Tekton: Trustless OTC Trading on Bitcoin

A Whitepaper

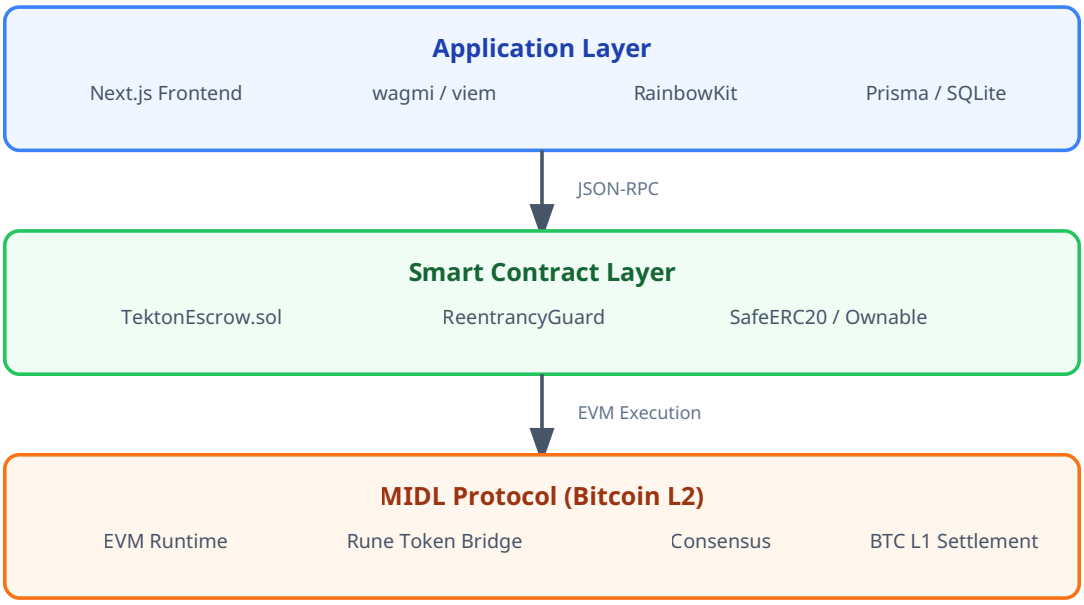
February 2026

Abstract

Tekton is a non-custodial, peer-to-peer OTC trading protocol built on Bitcoin through MIDL Protocol. It uses a Solidity escrow contract to enforce atomic settlement of BTC and ERC-20 token trades, eliminating counterparty risk without requiring trusted intermediaries. The protocol incorporates on-chain reputation tracking, anti-spam collateral requirements, and time-locked cancellation mechanics to create a self-regulating marketplace. This paper describes the protocol's architecture, contract design, state machine, fee economics, reputation model, security properties, and multi-chain roadmap.

1. Introduction

Figure 1: System Architecture



1.1 The OTC Problem

Figure 5: Reliability Score Components

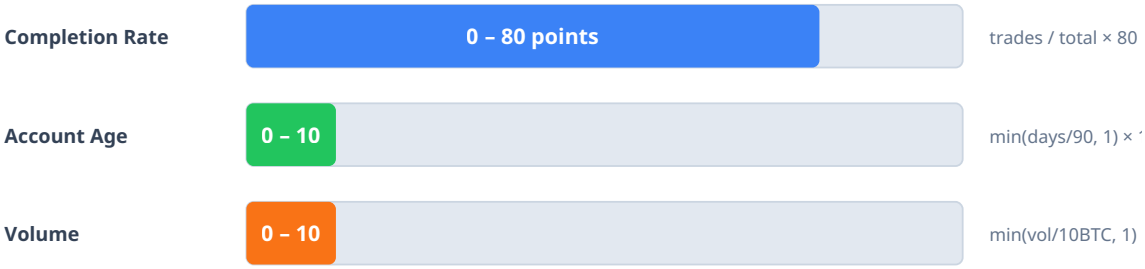


Figure 4: Fee Routing

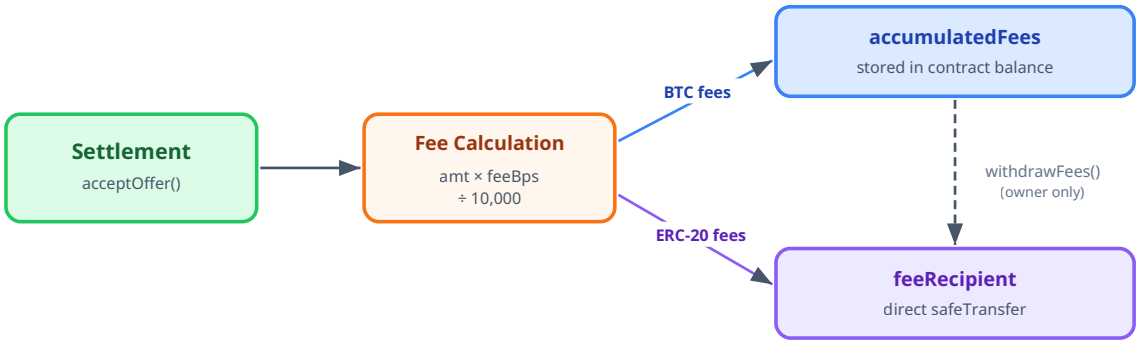
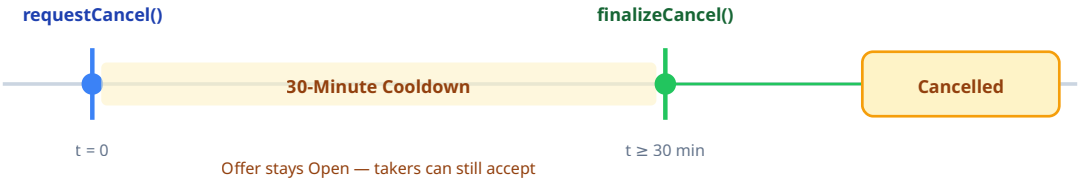


Figure 6: Two-Phase Cancellation Timeline



If acceptOffer() is called during cooldown, the trade settles and cancellation is voided.

