THE FORGE

PROTOCOL PUBLICATION / 01

TECHNICAL WHITE PAPER

Forge Protocol White Paper

Flare-native liquidity, liquid staking, and verifiable settlement evidence

MARKETS | LIQUID STAKING | GOVERNANCE | SHIELDED TRANSFERS | VERIFIABLE EVIDENCE

IMPLEMENTATION-ALIGNED DRAFT V1.3 — 30 AUGUST 2026

Architecture, implementation boundaries, security model, and deployment evidence.

BUILT FOR FLARE

FORGE PROTOCOL / TECHNICAL PAPER

Document status

This paper describes the Forge system implemented in this repository: the Forge exchange and governance protocol, the hpFLR liquid-staking system, and the integrated ProofRails evidence service. It is a technical description, not an offer of securities, an investment solicitation, a legal opinion, or a promise of yield.

Forge's active market/governance deployment is currently a Coston2 test deployment. Its recorded 29 August 2026 verification passes live binding checks for four pools, four gauges, three vaults, three aggregator venues, twelve directed oracle policies, and all 36 browser-to-contract bindings. A fresh privacy deployment is also live on Coston2: its replacement incremental Merkle tree was independently reconstructed after multiple deposits, three deposit/proof/withdrawal/disclosure/ProofRails/write-back rehearsals passed, and the attestation-bound interface is published at `https://forge-dex.pages.dev`. The superseded privacy deployment remains quarantined. The ProofRails API, worker, PostgreSQL, Redis, and Forge adapter are a separate service plane and are not hosted by static Cloudflare Pages. The live hpFLR instance is liquid and reconciled, its obsolete release caps are removed, and its canonical validator-reward claim path is authorized. Its current queue, reward, recipient, and eligible-position modules were upgraded through an eleven-call atomic timelock operation after the exact batch passed on a live-state fork; the resulting manifest passes 145/145 independent live checks. It still has no current mirrored P-chain stake or validator rewards because the twelve-wallet faucet cohort has not yet been funded. A testnet deployment is not approved for promotion to Flare mainnet. Mainnet parameters, contract addresses, custody arrangements, and production keys do not exist merely because a testnet value appears in a script. A mainnet release requires a clean, immutable source commit, testnet evidence bound to that commit, verified deployment evidence, and distinct newly generated mainnet credentials.

The contracts and exact deployment manifest are authoritative if this document ever differs from deployed bytecode. Values in this paper are classified as:

- **Protocol constants:** enforced by the current contract implementation and changeable only through a reviewed upgrade or new deployment.
- **Governed parameters:** bounded values that an authorized governance or operational role may change.
- **Candidate configuration:** values used by current deployment scripts or proposed for rehearsal; they are not final mainnet policy.

STATUS / SCOPE / AUTHORITY

FORGE PROTOCOL / TECHNICAL PAPER

Abstract

Forge is a modular financial protocol designed around Flare's native data and reward infrastructure. It combines five functions that are often deployed as unrelated systems:

1. a stable and volatile automated market maker with typed cross-venue routing;
2. vote-escrow governance, fee-directed gauge voting, launch tooling, and multiple independently funded sources of liquidity-provider return;
3. hpFLR, a principal-preserving liquid-staking receipt whose validator rewards are distributed separately and reproducibly;
4. an opt-in shielded-transfer pool that obscures the on-chain link between a deposit and a later withdrawal without presenting ordinary AMM activity as private; and
5. ProofRails, a project-scoped evidence service that reconciles settlement, constructs a deterministic signed archive, and anchors the archive hash on Flare.

The design separates principal, rewards, governance, and evidence. hpFLR principal is minted and redeemed one-for-one with FLR; reward accounting does not increase the hpFLR exchange rate. AMM reserves are separated from fees. Governance voting power is historical and time-weighted rather than derived from a live balance at execution time. ProofRails anchors exact bytes but does not claim that a hash alone proves every business assertion inside those bytes.

Security is treated as a system property. Contracts fail closed when mandatory price data, settlement provenance, dependency identities, or production configuration are missing. Privileged actions are bounded, role-separated, and observable. Off-chain calculations commit their complete inputs and outputs, allowing independent reproduction. The web application and backend improve usability but are not trusted as the final enforcement boundary.

