

Injective: A Finance-Native Layer-1 Blockchain

Injective Research

October 2026

Abstract

Injective is a proof-of-stake Layer-1 blockchain designed for finance. It implements core components of financial systems as protocol-native modules: a fully onchain orderbook with front-running-resistant frequent batch-auction matching, oracle infrastructure, insurance funds for derivative markets, real-world asset issuance and permissioning, and recurring token buy-backs. Applications compose against these modules directly, sharing protocol state and accessing a common liquidity layer through the exchange module.

The network embeds the Ethereum Virtual Machine (EVM) and a WebAssembly (WASM) virtual machine within one canonical state, unified by the MultiVM Token Standard, and reaches deterministic finality through Byzantine fault-tolerant (BFT) consensus. Injective-developed modules provide the common financial layer, while EVM and WASM applications access it without creating separate settlement domains. Deterministic module interfaces and machine-payment pathways provide core agentic infrastructure, enabling users, institutions, and AI agents to operate side by side against the same financial state. INJ is utilized to pay transaction fees, secure the network through staking, and govern the protocol; it is also the asset acquired and burned through the Community BuyBack.

This paper traces Injective’s evolution from an exchange settlement protocol into a sovereign network and presents the resulting architecture as an integrated financial state machine. It describes and formalizes the consensus threshold, block lifecycle, frequent batch auction clearing rule, oracle accounting, derivative risk checks, permission model, and dynamic issuance controller that govern the network’s operation. This document supersedes the original Injective Protocol whitepaper of December 2018, which described a front-running-resistant exchange settlement protocol on Ethereum.

This document supersedes the original Injective Protocol whitepaper (December 2018).

1 Introduction

1.1 From settlement protocol to sovereign network

The original Injective Protocol whitepaper, published in December 2018, described a decentralized exchange settlement protocol on Ethereum. Its design combined a trustless relayer network for sharing liquidity with verifiable delay functions intended to resolve same-block order conflicts. The protocol was framed by five properties: trustlessness, resolvability, public verifiability, liquidity neutrality, and resistance to front-running.¹

That architecture reflected the constraints of the time. Because the proposed protocol operated above Ethereum, it could not directly determine the host chain’s consensus, transaction ordering, or finality. It therefore needed additional machinery to establish fair ordering and coordinate exchange settlement. The central insight, however, was durable: the quality of an onchain market depends on the rules by which orders are admitted, sequenced, matched, and settled.

Injective subsequently developed into a sovereign Layer-1 blockchain, moving the objectives of the 2018 design deeper into the stack rather than abandoning them. Consensus now supplies authoritative ordering and deterministic finality, while the exchange module implements a decentralized, fully onchain central limit orderbook (CLOB), applies matching and risk rules, and settles each market transition. Liquidity comes from participants quoting into those markets, and any application can reach the same books through the exchange module.

In making that transition, Injective structured the Layer-1 around modules: self-contained units of state and execution logic that interoperate through defined interfaces. This architecture allowed performance-sensitive financial primitives to become shared, validator-secured infrastructure and evolve through targeted protocol upgrades. Around that market core, successive upgrades added Injective-native modules for oracle integration, insurance infrastructure for derivative markets, buy-back auctions, token issuance, asset-level permissioning, and scheduled contract execution. The addition of a native EVM alongside WASM extended the system into a MultiVM architecture in which different execution environments operate against one chain, one finality domain, and one canonical asset state.²

This 2026 paper therefore does not introduce a replacement for the original protocol. It documents the live state machine that emerged from it. What began as an exchange settlement design above another blockchain became a sovereign financial network able to define its own sequencing, execution, risk, and settlement rules at the base layer.

1.2 Design thesis

Injective is organized around a single thesis: the common machinery of a financial system belongs in the protocol layer. Four design principles follow from that thesis.

Financial primitives and core onchain functions are implemented as shared protocol modules, executed by validators as part of the chain’s state transition. This reduces duplicated infrastructure and places common financial rules under the network’s security and governance, rather than requiring each application to rebuild and separately secure them in isolated contracts.

Markets require deterministic, final, and transparent execution. Financial positions and risk controls depend on an authoritative order of actions, a common matching and risk model, and settlement that does not expose participants to a probabilistic confirmation window.

One state can serve multiple execution environments. Developers, applications, and institutions bring different technical requirements, and no single execution environment is optimal for every use case. A MultiVM architecture allows complementary execution models to be used together without fragmenting assets, liquidity, or settlement.

The token economy should evolve with the financial system it secures. Network security and ecosystem growth place different demands on the native asset, so its economic design must support both while remaining adaptable as participation and activity change over time.

Together, these principles define a Layer-1 network whose protocol layer supplies common financial infrastructure while leaving its application layer open to developers, institutions, and autonomous software.

1.3 Scope and implementation baseline

This paper describes Injective mainnet, chain identifier `injective-1`, as of October 1, 2026.

Unless otherwise stated, descriptions of native modules, EVM, WASM, the MultiVM Token Standard, exchange behavior, interoperability, governance, and token parameters refer to the software and network state active on that date. Later software releases and governance actions may change interfaces, parameters, or feature status. Source-code citations identify immutable commits; claims about live behavior and active parameters identify a versioned release artifact or the relevant governance record.²

2 Network architecture

2.1 Replicated financial state

Injective’s higher-level financial capabilities depend on a common systems requirement: every validator must maintain the same state and apply the same ordered transitions to it. This replicated-state foundation allows modules, virtual machines, and settlement paths to operate as parts of one coherent network rather than as separate systems.

At block height h , Injective can be modeled as a replicated state S_h maintained by validators. This state includes accounts and bank balances, bonded stake and validator voting power, and governance proposals and parameters. It also records exchange deposits, orders, positions, and market metadata, together with oracle observations, insurance-fund state, token and permission namespaces, smart-contract state, and cross-chain client or bridge state. A valid block applies an ordered transition function to the previous state.

$$S_h = T(S_{h-1}, B_h) \tag{1}$$

Equation 1. Canonical state transition

This formulation reduces a diverse financial system to one consensus requirement: every honest validator computes the same S_h from the same prior state and block. Markets can be processed in parallel where their state is independent, but all results are combined deterministically before commitment.

