128.trade: A Smart-Contract Callable Trading Engine for Any Chain

128.trade Team

November 2025

Abstract

Decentralised Finance (DeFi) has grown into a multi-chain ecosystem of programmable financial primitives, yet high-performance trading engines, centralised or decentralised, remain isolated from smart contracts and inaccessible as callable infrastructure. This paper introduces `128.trade`, the first smart-contract callable trading engine designed for any blockchain, enabling CEX-grade spot and perpetual execution to be invoked directly from on-chain applications without bridging or network switching. By combining a deterministic CLOB engine, a secure cross-chain oracle–relayer gateway, and an asset-mapping framework that unifies liquidity across chains, `128.trade` transforms trading from an isolated venue into a programmable primitive for the entire DeFi stack. This integration opens a new design space for vaults, agents, routers and structured products, establishing a cross-chain execution layer that expands the reach, composability and performance of decentralised markets.

1 Introduction

Decentralised Finance (DeFi) now spans automated market makers, on-chain orderbooks, perpetual derivatives, cross-chain bridges, and strategy protocols. Despite this breadth, a fundamental limitation persists: **no high-performance trading engine is directly callable by smart contracts, nor is there a mechanism for cross-chain programmable access to deep liquidity**. Existing trading venues, whether decentralised (e.g., Hyperliquid, Aster) or centralised (e.g., Binance, Bybit), remain operationally isolated and cannot be integrated as programmable components within DeFi applications.

This disconnect imposes structural constraints on what DeFi can build. Smart contracts cannot open or manage positions on high-performance venues, cannot access unified liquidity beyond their native chain, and cannot compose trading logic into higher-order financial primitives. As a result, trading, one of the most fundamental financial operations, remains external to DeFi's programmable architecture.

1.1 Structural Problems in Today's DEX Architectures

Across contemporary decentralised exchange architectures, three structural limitations consistently emerge:

1. **Fragmented Liquidity and Execution Environments.** Each chain hosts its own liquidity pools and orderbooks. Pricing, depth, and execution quality vary across networks, and no mechanism exists for smart contracts to access deep liquidity outside their native chain.
2. **Non-Programmability of High-Performance Engines.** The most efficient trading engines, CEX-grade or DEX-grade CLOB systems, are not exposed as callable endpoints.

They cannot be invoked by arbitrary smart contracts, preventing trading from functioning as a composable, on-chain primitive.

3. **Chain-Local Execution without Cross-Chain Workflow Support.** Existing DEXes operate within single-chain execution environments. Multi-chain strategies require manual bridging or off-chain orchestration, blocking the emergence of cross-chain trading automation or integrated DeFi–DEX applications.

These are *structural* issues: they arise from architectural boundaries, not from particular implementations.

1.2 Evolution of Decentralised Exchange Architectures

The trajectory of DEX design reflects successive attempts to address liquidity and execution constraints:

- **DEX 1.0 — Automated Market Makers (AMMs).** Enabled permissionless swaps but suffer from slippage, impermanent loss, and limited expressiveness for derivatives or advanced order logic.
- **DEX 2.0 — On-chain Orderbooks and Perpetual Engines.** Achieved significant improvements in performance and capital efficiency. However, these systems remain chain-bound, offer non-shared liquidity, and cannot be invoked by arbitrary smart contracts.

Despite their contributions, both generations share a critical limitation: **they treat trading engines as venues, rather than as callable infrastructure**. Consequently, DeFi lacks a mechanism to integrate high-performance execution into its programmable and composable financial stack.

1.3 Why Smart-Contract Callability Matters

Smart-contract callability marks a category shift in how trading infrastructure can be consumed. If a high-performance engine were callable:

- DeFi protocols could embed spot, perpetual, or strategy execution directly into on-chain logic.
- Aggregators, vaults, structured-product protocols, and on-chain agents could access deep liquidity without bridging or network switching.
- High-performance DEX engines would no longer be isolated venues, but *programmable execution modules* integrated into the broader DeFi ecosystem.

In short, smart-contract callability connects DeFi with CEX-grade and advanced DEX-grade performance, unlocking an execution model that existing systems cannot support.

1.4 Our Solution: A Smart-Contract Callable Trading Engine Accessible from Any Chain

128.trade introduces the first trading engine designed from inception to be callable by smart contracts across heterogeneous chains. The system consists of a deterministic, high-performance matching engine coupled with a cross-chain oracle–relayer gateway that transports requests and delivers verifiable execution results back to the originating chain.

This architecture provides three key properties:

1. **High-performance deterministic execution** for spot and perpetual markets, with auditability and multi-node consensus.
2. **Programmable trading liquidity**, allowing any contract on any supported chain to open positions, place orders, subscribe to vaults, or execute strategies.
3. **Cross-chain accessibility**, enabling protocols to integrate trading operations without bridging assets or leaving their native execution environment.

This paper contributes:

1. The design of a **smart-contract-callable trading engine** capable of CEX-grade execution and deterministic state transitions.
2. A **cross-chain oracle–relayer gateway** supporting secure request forwarding, permissionless callbacks, and liveness guarantees.
3. An **asset-mapping and settlement framework** enabling shared liquidity across heterogeneous chains.
4. A roadmap toward **verifiable execution**, including TEE attestation and zero-knowledge proof generation for state transitions.

Beyond these technical contributions, 128.trade aims to *connect high-performance DEX infrastructure with the programmable composability of DeFi*, transforming trading from an isolated venue into a shared execution layer. This integration opens the path toward a broader vision: the convergence of liquidity, performance, and programmability across all chains, a foundation for the next generation of decentralised financial systems.

