UKIYO NETWORK · TECHNICAL WHITE PAPER

SPOOL Protocol

Production-verified credit on Arbitrum

Document type: Technical white paper · Version 3.0

Execution environment: Arbitrum One · Settlement root: Ethereum L1

Classification: Public

Contents

1. Executive Summary
2. System Context and Trust Boundaries
3. Order NFT: Token Standard and State Machine
4. Industry Spool Contracts
5. Stablecoin Advance and Repayment Logic
6. Agent Execution Pipeline
7. Oracle Integration Interface
8. Arbitrum L2 Architecture: Cost, Finality, and Sequencer Risk
9. Stylus and the Rust Execution Path
10. Security Model and Failure Modes
11. Testing and Simulation Strategy
12. Deployment and Observability

1. Executive Summary

SPOOL Protocol is a production-verified credit network in which AI agents watch fulfillment oracles and execute allowlisted smart-contract actions: minting order NFTs, funding industry spools, and advancing stablecoins to suppliers. The protocol's operating principle is continuous rather than quarterly risk response: agents observe, decide, simulate, and then act—pausing advances, repricing spreads, and notifying buyers as production risk moves.

SPOOL deploys on Arbitrum One as its primary execution environment, with Ethereum L1 as the settlement root. Arbitrum's ~250ms block time and low fees make per-order state transitions and frequent oracle attestations economically viable, while Stylus—Arbitrum's Rust-to-WASM execution layer—runs compute-heavy credit logic at a fraction of EVM gas costs. This white paper specifies the contract architecture, finality assumptions, sequencer failure handling, Stylus split, and settlement mechanics.

2. System Context and Trust Boundaries

SPOOL operates across four trust domains. The boundaries between them are where security properties are either preserved or lost.

Domain	Components	Trust assumption	Failure impact
Industrial	ERP systems, production lines, logistics providers	Data reflects physical production, with latency	Corrupted evidence propagates to credit decisions
Oracle	Fulfillment Oracle agents, attestation signers	Scores are attested and fail closed	Stale or absent evidence; credit pause
Execution	SPOOL agent runtime, policy engine, simulation	Agent stays within allowlist; simulation precedes execution	Erroneous but bounded action
On-chain (L2)	SPOOL contracts on Arbitrum One	Arbitrum sequencer liveness + fraud-proof finality	Sequencer downtime; delayed hard finality

On-chain (L1)	Settlement root on Ethereum	Ethereum L1 finality	Fund loss or invariant violation

The separation of evidence, authority, and execution is structural. A compromised oracle cannot move funds; a compromised agent cannot exceed its allowlist; a compromised contract cannot alter the attestation history.

3. Order NFT: Token Standard and State Machine

The order NFT is SPOOL's primary coordination primitive. It represents a production order and carries the state that credit logic reads and writes.

3.1 Token standard

Each order is unique—distinct supplier, buyer, milestones, and risk parameters—so SPOOL uses ERC-721 rather than ERC-1155.

```
interface IOrderNFT is IERC721 {
    struct OrderData {
        bytes32 supplierId;
        bytes32 buyerId;
        uint256 notional;
        uint64  createdAt;
        uint64  deadline;
        OrderState state;
        bytes32 latestAttestationHash;
        uint16  fulfillmentScoreBps; // 0–10000
    }
    enum OrderState { Created, Funded, InProduction, Shipped, Delivered, Settled, Defaulted }

    function mintOrder(OrderData calldata data) external returns (uint256 tokenId);
    function updateState(uint256 tokenId, OrderState newState) external;
    function attachAttestation(uint256 tokenId, bytes32 attestationHash) external;
    function orderData(uint256 tokenId) external view returns (OrderData memory);
}
```

3.2 State machine invariants

Transition	Precondition	Authorized caller	On-chain effect
Created → Funded	Spool has sufficient liquidity; agent action approved	Spool contract	Advance recorded against order
Funded → InProduction	Supplier confirms start; score above threshold	Agent (allowlisted)	Milestone logged
InProduction → Shipped	Logistics event attested	Oracle callback	Score updated
Shipped → Delivered	Buyer receipt attested	Oracle callback	Repayment scheduled
Any → Defaulted	Score below floor or deadline exceeded	Agent or multisig	Spool loss allocation

A single `onlyAuthorized` modifier checks both the caller and the current state, so an agent cannot skip states or re-enter a settled order.