The state transition is produced and used by several classes of participants:

- validators, which propose, execute, vote on, and commit blocks;
- delegators, which assign stake and voting power to validators;
- users and applications, which submit signed messages and smart-contract calls;
- oracle reporters and transmitters, which bring external observations on chain;
- relayers, bridge operators, and issuers, which connect external systems or assets;
- governance participants, which can change parameters and authorize upgrades;
- asset administrators, which may control permissioned namespaces; and
- agents and automated contracts, which act within delegated keys and policies.

2.2 Layered architecture

Injective organizes the system as a stack whose components converge upon one committed state. Users, applications, institutions, and agents interact through native module messages, EVM, or WASM. These interfaces invoke protocol state for accounts, assets, finance-specific modules, and governance. Validators replicate that state and reach onchain consensus to agree on transaction ordering and establish the canonical commit. Inter-Blockchain Communication (IBC), the Cross-Chain Transfer Protocol (CCTP), Peggy, and other bridge infrastructure connect external domains to the state machine through path-specific verification rather than forming sequential stages through which every transaction passes.

One canonical state, multiple execution and interoperability interfaces

Execution environments and verified cross-chain paths converge on one validator-secured state.

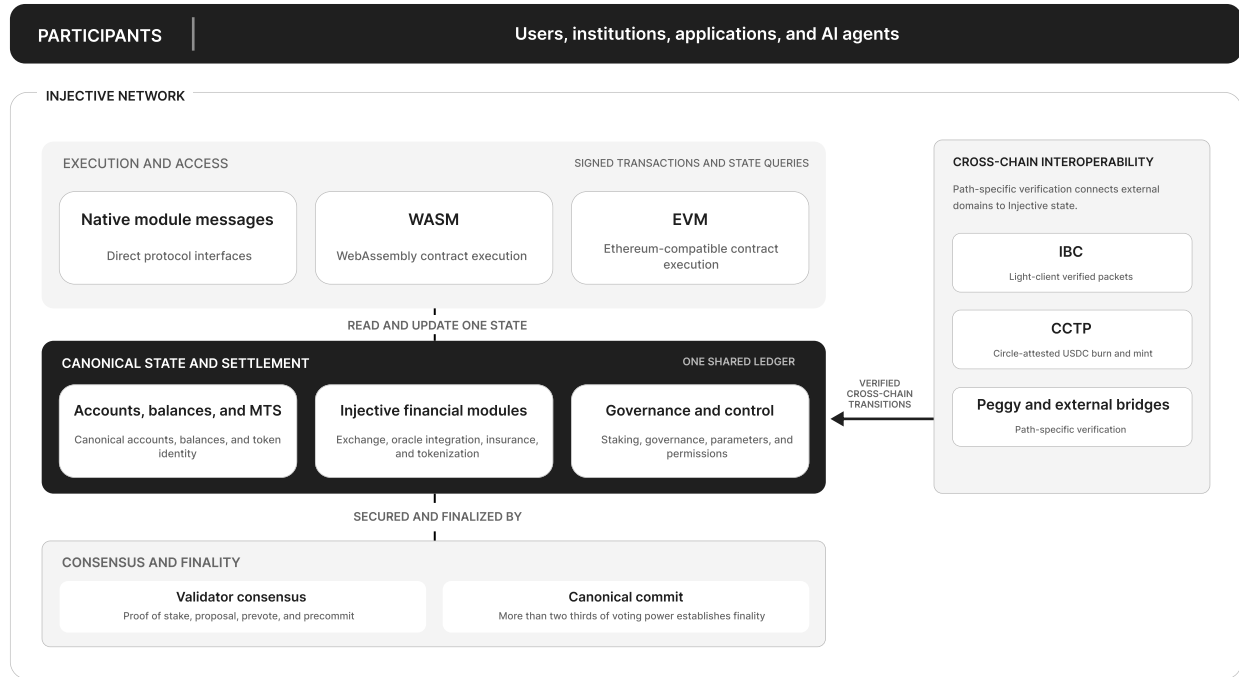


Figure 1: Injective's multi-layered architecture

The distinction between a module and a smart contract is central to this architecture. A contract is application code deployed within an execution environment, whereas a module forms part of the chain's own transition function and is executed by every validator. By placing common financial primitives in modules, Injective provides an onchain orderbook, risk state, balances, and settlement logic as public infrastructure while leaving product-specific behavior to applications.

2.3 Accounts, signatures, and transactions

Injective accounts are controlled by public and private key pairs and use a native Bech32 address beginning with inj. The same secp256k1 key can also be represented as a 20-byte Ethereum-style address, giving one externally controlled account a deterministic native and EVM representation rather than requiring a bridge between identities. An account number identifies the account, while a sequence number advances with each accepted native transaction to prevent replay.²

At the native protocol interface, a transaction contains one or more module-specific messages together with context such as the account number, sequence, chain identifier, fees, gas limit, memo, and signatures. A message may transfer an asset, place an order, change a staking position, vote in governance, or call another module function. Regardless of its contents, the transaction follows the same preparation, signing, and broadcasting lifecycle before validators check it and include it in a proposed block.²

Native transactions and EVM transactions use different encodings and client interfaces, but both enter the same chain and ultimately modify the same replicated state. Native signing paths support module-oriented transaction formats, while EIP-712 allows Ethereum wallets to authorize compatible Injective transactions using typed data. Across mixed native, WASM, and EVM operations, Injective’s MultiVM design preserves transaction atomicity so that a failed operation does not leave a partially applied state transition.^{2,3}

3 Consensus and deterministic finality

3.1 Consensus engine and commit rule

Injective uses proof of stake with an Injective-maintained implementation of CometBFT, the successor to Tendermint Core. The current release branch explicitly replaces upstream CometBFT with the Injective fork. The implementation preserves the Tendermint BFT proposal, prevote, precommit, locking, and greater-than-two-thirds commit structure formalized below. Let W be the total bonded voting power, F be the voting power controlled by faulty validators, and $C_r(h)$ be the precommit voting power for block h in round r . A block commits only when the precommit voting power for that block exceeds two thirds of W .^{2,4}

$$C_r(h) > \frac{2W}{3}, \quad F < \frac{W}{3} \quad (2)$$

Equation 2. Commit threshold and fault assumption

With $F < W/3$, conflicting blocks cannot both commit. In one round, precommit sets above $2W/3$ overlap by more than $W/3$ and must include honest voting power; honest validators do not precommit conflicting blocks. Locking and round changes extend safety across rounds; validators advance past unavailable or invalid proposals under the same commit rule.⁴