2 Vision

2.1 The Name and Its Significance

The name `128.trade` was chosen with deliberate intent. It draws inspiration from Massachusetts Route 128 (commonly "Route 128"), the highway encircling the greater Boston area that evolved from a regional transportation beltway into a symbol of technological progress, industrial clustering, and economic transformation. Once an ordinary ring road, Route 128 became the backbone of what was later called "America's Technology Highway," hosting one of the earliest and most influential high-technology ecosystems in the United States.

Historically, Route 128 did far more than shorten commutes. It connected research institutions, emerging technology companies, and industrial hubs, enabling knowledge flow, resource sharing and innovation at unprecedented scale. As laboratories, startups, venture investors and engineering talent concentrated along the corridor, the region shifted from a disparate set of towns into a coherent innovation ecosystem. Infrastructure catalysed economic behaviour: connectivity fostered collaboration; shared access enabled new industries; and the network itself reshaped the technological landscape of Massachusetts.

This history captures three principles foundational to our vision:

- **Connectivity.** Route 128 physically connected nodes that once existed in isolation. In doing so, it allowed information, talent and resources to flow in ways previously impossible.
- **Ecosystem Formation.** The corridor did not merely host activity; it enabled the emergence of an integrated technological ecosystem, where individual components—research centres, business parks, industry groups—could interoperate and reinforce one another.

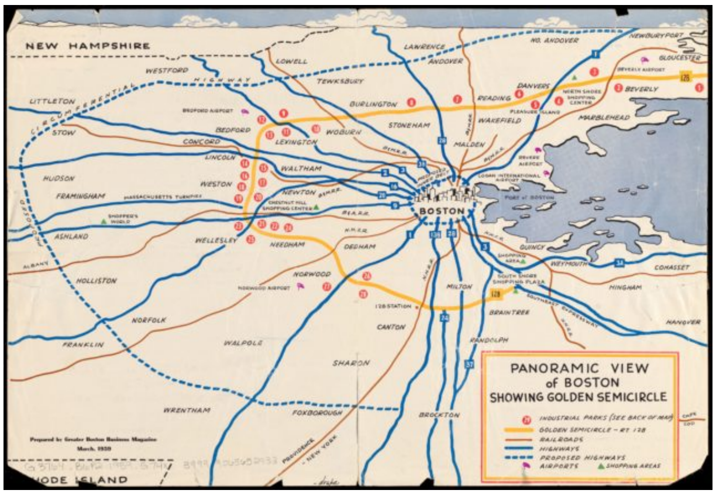

Figure 1: Massachusetts Route 128

- **Transformation.** Infrastructure, once deployed, altered the economic and cultural trajectory of an entire region. It shifted what could be built, how enterprises could grow and what forms of innovation were feasible.

We view `128.trade` through the same lens. Our goal is to build an infrastructure *corridor* for decentralised finance: not one carrying cars, but one carrying liquidity, state transitions, cross-chain execution and programmable trading workflows. In today's multi-chain environment, trading engines, liquidity pools, protocols and assets exist as isolated "towns" separated by chain boundaries, execution silos and incompatible interfaces. By providing a unified access layer and a callable high-performance trading engine, `128.trade` aims to link these previously disconnected components into a coherent, interoperable financial ecosystem.

Just as Route 128 enabled the rise of a high-technology economy by connecting institutions and industries, `128.trade` aspires to enable a high-functioning trading economy in DeFi—one in which chains, protocols and liquidity sources interoperate seamlessly, and where composable trading becomes a foundational primitive for the next generation of decentralised finance.

2.2 Our Vision for the Future

Our long-term vision is to establish **a programmable trading infrastructure layer that underpins all of DeFi across all chains**. This layer transforms trading from an isolated venue-level activity into a shared, permissionless and composable execution fabric, accessible to any protocol, on any chain, at any time. Within this vision:

- **Deep liquidity becomes universally accessible.** Smart contracts on any supported chain can call the 128.trade engine directly, accessing liquidity and execution quality traditionally confined to specialised high-performance venues. Liquidity is no longer siloed by chain boundaries, but forms a shared resource across the ecosystem.

- **Every tradable object becomes a programmable financial primitive.** Spot tokens, perpetual positions, vault shares and strategy outputs can be represented, transferred, composed and settled across chains with deterministic semantics. Assets evolve from chain-local representations into *interoperable primitives* that participate in a unified financial substrate.

- **Contracts become the primary economic agents.** Through standardised gateway interfaces, smart contracts themselves can initiate, manage and automate trading workflows—opening positions, executing hedges, rebalancing portfolios or constructing structured strategies. Trading becomes as natively programmable as swaps, lending or staking.
- **Trading integrates as a plug-and-play primitive for every protocol.** Aggregators, fund managers, structured-product protocols, automated agents and routers can incorporate the 128.trade engine as a backend module—much like protocols today integrate oracles, data feeds or messaging layers. The engine becomes an infrastructural component rather than a venue.

In essence, we envision 128.trade as *the connective tissue of decentralised markets*: an execution layer that unifies chains, assets and protocols into a coherent, high-performance and interoperable financial ecosystem. By enabling composable access to deep liquidity and deterministic execution across chains, 128.trade aims to provide the foundational infrastructure for the next generation of programmable finance.

3 Architecture Overview

This section presents a high-level overview of the `128.trade` architecture. The system is structured around two foundational modules—(i) the *Trading Engine*, which serves as the deterministic execution layer, and (ii) the *Oracle Services* module, which forms the access and cross-chain messaging layer. Together, these layers enable two complementary modes of interaction: high-performance API access and chain-agnostic smart-contract programmability. This duality allows `128.trade` to operate simultaneously as a next-generation trading venue and as a universal execution primitive for DeFi.