Figure 3: Atomic Settlement Sequence

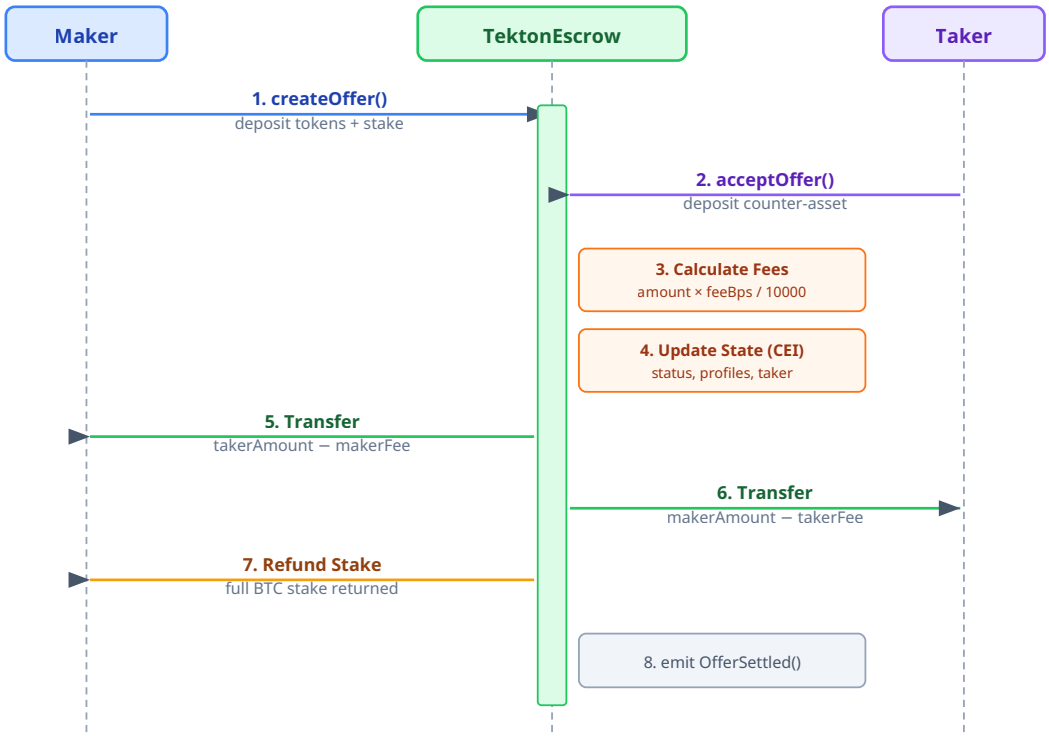
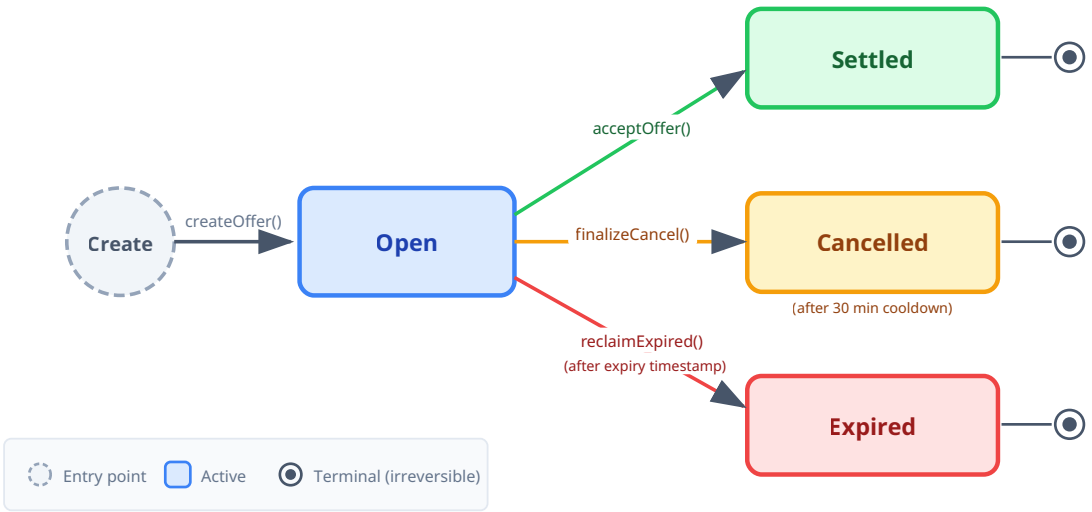


Figure 2: Offer State Machine



Over-the-counter cryptocurrency trading accounts for a significant portion of total market volume. Large holders, institutional desks, and market makers use OTC channels to execute block trades that would cause excessive slippage on public order books. Despite this volume, the infrastructure supporting these trades remains structurally fragile across four dimensions.

Counterparty risk. The fundamental problem in bilateral trading is sequencing: one party must send funds before receiving the counter-asset. The party who moves first assumes the full risk that the other will not deliver. Recourse is limited to off-chain negotiation or legal action, neither of which is fast, inexpensive, or guaranteed to succeed.