FORGE PROTOCOL / TECHNICAL PAPER

Contents

FORGE / 01

1. Motivation

Flare exposes native capabilities that allow a decentralized exchange to do more than price assets against its own reserves. FTSOv2 supplies network price data, wrapped native FLR can participate in delegation, and canonical reward-epoch and randomness systems can support reproducible distribution. Using these capabilities safely requires protocols to reconcile several competing objectives:

- Capital should remain useful while participating in staking, liquidity, or governance.
- Fee and delegation-yield routing should be directed by stakeholders without allowing flash-acquired voting power.
- Market execution should remain permissionless while resisting clearly anomalous oracle-relative output.
- Rewards should follow economic ownership, including ownership represented through approved LP, gauge, or vault positions.
- Operational facts should be provable without confusing an on-chain timestamp with proof that all submitted metadata is true.
- Emergency controls should stop new risk without unnecessarily trapping existing claims.

Forge addresses these objectives through explicit modules rather than a single contract with broad authority. The architecture accepts that not every trust boundary can be eliminated. P-chain custody, stake-mirror accuracy, reward-root posting, governance upgrades, signing keys, and data-provider operations remain real responsibilities. The protocol therefore bounds those responsibilities, makes their effects observable, and documents what cryptography and on-chain execution do—and do not—prove.

FORGE / 02

2. Design principles

2.1 Principal and rewards are different obligations

hpFLR represents principal at a one-to-one unit relationship with FLR. Validator rewards are recognized as a separate liability and distributed as native FLR or optionally restaked. This prevents an ambiguous exchange rate from mixing deposits, stake returns, losses, and rewards.

Forge AMM fees are likewise moved out of pool reserves and accounted for separately. LP claims, staking-fee distributions, and delegation-reward streams each have explicit liabilities.

2.2 Historical ownership governs historical rights

Governance votes use timestamped checkpoints. hpFLR reward calculations use past block checkpoints. An account cannot acquire tokens after a proposal starts or after a reward epoch closes and retroactively claim the associated right.

2.3 Constrain every privileged outflow

Vault staking releases are restricted to the configured, mirror-registered destination, the operations role, the physically available principal reserve, and live solvency. They are not split by an arbitrary amount cap or refill wait. Reward allocations cannot exceed recognized funding. Gauge-held fees, deployer-funded pool rewards, referral shares, protocol fees, and staking fees are all conserved and bounded.

2.4 Reproducibility before discretion

Where fully on-chain enumeration would be impractical, off-chain work is deterministic and committed. hpFLR reward roots bind the calculation block, snapshot set, complete exclusions, eligible supply, leaves, proofs, allocation total, and dust. ProofRails binds the exact bytes, file sizes, and hashes of a fixed archive inventory.

2.5 The client is not the policy

Frontends quote routes, explain outcomes, and prepare transactions. Backends perform bounded searches and operational work. Neither can authorize a pair, exceed a fee cap, bypass an oracle policy, mint unbacked hpFLR, over-allocate a funded reward epoch, or turn an untrusted archive signer into a trusted one.

2.6 Fail closed at production boundaries

Production startup and deployment gates reject missing or ambiguous configuration. A stale or unmapped mandatory FTSO feed rejects protected execution. ProofRails refuses production signing without exact settlement reconciliation and refuses to trust a signer merely because its public key is bundled with its signature. Mainnet deployment refuses testnet credentials and uncommitted source.

FORGE / 03

3. System overview

The system consists of three cooperating but independently secured planes.

3.1 Forge market and governance plane

The market plane contains stable and volatile AMM pairs, the router, typed cross-DEX aggregation, LP fee accounting, ERC-4626 LP vaults, FTSO price protection, WFLR delegation management, fee-only gauges, bribes, launch pools, and governance.

3.2 hpFLR staking plane

The staking plane accepts native FLR, mints hpFLR one-for-one, releases available principal to an identified staking operation, tracks recognized stake, queues redemptions, recognizes received validator rewards, and distributes those rewards using reproducible historical ownership.

Each provider may deploy an hpFLR instance. The implementation does not hard-code a provider. The provider's external P-chain custody and validator operations remain outside the EVM contracts and are an explicit trust boundary.