3.1 Core Modules

Trading Engine (Execution Layer). At the core of the architecture is a high-performance central-limit-order-book (CLOB) style engine capable of deterministic matching, margining, funding, liquidations and state transitions. The engine exposes standard low-latency APIs (REST, WebSocket or equivalent), enabling integration patterns familiar to high-performance trading venues. Applications, market makers, automation systems or exchanges can place and cancel orders, open and close positions, and query market or account state directly. This mode delivers CeFi-grade throughput and serves users who require real-time execution, algorithmic trading or direct venue connectivity.

Oracle Services (Access & Messaging Layer). In parallel, the `128.trade` architecture provides a trust-minimised cross-chain access layer that makes the trading engine callable from any smart contract. Gateway contracts deployed on supported chains expose functions such as `requestPlaceOrder`, `requestOpenPosition` or `queryPosition`. These calls are observed by an off-chain relayer network, which packages and forwards the requests to the engine. After execution, the relayers return results, together with threshold signatures, TEE attestations or future proof systems, via callbacks to the originating chain. This module elevates the trading engine from an isolated venue into a composable backend service, enabling trading to be invoked programmatically across heterogeneous chains without bridging or network switching.

3.2 Dual Service Modes

A defining feature of `128.trade` is that the same deterministic engine and unified liquidity pool are accessible through two distinct service modes:

Mode 1 — Direct API Access ("Engine-Native"). This mode provides full-performance venue-style interaction for high-throughput clients. It is optimal for traders, exchanges, liquidity managers and automation systems that require tight latency and granular execution control. It reflects the engine's role as a high-performance trading venue.

Mode 2 — Oracle-Based Smart-Contract Access ("Any-Chain Programmability"). This mode enables any smart contract on any chain to access the engine as a programmable primitive. Trading operations—spot orders, perpetual positions, vault subscriptions or strategy execution—can be initiated entirely on-chain and resolved via verifiable callbacks. It reflects the engine's role as a composable execution layer for DeFi.

These dual modes demonstrate a key property of the system: *one engine, two access paths, shared liquidity, different consumers.* `128.trade` thus serves both CeFi-level throughput and DeFi-level composability, positioning it as a universal trading infrastructure layer rather than a single-venue DEX.

3.3 End-to-End System Flows

The two interaction pathways operate as follows.

Direct Engine Access.

1. A client connects to the engine's API interface.
2. It issues commands such as `placeSpotOrder`, `placePerpOrder`, `cancelOrder`, or `openPosition`.
3. The engine processes the request through matching, risk checks, margin accounting, funding and settlement.
4. A synchronous or asynchronous response is returned with execution results, identifiers and updated state.

Smart-Contract Access via Oracle Services.

1. A contract on chain X calls a gateway function (e.g., `requestOpenPosition(params)`).
2. The gateway emits an event encoding request metadata.
3. Relayers observe the event and submit the request to the engine.
4. The engine processes the request deterministically and outputs the result.
5. Relayers return the result, signed or accompanied by proof data, via a callback (e.g., `callbackOnOpenPosition(result, sig)`).
6. The gateway verifies the proof or signature, applies idempotency checks and materialises engine-native state on-chain (e.g., minting a position NFT or recording an order ID).

This pathway allows smart contracts on any chain to access `128.trade` natively, without bridging assets or changing networks, thereby enabling fully programmable cross-chain trading workflows.

3.4 Key Architectural Considerations

- **Deterministic Execution.** The engine operates as a deterministic state machine, enabling auditability, replayability and verifiable state transitions.
- **Secure Gateway Semantics.** Gateway contracts ensure safe request/response handling through signature validation, proof verification, replay protection and strict idempotency.
- **Robust Cross-Chain Messaging.** The relayer network handles asynchronous requests, multiple-chain monitoring and verifiable callback delivery, forming a reliable messaging substrate.
- **Asset Mapping.** Engine-native objects such as perpetual positions or vault shares can be represented as on-chain tokens or NFTs, enabling composable use across chains.
- **Cross-Chain Composability.** Protocols can integrate trading directly into higher-level logic—automated hedging, strategy rebalancing or multi-chain structured products—treating `128.trade` as a programmable execution layer.

4 Trading Engine

The core of `128.trade` is its Trading Engine, a high-performance, deterministic execution environment that supports spot markets, perpetual contracts, hierarchical accounts and sub-accounts, and verifiable state transitions. While the design draws inspiration from modern high-performance venues such as `Hyperliquid` and `Aster`, the engine extends these paradigms with a Byzantine-Fault-Tolerant (BFT) node network, deterministic audit logs, and a progressive roadmap toward zero-knowledge verification of execution.

4.1 Accounts, Sub-Accounts, and Market Support

The engine supports spot and perpetual markets and provides a hierarchical account structure. Sub-accounts allow for logical separation of strategies or risk pools under a master account. Clients may:

- place or cancel spot orders,
- open or close perpetual positions,
- query market, account, or position states,
- execute algorithmic or automated trading strategies.

These capabilities are available uniformly regardless of how a request enters the engine (API or on-chain), ensuring consistent semantics across access modes.

4.2 Request Ingestion and Authentication

Although Section 3 describes the two high-level access modes (direct API access and oracle-based contract access), the Trading Engine itself abstracts over these differences. Internally, every action is processed as a canonical *engine request*. The ingestion pipeline verifies request provenance, authenticity, and sequencing through two complementary validation mechanisms:

API-Origin Requests (Signature-Based Authentication). Clients integrating via REST/WebSocket must sign each operation using registered keypairs. The engine validates:

- request signatures and key ownership,
- nonces and replay protection,
- account-level permissions and role scopes,
- rate limits and throttling rules.