Custodial intermediaries. To mitigate counterparty risk, custodial OTC desks hold both sides' funds during settlement. While effective, this model introduces fees (typically 0.1-1%), creates single points of failure, requires KYC/AML procedures, and concentrates risk in entities that may themselves be vulnerable to insolvency or compromise.

Fragmented discovery. Most OTC deals are negotiated through Telegram groups, Discord channels, and direct messages. There is no unified order book, no transparent pricing mechanism, and no standardized way to compare offers across venues.

No accountability. Traders who fail to deliver face no on-chain consequences. Without a reputation system tied to persistent identities, bad actors can repeatedly exploit new counterparties by creating fresh wallets.

1.2 The Tekton Approach

Tekton addresses these problems by encoding the complete OTC trade lifecycle into a single smart contract. The protocol's design rests on four principles:

1. **Non-custodial escrow.** All funds are held by the smart contract during the trade lifecycle, never by a human intermediary. The contract enforces deterministic release conditions.
2. **Atomic settlement.** Both sides of a trade are settled in a single transaction. Either both parties receive their funds, or the entire transaction reverts and no assets move.
3. **Persistent reputation.** Every trade, cancellation, and expiry is permanently recorded against the trader's wallet address on-chain. History is append-only and cannot be reset.
4. **Economic spam resistance.** Every offer requires a BTC collateral stake, and cancellations enforce a 30-minute cooldown period. These mechanisms raise the cost of abusive behavior while keeping legitimate trading frictionless.

---

2. Architecture

2.1 MIDL Protocol

Tekton's initial deployment targets MIDL Protocol, a Bitcoin Layer 2 that provides EVM-compatible smart contract execution with Bitcoin Layer 1 settlement. This architecture offers three properties relevant to the protocol:

- **EVM Compatibility.** Standard Solidity contracts compile and deploy using existing tooling (Hardhat, wagmi, viem). No custom bytecode formats or non-standard opcodes are required.
- **Bitcoin Settlement.** Transactions are ultimately anchored to Bitcoin's Proof-of-Work consensus, inheriting its security guarantees.
- **Rune Token Bridging.** Native Bitcoin Rune tokens are represented as ERC-20 contracts on the L2, enabling programmatic trading of Bitcoin-native assets through Solidity smart contracts.

The escrow contract contains no MIDL-specific dependencies. It is standard Solidity using only OpenZeppelin libraries and deploys without modification on any EVM-compatible chain. MIDL serves as the launch chain given the strength of the Bitcoin-native use case, but the protocol is architecturally chain-agnostic (see Section 8).

2.2 System Components

The protocol consists of three components with clearly separated responsibilities:

TektonEscrow.sol is the protocol's core contract. A single 541-line Solidity contract handles all on-chain operations: offer creation with fund deposit, offer acceptance with atomic settlement, two-phase cancellation, expiry reclamation, fee collection and withdrawal, and reputation tracking. All operations involving fund custody or transfer are executed on-chain through this contract. There are no off-chain dependencies for trade execution.

Frontend Application. A Next.js web application provides the user interface for browsing offers, creating trades, managing positions, viewing trader profiles, and communicating with counterparties. The frontend is stateless with respect to trade logic – all offer data, balances, and reputation scores are read directly from the smart contract. If the frontend application were to go offline, every active trade would remain on-chain and every deposited fund would remain recoverable through direct contract interaction.

Off-Chain Database. A SQLite database stores peer-to-peer chat messages between traders and wallet-signature authentication sessions. No trade state, fund balances, or reputation data is stored off-chain. This data is strictly supplementary and its loss would not affect any trade's integrity or any user's funds.

2.3 Contract Inheritance

The escrow contract inherits from three OpenZeppelin base contracts:

```
contract TektonEscrow is ReentrancyGuard, Ownable {
    using SafeERC20 for IERC20;
    // ...
}
```

- **ReentrancyGuard** prevents reentrant calls to state-changing functions that execute external transfers.
- **Ownable** restricts administrative functions (fee updates, stake adjustments, fee withdrawal) to the contract deployer.
- **SafeERC20** (via `using` directive) wraps all ERC-20 `transfer` and `transferFrom` calls to handle tokens with non-standard return values.

---

3. Protocol Design

3.1 State Machine

Every offer exists in one of four states. Transitions are enforced by the contract and cannot be bypassed by any party, including the contract owner.

States:

State	Value	Description
Open	0	Offer is active; maker's funds and stake are held in escrow
Settled	1	Both sides have been atomically distributed; trade is complete
Cancelled	2	Maker has reclaimed their deposit after the cancellation cooldown
Expired	3	Maker has reclaimed their deposit after the offer's expiry timestamp

Transitions:

From	To	Trigger	Conditions
Open	Settled	`acceptOffer()`	Taker deposits counter-asset; not expired; taker is allowed

From	To	Trigger	Conditions
Open	Cancelled	`finalizeCancel()`	Maker requested cancel >= 30 min ago; still Open
Open	Expired	`reclaimExpired()`	Current timestamp >= offer expiry; still Open

Once an offer leaves the `Open` state, it is terminal. No further transitions are possible.