4. Industry Spool Contracts

An industry spool is a sector-specific liquidity pool. Lenders deposit stablecoins; the spool allocates capital to order NFTs within its vertical (Auto, Pharma, Electronics).

4.1 Core accounting

```
interface ISpool {
    function deposit(uint256 assets, address receiver) external returns (uint256 shares);
    function withdraw(uint256 shares, address receiver, address owner) external returns (uint2
    function fundOrder(uint256 orderTokenId, uint256 amount) external onlyRole(FUNDER_ROLE);
    function totalAssets() external view returns (uint256);
    function totalShares() external view returns (uint256);
    function convertToShares(uint256 assets) external view returns (uint256);
    function convertToAssets(uint256 shares) external view returns (uint256);
}
```

The share/asset conversion uses a virtual offset and dead-shares mechanism to prevent first-depositor inflation attacks, the standard ERC-4626 mitigation.

4.2 Risk tranching

Senior and junior tranches are held in separate share classes, with a subordination invariant enforced after every loss allocation:

```
juniorAssets >= seniorAssets * juniorRatioBps / 10000
```

When a default occurs, losses apply to junior capital first. A breach of the invariant triggers the spool into a `Paused` state.

4.3 Merkle-verified order metadata

Supporting documents—ERP snapshots, inspection reports, logistics records—are batched into Merkle trees. Only the Merkle root is stored on-chain; per-order leaves are published off-chain and verified on demand. This preserves verifiability while keeping storage costs low, following the pattern established by Centrifuge V3 and other production RWA systems.

5. Stablecoin Advance and Repayment Logic

Suppliers receive stablecoin advances against verified order NFTs. Repayment is automated as fulfillment progresses and buyer payments settle on-chain.

5.1 Advance calculation

```
advance = notional * advanceRatioBps / 10000 * scoreBps / 10000
```

`advanceRatioBps` is a per-vertical parameter and `scoreBps` is the latest attested Fulfillment Probability Score. The product form means a declining score linearly reduces available liquidity, producing the intended continuous-response behavior.

5.2 Repayment waterfall

1. Buyer payment is received by an escrow contract or a designated settlement address.

2. Principal is returned to the spool.
3. Accrued interest is distributed according to the tranche structure.
4. Any residual is released to the supplier.

The waterfall uses a pull-payment pattern so a failing recipient cannot block settlement.

6. Agent Execution Pipeline

The SPOOL agent executes a four-stage cycle: observe, decide, simulate, act. This is a pipeline with a hard gate between simulation and execution.

Stage	Input	Output	Failure behavior
Observe	Latest oracle attestations, on-chain state	Structured observation vector	If attestation is stale, halt
Decide	Observation vector, policy bounds	Proposed action + calldata	No action if outside allowlist
Simulate	Proposed calldata, Arbitrum fork of current state	Simulation trace, gas estimate, state diff	Abort if simulation reverts or diff exceeds bounds
Act	Simulation-approved calldata	Submitted transaction	Escalate to multisig if value above threshold

Simulation runs against a forked Arbitrum state pinned to the latest block. Foundry's fork primitives (`vm.createFork` , `vm.selectFork` , `vm.rollFork`) allow the agent to test the exact call it intends to submit against the same contract versions deployed on Arbitrum One.

```
function test_simulateAdvance_onArbitrum() public {
    vm.createSelectFork(vm.envString("ARBITRUM_RPC_URL"));
    uint256 orderId = _mintTestOrder();
    uint256 advance = spool.previewAdvance(orderId);
    vm.prank(agent);
    spool.fundOrder(orderId, advance);
    assertEq(spool.advances(orderId), advance);
}
```

State can change between simulation and inclusion, so the simulation gate is complemented by post-execution invariant checks rather than treated as a guarantee on its own.