Only authenticated requests enter the engine's canonical request queue.

On-Chain-Origin Requests (Gateway-Provenance Verification). Requests submitted via gateway contracts undergo an alternative verification pipeline. The engine validates:

- the gateway contract address and chain identifier,
- block-header inclusion or light-client verification of emitted gateway events,
- relayer signature bundles or TEE attestations,
- canonical encoding of parameters and idempotency markers.

This ensures that cross-chain requests reflect genuine on-chain intent and prevents forged or replayed messages.

Unified Request Queue. After authentication, all requests—API or on-chain—enter the same deterministic request queue. This guarantees that:

- identical execution logic applies to both modalities,
- liquidity and state transitions are fully shared,
- no ordering or settlement differences arise between access paths.

4.3 BFT Multi-Node Network and Deterministic Execution

To provide decentralised integrity and auditability, the Trading Engine executes within a distributed BFT network of matching/risk nodes. Each state transition (order placement, matching, fills, margining, funding payments, liquidations) is processed deterministically and committed to an append-only execution log. Because the engine is deterministic, any auditor can replay the log from genesis and reconstruct the full state exactly.

This deterministic audit trail is fundamental for transparency, interoperability and eventual zero-knowledge verification.

4.4 Enhancement Roadmap: TEE and ZK Proofs

The engine supports a long-term trust-minimisation trajectory:

Phase 1: TEE-Hardened Execution. Matching nodes may run inside hardware-secured TEEs. Remote attestation allows the Oracle Services module and external verifiers to confirm that the correct engine binary and configuration are executing.

Phase 2: ZK-Proof Generation. Over time, the engine will output zero-knowledge proofs for each batch of trades, demonstrating that:

> "Given the previous state and the submitted orders, the new state was computed exactly according to public rules."

Such proofs enable external systems, including on-chain contracts, to verify correctness without trusting operator nodes.

4.5 Risk Engine, Liquidation, and Auditability

The risk module continuously evaluates each account's collateral ratio, exposure, leverage and funding accruals. When thresholds are breached, deterministic liquidation procedures are triggered and committed to the execution log. Because the log is append-only and replayable, it supports:

- forensic analysis,
- regulatory or protocol-level audits,
- state reconstruction by new nodes,
- verifiable cross-chain settlement via Oracle Services.

5 Oracle Services

The Oracle Services module constitutes the access and cross-chain messaging layer of `128.trade`. Because smart contracts cannot interact directly with an off-chain high-performance engine, this module provides the infrastructure required to transport requests from heterogeneous chains to the Trading Engine and to return authenticated, verifiable execution results back on-chain. It consists of four coordinated subcomponents, gateway contracts, a decentralised relayer network, callback and proof-verification logic, and the asset-mapping interface—which together expose the engine as a contract-callable execution primitive for the multi-chain DeFi ecosystem.

5.1 Overview and Design Goals

The Oracle Services layer is designed around four primary goals:

- **Any-chain programmability.** Smart contracts on heterogeneous chains must be able to invoke the Trading Engine as if it were a native module, without introducing new trust assumptions or bespoke bridging interfaces.
- **Verifiable correctness.** Returned results must be accompanied by proof bundles that allow the gateway contract to verify that off-chain execution was performed faithfully and deterministically.
- **Deterministic and replay-safe integration.** Each request is uniquely identified, idempotent and safely sequenced, ensuring consistent state updates across chains.
- **High throughput and asynchronous workflows.** The messaging layer must support concurrent requests, pipelined callbacks and scalable multi-chain operation.

To realise these goals, Oracle Services comprise four coordinated submodules:

1. **Gateway Contracts** (on-chain access layer),

2. **Relayer Network** (off-chain messaging and validation layer),
3. **Callback & Proof Verification Logic** (on-chain authenticity enforcement),
4. **Asset Mapping & Settlement Interface** (representation and settlement of engine-native state across chains).

5.2 Request Lifecycle: From On-Chain Intent to Verified Execution

The end-to-end lifecycle linking an on-chain contract to the Trading Engine proceeds through the following verifiable stages:

1. **On-chain request initiation.** A contract on chain X calls a gateway `request()` function (e.g., `requestOpenPosition`), which generates a unique `requestId`. The gateway emits an event encoding parameters, chain context and caller metadata.
2. **Relayer event detection and provenance validation.** The decentralised relayer network monitors gateway events, verifies origin-chain metadata and optional light-client proofs, and constructs a canonical request payload for off-chain execution.
3. **Delivery to the Trading Engine.** Relayers submit the validated request to the engine through authenticated channels. The engine authenticates the message and inserts it into its deterministic execution queue.
4. **Engine execution and result generation.** The Trading Engine processes the request, matching, margining, funding, liquidating or settling as appropriate, and outputs a deterministic execution result and associated batch identifier.
5. **Proof generation and attestation.** Relayers collectively generate a *proof bundle* attesting to the correctness of the engine result. This bundle may include threshold signatures, TEE attestation metadata, or future ZK proofs generated from the engine's deterministic audit log.
6. **Callback delivery to chain X.** A relayer (or any permissionless actor) submits the proof-backed result to the gateway's `callback()` function, along with identifiers such as `requestId` and `batchId`.
7. **Gateway verification and state materialisation.** The gateway verifies the proof bundle, enforces idempotency and replay protection, and applies the result on-chain, minting or transferring position NFTs, updating balances or notifying the invoking contract.

This lifecycle ensures that the Trading Engine can be invoked from any chain while maintaining correctness, authenticity and deterministic reconciliation.