3.2 Data Structures

The contract defines two primary structs that constitute the protocol's on-chain data model:

```
struct Offer {
    address maker;            // Creator of the offer
    address makerToken;       // Token being sold (address(0) = native BTC)
    uint256 makerAmount;      // Amount being sold
    address takerToken;       // Token wanted in return (address(0) = native BTC)
    uint256 takerAmount;      // Amount wanted in return
    uint256 stake;            // Anti-spam BTC stake held in escrow
    uint256 expiry;           // Unix timestamp after which offer can be reclaimed
    uint256 cancelRequestedAt; // Timestamp of cancel request (0 = not requested)
    address allowedTaker;     // Restricted taker address (address(0) = public)
    address taker;            // Address that accepted (set on settlement)
    OfferStatus status;       // Current state: Open, Settled, Cancelled, Expired
}

struct TraderProfile {
    uint256 tradesCompleted;  // Count of successful settlements
    uint256 totalVolume;      // Cumulative trade volume in wei
    uint256 offersCancelled;  // Count of cancelled offers
    uint256 offersExpired;    // Count of expired, unfilled offers
    uint256 firstTradeAt;     // Unix timestamp of first trade (0 = never traded)
}
```

Storage layout:

```
mapping(uint256 => Offer) public offers;         // offerId -> Offer
mapping(address => TraderProfile) public profiles; // trader -> profile
mapping(address => uint256[]) public userOfferIds; // trader -> list of offer IDs
uint256 public nextOfferId;                       // auto-incrementing counter
```

3.3 Offer Creation

The `createOffer()` function establishes a new trade offer by depositing the maker's sell-side assets and anti-spam stake into the contract.

Function signature:

```
function createOffer(
    address makerToken,   // address(0) for BTC
    uint256 makerAmount,
    address takerToken,   // address(0) for BTC
    uint256 takerAmount,
    uint256 expiry,       // Unix timestamp, must be >= now + 1 hour
    address allowedTaker  // address(0) for public offers
) external payable nonReentrant
```

Validation checks: - `makerAmount > 0` and `takerAmount > 0` - `expiry > block.timestamp` and `expiry >= block.timestamp + 1 hour` - `makerToken != takerToken` (cannot trade a token for itself)

Fund handling depends on the maker's sell-side token:

When selling BTC (makerToken = address(0)): - `msg.value` must be >= `makerAmount + minStake` - Both the trade amount and stake are held in the contract's native balance - Any excess `msg.value` beyond `makerAmount + minStake` is immediately refunded

When selling an ERC-20 token: - `msg.value` must be >= `minStake` (BTC stake only) - The maker must have previously approved the contract for >= `makerAmount` of the ERC-20 - The contract pulls `makerAmount` via `safeTransferFrom` - Any excess `msg.value` beyond `minStake` is immediately refunded

The offer is assigned a sequential ID from `nextOfferId++`, stored in the `offers` mapping, appended to the maker's `userOfferIds` list, and an `OfferCreated` event is emitted with all parameters.

3.4 Atomic Settlement

The `acceptOffer()` function executes the complete trade in a single atomic transaction. This is the core of the protocol's trust model: both sides are settled simultaneously, making it impossible for one party to receive funds without the other also receiving theirs.

Function signature:

```
function acceptOffer(uint256 offerId) external payable nonReentrant
```

Preconditions: - Offer status must be `Open` - Current timestamp must be before `offer.expiry` - Caller cannot be the offer maker (`offer.maker != msg.sender`) - If `allowedTaker` is set, caller must match it

Execution sequence (CEI pattern – Checks, Effects, Interactions):

1. **Deposit counter-asset.** If the taker token is BTC, `msg.value` must cover the `takerAmount` (excess is refunded). If ERC-20, the taker must have approved the contract, and funds are pulled via `safeTransferFrom`.
2. **Calculate fees.** Platform fees are computed as basis points of each side's counter-asset: `makerFee = takerAmount * platformFeeBps / 10000 takerFee = makerAmount * platform-FeeBps / 10000 makerReceives = takerAmount - makerFee takerReceives = makerAmount - takerFee`
3. **Update state (Effects).** Before any external transfers:
4. `offer.taker = msg.sender`
5. `offer.status = OfferStatus.Settled`
6. `offer.cancelRequestedAt = 0` (void any pending cancel)
7. Both traders' `TraderProfile` structs are updated: `tradesCompleted++`, `totalVolume += volume`, `firstTradeAt` set if zero
8. **Execute transfers (Interactions).**
9. BTC fees added to `accumulatedFees`; ERC-20 fees sent directly to `feeRecipient`
10. Maker receives `makerReceives` of the taker's token
11. Taker receives `takerReceives` of the maker's token
12. Maker's anti-spam stake is refunded in full
13. **Emit event.** `OfferSettled(offerId, taker, makerReceived, takerReceived, totalFee)`

If any step in this sequence fails (insufficient approval, failed transfer, arithmetic error), the entire transaction reverts. No partial state changes persist. This atomicity guarantee is the protocol's core security property.

3.5 Two-Phase Cancellation

Cancellation uses a time-locked two-phase mechanism designed to prevent a specific attack pattern: a maker posts an attractive offer, waits for a taker to submit an acceptance transaction, then front-runs it with a cancellation.

Phase 1: Request (`requestCancel`)

```
function requestCancel(uint256 offerId) external
```

- Only the maker can call this
- Offer must be `Open` with no prior cancel request

- Sets `offer.cancelRequestedAt = block.timestamp`
- Emits `CancelRequested(offerId, cancelableAfter)` where `cancelableAfter = block.timestamp + 30 minutes`
- The offer remains `Open` and fully accepting during the cooldown

Phase 2: Finalize (`finalizeCancel`)

```
function finalizeCancel(uint256 offerId) external nonReentrant
```

- Only the maker can call this
- Offer must still be `Open` (not accepted during cooldown)
- `block.timestamp >= cancelRequestedAt + CANCEL_COOLDOWN` (30 minutes)
- Sets `offer.status = OfferStatus.Cancelled`
- Increments `profiles[maker].offersCancelled`
- Refunds the maker's deposited tokens and stake

Key property: If a taker calls `acceptOffer()` during the 30-minute cooldown window, the offer is settled normally. The settlement sets `cancelRequestedAt = 0`, voiding the pending cancellation. The maker receives the trade proceeds rather than their original deposit. This means the cooldown serves as a "last chance" window for legitimate takers.