7. Oracle Integration Interface

SPOOL consumes attested Fulfillment Probability Scores through a narrow interface that fails closed when attestations are absent.

```
interface IFulfillmentConsumer {
    function onScoreUpdate(
        bytes32 orderId,
        uint16  scoreBps,
        uint64  timestamp,
        bytes   calldata attestation
    ) external;
}
```

The callback verifies that `timestamp` is within a freshness window and that the attestation recovers to an authorized oracle signer before updating any order state. If verification fails, the callback reverts and the order remains in its prior state.

8. Arbitrum L2 Architecture: Cost, Finality, and Sequencer Risk

Arbitrum One is an optimistic rollup using the Nitro stack. Execution happens on L2, data is posted to Ethereum L1, and correctness is guaranteed by interactive fraud proofs with a challenge window. The choice of Arbitrum has direct consequences for SPOOL's cost model, finality assumptions, and agent behavior.

8.1 Cost model

Arbitrum fees have two components: L2 execution gas and L1 data posting. EIP-4844 blobs reduced L2 posting costs by 10–100× compared to calldata, with post-Dencun typical fees falling to fractions of a cent. Per-order state transitions and frequent oracle attestations are therefore economically viable.

Operation	Typical cost driver	Design
Mint order NFT	Storage writes	Pack state, score, and timestamps into one slot
State transition	Storage write + event	Event-based indexing; registry stores only latest
Oracle attestation	Calldata + signature verify	Batch multiple orders per attestation tx
Advance / repay	Token transfer + accounting	Pull-payment waterfall to avoid push failures

8.2 Finality model

Arbitrum exposes three confirmation levels. **Soft finality** (`latest`) is a sequencer promise, sub-second but reversible. **Parent-chain safe** (`safe`) means the batch is posted and the containing block has reached Ethereum's safe block—roughly 12–15 minutes, revertible only by a deep parent-chain reorg. **Parent-chain final** (`finalized`) means the batch has reached Ethereum's finalized block, revertible only if Ethereum itself reverts a finalized block. Each SPOOL action states which level it requires.

Action	Finality required	Rationale
Pause advance	Soft (sequencer, sub-second)	Risk-reducing; safe under any outcome
Reprice spread	Soft	Reversible; next tick can correct
Advance stablecoins	Parent-chain safe (~12–15 min)	Irreversible value transfer
Repayment settlement	Parent-chain final	Closes credit position

The native L1 withdrawal bridge carries a ~7-day optimistic challenge period. Small repayments settle entirely on Arbitrum; large repayments requiring L1 settlement either disclose the challenge period or route through a fast bridge with explicit counterparty risk disclosure.

8.3 Sequencer failure

Arbitrum's sequencer is currently operated by the Arbitrum Foundation with a decentralization roadmap. Sequencer unavailability is distinct from oracle feed loss and is handled separately.

- **Detection:** the agent runtime monitors Arbitrum block height; a stall beyond 30 seconds (roughly 120 missed blocks) triggers degraded mode.
- **Containment:** no new advances are proposed; outstanding positions remain open; submitted risk-reducing actions remain valid.
- **Recovery:** critical pause actions are force-included via the Arbitrum delayed inbox on L1.
- **Policy:** the policy engine records the maximum acceptable staleness of sequencer confirmation; beyond it, escalation triggers automatically.

8.4 Network allocation

Workload	Network	Reason
SPOOL credit contracts	Arbitrum One	Stylus for compute, Timeboost for agent latency
Consumer-facing UI, supplier portal	Base	Coinbase distribution, cheaper simple transfers
L1 settlement root	Ethereum	Canonical finality

Cross-chain state is reconciled explicitly: a supplier advancing on Arbitrum and receiving on Base implies a bridge hop that is accounted for in repayment timing and risk disclosure.