5.3 Gateway Contracts

Gateway contracts form the canonical on-chain interface to `128.trade`. They expose a family of deterministic functions that encode decentralised intent:

- **request functions** such as `requestPlaceOrder`, which allow smart contracts to initiate engine-side actions;
- **callback functions** such as `onOrderResult`, through which verifiable engine results and proof bundles are delivered back on-chain.

The gateway contract enforces strict checks: correct request identifiers, replay-protection via nonces or requestIds, verification of relayer signatures or threshold signatures, and (where applicable) verification of state roots or proofs from the underlying bridge or light-client. This design ensures that only authorised and valid results are applied on-chain.

5.4 Relayer Network

The decentralised relayer network forms the off-chain messaging and validation layer responsible for transporting requests to the Trading Engine and returning authenticated results. Its responsibilities include provenance verification, request canonicalisation, delivery to the engine, and generation of cryptographic attestations that certify the correctness of the engine's deterministic execution.

- **Event observation & provenance validation.** Relayers monitor gateway events across supported chains, verify origin-chain metadata, and (when available) validate light-client or state-root proofs to ensure that the request originated from the correct chain and gateway.
- **Canonical request construction.** Relayers normalise request parameters, constructing a canonical payload that the Trading Engine can authenticate and execute deterministically.
- **Authenticated request delivery.** Validated requests are forwarded to the Trading Engine via authenticated channels. Once inside the engine, all requests—API or on-chain—enter the same deterministic execution queue.
- **Execution-proof generation.** After the engine produces a deterministic result, relayers jointly generate an *execution proof bundle* that attests to correctness. The proof bundle may be instantiated using multiple verification mechanisms:
  - *Threshold-signature attestation* (e.g., M-of-N signatures certifying agreement on engine output);
  - *TEE-based remote attestation* (proving the result was generated by an audited engine binary running in a secure enclave);
  - *Light-client provenance proofs* (showing the request was included in the canonical chain state);
  - *Zero-knowledge proofs* (future extension), generated from the engine's deterministic audit log to prove state transitions without revealing internal data.
- **Callback preparation.** Equipped with a valid proof bundle, any relayer (or permissionless actor) can construct and submit a callback transaction to the destination chain.

The design follows principles from modern off-chain reporting networks such as Chainlink OCR, but extends them: relayers here certify not just observed data, but the correctness of state transitions performed by a high-performance deterministic execution engine.

5.5 Callback & Proof Verification Logic

Once a callback transaction carrying an execution proof bundle reaches the gateway contract, verification proceeds entirely on-chain. This separation provides a clean security boundary: relayers produce proofs; gateways verify them.

- **Proof-bundle verification.** The gateway verifies the cryptographic proof bundle attached to the execution result. Supported proof types include:
  - threshold-signature sets (certifying multi-relayer agreement),
  - TEE attestation metadata (certifying the engine binary and execution environment),

    - zero-knowledge proofs (certifying state-transition correctness directly from deterministic logs).

- **Request and batch integrity.** The gateway checks the `requestId`, `batchId`, and canonical payload fields to ensure the callback corresponds exactly to a previously issued request.

- **Replay protection & idempotency.** State transitions are applied exactly once. Any callback failing nonce or idempotency checks is rejected.

Upon successful verification, the gateway materialises the validated result on-chain—for example, minting or updating position NFTs, transferring vault shares, or notifying the invoking contract. This ensures that on-chain state transitions faithfully reflect authenticated deterministic execution of the Trading Engine, even in adversarial conditions.

5.6 Asset Mapping & Settlement Integration

The Oracle/Relayer layer also interacts with the asset-mapping system of `128.trade`. For example:

- Spot assets from various chains may be mapped into the engine as virtual liquidity endpoints; the gateway contract and relayer network ensure the transfer of asset-mapping instructions and settlement notifications.

- Perpetual positions (represented as NFTs) may be created or transferred via callback events, with proofs of execution delivered through the oracle network.

- Vault shares (ERC-20 tokens) may be minted on chain X after the trading engine executes a vault-subscription request; the relayer network ensures the correct state has been achieved before the minting callback.

5.7 Safety and Liveness Guarantees

To ensure robust operation of the oracle/relayer subsystem in `128.trade`, we design specific mechanisms that address both **safety** (i.e., "bad things do not happen") and **liveness** (i.e., "good things eventually happen") in the context of cross-chain request/response flows. In classical distributed systems literature, safety and liveness are fundamental properties of protocols.

Fully Permissionless Callback Mechanism Even if the relayer set becomes unavailable or incapacitated (for example by network faults, censorship or operator failure), users on any chain can still complete the request-to-engine workflow and apply the result on-chain. Concretely: when a smart contract on chain X submits a `request()` via the gateway contract, the matching/trading engine executes the request and emits a signed or verifiable proof of execution (e.g., threshold-signature, ZK-proof or attested state root). A *third-party user or contract* may then invoke the `callback()` on chain X by presenting the execution result and proof directly, without relying on any designated relayer. This design ensures that funds and positions are not locked indefinitely due to relayer downtime: the safety property holds because only valid proofs are accepted, and liveness is improved because any actor can progress the callback.

Emergency Exit via Fraud-Proof Roll-up Escape Hatch To further guarantee liveness and protect users in worst-case failures of the engine or the relayer network, we incorporate an emergency escape mechanism akin to optimistic roll-up fraud-proof systems. When a user has placed an order on chain X and the engine or relayer network fails to deliver a callback within a predetermined *challenge period*, the user may initiate an on-chain "challenge" via the gateway contract:

1. The user submits their `requestId`, evidence of submission (e.g., event log, timestamp) and a proof that no valid callback has arrived within the window.
2. If the engine/relayer system does not supply a valid proof of execution within the challenge period, then the gateway contract allows the user to withdraw or reclaim funds/assets through a fallback path.