3.6 Expiry Reclamation

Offers that are not filled and not cancelled before their expiry timestamp can be reclaimed by the maker:

```
function reclaimExpired(uint256 offerId) external nonReentrant
```

- Only the maker can call this
- Offer must be `Open` and `block.timestamp >= offer.expiry`
- Sets `offer.status = OfferStatus.Expired`
- Increments `profiles[maker].offersExpired`
- Refunds the maker's deposited tokens and stake

The minimum expiry duration is 1 hour (`MIN_EXPIRY_DURATION = 1 hours`), ensuring that offers are visible for a reasonable period before they can be reclaimed.

3.7 Fee Model

The protocol charges a symmetric platform fee on settlement, deducted from each party's received counter-asset.

Parameters:

Parameter	Value	Constraint
`platformFeeBps`	30 (0.3%)	Owner-adjustable
`MAX_FEE_BPS`	500 (5%)	Immutable hard cap (`constant`)

Fee calculation:

```
makerFee = takerAmount * platformFeeBps / 10000    // Deducted from what maker receives
takerFee = makerAmount * platformFeeBps / 10000    // Deducted from what taker receives
```

Fee routing differs by token type: - **BTC fees** accumulate in the contract's `accumulatedFees` counter and are withdrawn in batch by the contract owner via `withdrawFees()` . - **ERC-20 fees** are transferred directly to the `feeRecipient` address at time of settlement.

This asymmetry exists because batching native BTC fee withdrawals is gas-efficient, while ERC-20 tokens cannot be aggregated in a single counter without tracking per-token balances.

Administrative constraints: - `setPlatformFee(bps)` requires `bps <= MAX_FEE_BPS` – the owner cannot set a fee above 5% - `setFeeRecipient(addr)` requires `addr != address(0)` - `withdrawFees()` requires `accumulatedFees > 0` and transfers to `feeRecipient`

3.8 Anti-Spam Stake

Every offer requires a minimum BTC stake (`minStake`) deposited by the maker alongside their trade tokens. The stake is always refunded – on settlement, cancellation, or expiry. There is no slashing mechanism.

Rationale for non-slashing design:

A slashing model (where the stake is forfeited on cancellation) would disproportionately penalize legitimate makers who have valid reasons to cancel (market movement, changed requirements). The anti-spam mechanism instead relies on two complementary deterrents:

1. **Capital lockup.** The stake must be available upfront, limiting the rate at which an attacker can create offers. An attacker creating 100 offers simultaneously would need 100x the minimum stake in liquid BTC.

2. **Reputation cost.** Each cancellation or expiry is permanently recorded in the maker's `TraderProfile`. Since the completion rate component accounts for 80 of 100 possible reputation points, even a few cancellations among otherwise successful trades will meaningfully degrade the maker's score. Experienced traders filter by reputation, making serial cancellers effectively invisible to the market.

The minimum stake is adjustable by the contract owner via `setMinStake()` to respond to changing network conditions and attack economics.

---

4. Reputation System

4.1 Profile Data

Every address that participates in a trade accumulates a permanent, append-only on-chain profile stored in the contract's `profiles` mapping:

```
struct TraderProfile {
    uint256 tradesCompleted;    // Incremented on settlement (both maker and taker)
    uint256 totalVolume;        // += trade amount in wei (both maker and taker)
    uint256 offersCancelled;    // Incremented on finalized cancellation (maker only)
    uint256 offersExpired;      // Incremented on expiry reclamation (maker only)
    uint256 firstTradeAt;       // Set to block.timestamp on first trade; never reset
}
```

Profile updates occur atomically within the settlement, cancellation, or expiry transaction. They cannot be reset, modified, or deleted outside of the contract's defined operations. There is no administrative override for profile data. A trader's on-chain history is permanent and immutable.

4.2 Reliability Score

The contract exposes a `getReliabilityScore(address)` view function that computes a 0-100 score from three weighted components:

Component 1: Completion Rate (0-80 points)

```
uint256 totalOffers = p.tradesCompleted + p.offersCancelled + p.offersExpired;
uint256 completionPts = (p.tradesCompleted * 80) / totalOffers;
```

This is the dominant component by design. In OTC trading, the most critical question is whether a counterparty will honor their commitment. A trader with 20 completed trades and 0 cancellations earns the full 80 points. A trader with 20 completed and 5 cancelled earns `(20 * 80) / 25 = 64` points.

Component 2: Account Age (0-10 points)

```
uint256 daysSince = (block.timestamp - p.firstTradeAt) / 1 days;
uint256 agePts = daysSince >= 90 ? 10 : (daysSince * 10) / 90;
```

Linear ramp from 0 to 10 over the first 90 days from the trader's first completed trade. This component rewards longevity and is difficult to fake (requires waiting).

Component 3: Volume (0-10 points)

```
uint256 volumeCap = 10 ether; // 10 BTC in wei
uint256 volumePts = p.totalVolume >= volumeCap ? 10 : (p.totalVolume * 10) / volumeCap;
```

Linear ramp from 0 to 10, capping at 10 BTC of cumulative volume. This component rewards meaningful participation and filters out accounts that only complete trivial test trades.