3.2 Validator participation and accountability

Bonded INJ connects the consensus rule to economic participation. Validators bond their own INJ and can receive delegated stake, which together determine stake-weighted voting power in the active validator set. Active validators propose blocks, validate transactions and state transitions, and exchange prevotes and precommits. Delegators contribute bonded stake to this security process without operating consensus infrastructure.²

Bonded stake also provides the accountability layer. Protocol-defined faults, including extended downtime and signing conflicting blocks at the same height, can trigger slashing and validator status changes. Because delegated stake contributes to a validator’s voting power, those penalties affect the stake bonded through that validator under the protocol’s rules. An unbonding period delays the release of stake after participation so faults can be detected and attributed before the associated

capital exits the security process. This section describes the consensus role of staking; Section 9 and the dedicated tokenomics paper address issuance and the broader reward economics.^{2,19}

3.3 Consensus implementation and block production

Block latency is determined by more than the voting rule alone. Transactions must propagate across the network, a proposal must reach validators, validators must execute and validate it, consensus messages must circulate, and the resulting state must be persisted. Injective reduces latency along this path through validator direct peering, optimized state synchronization and data handling, more efficient disk and resource management, reduced I/O load, and refinements to consensus processing.²

These changes reduce latency along the block-production path without introducing a post-commit confirmation period. Injective publishes a block time of approximately 600 milliseconds and deterministic finality at commit. Block time and finality are distinct: block time describes the cadence at which blocks are produced, while finality describes when a state transition becomes irreversible under the consensus model.^{4,5}

Short block intervals must still be balanced against block capacity and execution cost. Larger proposals take longer to propagate and validate, while heavier application execution delays the vote cycle. Block gas, mempool capacity, and consensus timing therefore form a coupled operating envelope rather than independent maxima.^{2,5}

3.4 Finality as a market property

A committed Injective block has deterministic finality: after commitment there is no longest-chain fork choice and no probabilistic confirmation window.⁴

For finance, this distinction is substantive. A trade that appears in a block but may later be reorganized creates credit and hedging uncertainty. Deterministic finality allows downstream systems to treat the committed transition as authoritative immediately. A fill, position update, token transfer, or liquidation is final with the block that contains it.

Finality also narrows the interval during which state can diverge across applications. Because native modules and smart contracts settle against the same committed ledger, a transfer completed through the EVM and a balance read through a native module converge on the same finalized result, giving downstream systems a common state against which to account and manage risk.

4 MultiVM execution, block construction, and shared state

4.1 EVM and WASM

Injective embeds EVM and WASM execution within one Layer-1 state machine, alongside purpose-built modules for exchange, oracle, insurance, token issuance, permissions, automation, and other network functions. At the implementation layer, the chain combines Injective-developed modules with foundational components built using the Cosmos software development kit (SDK); at the developer layer, the relevant execution choices are EVM, WASM, and direct module messages.²

The EVM uses chain ID 1776 and belongs to the same Injective network identified natively as `injective-1`; it is neither a separate rollout nor a bridged execution domain. Transactions invoking Solidity and WASM contracts pass through the same validator consensus and reach finality with the same canonical commit. Through EVM precompiles and WASM native-module messages, both environments can update shared native module state. Where Solidity requires an EVM-native interface to a module, Injective exposes that interface through a precompile, while WASM contracts use native bindings and module messages. These are execution gateways within one chain rather than merely interoperable bridges between separate ledgers.²

4.2 MultiVM Token Standard

The MultiVM Token Standard, or MTS, gives a token one canonical balance and identity across execution environments. Rather than synchronizing an EVM token with a separate native copy, the bank module maintains the authoritative balance and total supply, while EVM, WASM, and native interfaces address that same underlying asset state.²

Two elements implement the EVM component of this model. The Bank precompile exposes balance, supply, transfer, mint, burn, and metadata functions to Solidity, while the ERC-20 module maps bank denominations to ERC-20 contracts. Existing TokenFactory, IBC, and Peggy denominations can therefore acquire an EVM interface without becoming economically separate tokens. ERC-20 allowances remain contract state because they are an EVM authorization concept, but balances remain bank state, so a transfer from either environment mutates one canonical ledger entry.²

Applications can use the same collateral and settlement assets across virtual machines and, through the exchange module, route trading into a common orderbook.

4.3 Transaction admission and proposal construction

Before execution, Injective groups pending transactions into four ordered lanes: Oracle, Governance, Exchange, and Default. Oracle updates receive the highest proposal priority, followed by exchange-administrator transactions in the Governance Lane. Exchange-only transactions enter the Exchange Lane, and other traffic enters the Default Lane. Validators apply lane rules to construct and validate proposals.²

Lane routing preserves account sequence so a later transaction cannot bypass an earlier transaction from the same signer. Within the Exchange Lane, priority accounts for fee-discount tier and transaction type, with liquidation-only transactions receiving the highest priority.²

Transaction admission also passes through the txfees module. It enforces a maximum gas request per transaction and minimum gas-price rules, including a separate threshold for high-gas transactions. Governance-controlled parameters can enable an EIP-1559-style adaptive fee market, define the target share of block gas, bound base-fee movement, and govern the rechecking of pending transactions as the base fee changes. When adaptive mode is enabled, let b_h be the base fee for block h , g_h the gas used, g^* the target gas, and α the per-block adjustment rate. Before configured bounds and periodic reset are applied, the update is:^{2,5,6}

$$b_{h+1} = b_h \left(1 + \alpha \frac{g_h - g^*}{g^*} \right) \quad (3)$$

Equation 3. Conditional adaptive base-fee update

Utilization above target raises the base fee; utilization below target lowers it. Lanes and fee admission therefore solve different parts of the transaction pipeline: lanes determine functional ordering within proposals, while txfees governs the economic and gas conditions under which transactions enter and remain in the mempool.^{2,5,6}