This mechanism ensures that *liveness* is preserved: honest users will not have funds indefinitely locked; and *safety* is preserved because the escape path is only triggered if the system fails to deliver a valid execution proof, thereby avoiding unjustified exits.

Combined Security Guarantees

- **Safety**: Only authorised, correctly signed/attested results from the trading engine are accepted. Replay attacks are prevented through unique `requestId` and batch identifiers; signature schemes or proof verification ensure authenticity and non-tampering.
- **Liveness**: The permissionless callback path and the fraud-proof escape mechanism guarantee that, under benign conditions or worst-case failure conditions, users retain the ability to complete their requests or withdraw their assets.
- **Hybrid trust-minimisation**: While relayers operate for performance and convenience, trust assumptions are minimised. The design does not depend exclusively on a single party or relayer set to sustain functionality.

In combination, these mechanisms extend the reliability of the oracle-relayer subsystem beyond typical designs and align with best practices in oracle/liveness research (e.g., ensuring freshness, fallback paths, decentralised relayers).

6 Applications and the New Design Space

The introduction of a smart-contract callable, high-performance trading engine opens a design space that has not previously existed in decentralised finance. Traditional DEXs such as Hyperliquid and Aster expose liquidity but not programmable execution; CEXs expose execution but cannot be invoked by smart contracts; and isolated L2 order-book venues cannot be composed with DeFi protocols. `128.trade` transforms trading into a *programmable primitive* that contracts can call directly, enabling new architectures in strategy automation, asset management, derivatives, and multi-chain coordination.

6.1 Trading as a Programmable Primitive

`128.trade` elevates trading from a user-driven activity to a system-level operation accessible by any smart contract. This shift introduces three fundamental changes:

- **Contracts initiate trades, not only users.** Trading actions become part of protocol logic—automated, verifiable, and permissionless.
- **Liquidity becomes an underlying resource.** Protocols across chains access deep and shared liquidity without fragmentation.
- **Execution becomes programmable.** Deterministic execution and verifiable callbacks allow protocols to depend on off-chain trading as if it were on-chain computation.

6.2 Foundational Financial Primitives

`128.trade` provides a set of *financial execution primitives* that function as "syscalls" for higher-level protocols:

- **Spot Execution Primitive.** Smart contracts can place, cancel, or modify spot orders using a full suite of exchange-grade order semantics: market orders, limit orders, stop-loss and take-profit conditions, time-in-force parameters, and algorithmic order types such as TWAP slicing or adaptive execution. These actions are processed through deterministic matching and settlement logic, enabling protocols to implement execution behaviour that is impossible in AMMs, where liquidity curves constrain price formation and order types are effectively non-expressive.
- **Perpetual Execution Primitive.** Contracts may open, close, or modify perpetual positions with margining, funding payments, and liquidation rules enforced by the engine's deterministic pipeline. The primitive allows protocols to construct programmable leveraged exposures, hedges, rolling strategies, or delta-adjusted portfolios. Conditional order types and automated position-management logic (e.g., stop-loss triggers, trailing stops, or dynamic leverage modulation) can be fully encoded in on-chain strategy logic.
- **Unified Margin & Subaccount Primitive.** Smart contracts can manage multiple subaccounts, each with isolated margin, PnL, and liquidation boundaries. Subaccounts serve as strategy containers: a protocol may create separate subaccounts for hedging, basis trading, yield enhancement, or user-specific exposures. Because subaccounts can be controlled programmatically, they enable portfolio-level reasoning—contracts can rebalance exposures, shift margin between strategies, or execute multi-leg workflows across chains while preserving risk isolation.
- **Vault Subscription Primitive.** Vaults or structured-product protocols may subscribe to engine-side strategies and mint ERC-20 vault shares backed by deterministic execution. Strategies may include mean-reversion, grid strategies, delta-neutral or basis arbitrage, trend-following, or adaptive market regimes. Execution happens on the Trading Engine, while vault logic, deposit, withdrawal, share accounting, and strategy parameters, remains fully on-chain. This primitive enables a new class of multi-chain asset managers whose strategies rely on high-performance execution rather than AMM-based heuristics.
- **Asset Mapping Primitive.** The Trading Engine exposes a universal asset-mapping layer that materialises any engine-native object, spot balances, perpetual positions, funding portfolios, vault shares, or entire subaccounts, as on-chain financial assets. These representations may take the form of ERC-20, ERC-721, or ERC-1155 tokens, enabling fungible exposures, position-specific ownership, or portfolio-level bundles. Once mapped on-chain, such assets can be transferred, collateralised, pledged, fractionalised, or integrated into lending, derivatives, or structured-product protocols. In effect, `128.trade` transforms all tradable exposures into programmable financial primitives for the broader DeFi stack.

Together, these primitives constitute a general-purpose financial execution layer: contracts express intent, the engine performs deterministic trading, and assets become first-class programmable objects within the multi-chain DeFi ecosystem.

6.3 A New Paradigm of Composable DeFi Applications

By transforming trading into a contract-callable primitive, `128.trade` does not simply expand the existing DeFi design space—it enables a fundamentally new paradigm in which execution, liquidity, risk transformation and asset representation become fully programmable. This section outlines several application directions that illustrate how `128.trade` interacts with the broader DeFi ecosystem and creates new possibilities that were previously unattainable.