3.3 ProofRails evidence plane

The evidence plane is a private, project-scoped API and worker system. It verifies a cited Flare settlement, generates ISO 20022-style evidence, signs a deterministic archive with Ed25519, and anchors the complete archive hash through an authorized EvidenceAnchor operator.

The three planes are composable without collapsing their trust models. A ProofRails anchor does not control AMM funds. A reward-root poster cannot release staking principal. An hpFLR operator cannot create an arbitrary Forge pair. Governance can change only the actions exposed by the governed contracts and their timelocks or role boundaries.

FORGE / 04

4. Flare as the protocol substrate

Forge uses Flare-native services in three principal ways.

4.1 FTSOv2 market references

The FTSOPriceOracle maps tokens to FTSOv2 feed identifiers, normalizes prices to 18 decimals, validates timestamps, and supplies reference values for protected swaps. It resolves the FTSOv2 endpoint from Flare's contract registry and can refresh the resolved system address after an authorized network upgrade.

4.2 Wrapped FLR delegation

Eligible WFLR pairs can delegate their WFLR voting weight to approved data providers. The delegation manager supports at most two provider destinations per pair. A keeper synchronizes weights and claims canonical Flare reward epochs through authorized executor paths. Claimed WFLR is measured and forwarded to the associated LP reward path rather than being treated as AMM reserve.

4.3 Canonical epochs and randomness

hpFLR reward distribution binds its accounting epochs to Flare's canonical reward-epoch source. When an epoch closes, the system commits a future Flare random round. After secure historical randomness becomes available, a deterministic algorithm derives a fixed number of distinct, sorted snapshot blocks from the closed epoch range. This removes discretionary snapshot timing from the root poster.

Deployment must verify the current canonical Flare contracts and timing values. Network values recorded in development documentation are evidence for a specific rehearsal, not timeless constants.

FORGE / 05

5. Forge automated market maker

5.1 Pair types

Forge supports two constant-function pair types.

For a volatile pair, the invariant is:

$$k = x \times y$$

where x and y are post-fee reserves.

For a stable pair, reserves are decimal-normalized and the invariant is:

$$k = x^3y + xy^3 = xy(x^2 + y^2)$$

The stable curve is intended for assets expected to trade near parity. It provides lower slippage near the equilibrium region while preserving a bounded constant-function market. Decimal normalization is essential: the curve must compare economic units, not raw token integers with potentially different decimal scales.

Initial liquidity permanently locks a minimum quantity of LP units. Stable pools also enforce a minimum invariant so small first deposits cannot create a mathematically unusable pool. Liquidity minting and burning remain proportional to pool ownership after initialization.

5.2 Pair creation and identity

Pairs are created through a factory with deterministic identity. Token order is canonical, duplicate token pairs are rejected, generated bytecode must exist, and the deployed pair must report the expected token addresses and curve type before the factory registers it.

Creation is not an arbitrary public factory call. ForgePairGovernance is the primary creator, while explicitly approved modules such as LaunchManager may receive a narrowly scoped creation permission. A reentrancy lock surrounds generator execution and registry writes so an external generator cannot consume a deterministic salt and leave inconsistent factory state.

5.3 Swap fees and allocation

The current implementation initializes stable fees at 0.04 percent and volatile fees at 0.18 percent. An authorized fee manager may configure a pair-specific or category fee up to the contract maximum of 0.25 percent.

For a gross swap fee F, the sequence is:

1. referral share, if enabled and a referral handler exists;
2. protocol treasury share from the remaining fee;
3. BBX staking share from the then-remaining fee; and
4. residual LP share.

If r, p, and s are the relevant basis-point shares of each successive remainder, then:

$$F_{ref} = F \times r / 10{,}000$$

$$F_{protocol} = (F - F_{ref}) \times p / 10{,}000$$

$$F_{staking} = (F - F_{ref} - F_{protocol}) \times s / 10{,}000$$

$$F_{LP} = F - F_{ref} - F_{protocol} - F_{staking}$$

The implementation caps referral share at 12 percent of the swap fee, protocol share at 20 percent of its applicable remainder, and staking share at 30 percent of its applicable remainder. Factory defaults are 15 percent protocol share, zero referral share, and zero staking share until a valid handler is configured. The Coston2 deployment candidate configures the staking share at 25 percent of the post-referral, post-protocol remainder. These are fee-revenue splits, not percentages of the entire swap amount.