4.4 Block execution lifecycle

After proposal construction, every validator executes the same ordered transaction sequence against the same pre-block state. Native module hooks perform scheduled work before and after transaction execution. The exchange module's EndBlock sequence is especially important because it defines how orders submitted throughout a block become a single deterministic market transition.²

At BeginBlock, the exchange can process scheduled funding updates, expiry settlements, trading rewards, and fee-accounting work. During transaction execution, messages create and cancel orders, update oracle state, move balances, or call contracts. At EndBlock, the exchange determines fee tiers, processes market orders against the beginning-of-block resting book, persists those results, processes matching limit orders, persists fills and new resting orders, updates funding and rewards, and emits the resulting events.²

Deterministic block and market execution lifecycle

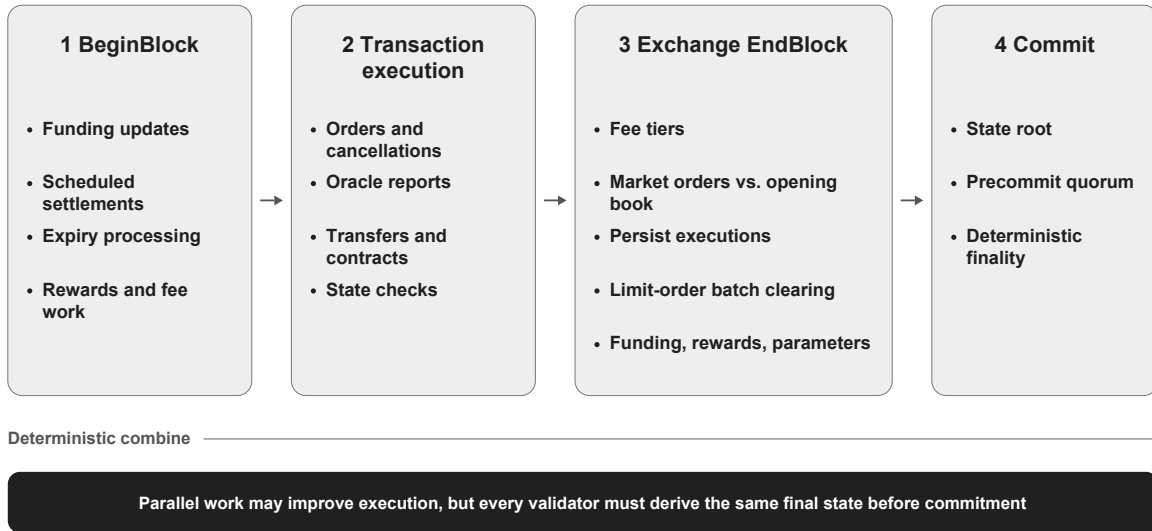


Figure 2: Deterministic block and exchange lifecycle

Some market groups are processed in parallel using a split, apply, and combine pattern. This concurrency is an implementation strategy rather than a relaxation of determinism: markets can be divided into independent work sets where their orderbooks and state updates do not conflict, then recombined in a defined order before commitment.²

Scheduled execution is one example of how module logic participates in this lifecycle. The Wasmx module allows governance-registered WASM contracts to receive a `begin_blocker` sudo call under a specified code identifier and gas configuration. Registered contracts may be pinned in memory and can be deactivated or removed under defined conditions. Wasmx therefore supplies deterministic, protocol-scheduled automation without introducing another virtual machine or relying on an off-chain keeper, and its effects remain subject to the same execution and consensus rules as every other state transition.²

5 Financial primitives as protocol modules

The financial layer is implemented as a set of modules within the state machine, which has three direct consequences. Module state is available through common interfaces, execution is synchronized with the block lifecycle, and protocol upgrades can evolve shared infrastructure without requiring each application to deploy a replacement market stack of its own.

5.1 Exchange state, subaccounts, and shared markets

The exchange module maintains spot and derivative markets, orderbooks, deposits, positions, market parameters, fee settings, and settlement state. Applications using native, EVM, or WASM interfaces can submit orders into the same module-managed markets, so each market has one protocol orderbook rather than an application-specific book. Liquidity is not produced by the module itself. It is supplied by participants that submit orders directly or through smart-contract vaults and other pooled strategies. These strategies may aggregate capital, but execution and price discovery still occur through the shared orderbook rather than an automated market-maker curve.²

Trading balances are held in subaccounts that distinguish total deposits from amounts available for new actions. Funds committed to resting orders, derivative margin, or other exchange obligations remain part of the subaccount's total balance but cannot be spent again while reserved. This separation allows the protocol to prevent double use of capital and to maintain position margin independently from a user's unrestricted bank balance.

Derivative markets also carry initial-margin and maintenance-margin parameters. The former constrains the risk that can be opened, while the latter determines when an existing position becomes eligible for liquidation. Because the checks are performed against exchange-module state, every interface admitting an order into a market inherits the same collateral and risk rules; Section 5.4 formalizes the resulting position-equity and maintenance tests.

5.2 Frequent batch auctions

Continuous matching creates a structural speed race because orders are processed one at a time, allowing small latency advantages to determine who reaches a stale quote first. On a public chain, transaction ordering can intensify that race by giving validators or sophisticated participants an opportunity to insert or reorder activity. Frequent Batch Auctions (FBAs) were proposed in market-design research as a response to this structural problem: discrete intervals and uniform-price clearing replace competition on speed with competition on price. Injective implements this model at the exchange-module layer, with each block serving as an auction interval.^{2,7}

The design has three defining properties. First, each block acts as a discrete auction interval rather than a continuous sequence of independent matches. Second, matching orders execute at a uniform clearing price within the interval. Third, new orders are sealed at the orderbook level: they are not published into the resting book until the interval closes and the EndBlock batch is executed. In this context, sealed describes when orders enter the public orderbook rather than a separate cryptographic commit-reveal scheme.^{2,7}

These properties change the basis of competition within the matching process. Because orders in the same batch are cleared together at a uniform price, being marginally earlier in the block does not produce a superior serial execution price. The ordinary FBA path is therefore resistant to

front-running, stale-quote sniping, and a major class of transaction-ordering maximal extractable value (MEV). By reducing the value of latency advantages and the risk of adverse execution against stale quotes, the design can also support tighter spreads and greater displayed depth. Experimental comparisons of FBA and continuous double auction (CDA) markets report less predatory trading, lower investment in low-latency technology, lower transaction costs, and lower volatility in spreads and liquidity under FBA.^{7,8}