Total score: `completionPts + agePts + volumePts` (0-100)

4.3 Confidence Adjustment

The raw on-chain score can overstate reliability for new traders. A single successful trade yields a 100% completion rate, earning 80 points. While mathematically correct, this does not represent proven reliability.

The frontend applies a confidence multiplier before displaying scores to users:

```
adjustedScore = rawScore * min(tradesCompleted / 10, 1)
```

Effect on displayed scores:

Completed Trades	Raw Score	Multiplier	Displayed Score
1	80	0.10	8
3	80	0.30	24
5	80	0.50	40

Completed Trades	Raw Score	Multiplier	Displayed Score
10	80	1.00	80
20	85	1.00	85

At 10 completed trades, the multiplier reaches 1.0 and the displayed score equals the raw on-chain score. This creates a clear growth path for new traders while protecting counterparties from trusting unproven addresses with high but statistically insignificant completion rates.

4.4 Trust Tiers

The frontend maps adjusted scores and trade counts into human-readable trust tiers displayed on trader profiles and offer cards:

Tier	Min Adjusted Score	Min Trades	Criteria
OG Trader	95	50	Near-perfect completion rate over a large sample
Trusted	80	15	Consistently reliable with meaningful volume
Reliable	60	5	Demonstrated track record with moderate history
New	–	–	Insufficient data to assess reliability

Trust tiers are computed client-side and are not stored on-chain. They serve as visual indicators only – the contract enforces no restrictions based on tier.

---

5. Security Model

5.1 Smart Contract Security

The contract employs multiple layers of defense against common smart contract vulnerabilities:

Reentrancy protection. All state-changing functions that make external calls (`acceptOffer`, `finalizeCancel`, `reclaimExpired`, `withdrawFees`) use the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`. Additionally, the contract follows the Checks-Effects-Inter-

actions (CEI) pattern throughout: all state mutations (status changes, profile updates, balance adjustments) complete before any external calls (`call{value}`, `safeTransfer`). Even without `ReentrancyGuard`, reentrant calls would encounter already-updated state.

Safe token transfers. All ERC-20 interactions use OpenZeppelin's `SafeERC20` library, which wraps `transfer` and `transferFrom` calls to handle tokens that: (a) return `false` instead of reverting on failure, (b) return no value at all, or (c) have other non-standard behaviors. This covers a broad class of deployed ERC-20 implementations.

Integer safety. Solidity 0.8.28 provides built-in overflow and underflow protection on all arithmetic operations. The contract does not use `unchecked` blocks for any fee calculation, amount computation, or counter increment.

Access control. Administrative functions are restricted via `Ownable`:

Function	Access	Constraints
`setPlatformFee(bps)`	Owner only	`bps <= 500` (MAX_FEE_BPS is immutable)
`setMinStake(amount)`	Owner only	None (can be set to 0 or any value)
`setFeeRecipient(addr)`	Owner only	`addr != address(0)`
`withdrawFees()`	Owner only	`accumulatedFees > 0`

No administrative function can modify an active offer's state, alter a trader's reputation profile, or redirect deposited trade funds. The owner has no ability to intervene in active trades.

Excess value refunds. All `payable` functions that accept `msg.value` compute the exact required amount and immediately refund any surplus. This prevents the common "accidentally sent too much ETH" problem:

```
uint256 excess = msg.value - requiredAmount;
if (excess > 0) {
    (bool ok, ) = msg.sender.call{value: excess}("");
    require(ok, "Excess refund failed");
}
```

5.2 Frontend Security

Wallet-signature authentication. The application uses `personal_sign` (EIP-191) with timestamped messages for authentication. Users sign a message containing the current timestamp; the server verifies the signature recovers to the claimed address and that the timestamp is within a 5-minute window. This prevents replay attacks. Sessions are valid for 24 hours.

CSRF protection. All mutating API endpoints (POST, PUT, DELETE) verify that the request's `Origin` header matches the server's `Host` header. Requests from different origins are rejected.

Rate limiting. The authentication endpoint enforces a per-IP limit of 10 attempts per minute. The messaging endpoint enforces 30 requests per minute per authenticated address. These limits are implemented as in-memory sliding windows.

Input sanitization. All user-provided text (chat messages, offer descriptions) is stripped of: control characters (U+0000-U+001F, U+007F-U+009F), bidirectional override characters (U+202A-U+202E, U+2066-U+2069), and zero-width characters (U+200B-U+200F, U+FEFF). This prevents homograph attacks, invisible text injection, and RTL/LTR override exploits.

5.3 Threat Model

Threat	Attack Description	Mitigation
Cancel-baiting	Maker posts attractive offer, cancels when taker submits acceptance	30-minute cancel cooldown; taker can accept during window, voiding the cancel
Offer spam	Attacker floods order book with fake offers	BTC stake required per offer; cancellations degrade reputation
Sybil scores	Attacker creates wash trades to inflate reputation	Confidence multiplier requires 10+ trades for full score weighting
Signature replay	Attacker captures and replays auth signature	5-minute timestamp expiry on signed messages
Session hijacking	Attacker steals session token	Client-side tokens; revocable via `DELETE /api/auth`; 24-hour expiry
Reentrancy	Attacker exploits callback during fund transfer	`ReentrancyGuard` + CEI pattern on all external call paths
Fee manipulation	Owner sets exploitative fees	`MAX_FEE_BPS = 500` is a compile-time constant; cannot be changed

6. Token Support

6.1 Token Model

Tekton supports two token types:

Native BTC. Represented by `address(0)` in the contract. BTC is transferred via `msg.value` on deposit and `call{value}` on withdrawal. The contract holds BTC in its native balance.

