposalStatus(uint256 proposalId)`
  Returns current status of a proposal: Pending, Active, Succeeded, Executed, or Failed.

Governance Parameters and Constraints

- **Proposal Threshold:** Must have >= 500 reputation points and >= 250 pUSDC staked to submit.
- **Quorum:** Minimum of 20 total votes or 15
- **Support Threshold:** Minimum 60
- **Execution Delay:** 24–72 hours, configurable by DAO.
- **Emergency Override:** PJDAO multisig can pause execution in case of exploits or malicious code.

Example: Protocol Parameter Update

1. Juror Alice submits a proposal to reduce minimum juror stake from 500 pUSDC to 250 pUSDC.
2. Proposal ID #34 is created, entering a 7-day voting window.
3. 46 jurors vote: 37 "Yes", 9 "No". Quorum met, majority achieved.
4. After a 24-hour delay, Bob (any user) calls `executeProposal(34)`.
5. The JurorPoolContract now reflects the new staking threshold.

5. pUSDC: Synthetic Stablecoin for Protocol Accounting

Introduction

The PeerHire protocol introduces `pUSDC` as a synthetic, non-transferable stablecoin used exclusively within the platform for escrow payments, juror staking, and DAO operations. Pegged 1:1 to the U.S. Dollar, `pUSDC` is not a traditional ERC-20 token, it cannot be bridged, traded, or withdrawn. It is a **unit-of-account token** backed indirectly by fiat deposits collected via off-chain gateways (e.g., Razorpay) and maintained through backend reconciliation.

This approach enables deterministic smart contract logic without exposing users to volatile assets or regulatory complexity. The use of `pUSDC` ensures that project agreements, dispute resolutions, and DAO rewards are denominated in a stable value, while abstracting away wallet UX and crypto volatility for end users.

Design Rationale

- **Fiat-First UX:** Users fund projects in INR/USD via trusted payment processors, with no exposure to wallets or tokens.
- **On-Chain Enforcement:** All internal logic like escrow locks, milestone payouts, juror rewards is handled via stable-value accounting.

- **Synthetic Model:** pUSDC is not issued on-chain like USDC; it is minted as a **protocol-internal representation** of real fiat.
- **Liquidity Pool Based Minting:** To avoid high gas fees and excessive on-chain calls, PeerHire mints pUSDC in bulk into a backend liquidity reserve. As users deposit fiat, equivalent pUSDC is assigned from this pool.
- **Compliance Simplicity:** As pUSDC never leaves the system, there are no custody, AML/KYC, or stablecoin licensing concerns.

Operational Lifecycle

1. Platform backend mints a bulk reserve of pUSDC into a liquidity vault contract.
2. When a user pays via Razorpay or fiat gateway, the backend allocates equivalent pUSDC from the vault to the user's smart contract wallet.
3. Funds are locked in the `EscrowContract` as per project milestone terms.
4. On project completion or dispute resolution, pUSDC is either released to the freelancer or burned if refunded.
5. For refunds, the backend triggers a fiat reversal and burns corresponding pUSDC tokens.

Smart Contract Interface: `pUSDCContract`

Key Functions:

- `mint(address to, uint256 amount)` — Admin-only function to increase supply (used only by backend or DAO vault).
- `burn(address from, uint256 amount)` — Deletes pUSDC during refunds or penalty slashing.
- `transfer(address to, uint256 amount)` — Enables internal contract-to-contract transfers (e.g., Escrow to Freelancer).
- `balanceOf(address user)` — Returns current pUSDC balance of a smart wallet.

Tokenomics Overview

- **Supply Model:** Total circulating pUSDC = Total fiat deposits - refunds - slashed amounts.
- **Non-Transferable:** Cannot be sent to external wallets or traded on DEXs.
- **Commission Handling:** Protocol deducts a commission (e.g., 6%) from fiat before pUSDC is allocated to escrow.
- **Dispute Fee Flow:** pUSDC used in juror staking, slashed for malicious votes, and redistributed to honest jurors.

- **Future DAO Use Cases:** Governance voting gas, grant distribution, juror bounties, and staking rewards.

Security, Auditability, and Risk Mitigation

- Only backend and DAO-approved contracts can mint/burn.
- All pUSDC movements are logged with event hashes and verifiable by auditors.
- Daily backend-to-chain reconciliation checks enforce 1:1 parity.
- If reconciliation fails, all minting halts until the discrepancy is resolved.
- Vault reserves can be paused or capped by governance for circuit-breaking.

Lifecycle Diagram (Resized)