PairFees segregates these assets from reserves. Per-user indices track LP fee ownership across balance changes so transferred, burned, vaulted, or withdrawn LP units do not claim fees earned by a prior owner.

5.4 Swap execution safety

Pair execution validates nonzero input, bounded output, the applicable invariant after fees, token identity, reserve updates, and optional callback behavior under a pair-level lock. User-specified minimum output protects against receiving too little.

An optional mandatory FTSO policy adds an output ceiling to direct pair execution. The pair hook deliberately does not impose a naïve two-sided pre-transfer price comparison: an attacker could temporarily donate input, force a victim below the apparent lower band, and skim the donation after the victim reverts. Instead, the pair-level hook limits excessive output relative to fresh oracle value, while the user's minimum output protects the lower side. The higher-level aggregator performs a two-sided post-settlement band check because it can observe complete route balance deltas safely.

5.5 FTSOv2 reference calculation

For token input amount Ain, token decimal counts din and dout, and fresh USD prices Pin and Pout, the oracle computes:

$$N_{in} = A_{in} \times 10^{18} / 10\text{^}d_{in}$$

$$Nout = Nin \times Pin / Pout$$

$$Eout = Nout \times 10^{\wedge}dout / 10^{18}$$

For maximum deviation d in basis points, the allowed band is:

$$Lower = floor(Eout \times (10{,}000 - d) / 10{,}000)$$

$$Upper = floor(Eout \times (10{,}000 + d) / 10{,}000)$$

Unmapped feeds, future timestamps, stale values, zero prices, unsupported decimals, malformed FTSO responses, and deviation settings above 50 percent fail validation. The governed freshness window is constrained between 90 and 3,600 seconds; its implementation default is 300 seconds. Batch feed reads are bounded at 64 identifiers.

The current static-call integration assumes zero-fee FTSO reads. If the network makes a required read fee-bearing, protected swaps fail closed until a reviewed integration change is deployed.

5.6 Optional concentrated-liquidity compatibility

The source tree also retains an optional Algebra Integral concentrated-liquidity integration. RouterV2 can quote and execute a concentrated route only after its swap router, Algebra factory, quoter, and pool-provenance storage are configured atomically. For every route, the advertised pool must match either the factory's canonical pool for the token pair or the factory's custom pool for the recorded deployer. This prevents a route from naming one pool while executing through another.

GaugeCL and GaugeFactoryCL are inherited compatibility modules for Algebra farming and community-vault fee collection. CustomPoolDeployer coordinates pool, community-vault, fee, tick-spacing, and farming-plugin setup through configured Algebra dependencies.

These contracts are not active in the supported deployment. GaugeManager explicitly rejects concentrated-gauge creation because the retained CL farming path assumes a primary incentive token. Concentrated trading may be reviewed independently in the future, but production activation would require a fee-only design plus all external Algebra contracts, plugins, vaults, factories, pools, roles, and runtime hashes in the exact deployment manifest and rehearsal. The legacy GlobalRouter rejects Algebra requests and payable concentrated-liquidity selectors rather than silently running the V2 path or retaining native value.

FORGE / 06

6. Typed cross-DEX aggregation

ForgeDexAggregator searches across approved venue types without exposing arbitrary target calls or arbitrary calldata. Supported adapters cover:

- Forge V2 routes;
- explicitly configured Uniswap V2-compatible venues; and
- explicitly configured Uniswap V3-compatible venues.

A route contains no more than four hops. A split execution contains no more than four routes. Every hop must preserve token continuity, venue identity, pool identity, and approved adapter type.

Execution uses exact temporary token approvals, clears allowances after use, settles by measured balance delta, and rejects stranded intermediate input. The caller supplies a total minimum output. When a route is subject to mandatory oracle policy, the aggregator checks the pre-execution expectation and the final observed result. Governance recovery is limited to assets not owed to an active execution.

The backend route service is a convenience layer. It applies approved-token and venue filters, bounded search, payload validation, cache and response controls, and rate limits. The on-chain aggregator independently enforces what can actually execute.