Within the documented lifecycle, market orders are processed first against the resting orderbook as it existed at the beginning of the block. Matching limit orders are then combined with that book and cleared while the spread remains crossed, after which fills, cancellations, and newly resting orders are persisted as part of the block transition.²

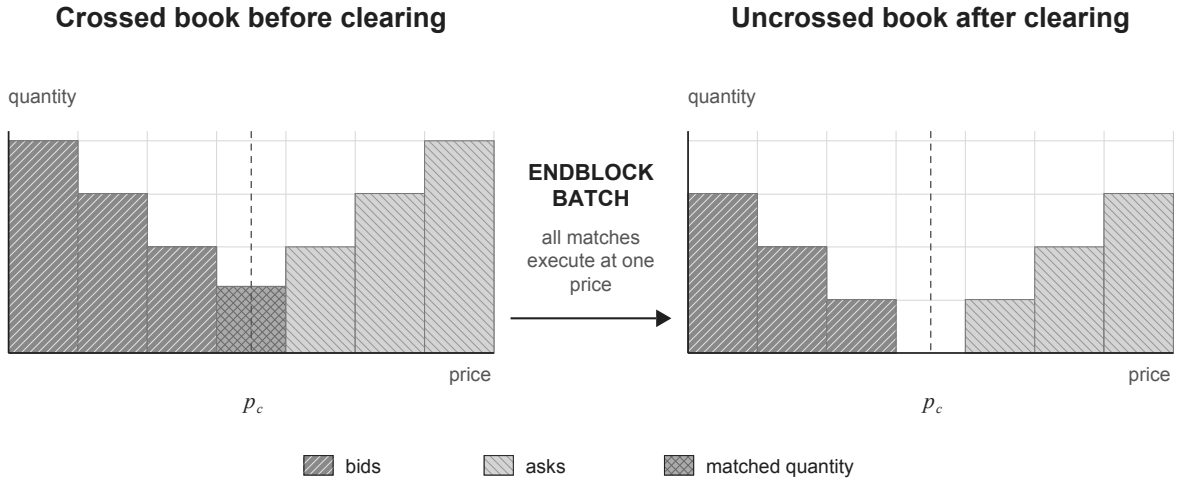


Figure 3: Uniform-price batch clearing

Let p_b and p_s denote the prices of the last matched buy and sell orders. For a derivative market, let p_m denote the mark price. The documented clearing rule can be represented as follows, with p_{cross} denoting the price of the last crossing order when the match crosses the spread in that direction.²

$$p_c = \begin{cases} p_{\text{cross}} & \text{if the final match crosses the spread,} \\ p_m & \text{if } p_s \leq p_m \leq p_b, \\ \frac{p_b + p_s}{2} & \text{otherwise.} \end{cases} \quad (4)$$

Equation 4. Batch clearing-price rule

The lifecycle distinguishes market orders, matching limit orders, and atomic market orders. Market orders use the beginning-of-block resting book, while the limit-order phase applies the batch clearing rule so that price and quantity determine execution without introducing a continuous nanosecond priority race. Atomic market orders form a deliberate exception for smart-contract

workflows that require immediate execution and therefore pay an additional fee rather than entering the ordinary batch.²

5.3 Oracle accounting

Derivative risk and settlement depend on external prices. For each price feed, the oracle module stores the latest normalized price $p(t)$, cumulative price $C(t)$, and timestamp t . The cumulative series permits a time-weighted average price (TWAP) over an interval without storing every intermediate observation.^{2,9}

$$\text{TWAP}(t_1, t_2) = \frac{C(t_2) - C(t_1)}{t_2 - t_1} \quad (5)$$

Equation 5. Time-weighted average from cumulative price state

Different providers follow different verification paths. In the off-chain reporting (OCR) path, N oracle nodes exchange signed reports over a peer-to-peer network before a designated transmitter submits an aggregate on chain. The module verifies the signatures and configuration, records the median observation, and uses epoch and round identifiers to reject stale or duplicated reports, while reward accounting can compensate contributors.^{2,10}

TWAPs reduce sensitivity to individual observations, while median aggregation reduces sensitivity to individual reporters. Together with update cadence and provider diversity, these mechanisms convert external market data into a deterministic state that the exchange module can use for funding, margin, and settlement.

5.4 Funding, margin, and liquidation

Perpetual markets use funding to connect the contract price to its reference market. The exchange computes a time-weighted premium from the relationship between execution prices and the mark price, incorporates the configured interest component, applies a cap, and converts the result into an hourly rate. The resulting payment transfers value between long and short positions without changing aggregate open interest.^{2,11}

For a simplified isolated linear position, let s equal $+1$ for a long and -1 for a short, Q be absolute quantity, p_e be entry price, p_m be mark price, M be posted margin, and F be accrued funding owed by the position. Equity before closing fees can be written as:

$$E(p_m) = M + sQ(p_m - p_e) - F \quad (6)$$

Equation 6. Simplified linear-position equity

If r_{maint} is the maintenance-margin ratio, a compact representation of the maintenance test is:

$$E(p_m) < r_{\text{maint}} Q p_m \quad (7)$$

Equation 7. Simplified maintenance-margin condition

Equations 6 and 7 provide a compact linear mark-to-market model rather than a complete reproduction of the implementation. The onchain check additionally incorporates market configuration, funding state, order state, and applicable fees.

When a position breaches the protocol’s liquidation conditions, the exchange creates a reduce-only market order that executes before ordinary matching. If closing the position leaves a negative payout, the market’s insurance fund can cover the shortfall; if value remains after the forced reduction, the surplus is returned according to the module’s liquidation accounting.²

5.5 Insurance funds

Injective’s insurance module manages pools of capital that can absorb specified liquidation shortfalls in derivative markets. Underwriters contribute assets and receive share tokens representing a proportional claim on the relevant fund. Redemptions follow the module’s notice process, reducing the risk that coverage disappears immediately during market stress. The module tracks fund balances, share supply, and pending redemptions.²

Insurance capital forms the final layer of the market’s loss-management sequence: posted margin absorbs loss first, forced reduction closes exposure, and available insurance capital may cover a residual deficit under the module’s rules.