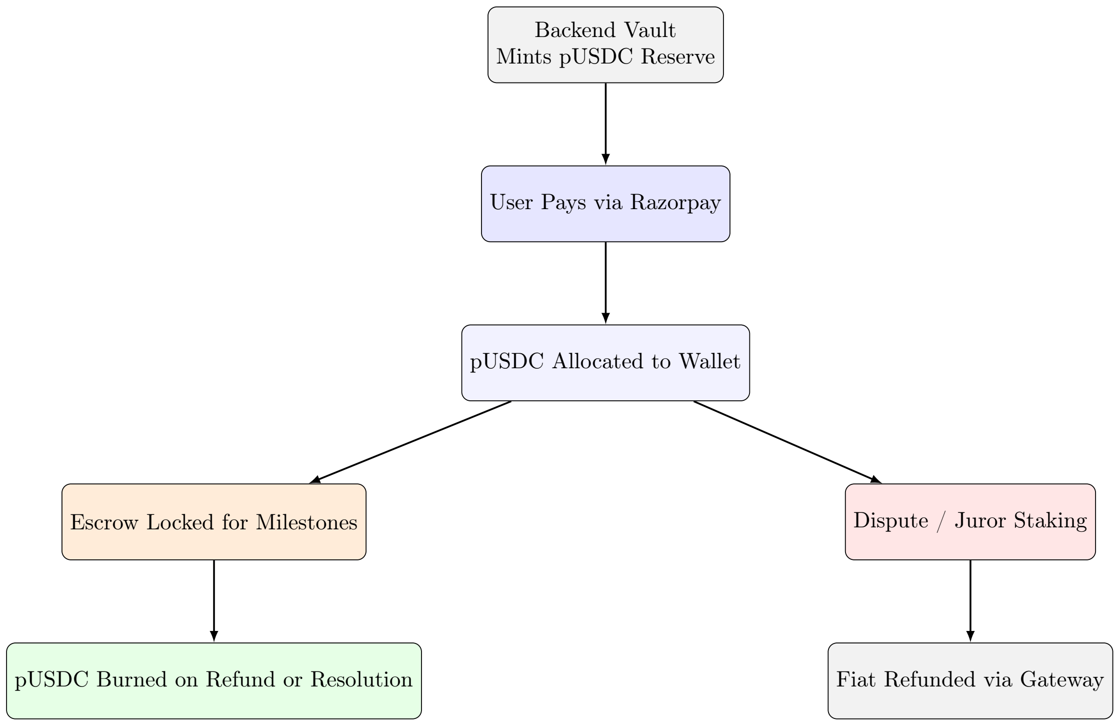
Backend Vault
Mints pUSDC Reserve
User Pays via Razorpay
pUSDC Allocated to Wallet
Escrow Locked for Milestones
Dispute / Juror Staking
pUSDC Burned on Refund or Resolution
Fiat Refunded via Gateway

Figure 9: pUSDC Lifecycle: Vault-based Minting, Allocation, Usage, and Burn

DAO Governance and Future Extensions

In future protocol upgrades, DAO may assume partial or full control over the pUSDC supply lifecycle:

- DAO-controlled grant funds may be allocated in pUSDC to approved wallets.

- Protocol upgrade proposals can define mint caps, burn cooldowns, or jurisdiction-based fiat handling.
- Mint and burn events can be linked to on-chain governance votes for enhanced transparency.

Conclusion

The pUSDC token is a foundational component of PeerHire's fiat-compatible, smart contract-native infrastructure. It bridges the gap between trustless automation and real-world usability by offering stable-value accounting without exposing users to crypto complexity. The synthetic, protocol-contained design allows secure, auditable, and regulatory-light operations, while enabling enforceable coordination for projects, disputes, and governance.

6. pJustice: Decentralized Dispute Resolution Protocol

Overview

The **pJustice Protocol** is PeerHire's decentralized arbitration framework designed to resolve conflicts between clients and freelancers. Built on a modular, cryptographically-enforced pipeline, pJustice integrates skill-weighted peer voting, off-chain AI advisories, and programmable incentive-slash models. This mechanism ensures that milestone disputes are adjudicated fairly, without central authority, while preserving platform efficiency and reputation integrity.

Triggering Disputes

Disputes are triggered under the following conditions:

- A milestone is marked complete by a freelancer, but the client disputes the quality or completeness.
- A client fails to approve or reject a milestone within a predefined timeout.
- Either party explicitly calls `raiseDispute()` with a justification.

Upon trigger:

1. The associated pUSDC in `EscrowContract` is frozen.
2. The project is flagged as "Disputed" on-chain.
3. `DisputeResolutionContract` initializes a juror pool selection process.

Evidence Submission and AI Mediation

Both client and freelancer are prompted to submit evidence (e.g., deliverables, Figma links, GitHub commits, communication logs), which are uploaded to IPFS. The IPFS hashes are committed on-chain via `submitEvidence()`.

An off-chain AI mediation layer powered by LLMs (e.g., GPT-4 via LangChain) summarizes the evidence:

- Generates side-by-side bullet point summaries.
- Flags discrepancies between scope and deliverables.
- Suggests an advisory verdict.

This AI opinion is non-binding and only available to jurors for reference.

Peer Jury Selection

Jurors are selected using the `JurorPoolContract` from among:

- Verified PeerHire users with domain-matched skill tags.
- At least Level 2 reputation.
- Active staking of minimum pUSDC amount.

Jurors are pseudo-randomly selected using:

- DAO reputation weights (normalized)
- On-chain verifiable randomness (e.g., Chainlink VRF)
- Skill tag match multipliers

Typically 5–11 jurors are selected per case.

Quadratic Voting Model

Jurors vote privately off-chain and sign their vote with session keys. On vote reveal, the votes are tallied using quadratic voting:

$$\text{Effective Weight}_i = \sqrt{s_i}$$

Where:

- s_i: number of stake units by juror i
- $\sum \text{Effective Weight}_i$ across all jurors determines winning side

This ensures that:

- Larger stakers have influence, but with diminishing returns.
- Vote buying and collusion are discouraged.

Example:

- Juror A stakes 100 $\rightarrow$ weight = 10
- Juror B stakes 400 $\rightarrow$ weight = 20 (not 4x)

Outcome Enforcement

The winning vote side (client or freelancer) is determined by:

- Simple majority or supermajority, depending on DAO parameters.
- `resolveDispute()` is called by the contract admin or DAO automation.

Consequences:

- Funds are released from escrow accordingly (fully, split, or refunded).
- The `ReputationContract` is updated.
- Losing party may receive a rating downgrade.
- Jurors are slashed or rewarded.

Appeal Mechanism

If either party challenges the verdict:

- They must post an appeal bond in pUSDC.
- Case escalates to **PeerHire Arbitration Council (PAC)**, a curated DAO panel.
- Final binding verdict is returned. All appeal data is logged on-chain.