9. Stylus and the Rust Execution Path

Arbitrum Stylus introduces a coequal WebAssembly virtual machine alongside the EVM, allowing contracts written in Rust to compile to WASM and run alongside Solidity contracts sharing the same state tree.

9.1 What Stylus changes

Rust-compiled WASM contracts execute faster with significantly lower gas fees: 4–10× cost savings in AMM-style workloads and 100–500× cheaper memory for compute-intensive operations. Solidity remains appropriate for storage-heavy and integration-heavy contracts. The split is explicit: **Solidity for state, Stylus for compute**.

9.2 SPOOL components on Stylus

Component	Why Stylus
Advance calculation	Product of notional, ratio, and score across batched orders is compute-heavy
Score aggregation	Combining multiple oracle attestations per order benefits from cheaper memory and compute
Invariant checking	Post-execution verification across all spools is compute-intensive

9.3 Interoperability and safety

Stylus contracts call Solidity and vice versa without disrupting Ethereum equivalence. Rust's memory safety and the Stylus SDK's default-disabled reentrancy reduce common vulnerability classes; the newer ecosystem implies less battle-tested tooling, so Stylus modules receive additional audit coverage before mainnet deployment.

The order NFT, spool accounting, and access control remain in Solidity. Advance preview, score aggregation, and invariant verification run in Stylus. Cross-contract calls use the shared state tree, so no bridge is needed between the two VMs.

10. Security Model and Failure Modes

Failure	Detection	Containment	Recovery
Oracle attestation forgery	Signature recovery mismatch	Callback reverts; no state change	Rotate oracle signer via governance
Stale score	Timestamp outside freshness window	Advances paused	Oracle resumes publishing
Agent compromised	Proposed selector outside allowlist	Policy contract rejects	Rotate agent key

Simulation–execution divergence	State diff exceeds bound	Post-execution invariant check reverts	Manual review
Spool insolvency	Junior tranche breached	Spool paused	Governance recapitalization or wind-down
Sequencer stall	Block height stagnates beyond threshold	New advances halted; degraded mode	Force-include pause via L1 delayed inbox

Invariant checks are implemented as view functions that the agent calls after every action and that any external party can call to detect a breach.

11. Testing and Simulation Strategy

Layer	Scope	Tooling
Unit	Individual functions, state transitions	Foundry `forge test`
Integration	Spool + order NFT + oracle callback	Foundry with mock oracle
Fork	Real Arbitrum state, real stablecoins	`vm.createSelectFork(ARBITRUM_RPC_URL)`
Invariant	System-wide properties under fuzzing	Foundry invariant testing

Fork testing against Arbitrum One is particularly important because SPOOL integrates with deployed stablecoin contracts and oracle infrastructure outside the team's control. The runtime pins the fork URL to the deployment chain recorded in the policy engine; simulating against the wrong chain produces misleading state diffs.

12. Deployment and Observability

Deployment targets Arbitrum One for SPOOL contracts, Ethereum L1 for the settlement root, and Base for consumer surfaces. Foundry scripts with deterministic broadcast and on-chain verification provide reproducibility.

Observability covers both chain-specific and cross-chain signals: Arbitrum sequencer liveness, L1 finality lag, attestation freshness, and spool utilization per tranche. Alerts fire when any of these crosses a threshold, and the agent runtime treats sequencer staleness as a fail-closed trigger independent of oracle status.

> **Disclaimer.** This white paper is for informational purposes and does not constitute financial, legal, or investment advice. Digital assets and DeFi protocols carry significant risk, including total loss of capital.

© 2025 Ukiyo Network. SPOOL Protocol technical white paper, version 3.0.
Execution environment: Arbitrum One. Settlement root: Ethereum L1.