(1) Higher Capital Efficiency Through Universal Asset Mapping. Because any engine-native object—spot balances, perpetual positions, vault shares, or even entire subaccounts—can be materialised as on-chain assets, `128.trade` enables unprecedented capital efficiency. Spot exposures become ERC-20 tokens, perpetual positions become ERC-721 or ERC-1155 tokens, and an entire subaccount may be represented as a portfolio-level asset capable of carrying margin, PnL, and risk parameters. These mapped assets can be supplied to lending protocols, deposited in AMMs, collateralised in derivatives platforms, or used as LP/ staking objects. *All tradable exposures become composable DeFi objects*, allowing capital to simultaneously participate in trading, collateralisation, yield strategies and cross-chain structured portfolios. This breaks the long-standing divide between "execution capital" and "DeFi capital", enabling a unified and far more efficient financial ecosystem.

(2) A More Expressive and Efficient Trading System for DeFi. `128.trade` supports exchange-grade order semantics: market orders, limit orders, stop-loss and take-profit triggers, time-in-force instructions, and algorithmic execution such as TWAP or adaptive slicing. Protocols can combine these order types to express complex behaviours spanning spot markets, perpetuals, vault shares, and strategy-level exposures. Because all liquidity is consolidated within a unified trading engine, applications can access deeper and more consistent liquidity across chains, and DeFi aggregators can route orders into a single high-performance execution layer. This creates a radically more efficient trading environment compared to AMMs or isolated order-book venues, enabling precision execution for strategies, vaults, funds and automated agents.

(3) Improving and Redefining Existing DeFi Protocols. As execution becomes programmable and assets become universally composable, existing DeFi protocols can evolve far beyond their current architectural constraints. The examples below illustrate how integrating `128.trade` transforms the behaviour, efficiency, and risk model of today's core DeFi primitives.

- **(3.1) The Evolution of AMM DEXs.** AMMs such as Uniswap v4 can integrate `128.trade` through hooks to perform automated hedging on perpetual markets. An LP position that would traditionally experience impermanent loss can dynamically open offsetting perp positions, neutralising directional risk. AMMs thus evolve from passive liquidity curves into *hybrid AMM–CLOB systems* with dramatically improved capital efficiency and risk-adjusted returns.
- **(3.2) Lending Protocols as Intelligent Risk-Management Banks.** Borrow/lend systems like Aave can directly invoke programmable hedging or conditional execution on the Trading Engine. Instead of relying solely on over-collateralisation and liquidators, lending markets can place automated stop-loss orders, delta hedges, or conditional sells to manage borrower risk—improving collateral efficiency, reducing systemic volatility, and enabling real-time portfolio-level risk automation.

These examples represent only initial steps in how existing DeFi systems may be redesigned when execution becomes a programmable primitive; many further integrations, across derivatives, stablecoin systems, asset managers, and liquidity networks, remain unexplored.

(4) Entirely New Protocol Classes Enabled by Programmable Execution. Beyond improving existing protocols, `128.trade` enables wholly new categories of applications that were previously impossible because no system combined high-performance execution with smart-contract programmability. The examples below illustrate how new financial architectures emerge once trading, asset representation, and cross-chain workflows all become programmable.

- **(4.1) Fully On-Chain Structured Products and Autonomous Funds.** Vaults become fully on-chain hedge funds: strategy logic is encoded in smart contracts, execution is performed deterministically by the Trading Engine, and vault shares are minted as ERC-20 tokens. Strategies such as mean-reversion, grid strategies, delta-neutral hedging, basis trades, trend-following, and regime-adaptive models can all be implemented without off-chain intermediaries. Every user can create their own fully on-chain fund, backed by verifiable, deterministic execution.
- **(4.2) On-Chain Autonomous Agents (AI or Algorithmic).** AI agents, LLM-driven traders, or algorithmic controllers can execute strategies, arbitrage across chains, manage vault portfolios, or rebalance structured products autonomously. Because execution correctness is guaranteed by proof-backed callbacks, agents can safely manage significant capital with deterministic guarantees. This marks the emergence of *on-chain autonomous traders and AI-powered financial DAOs*, a protocol category previously impossible due to the lack of programmable high-performance execution.

Together, these application directions illustrate a new DeFi paradigm: trading becomes a universal execution primitive, assets become composable financial objects, and protocols gain the ability to express complex, cross-chain trading logic with deterministic guarantees. `128.trade` thus forms the execution backbone for a more unified, efficient and expressive multi-chain financial ecosystem.

6.4 Why These Applications Were Previously Impossible

The design space unlocked by `128.trade` has remained inaccessible in prior architectures because the two dominant models in decentralised and centralised trading each lack a critical capability.

(1) AMMs provide programmability but lack expressive execution. AMMs can be composed by smart contracts, but their financial expressivity is fundamentally limited:

- most AMMs cannot support perpetual futures or margining;
- order types such as limit orders, stop orders, or conditional execution are not natively representable;
- liquidity is passive and cannot express dynamic strategy logic or risk management.

Thus, while AMMs are programmable, they cannot serve as a general-purpose execution substrate for financial applications.

(2) CEXs and order-book DEXs support rich execution but cannot be composed. Centralised exchanges and high-performance order-book DEXs offer the full range of financial operations—spot, perpetuals, leverage, advanced order types—but:

- they remain isolated venues with no smart-contract callable interface;
- they cannot be invoked by DeFi protocols or integrated into on-chain workflows;
- their liquidity and execution cannot participate in cross-chain composability.

These systems provide execution richness but cannot act as infrastructure for decentralised programmability.

A structural gap. For over a decade, programmable systems lacked expressive execution, and expressive systems lacked programmability. As a result, an execution layer for DeFi—one that contracts could call directly—did not exist.

How `128.trade` resolves this. `128.trade` is the first system to combine:

- the *programmability and composability* characteristic of on-chain primitives, and
- the *rich execution semantics and performance* characteristic of advanced order-book venues.