Incentives and Slashing Logic

Juror Action	**Outcome**
Votes with consensus	Earns fee share
Votes against consensus (with justification)	Partial reward
Malicious or lazy vote (flagged)	Slashing of stake
Fails to vote	Temporary ban from juror pool

Table 1: Juror Incentive Model

DAO Constitution Summary

PeerHire Justice DAO (PJDAO) governs the entire dispute pipeline. Key principles:

- Jurors must KYC and stake to gain entry.
- DAO can vote to change:
  - Jury size
  - Minimum stake amount
  - AI integration thresholds

  - Dispute fees or appeal costs

- DAO reputation grows via correct votes and proposals.
- DAO proposals are executed via `GovernanceContract`.

pJustice Flow

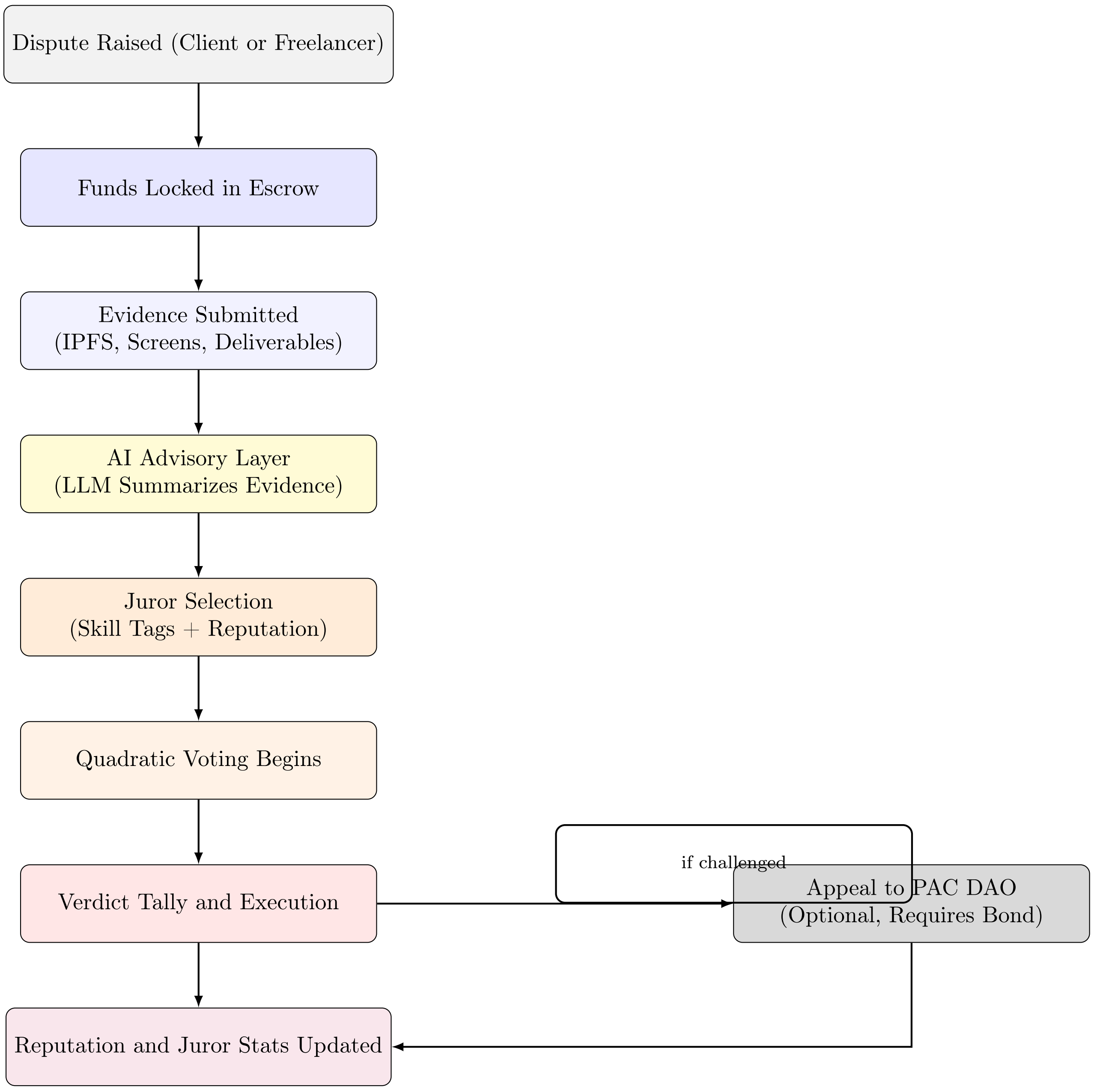
Dispute Raised (Client or Freelancer)
Funds Locked in Escrow
Evidence Submitted (IPFS, Screens, Deliverables)
AI Advisory Layer (LLM Summarizes Evidence)
Juror Selection (Skill Tags + Reputation)
Quadratic Voting Begins
Verdict Tally and Execution
if challenged
Appeal to PAC DAO (Optional, Requires Bond)
Reputation and Juror Stats Updated

Figure 10: Lifecycle of a Dispute in the pJustice Protocol

Conclusion

pJustice offers a transparent, decentralized alternative to the opaque, biased dispute systems found in legacy freelancing platforms. By combining domain-matched juries, cryptographic enforcement, quadratic voting, and optional AI mediation, PeerHire achieves verifiable fairness without compromising user experience or platform neutrality.

7. Wallet UX and ERC-4337 Abstraction

Motivation: Why Wallet UX Breaks Web3 Onboarding

For non-crypto-native users, wallet setup is often the single largest barrier to interacting with decentralized applications. Concepts like seed phrases, signing prompts, gas fees, and token approvals introduce friction and anxiety especially for students, early professionals, or mainstream users unfamiliar with Web3 tooling.

In the PeerHire ecosystem, every user should be able to join, post or accept a job, resolve disputes, and get paid all without having to manage or even see a wallet. This requirement led us to adopt full **account abstraction via ERC-4337** and session-based access patterns.

Architecture: Invisible Smart Wallets per User

Every PeerHire user is assigned a **smart contract wallet** upon signup. This wallet conforms to the ERC-4337 standard, making it programmable, upgradeable, and custodial-free.