6 Permissioned assets and real-world asset infrastructure

6.1 TokenFactory

TokenFactory allows an account to create a native denomination in the namespace `factory/{creator}/{subdenom}`. The namespace administrator can mint, burn, transfer administrative control, and update metadata subject to module rules. The resulting asset is a bank token, so it can participate in the same canonical state used by native modules and MTS interfaces.²

TokenFactory answers how an asset is created and who controls its supply. It does not by itself answer who may hold or transfer that asset. The Permissions module adds that policy layer.

6.2 Role-based asset controls

The Permissions module applies role-based controls to TokenFactory assets, including minting, receiving, burning, and sending tokens. An action is accepted only when the address’s combined role permissions allow it, namespace policy permits it, no overriding blacklist applies, and any configured receive hook succeeds. Receive hooks can apply deterministic contract logic using the sender, recipient, action, and amount.^{2,12}

The namespace can be sealed to make specified administrative settings irreversible. Issuers can also opt into governance override, creating a public recovery path. The resulting roles, action masks, hooks, and administrative configuration are represented directly in protocol state.

6.3 Real-world assets and iAssets

Injective treats real-world asset infrastructure as a protocol capability rather than a single application category. TokenFactory and Permissions support native issuance with configurable asset policy. The exchange and oracle modules support a second route: markets that provide price exposure to an external instrument through a cash-settled derivative.

These routes produce economically different instruments. A native permissionless token is a bearer-style onchain asset with no protocol-level holder restrictions. A permissioned token can represent a claim administered by an issuer and constrained by role policy. An iAsset is a cash-settled derivative that tracks the price of an external asset without representing ownership of the underlying asset.^{2,13}

Table 1: Economically distinct asset models

Model	Onchain object	Primary authority	What the holder owns
Permissionless native token	Bank denomination	Token administrator, if retained	The token itself; any off-chain claim depends on separate terms
Permissioned token	TokenFactory denomination plus Permissions policy	Issuer roles, hooks, and governance path	A token governed by the issuer-defined claim and redemption framework
iAsset	Cash-settled derivative position	Exchange rules, oracle, market governance	Price exposure without title to the referenced asset

iAssets solve a market-access problem rather than an asset-registry problem. They combine stablecoin margin and oracle-based pricing within exchange-module markets to create programmable price exposure without pre-funding or wrapping the referenced asset. Participants supply liquidity by quoting into those markets, while applications can expose the same module-managed book through different interfaces. The distinction between issued ownership claims and derivative exposure lets builders select the asset model appropriate to the market they are creating.

7 Interoperability and settlement assets

7.1 External assets and settlement paths

Injective connects its financial state to assets and messages from external networks through IBC, Peggy, and Circle’s CCTP. These paths do not share a single trust model: each verifies a different external event and creates an asset or message whose provenance remains identifiable after it enters Injective’s state machine.^{2,14,15}

IBC uses onchain light clients and packet commitments to verify state transitions between compatible chains. Relayers transport packets, but they do not determine validity, which instead derives from the connected chains, client state, submitted proofs, and timeout rules.¹⁴

Peggy is Injective’s validator-secured, bidirectional bridge for ERC-20 assets on Ethereum and has historically served assets including the legacy ERC-20 representation of INJ and Ethereum-origin stablecoins such as USDT. When an Ethereum-origin asset moves to Injective, the ERC-20 is locked in the Peggy contract and Peggo orchestrators submit deposit claims; after more than two thirds of Injective voting power attests to the event, the Peggy module mints a representative denomination beginning with peggy. A withdrawal reverses the process by burning the Injective representation and authorizing release of the ERC-20 from the Ethereum contract.²

Circle’s Cross-Chain Transfer Protocol, or CCTP, uses a different model for native USDC. A source-domain burn is observed and attested by Circle’s service, and that attestation authorizes native USDC minting on the destination. CCTP V2 support on Injective also enables optional fast-transfer and hook patterns.¹⁵

The EVM and native execution environments on Injective are different. They share one chain and one finalized state, so moving a token between their interfaces through MTS is not a cross-chain bridge event.

Table 2: Settlement and verification models

Path	What is verified	Primary dependency	Result on Injective
MTS	Internal state transition	Core modules and mappings	One canonical balance
IBC	Counterparty state through a light client	Both chains, clients, relayers	Voucher or packet state
CCTP	Circle attestation of a USDC burn	Circle issuer and attester	Native USDC mint
Peggy	Ethereum deposit event or Injective withdrawal batch	Ethereum contract, Peggo, and more than two thirds of Injective voting power	peggy denomination or ERC-20 release

7.2 Settlement composition

Cross-chain composition can be understood as a sequence of finality domains. After a source chain finalizes an event, a light client, attester, or bridge mechanism validates that event under its own security model; only then does Injective accept the corresponding message and update its state so that an application can use the resulting asset in a market or contract.

This architecture preserves provenance across different settlement paths. Native bank assets, IBC denominations, bridge-originated assets, and issuer-minted assets can be represented in the same financial state while retaining distinct origins and verification models.

8 Artificial intelligence and agentic finance

8.1 A programmable financial environment for agents

Injective’s architecture is naturally suited to autonomous software because financial actions are exposed through stable module interfaces, market state is public, and settlement is deterministic. An agent can discover available functions, query balances and markets, construct transactions, and request signatures against the same infrastructure used by human-facing applications, without receiving any special authority from consensus.

Current developer tooling demonstrates this model by exposing balances, markets, orders, transfers, and other workflows through command-line interfaces, Model Context Protocol (MCP) servers, and reusable agent skills. These tools remain clients of the protocol rather than components of consensus: transaction authority stays in the signing layer, allowing an agent to research and prepare broadly while holding only the execution authority assigned to it.^{16,17}

Within this broader toolchain, MCP provides a standardized interface through which compatible agents can discover and invoke Injective capabilities. The main Injective MCP server exposes chain queries and transaction workflows, including market data, orders, transfers, bridging, and EVM transaction submission. A separate documentation server provides semantic search and retrieval across Injective’s developer materials. Together, they give agents both an execution surface and a current source of protocol context without defining the agent’s reasoning process or expanding its authority.^{16,17}

The resulting architecture separates reasoning, authorization, and execution. A reasoning layer selects an action, a policy and signing layer determines whether that action may be authorized, and the chain applies deterministic transaction validity, market checks, and settlement. Autonomous systems can therefore operate against a deterministic financial core without placing probabilistic reasoning inside consensus.