This unification allows trading to function as a programmable primitive for the first time, enabling application classes that were structurally impossible in prior architectures.

7 Additional Features

7.1 DEX Aggregator

Liquidity in the decentralized-exchange (DEX) ecosystem is increasingly fragmented across numerous independent venues and siloed asset pools. Traders face sub-optimal routing, elevated slippage, and constrained asset coverage when relying on a single DEX. Recent research confirms that decentralized-exchange aggregators play a crucial role in mitigating these inefficiencies by intelligently routing trades across multiple sources and thereby improving execution quality.

In the context of `128.trade`, we propose to embed a *DEX aggregator feature* directly within the trading-engine side, complementing the core high-performance CLOB engine and the cross-chain gateway infrastructure. This design enables the engine not only to serve its native liquidity, but also to interface with external DEXs (and even centralized-exchange orderbooks) to improve overall trade execution, especially for large volumes, niche pairs or fragmented markets.

Architecture & Technical Design

- **External liquidity integration**: The engine exposes an interface to accept external routing orders or sub-orders to other venues. From the engine's viewpoint, these external venues appear as additional "liquidity legs" that can be selected by the aggregator logic when optimal.
- **Smart-order routing (SOR) logic**: The aggregator module analyses multiple trading paths—including native engine liquidity and external venues—by evaluating depth, fee structure, expected slippage, gas/settlement cost and chain context. It then splits or directs orders along the optimal route.
- **Secure routing execution**: To ensure funds safety when interacting with external venues, the aggregator feature utilises *threshold ECDSA / multi-signature protocols* for control of funds dispatch to external venues. It may also leverage *Trusted Execution Environments (TEEs)* and *zkTLS-style proofs* to guarantee that node-operators cannot arbitrarily move assets. Research shows that dynamic threshold ECDSA architectures are viable for custodial and routing infrastructure.
- **Verification of external execution results**: After routing and execution, the aggregator receives proof or signed results from the external venue. These results are then subject to integrity checks (signature thresholds, TEE attestations, optionally light-client verified state roots) before the final settlement is applied on-chain or in the engine state.

- **Composable pooling of liquidity**: By incorporating external venues, `128.trade` effectively becomes a *virtual liquidity super-source*, thereby allowing aggregators, routers or smart-contracts on any chain to route into our engine and receive better execution than any single siloed DEX.

7.2 Fund Management & Smart-Contract Governance

The market for algorithmic or strategy-based trading in DeFi remains significantly underserved. Existing public funds or strategy products often employ relatively simple methodologies and provide limited transparency. At the same time, many hedge-fund-style offerings remain opaque, suffer from weak gate-keeping (for example poor holdings disclosure, potential self-dealing), and are inaccessible to smaller investors. In the context of 128.trade, we address these issues directly by embedding fund-management logic into smart-contracts and introducing enforceable constraints that align manager incentives with investor protection.

Smart-Contract-Governed Fund Structures Within the 128.trade environment, fund creators deploy smart contracts that define the operational rules of each strategy or fund. Critical parameters are encoded, including: deposit and withdrawal conditions, revenue-share and performance-fee models, permissible asset classes and trading instruments (for example spot assets, perpetual contracts), maximum position sizes or leverage, delta-neutral constraints (if appropriate), and other risk-control rules. Because these mechanics are enforced on-chain, the fund creator is unable to deviate from the defined rules. The governance logic thereby mitigates common risks such as self-dealing, fund drift or hidden holdings, and enhances transparency and auditability for all participants.

Strategy Library and Tokenised Participation 128.trade also supports a built-in library of trading strategies (for example: funding-rate arbitrage, price-spread trades, cross-DEX arbitrage) which can be adopted directly by users or further customised by strategy authors. Each fund or strategy issues tokenised shares (e.g., ERC-20 tokens) representing participation in the strategy. Investors on any chain can subscribe, redeem or trade these tokens. The alignment is improved: strategy authors earn only under prescribed performance conditions, and the smart-contract logic ensures that revenue-sharing and redemptions follow the pre-defined rules.

Risk-Control, Transparency and Composability Smart-contract enforcement allows for granular control: holdings limits, on-chain restrictions to mitigate front-running, predefined strategy-scope limits, and continuous real-time visibility into fund state. This level of transparency and governance is rare in traditional hedge-fund structures, but achievable in a DeFi-native framework. Integrating with 128.trade's cross-chain architecture, funds and strategies become composable primitives: they can be embedded into other protocols, used as collateral, or layered into structured products on any supported chain.

7.3 Privacy Protection

To safeguard both strategy confidentiality and user trust, 128.trade supports private transaction flows combined with on-chain verifiable compliance. For example, trades can be routed through dark-pool-style execution or private transaction pools (thus reducing MEV and front-running exposure). Following execution, strategy details remain hidden, yet the smart contracts verify compliance with predefined rules (via Trusted Execution Environments (TEEs) or zero-knowledge proofs (ZKPs)). As a result, investors obtain full transparency into rule enforcement, ensuring custody and execution integrity, while strategy providers retain their proprietary edge with preserved privacy.

8 Conclusion

This whitepaper has presented 128.trade as a foundational infrastructure layer for DeFi: bridging high-performance trading engines with chain-agnostic smart-contract access, and enabling composable liquidity, positions and fund strategies across multiple chains. With safety and liveness guarantees built into its architecture, such as permissionless callbacks, threshold-signed relayer results and fraud-proof escape hatches, 128.trade mitigates the fragmentation and trust-bottlenecks of existing systems. By enabling any chain to access deep liquidity and any asset or contract to participate in advanced markets, 128.trade is poised to become the universal trading layer for the next generation of programmable finance.