- The wallet is deployed deterministically (via a counterfactual address) only when it receives its first funds (pUSDC).
- It is managed by a session key or third-party auth (Web3Auth), not a private key exposed to the user.
- Users never interact with signing prompts, MetaMask popups, or gas selection.

Web3Auth and Social Login

To ensure seamless login without seed phrases, we integrate with `Web3Auth`, an open-source identity aggregator that allows users to authenticate using:

- Google, GitHub, Twitter, Discord (OAuth2)
- Passwordless email-based login
- Mobile biometrics or device keys (via WebAuthn)

The authenticated session key (temporary) is then used to control the user's smart wallet, with full cryptographic accountability.

Session Keys and Delegated Access

PeerHire uses session keys to temporarily authorize UI/browser-level access to the user's wallet. This enables:

- Secure short-lived sessions (e.g., 24h)
- Multi-device login and resume
- Controlled access with auto-expiry

These session keys are bound to access scopes and expiry timestamps. If a session key expires or is revoked, no further transactions are possible until re-authentication.

Gasless Transactions via Bundlers

In traditional Ethereum transactions, users need ETH to pay gas fees. This is unacceptable in a fiat-onboarded platform like PeerHire. To solve this, we rely on:

- ERC-4337's **UserOperation** model
- **Bundlers** (via infrastructure providers like Biconomy or Gelato)

Here's how it works:

1. The user's transaction is bundled and relayed by a third-party operator.
2. The platform pays gas fees on their behalf (sponsored relaying).
3. Optionally, gas costs can be deducted later in pUSDC as a hidden fee.

This provides a fully **gasless UX** and users never see or think about gas.

Security and Replay Protection

Each smart wallet includes:

- **Nonce tracking** to prevent replay attacks.
- **Guardian logic** for recovery and device loss.
- Optional multisig or 2FA enforcement for high-value accounts.

Additionally, since every transaction is a smart contract method (not raw ETH transfer), custom rules like escrow locks, dispute checks, and signature verification can be encoded directly into wallet logic.

Architecture Flow Diagram

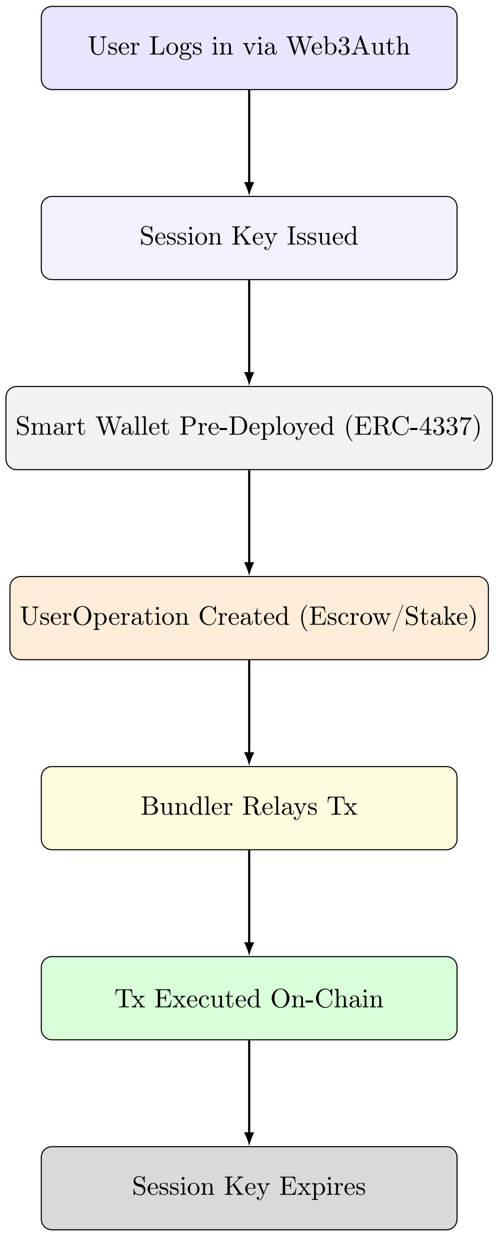
User Logs in via Web3Auth
Session Key Issued
Smart Wallet Pre-Deployed (ERC-4337)
UserOperation Created (Escrow/Stake)
Bundler Relays Tx
Tx Executed On-Chain
Session Key Expires

Figure 11: Workflow for Gasless Smart Wallet UX in PeerHire

Conclusion

Through full ERC-4337 account abstraction, gasless relaying, and social login, PeerHire eliminates the friction that typically repels non-crypto users. This architecture bridges the gap between fiat onboarding and on-chain enforcement, letting users benefit from smart contract guarantees without knowing they're even using crypto. Wallets should be invisible, secure, and boring and that's exactly how we've built them.

8. Security and Attack Vectors

PeerHire's architecture introduces several novel layers such as synthetic escrow tokens, decentralized dispute resolution, and invisible smart wallets. While these abstractions enhance user experience, they also introduce complex attack surfaces.

This section outlines the major categories of threats, their potential impact, and the preventive measures we've integrated into the protocol's design.

1. Juror Collusion and Voter Bribery

Threat: Jurors could collude to favor one party in a dispute, especially in high-value projects. External parties might bribe jurors off-chain or via alternate wallets.

Mitigations:

- Jurors are selected pseudorandomly based on skill-tags, DAO reputation, and stake balance.
- Voting is anonymous during the dispute window and revealed only at tally.
- **Quadratic voting** limits power concentration and stake-based dominance.
- Incorrect or malicious votes are penalized by slashing the juror's staked pUSDC.
- Juror performance history is permanently tracked via the `ReputationContract`.

2. Dispute Spamming

Threat: A malicious actor could flood the platform with low-value disputes to consume DAO attention, block payment flows, or cause denial-of-service to the arbitration system.