8.2 Machine payments through x402

x402 embeds payment into an HTTP request cycle using the 402 Payment Required status. A client that requests a protected resource receives payment requirements, signs an authorization, and the signed authorization is relayed to a facilitator, which sends the onchain transaction and waits for settlement before the client retries the request with proof of payment. Injective's implementation uses USDC on the EVM path.¹⁸

This protocol turns a web endpoint into a metered service without a conventional account or subscription. It is especially useful for machine-to-machine access to data, models, or compute, and it gives agents a payment path that can be composed directly with onchain USDC settlement.

8.3 Bounded authority and deterministic automation

Agent authority can be bounded through narrow keys or smart accounts, asset and amount limits, market allowlists, price-impact limits, simulation, approvals, rate limits, and execution logs. Because these controls are enforced outside the reasoning model, they can remain deterministic even when the agent's planning process is probabilistic.

Scheduled contracts offer a complementary model by executing deterministic logic at defined block stages without relying on a language model. Agents are well suited to interpretation and planning, whereas chain-scheduled contracts are better suited to invariant execution; combining them can keep discretionary reasoning outside the consensus boundary while preserving deterministic rules for authorization and settlement.

9 The INJ token economy

9.1 Network functions

INJ is the native asset used for proof-of-stake security, governance, and network fees. Validators bond INJ directly or receive delegated stake, and the resulting voting power determines participation in consensus; governance uses staked participation to decide proposals and parameter changes, while transaction fees compensate the network for computation and state use.

The token also connects network security and financial activity to supply through two separate mechanisms: dynamic protocol issuance and recurring burns. The separate paper *INJ: A Programmable Token Economy for Deflationary Acceleration* provides the dedicated historical and economic treatment, while this section focuses on the mechanisms directly relevant to the protocol architecture.¹⁹

9.2 Moving Change Rate Mechanism

The mint module adjusts the annualized supply rate block by block according to the difference between the actual bonded proportion and a target. Let SR be the current supply rate, SRC the supply-rate-change parameter, B the configured blocks per year, BP_{actual} the actual bonded percentage, BP_{goal} the target, and SR_{lower} and SR_{upper} the approved bounds. The Moving Change Rate Mechanism is:¹⁹

$$SR_{\text{new}} = \begin{cases} \min\left(SR + \frac{(BP_{\text{goal}} - BP_{\text{actual}})SRC}{B}, SR_{\text{upper}}\right) & BP_{\text{actual}} < BP_{\text{goal}}, \\ SR & BP_{\text{actual}} = BP_{\text{goal}}, \\ \max\left(SR - \frac{(BP_{\text{actual}} - BP_{\text{goal}})SRC}{B}, SR_{\text{lower}}\right) & BP_{\text{actual}} > BP_{\text{goal}}. \end{cases} \quad (8)$$

Equation 8. Moving Change Rate Mechanism

When bonded participation falls below the target, the controller raises the supply rate toward the upper bound; when participation rises above the target, it lowers the rate toward the lower bound. At the target, the rate remains unchanged, making the mechanism a feedback controller for staking participation rather than a fixed emission schedule.

Governance proposal 617, passed in January 2026, set the annual issuance bounds to 2.2% and 4.4%, retained a 60% goal-bonded ratio and 50% rate-change parameter, and set blocks per year to 63,072,000. These were the active parameters documented for this October 2026 edition of the Injective Whitepaper.²⁰

9.3 Community BuyBack

The Community BuyBack is a recurring onchain process in which participants commit INJ for a pro rata share of contributed ecosystem revenue, with the committed INJ then burned. The program replaced the earlier winner-take-all burn auction in October 2025, broadening participation while preserving permanent token removal.

Under the exchange module's fee design, 40% of positive trading fees is paid to the designated fee recipient, while the remaining 60% accrues to the protocol revenue pool that supplies the Community BuyBack. Other applications and participants can also contribute revenue. This makes the burn mechanism horizontally scalable: as the application ecosystem expands and generates more revenue, the potential buyback pool can grow without requiring base-layer transaction fees or congestion to rise.

Dynamic issuance and Community BuyBack burns remain separate flows: the first adapts staking incentives to bonded participation, while the second converts contributed ecosystem value into token removal.^{2,19}

10 Governance

Injective governance coordinates software upgrades, module parameters, market settings, community spending, and other state changes authorized by the protocol. Staked INJ supplies voting power, and proposals pass or fail under configured quorum, threshold, and veto conditions.

This process has carried the network from its early exchange-focused architecture into a MultiVM financial platform. Consensus configuration, EVM activation, module upgrades, oracle providers, scheduled-contract registrations, and token-economic parameters can all be coordinated through the same onchain mechanism.

Protocol governance and asset administration nevertheless remain distinct. Chain governance controls shared protocol state, while a TokenFactory administrator controls its namespace, a permissioned-asset issuer controls retained roles, Circle controls native USDC issuance and CCTP attestations, and each application controls its own contracts and interface. The separation allows the base protocol to remain common even when product and asset policies are independently configured.

11 Security

11.1 Network security

Network security rests on proof-of-stake consensus. Validators commit blocks through the greater-than-two-thirds voting-power threshold formalized in Equation 2, while validators and delegators are economically aligned through staking rewards and slashing. Deterministic finality gives each committed state transition one authoritative result.^{2,4}

11.2 Financial security

Security extends into the financial layer through several coordinated mechanisms. Frequent Batch Auctions reduce the continuous time-priority race at matching. Oracle medians and cumulative price state support deterministic marks and time-weighted pricing. Initial and maintenance margin constrain leverage, reduce-only liquidations close impaired positions, and available insurance capital may absorb eligible residual shortfalls.

The security model is layered because consensus finalizes state but does not originate every input. Oracle modules govern how external prices enter protocol state; exchange modules apply market and risk parameters; IBC, Peggy, and CCTP validate external events under different verification models; governance and asset administrators act within their respective scopes; and agents operate through independent signing and authorization controls. These components converge on one finalized ledger, but they do not collapse into one trust model. Finality makes an accepted state

transition authoritative, while the validity of its inputs and authorizations remains governed by the subsystem that produced them.