ERC-20 tokens. Any token implementing the ERC-20 interface can be used in offers. The contract interacts with ERC-20 tokens exclusively through `SafeERC20.safeTransferFrom` (deposits) and `SafeERC20.safeTransfer` (withdrawals). No token whitelist is enforced at the contract level.

The contract requires that `makerToken != takerToken` – a maker cannot create an offer to trade a token for itself. Both sides of a trade can be BTC, both can be ERC-20, or one of each (the most common case).

6.2 Deployed Tokens

On MIDL Protocol, Bitcoin Rune tokens are bridged to the L2 and represented as standard ERC-20 contracts. At launch, the following tokens are available:

Token	Contract Address	Decimals	Type
BTC	`0x0000000000000000000000000000000000000000`	18	Native
WBTC	`0x1736866b6CA02F2Ec69a4b7E6A70fd15700d71bE`	18	ERC-20 (Rune)
USDC	`0x323177Bac995D133062DC8F5f2D390B3eaC4577C`	6	ERC-20 (Rune)
TEKTON	`0x62865D0bD2576cf10dd261ADB2fC1d6Ca1485f2c`	0	ERC-20 (Rune)

The frontend displays rich metadata (icons, names, decimal-aware formatting) for known tokens and falls back to displaying the contract address for unrecognized tokens.

6.3 Decimal Handling

The contract operates on raw `uint256` amounts and is agnostic to token decimals. A trade of 1,000,000 USDC (6 decimals, = 1 USDC) and 100000000000000000 BTC (18 decimals, = 0.1 BTC) are stored as their raw values. The frontend is responsible for decimal-aware display formatting, parsing user input into raw amounts, and computing human-readable exchange rates.

Volume tracking in `TraderProfile.totalVolume` sums raw `takerAmount` values without decimal normalization. This means volume figures are meaningful only when comparing trades of the same token. The frontend accounts for this when displaying trade statistics.

---

7. Economic Model

7.1 Revenue

The protocol generates revenue through platform fees on settled trades. At the current rate of 30 basis points (0.3%), a $100,000 trade generates $300 in fees split across both sides. Fees are denominated in the traded tokens themselves, meaning the protocol's revenue composition mirrors its trading volume by token.

BTC fees accumulate in the contract and are withdrawable in batch. ERC-20 fees are routed directly to the fee recipient on each settlement. This dual model avoids the complexity of tracking accumulated balances for arbitrary ERC-20 tokens.

7.2 Incentive Alignment

The protocol's incentive structure ensures that completing trades is the dominant strategy for any rational, repeat participant:

Action	Stake Outcome	Reputation Effect	Score Impact
Complete a trade	Refunded in full	`tradesCompleted++`, `totalVolume += amount`	Strongly positive
Cancel an offer	Refunded in full	`offersCancelled++`	Negative (reduces completion rate)
Let an offer expire	Refunded in full	`offersExpired++`	Negative (reduces completion rate)
No participation	N/A	No profile created	Neutral

The completion rate component dominates the score formula at 80 of 100 possible points. Consider a trader with 19 successful trades and 1 cancellation:

- Completion rate: 19/20 = 95%, yielding 76 of 80 points
- Total score (with full age and volume bonuses): 96/100

Now consider the same trader with 5 cancellations:

- Completion rate: 15/20 = 75%, yielding 60 of 80 points
- Total score: 80/100 – dropped from “OG Trader” to the border of “Trusted”

This steep sensitivity to cancellations creates strong pressure to honor committed offers. The stake refund ensures legitimate cancellations are not financially punitive, while the reputation cost makes serial cancellation self-defeating.

---

8. Roadmap

Tekton is architecturally chain-agnostic. The escrow contract contains no chain-specific dependencies – the state machine, fee model, reputation system, and settlement logic operate identically on any EVM-compatible network. Multi-chain deployment requires only redeployment and configuration; no contract modifications are necessary.

Phase 1: Bitcoin Foundation (Current)

The initial deployment on MIDL Protocol Regtest establishes the core trading engine and validates the protocol’s viability for Bitcoin-native OTC trading.

Delivered capabilities: - Atomic escrow settlement for BTC and ERC-20 tokens - On-chain reputation tracking with confidence-adjusted scoring - Anti-spam collateral stakes with two-phase cancellation - Peer-to-peer messaging between traders - Wallet-signature authentication (no passwords, no centralized accounts)

Next milestone: Production deployment on MIDL Protocol mainnet upon network availability, with finalized Rune token registry for bridged Bitcoin assets.

Phase 2: Multi-Chain Expansion

The same contract deploys on any EVM-compatible chain without modification. Phase 2 extends Tekton to additional L1 and L2 networks where OTC demand and liquidity exist.

Target networks: - **Arbitrum / Base / Optimism.** Lower transaction costs, high throughput, and established DeFi user bases make these chains suitable for retail and mid-size OTC trades. - **Ethereum Mainnet.** Higher gas costs are immaterial for institutional-scale OTC trades. Ethereum’s liquidity depth and institutional presence make it a high-priority target. - **Additional Bitcoin L2s.** As the Bitcoin L2 ecosystem matures and EVM compatibility expands, Tekton can deploy on competing chains to capture Bitcoin-native OTC demand.

Each deployment operates as an independent instance with its own order book, reputation ledger, and fee configuration. This architecture avoids cross-chain complexity while immediately serving traders on each network.