Mitigations:

- Every dispute requires a fixed minimum stake in pUSDC, acting as a deterrent for spam.
- Repeated spam behavior triggers automated stake slashing and dispute rate-limiting.
- PJDAO has emergency powers to freeze actors showing coordinated spam patterns.

3. Arbitrage or Abuse of pUSDC

Threat: Since pUSDC is minted off-chain in bulk and allocated to users upon fiat deposit, mismatches or oracle lags could be exploited to create fake balances or arbitrage flows.

Mitigations:

- pUSDC is **non-transferable** outside of PeerHire contracts, removing external liquidity risk.
- All mint and burn events are logged on-chain and reconciled daily against backend accounting.
- Protocol fees (e.g., 6%) are deducted before pUSDC reaches user wallets, reducing surface area for edge-case overflows.
- A real-time auditor script flags anomalies between fiat logs and on-chain supply.

4. Off-Chain Sync Failures (Fiat pUSDC)

Threat: If the fiat payment backend (e.g., Razorpay) goes offline, fails, or delays refunds, there could be a mismatch between user expectations and protocol state.

Mitigations:

- Minting of pUSDC is triggered only after confirmed webhook events from payment gateways.
- Refunds are handled by burning the corresponding pUSDC before triggering off-chain fiat refund flows.
- A watchdog system halts further minting if reconciliation drifts beyond a safety threshold (e.g., 0.5%).
- In severe failures, the DAO can invoke a mint/burn lock through the `GovernanceContract`.

5. Front-Running and Flash Loan Attacks

Threat: In scenarios like dispute resolution or DAO votes, attackers may use mempool monitoring or flash loans to influence outcomes, exploit last-minute proposal changes, or manipulate escrow state.

Mitigations:

- All core transactions (e.g., jury votes, milestone releases) are submitted via ERC-4337 UserOperations — these are batched and do not sit in the public mempool.
- Juror selection and vote execution use hashed commitments to prevent last-block manipulation.
- Escrow and resolution flows are protected by time locks, sequencer finality, and nonce tracking.
- Critical governance votes (e.g., protocol fee changes) require delayed execution windows and multi-phase approval.

6. Recovery and Session Key Exploits

Threat: If a session key is compromised, it could drain a user's pUSDC or tamper with their project agreements.

Mitigations:

- All ERC-4337 wallets are equipped with guardian logic, allowing recovery via email/social re-authentication.
- Session keys are time-bound, IP-bound, and limited in permission scope (e.g., can't release escrow unless verified).
- Wallets can include 2FA-based approvals for certain high-risk actions (e.g., staking over X amount).

Conclusion

Security in PeerHire is not an afterthought, it's a foundational design principle. By combining programmable wallet logic, modular smart contracts, transparent dispute resolution, and reconciliation-aware stablecoin logic, PeerHire defends against both traditional Web3 threats and protocol-specific risks.

This threat model will be continually updated as usage patterns, DAO activity, and external integrations evolve.

9. Governance: PJDAO and Protocol Upgradeability

PeerHire is governed by the **PeerHire Justice DAO (PJDAO)**, a decentralized autonomous organization that controls dispute resolution, staking incentives, juror eligibility, and future protocol upgrades. This system is not merely cosmetic governance, it is structurally tied to how justice is enforced and how rules evolve.

This section details the mechanics of PJDAO, its reputation-based voting model, upgrade paths, circuit breakers, and the roadmap to becoming a fully community-owned protocol.

1. Juror-Centric Reputation System

In contrast to token-weighted governance seen in many DAOs, PeerHire ties governance rights to juror performance. This includes:

- **Positive Contributions:**
  - Voting with majority consensus in dispute cases
  - Submitting high-quality rationales (if required in appeals)
  - Proposing or supporting constructive protocol upgrades
- **Penalties:**
  - Slashing for misaligned votes or abstention
  - Decay of inactive jurors' reputation over time
  - Exclusion from jury pools after repeated poor performance

Reputation Points (DAORep): This non-transferable unit is accrued over time and defines a juror's:

- Voting weight in DAO proposals
- Eligibility to submit or co-sponsor protocol upgrades
- Priority in juror selection

2. Proposal Lifecycle and Submission Rules

Any DAO member with `>100 DAORep` may submit a governance proposal. These are categorized into:

- **Parameter Change:** Adjusting protocol values (e.g., jury size, dispute fees)
- **Upgrade Proposal:** Introducing new contracts or modifying execution logic
- **Policy or Treasury Action:** Grant disbursements, juror onboarding programs, etc.

Required Components for Proposal Submission:

1. **Title and Description**
2. **Rationale and Expected Outcome**
3. **Smart Contract Calldata** (for upgrade proposals)
4. **DAORep threshold check** (enforced on-chain)

Proposals are submitted to the `GovernanceContract` and follow a time-bound voting window (typically 7 days), after which results are queued for delayed execution.

3. Voting Logic and Quadratic Weighting

PJDAO employs **quadratic voting**, which grants influence based on the square root of DAORep. This balances influence between long-term contributors and prevents whale dominance.
Formula:

$$\text{Effective Votes} = \sqrt{\text{DAORep Points Staked}}$$

Why Quadratic?

- Prevents Sybil attacks from users creating multiple low-rep accounts
- Incentivizes meaningful participation instead of passive staking
- Gives minority but high-context voters a proportional voice

Votes are committed via a `commit-reveal` scheme:

- Phase 1: Commit vote hash
- Phase 2: Reveal vote and salt
- Phase 3: Aggregate, verify, and compute majority

4. Upgrade Path and Execution Mechanism

Upgrades are managed through a proxy pattern or beacon architecture (depending on contract type). The DAO has access to a secure upgrade executor, which:

- Verifies proposal quorum and support threshold
- Waits for the time-delay (48–72 hours)
- Allows for audits, external scrutiny, and appeals
- Executes the calldata via the designated upgrade controller

5. Emergency Breaks and Circuit Control

For protocol safety, PJDAO includes:

- **Mint Lock:** Prevents `pUSDC` minting if fiat reserves diverge
- **Dispute Freeze:** Pauses new disputes if manipulation is suspected
- **Governance Pause:** 24-hour emergency delay for high-risk proposals

Emergency proposals require a `2/3 supermajority` and **instant execution** after the voting phase.