11.3 Software assurance

Because native modules serve many applications, protocol correctness has system-wide value. Deterministic tests, simulation, property and invariant testing, audits, controlled upgrades, bug bounties, and monitoring form the assurance process around modules, VM integrations, contracts, and MultiVM mappings. Role separation, namespace sealing, and transparent governance add complementary controls for administrative state.

Table 3: Layered security mechanisms

Layer	Core mechanism	Protected transition	Complementary controls
Consensus	Greater-than-two-thirds precommit threshold	Canonical block commitment	Staking, slashing, validator operations, monitoring
Oracle	Signed reports, medians, cumulative price state	External data to protocol price	Provider diversity, cadence, bounds, TWAP
Market risk	Margin checks and reduce-only liquidation	Position state to orderly close	Parameters, liquidity, available insurance capital
Software	Deterministic module and contract execution	Valid transaction to state update	Tests, audits, invariants, staged upgrades, bounties
Settlement	Light clients, attestations, and bridge verification	External event to Injective asset state	Asset provenance and path-specific limits
Administration	Governance and namespaced role policy	Authorized parameter or asset action	Role separation, sealing, multisignature controls
Agents	Policy-controlled signing authority	Machine intent to signed transaction	Simulation, approvals, limits, and execution logs

12 Conclusion

Injective is a Layer-1 blockchain whose protocol implements the common machinery of a financial system. Consensus establishes deterministic finality; the exchange module converts orders into a shared market transition; oracle and insurance modules support pricing and loss absorption; and TokenFactory and Permissions provide native issuance and asset policy. MTS gives EVM and WASM one canonical asset state, while IBC, CCTP, and bridge infrastructure connect external assets under distinct verification models.

Its defining choice is to provide this common machinery once, at the protocol layer. Applications built for EVM, WASM, or native module interfaces can operate using the same unified assets, orderbooks, risk state, and settlement, even when they present very different products to users. The onchain orderbooks remain protocol infrastructure, while liquidity comes from participants that quote into them.

The 2018 whitepaper set out to build an exchange protocol that was trustless, publicly verifiable, liquidity neutral, and resistant to front-running. The live network now realizes those objectives at the base layer and extends them through a broader system of native markets, programmable execution, asset issuance, interoperability, and automated finance. What began as an exchange settlement protocol has become a sovereign state machine for programmable markets, open to developers, institutions, and autonomous agents alike.

References

- [1] Eric Chen and Albert Chon, “Injective Protocol: A Collision Resistant Decentralized Exchange Protocol,” December 2018.
- [2] Injective Foundation, “Injective Core,” public source baseline v1.20.3-safeharbor.2, GitHub commit de8a811.
- [3] Remco Bloemen, Leonid Logvinov, and Jacob Evans, “EIP-712: Typed Structured Data Hashing and Signing,” Ethereum Improvement Proposals, <https://eips.ethereum.org/EIPS/eip-712>.
- [4] Ethan Buchman, Jae Kwon, and Zarko Milosevic, “The Latest Gossip on BFT Consensus,” 2018, <https://arxiv.org/abs/1807.04938>.
- [5] Petar Zhivkov and Eric Chen, “Dynamic Fee Markets at Sub-Second Timescales: Adapting EIP-1559 for High-Throughput Blockchains,” *Mathematics* 14, no. 7 (2026): 1232, <https://doi.org/10.3390/math14071232>.
- [6] Tim Roughgarden, “Transaction Fee Mechanism Design,” *Journal of the ACM* 71, no. 4 (2024): Article 30, <https://doi.org/10.1145/3674143>.
- [7] Eric Budish, Peter Cramton, and John Shim, “The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response,” *Quarterly Journal of Economics* 130, no. 4 (2015): 1547-1621, <https://doi.org/10.1093/qje/qjv027>.
- [8] Eric M. Aldrich and Kristian Lopez Vargas, “Experiments in High-Frequency Trading: Comparing Two Market Institutions,” *Experimental Economics* 23, no. 2 (2020): 322-352, <https://doi.org/10.1007/s10683-019-09605-2>.
- [9] Hamda Al-Breiki, Muhammad Habib ur Rehman, Khaled Salah, and Davor Svetinovic, “Trustworthy Blockchain Oracles: Review, Comparison, and Open Research Challenges,” *IEEE Access* 8 (2020): 85675-85685, <https://doi.org/10.1109/ACCESS.2020.2992698>.
- [10] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Ari Juels, and Andrew Miller, “Chainlink Off-chain Reporting Protocol,” version 1.2, February 24, 2021, <https://research.chain.link/ocr.pdf>.
- [11] Damien Akerer, Julien Hugonnier, and Urban Jermann, “Perpetual Futures Pricing,” *Mathematical Finance* 36, no. 3 (2026): 481-499, <https://doi.org/10.1111/mafi.70018>.
- [12] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman, “Role-Based Access Control Models,” *Computer* 29, no. 2 (1996): 38-47, <https://doi.org/10.1109/2.485845>.
- [13] Siddhant Shah and Eugene Pinsky, “Perpetual Futures in Decentralised Finance: Mechanics, Economic Claims, and the Drivers of Trading Volume,” *International Journal of Financial Studies* 14, no. 7 (2026): 178, <https://doi.org/10.3390/ijfs14070178>.

- [14] Christopher Goes, “The Interblockchain Communication Protocol: An Overview,” 2020; Cosmos, “Interchain Standards.”
- [15] Circle Internet Financial, “Cross-Chain Transfer Protocol V2.”
- [16] Model Context Protocol, “Model Context Protocol Specification,” 2026-07-28.
- [17] Injective Labs, “Injective MCP Server”; “Injective Agent SDK.”
- [18] x402 Foundation, “x402 Specification”; “Exact Scheme on EVM.”
- [19] Injective Research, “INJ: A Programmable Token Economy for Deflationary Acceleration,” May 2024, https://injective.com/INJ_Tokenomics_Paper.pdf.
- [20] Injective governance proposal 617, “The INJ Supply Squeeze,” onchain proposal, MsgUpdateParams, and final tally, January 2026, <https://injhub.com/proposal/617/>.