Planned enhancements for Phase 2: - Off-chain order book indexing via event listeners (`OfferCreated` , `OfferSettled` , `CancelRequested`), replacing on-chain iteration with indexed queries for efficient filtering, sorting, and search - WalletConnect and MetaMask integration for broader wallet compatibility - Price oracle integration (Chainlink, Pyth) for market-rate offers and limit-order functionality

Phase 3: Cross-Chain Trading

Once Tekton operates on multiple networks, the protocol extends to enable trades across chains. A trader on Arbitrum should be able to fill an offer posted on Ethereum without manually bridging funds.

Settlement approaches under evaluation:

- **Hash Time-Locked Contracts (HTLCs).** Atomic swaps using cryptographic hash locks and time locks, requiring no trusted intermediary. Each party locks funds with a hash; revealing the preimage on one chain unlocks funds on the other.
- **Bridge-mediated settlement.** Integration with established bridge protocols (e.g., Across, Stargate) to move assets between chains as part of the settlement transaction flow.
- **Intent-based execution.** Makers express desired trades as intents; solvers compete to fill orders from any source chain, with the escrow contract verifying fulfillment on each side.

Cross-chain reputation is a related challenge. A trader's history on one chain should be portable to others. Approaches include: - Cross-chain messaging protocols (LayerZero, Hyperlane) to synchronize profile data between deployments - Off-chain attestation layers that aggregate reputation across deployments into a single verifiable credential - EAS (Ethereum Attestation Service) for cryptographically signed reputation attestations

Phase 4: Protocol Governance and Ecosystem

As the protocol matures beyond a single-operator model, governance and extensibility become priorities.

- **Governance transition.** Move fee parameter control from a single owner to a multi-sig or DAO governance structure, ensuring no single party can unilaterally modify protocol economics.
- **Dispute resolution.** Multi-signature arbitration mechanism for contested trades, allowing designated arbiters to pause settlement pending review. Particularly relevant for cross-chain scenarios where on-chain verification is incomplete.

- **SDK and API.** Published interfaces enabling third-party frontends, trading bots, and aggregators to interact with Tekton deployments programmatically. This transforms Tekton from an application into infrastructure.
- **Mobile-native experience.** WalletConnect and MetaMask mobile connectors to extend accessibility beyond desktop browsers.

The long-term objective is a unified OTC liquidity layer spanning every major EVM chain, where reputation is portable across networks, offers can be filled from any chain, and settlement remains trustless at every step.

---

9. Conclusion

Tekton demonstrates that trustless OTC trading on Bitcoin is not only feasible but practical with current technology. By combining MIDL Protocol's EVM compatibility with a purpose-built escrow contract, the protocol eliminates counterparty risk without requiring trusted intermediaries or custodial infrastructure.

The on-chain reputation system provides what OTC markets have historically lacked: a permanent, public, and tamper-proof record of every participant's trading behavior. Reputation cannot be reset, selectively edited, or transferred between addresses. The confidence-adjusted scoring formula ensures that scores reflect statistically meaningful sample sizes, preventing new accounts from appearing more reliable than their history warrants.

The two-phase cancellation mechanism and anti-spam stake requirements create structural disincentives against the most common OTC abuse patterns – cancel-baiting, offer spam, and score manipulation – without restricting legitimate trading activity.

Atomic settlement is the protocol's foundational guarantee: every trade either completes exactly as specified in the offer terms, or no assets move at all. There are no partial settlements, no stuck funds, and no administrative overrides for active trades.

MIDL Protocol provides the initial deployment environment, but the protocol's chain-agnostic Solidity contract positions it for deployment across the EVM ecosystem. The roadmap extends from single-chain operation through multi-chain deployment to cross-chain settlement, progressively building toward a unified OTC liquidity layer that operates wherever smart contracts run.

References

1. MIDL Protocol. “MIDL JS SDK Documentation.” https://js.midl.xyz/
2. OpenZeppelin. “OpenZeppelin Contracts v5.” https://docs.openzeppelin.com/contracts/5.x/
3. MIDL Protocol. “MIDL GitHub Repository.” https://github.com/midl-xyz/midl-js
4. Ethereum Foundation. “EIP-191: Signed Data Standard.” https://eips.ethereum.org/EIPS/eip-191
5. Ethereum Foundation. “ERC-20: Token Standard.” https://eips.ethereum.org/EIPS/eip-20

Appendix A: Contract Summary

Property	Value
Contract	TektonEscrow.sol
Source Lines	541
Solidity Version	^0.8.28
Chain	MIDL Protocol Regtest (Chain ID 15001)
Deployed Address	`0x0FCF1E8F42B98299a44C2A4d1F06298808A5E326`
Platform Fee	30 bps (0.3%)
Max Fee Cap	500 bps (5%) – immutable constant
Cancel Cooldown	30 minutes – immutable constant
Min Expiry Duration	1 hour – immutable constant
Dependencies	OpenZeppelin ReentrancyGuard, SafeERC20, Ownable
License	MIT (contract only)

Appendix B: Event Reference

Event	Emitted When	Key Parameters
`OfferCreated`	New offer deposited	offerId, maker, tokens, amounts, stake, expiry
`OfferSettled`	Trade atomically completed	offerId, taker, net amounts, total fee
`CancelRequested`	Maker initiates cancellation	offerId, cancelableAfter timestamp
`CancelFinalized`	Cancellation completed after cooldown	offerId
`OfferReclaimed`	Expired offer reclaimed	offerId
`FeesWithdrawn`	Owner withdraws accumulated BTC fees	recipient, amount
`PlatformFeeUpdated`	Fee rate changed	old bps, new bps
`MinStakeUpdated`	Stake requirement changed	old stake, new stake
`FeeRecipientUpdated`	Fee recipient changed	old address, new address