6. DAO Treasury and Juror Compensation

PJDAO will gradually own and control:

- **Juror Fees:** Collected from dispute filing costs
- **Commission Revenue:** From project commissions (e.g., 6%)
- **Grant Pools:** To bootstrap new features, tools, or integrations

DAO-led Disbursement: Jurors vote on grant allocations, bug bounties, and feature rewards via `GovernanceContract` proposals.

7. Roadmap to Full Decentralization

- **Phase 1: Bootstrap**
  - Core team holds admin keys
  - Governance is semi-decentralized
  - Emergency powers retained
- **Phase 2: DAO-Driven Governance**
  - All parameter changes and upgrades routed via PJDAO
  - Multisig replaced with on-chain quorum and delay logic

- **Phase 3: Immutable DAO**
  - No privileged roles
  - Treasury fully managed by DAO
  - PJDAO becomes the sole legal and operational entity

8. TikZ Governance Lifecycle Diagram

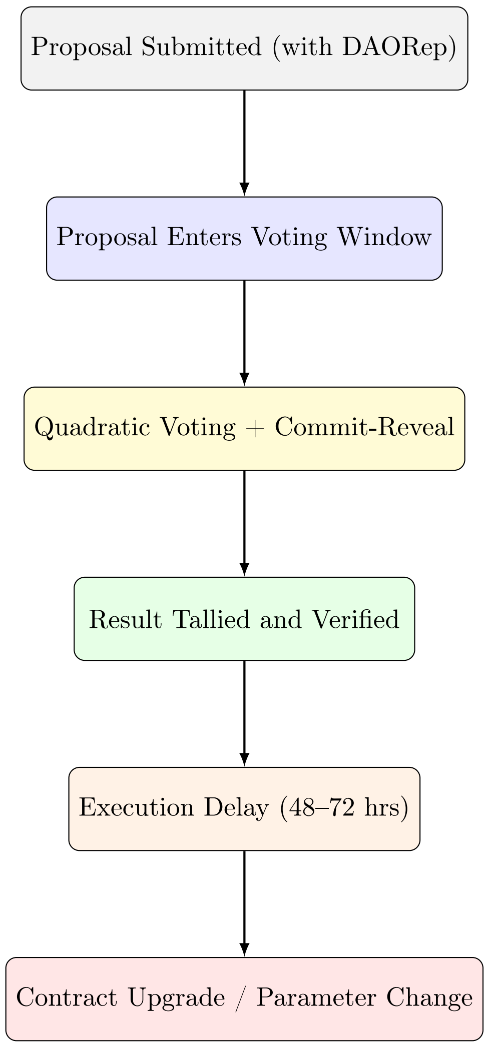
Proposal Submitted (with DAORep)
Proposal Enters Voting Window
Quadratic Voting + Commit-Reveal
Result Tallied and Verified
Execution Delay (48–72 hrs)
Contract Upgrade / Parameter Change

Figure 12: PJDAO Proposal Lifecycle and Governance Flow

Conclusion

PeerHire's governance is engineered for long-term resilience and fairness. By tying power to verified contributor activity, enforcing reputation-aware voting, and incorporating transparent upgrade paths with circuit breakers, PJDAO aims to become a gold standard in dispute-centric protocol governance.

It is not just a DAO that votes, it adjudicates, upgrades, and evolves.

10. Roadmap

PeerHire's development and deployment strategy is structured in sequential phases starting with a tightly scoped MVP focused on student freelancers, and progressively evolving into a globally interoperable, community-governed protocol for remote work. Each milestone introduces new capabilities around decentralization, identity, governance, and ecosystem expansion.

Quarter	**Milestone**
Q3 2025	**MVP Launch (Students, grads and early professionals Only)**: Web app release with fiat onboarding, project posting, milestone-based escrow (pUSDC), and basic freelancer-client matching for student communities. Dispute resolution is manual but logged for future replay.
Q4 2025	**Dispute Resolution v1 (Testnet)**: Launch of smart contract-based dispute logic (pJustice), with mock juror voting, IPFS evidence submission, and minimal AI assistance. Run mock disputes with real students.
Q1 2026	**Reputation System via SBTs**: Soulbound Tokens (SBTs) begin issuing for verified skills, reviews, and dispute performance. Freelancers start building a cross-platform, blockchain-backed CV.
Q2 2026	**DAO Governance v1 + Juror Staking**: Launch of PJDAO, enabling juror staking, quadratic voting on disputes, and DAO proposals for upgrades and parameter changes (e.g., minimum stake amount, juror pool config).
Q3 2026 and beyond	**Global Ecosystem Expansion**: Open the platform to verified professionals, DAOs, and task bounty networks. Add support for programmable task pools, multi-chain integration (e.g., Linea, Base), and treasury-funded grants via governance. Expand dispute protocol to include cross-border arbitration and multilingual juror selection.

The roadmap is not static. Future iterations will evolve based on real-world usage, DAO proposals, and insights gathered during earlier phases. Upgrades will always prioritize community trust, governance security, and the protocol's mission to make freelance collaboration fair, verifiable, and sovereign-by-design.

11. Scalability and Ecosystem Vision

PeerHire's architecture is not limited to solving problems for students alone, it is engineered to scale horizontally across sectors, platforms, and protocols. The primitives we're building trustless escrow, reputation-bound identity, dispute resolution via DAO, and programmable incentive layers are extensible across the global freelance economy, DAO ecosystems, and decentralized talent networks.

From Campus to Cosmos

The MVP targets students and early professionals, who face the sharpest asymmetries in power, payment, and proof-of-work. But the same problems lack of verifiable trust, high intermediary fees, and opaque arbitration plague professionals, DAO contributors, and gig workers worldwide.

The long-term vision is to abstract away institutional gatekeeping and offer a global protocol for verifiable, enforceable remote collaboration.

DAO-to-DAO Collaboration

DAOs today often coordinate via Discord and spreadsheets, lacking formal contracting or escrow mechanisms. PeerHire can provide:

- On-chain project contracts between DAOs and contributors.
- Reputation accumulation across projects, tied to verifiable delivery.
- DAO-to-DAO work bounties, grants, and co-funding workflows.

Towards a Decentralized HR Stack

PeerHire can serve as the infrastructure for decentralized human resource management, including:

- Verified contributor identity (DID + SBT).
- Work histories and reviews stored immutably.
- DAO-curated reputation scores for hiring and referrals.
- Token-gated access to job boards or juror roles.

Modular Plugin Ecosystem

Our smart contract suite can be extended via permissioned or open plugins, including:

- **Grant Allocators:** Automate DAO grant issuance tied to deliverables.
- **Bug Bounty Funnels:** Track submissions, handle escalations, and auto-pay based on triage.
- **Hackathon Pipelines:** Let hackathon organizers issue challenges, rewards, and jury-based scoring.

Each plugin is built on the same foundation: escrow, agreement, dispute resolution, and reputation enforcement.

Social Graph Interoperability

As trust shifts from institutions to peer graphs, PeerHire is designed to integrate with decentralized identity protocols and social primitives:

- **Lens Protocol:** Reputation overlays on Lens profiles.
- **Farcaster / ENS / Ceramic:** Ties between pseudonymous handles and work history.
- **Proof-of-Peer DAO Badges:** Display jury history, project completions, and disputes resolved on public handles.

This creates portable credibility and context-aware access controls.

Cross-Chain Scaling and Multi-Network Strategy

Although PeerHire is initially deployed on Polygon for affordability and ease of integration, our architecture is cross-chain portable:

- **pUSDC Bridging:** Mirror pUSDC balances across EVM chains for interoperability.
- **Multichain Reputation Oracles:** Sync reputation snapshots across chains via APIs or ZK oracles.
- **Cross-chain Disputes:** Leverage relayers to route dispute proofs across chains while enforcing outcomes locally.

A Protocol for Verifiable Work

At scale, PeerHire becomes more than a marketplace. It becomes a decentralized, programmable infrastructure for:

- Verifying who did what, for whom, and when.
- Automating conflict resolution without central control.
- Incentivizing good behavior and punishing misconduct without needing trust.

This turns any ecosystem—education, gaming, research, DeFi into a verifiable, self-enforcing work economy.

12. Team

Founders

Sumit Nirmal

Co-Founder and Chief Executive Officer
Sumit is responsible for protocol architecture, product direction, and technical execution at PeerHire. He holds a degree from the Indian Institute of Technology (IIT) Kharagpur and brings prior experience building agentic automation systems and decentralized applications. His focus spans

end-to-end protocol design, ERC-4337 wallet abstraction, and agent-based DevOps for blockchain infrastructure. He has led multiple 0-to-1 product launches across fintech and Web3.

Chanchal Kuntal
Co-Founder and Chief Operating Officer
Chanchal oversees operations, ecosystem partnerships, and governance workflows. She was awarded the Global Tech APAC title and will represent the Asia-Pacific region at the Global Tech Awards 2025 in Paris. As an AI mentor for Microsoft Azure's student program, she has led technical outreach and upskilling initiatives across India. Her work centers on building resilient user communities and creating inclusive onboarding strategies for decentralized ecosystems.

Governance and Research Network

PeerHire is advised by contributors across the fields of cryptographic protocols, DAO coordination theory, and dispute resolution mechanisms. The protocol's evolution is supported by an extended network of open-source researchers, product designers, and legal analysts. As of this publication, governance decisions are managed by the founding team, with plans to transition control to the PJDAO (PeerHire Justice DAO) by mid-2026, in accordance with the roadmap in Section 10.

13. Legal Considerations

Overview

PeerHire is designed with legal and regulatory clarity from the outset. Unlike traditional token-based platforms that expose users to crypto market volatility, regulatory ambiguity, and custody risks, PeerHire avoids speculative financialization. The platform's architecture explicitly separates user interaction (fiat-based) from protocol-level enforcement (blockchain-based), offering the benefits of decentralization without the compliance burden typically associated with cryptocurrency systems.

No Speculative Token, No ICO Risk

PeerHire does not issue or promote any freely tradable token. The internal accounting unit, `pUSDC`, is a synthetic, non-transferable stablecoin exclusively used within the platform's smart contracts. It cannot be traded on exchanges, speculated upon, or withdrawn from the protocol. As such:

- There is no Initial Coin Offering (ICO) or Token Generation Event (TGE).
- `pUSDC` does not qualify as a security or financial instrument under most global regulations, including the U.S. SEC's Howey Test or India's FEMA guidelines.
- Users do not purchase tokens directly. All economic activity begins with fiat deposits via regulated gateways (e.g., Razorpay, UPI).

This structure reduces legal exposure and ensures that PeerHire's protocol remains compliant with existing and evolving fintech and crypto regulatory environments.

Fiat Onboarding via Compliant Payment Gateways

All payments on PeerHire originate from fiat deposits made through fully regulated payment processors. These include:

- **Razorpay (IN)** — PCI-DSS compliant, RBI-regulated PSP (Payment Service Provider)
- **UPI/NEFT/IMPS** — Standard Indian banking rails with full KYC support
- **Bank Transfers (future)** — For larger clients or institutional users

Fiat onboarding is entirely off-chain. Users are not required to hold or interact with any crypto wallet or assets. The platform backend performs minting of the synthetic pUSDC token in response to fiat confirmations, ensuring:

- Fiat custody is handled exclusively by compliant institutions.
- No conversion to real cryptocurrencies (e.g., USDC, ETH) is performed on behalf of users.
- No exposure to crypto volatility, exchange risks, or AML triggers.

Synthetic Token Design for Internal Use Only

The `pUSDC` token exists solely as a unit-of-account within the PeerHire protocol. It is not pegged to any real stablecoin or custodial reserve; instead, it is mapped to fiat deposits held in bulk by PeerHire Technologies Pvt. Ltd. Key legal implications of this architecture:

- `pUSDC` is not a digital asset or stablecoin as defined by the FATF or RBI.
- It cannot be transferred between users, traded externally, or listed on exchanges.
- It is non-custodial and non-redeemable by the user directly all fiat flows are controlled by PeerHire via refund and withdrawal requests.
- Internal accounting balances are reconciled daily and audited periodically to ensure 1:1 mapping of backend fiat to on-chain synthetic supply.

DAO Governance and Slashing Logic

While the protocol includes DAO-like voting features for juror selection, slashing, and upgrade proposals, all token movements remain internal and are limited to functional flows such as:

- Dispute resolution incentives
- Slashing of malicious jurors
- Staking requirements for tribunal participation

Because these operations involve a synthetic token with no secondary value or external market presence, they do not trigger regulatory burdens under commodity or securities law.

Risk Disclosures

PeerHire provides full upfront disclosures on potential user and protocol-level risks:

- **Refund Delays:** In rare cases, fiat refunds (following burn events) may be delayed due to banking APIs or payment partner limitations.
- **Backend Custody:** While the system avoids self-custody risks for users, it introduces custodial obligations on PeerHire Technologies Pvt. Ltd. to maintain fiat-liquidity integrity.
- **Reconciliation Mismatch:** In the event of backend accounting mismatches, minting is halted and an emergency resolution protocol (including DAO vote) may be invoked.
- **Juror Behavior:** DAO juror actions, though slashed and reputation-weighted, may result in perceived bias. All verdicts are final unless escalated via appeal.

Compliance Roadmap

PeerHire is committed to regulatory compliance across all jurisdictions where operations are active. Key milestones include:

- ISO 27001 and SOC-2 readiness by mid-2026
- RBI OPGSP registration for global remittances (if applicable)
- PCI-DSS compliance for payment flows and token vaulting
- Smart contract audits by certified blockchain security firms

Additionally, the platform maintains internal legal advisory support and partners with fintech lawyers to ensure up-to-date alignment with India's Digital Personal Data Protection Act (DPDP), EU's GDPR, and future global frameworks around digital labor, synthetic tokens, and platform liability.

Conclusion

PeerHire is not a crypto exchange, not a token economy, and not a speculative asset issuance platform. It is a fiat-onboarding, smart contract-powered coordination engine that uses a synthetic stable accounting unit for enforcing work agreements, distributing rewards, and enabling decentralized conflict resolution all within clear legal and compliance boundaries.

References

Academic and Protocol References

- Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008.
`https://bitcoin.org/bitcoin.pdf`
- Vitalik Buterin. *Ethereum Whitepaper.* 2013.
`https://ethereum.org/en/whitepaper/`
- Kleros Team. *Kleros: Short Paper.* Decentralized Justice Protocol. 2018.
`https://kleros.io/static/whitepaper_en-f21d7a1a8d0fb8f06a1a6c7aa4c34c41.pdf`
- Ethereum Foundation. *EIP-4337: Account Abstraction via EntryPoint Contract.* 2021.
`https://eips.ethereum.org/EIPS/eip-4337`
- Glen Weyl, Zoë Hitzig, E. Glen Weyl. *Quadratic Voting in Democratic Politics.* Journal of Political Economy, 2019.
- Aragon Association. *Aragon Court Whitepaper.* 2020.
`https://aragon.org/blog/aragon-court-whitepaper`
- ERC-20 Token Standard. *Ethereum Improvement Proposals.*
`https://eips.ethereum.org/EIPS/eip-20`
- World Bank Group. *The Digital Gig Economy and Developing Countries.* 2018.
`https://documents.worldbank.org/en/publication/documents-reports/documentdetail/681641529722201141/`

Development Libraries and Tools Used

- **OpenZeppelin Contracts** — Secure, community-vetted smart contract libraries.
`https://github.com/OpenZeppelin/openzeppelin-contracts`
- **Chainlink** — Decentralized oracle infrastructure for real-world data feeds.
`https://chain.link/`
- **LangChain** — Agentic LLM orchestration framework for Python and JavaScript.
`https://www.langchain.com/`
- **Gelato Network** — Automation for smart contract tasks (gasless tx, keepers).
`https://www.gelato.network/`
- **Web3Auth** — Social login and wallet abstraction infrastructure.
`https://web3auth.io/`
- **Polygon PoS** — High-throughput EVM-compatible L2 chain for smart contract deployment.
`https://polygon.technology/`
- **Hardhat** — Ethereum development environment and task runner.
`https://hardhat.org/`

- **IPFS** — Decentralized file storage protocol for off-chain evidence.
  `https://ipfs.io/`
- **Razorpay API** — Compliant fiat payment processing stack in India.
  `https://razorpay.com/docs/api/`
- **Biconomy SDK** — Gasless transaction infrastructure.
  `https://www.biconomy.io/`

Legal and Regulatory References

- Financial Action Task Force (FATF). *Guidance for a Risk-Based Approach to Virtual Assets and VASPs.* 2021.
- Securities and Exchange Commission (U.S.). *Framework for "Investment Contract" Analysis of Digital Assets.* 2019.
- Reserve Bank of India (RBI). *FAQs on Digital Lending Guidelines and Payment Gateways.* 2023.
- Ministry of Electronics and IT, Government of India. *Digital Personal Data Protection Act.* 2023.