FORGE / 07

7. Liquidity-provider return model

Forge separates three active LP return sources:

1. **AMM trading fees.** The residual fee share is attributed to LP ownership through Pair and PairFees accounting.
2. **Optional pool-deployer rewards and vote bribes.** An approved pool deployer may supply its own external reward token to its own approved gauge, and bribe contracts may hold bounded reward-token sets associated with gauge votes. These assets originate from their disclosed funders, not Forge or a Forge token budget.
3. **WFLR delegation rewards.** WFLR pairs may earn Flare delegation rewards, which are claimed, measured, and streamed to the designated LP vault path.

ForgeLPVault is an ERC-4626 vault whose asset is a Forge LP token. It can collect the pair's underlying fee tokens and receive a WFLR delegation reward stream. Delegation rewards are streamed over seven days rather than assigned instantaneously. If a reward arrives while share supply is zero, it is queued instead of being lost or claimable by an unrelated party.

Standard Forge gauges are created with a zero primary reward token. Gauge voting therefore organizes fee and bribe attribution without creating a protocol-funded farming stream. An optional pool rewarder has separate custody and accounting, must be bound to the exact gauge and approved deployer, and rejects the platform governance token as its funded asset. The supplying deployer selects a positive distribution period for every funded or renewed stream; Forge imposes no seven-day incentive default.

GaugeFactory records one incentive deployer for each approved gauge. GaugeIncentiveFactory permits only that address to create the canonical rewarder and makes the deployer its owner and sponsor. A governance-approved LaunchPad project assigns its project owner automatically; another pool requires an explicit gauge-administrator assignment. Revoking that assignment blocks new funding but does not confiscate a funded stream or user liabilities. The first rewarder may be attached after stake exists because it initializes accounts lazily from canonical gauge balances. An active program cannot be silently replaced while stake remains.

The reward asset, its contract behavior, available balance, rate, duration after each renewal, disclosures, continuity, and legal treatment are the supplying deployer's responsibility. Forge does not issue, fund, custody, curate, endorse, guarantee, insure, or maintain it. Appearance in the interface means only that the on-chain pool/deployer binding is valid; it is not token due diligence or a promise of value. GaugeV2 isolates failures in the optional hook so a paused or reverting reward token cannot prevent withdrawal of LP principal, although the external incentive itself may fail or become unavailable.

The system does not promise a fixed annual percentage yield. Returns depend on trading activity, pool ownership, delegation performance, deployer-funded external rewards or bribes, and the configured fee policy. A deployer may stop renewing a program. The platform governance token is not an LP return source.

FORGE / 08

8. BBX governance token

8.1 Fixed-supply, governance-only model

BBX is the Forge governance token. Its total supply is created once in the token constructor, it is burnable, and it exposes no ongoing mint role. BBX may be held, transferred, staked for historical governance power, or locked into a vote-escrow position. The active protocol does not dispense BBX to gauges, pay BBX rebases, run BBX reward seasons, or fund liquidity with a BBX emissions allocation.

The Coston2 deployment script uses a candidate initial issuance of 1,000,000,000 BBX for governance testing. It may create a 50,000,000 BBX protocol permanent governance position and may transfer candidate team and partner governance allocations of 30,000,000 and 20,000,000 BBX respectively. All other issuance remains unallocated in the disposable test deployer account; the script does not assign that remainder an emissions, bridge, treasury, or public-distribution purpose.

Those figures are rehearsal inputs, not finalized mainnet tokenomics. Any production distribution must separately identify recipients, custody, vesting or lock treatment, governance concentration, and exact immutable deployment transactions. No allocation may be inferred from the prior incentive design or from balances in a superseded Coston2 deployment.

Creating a supermassive governance position uses VotingEscrow's explicit permanent-position accounting. Its principal and voting-power treatment must be reconciled independently in a production manifest because it can materially concentrate governance. It does not create a right to token rewards.

8.2 Enforced absence of protocol rewards

The governance-only policy is enforced at several layers:

- GaugeV2 and GaugeFactory both require the primary reward-token argument to be the zero address.
- GaugeManager creates standard gauges with a zero primary reward token and rejects every primary-reward notification.
- GaugeManager's legacy minter slot can only be set to zero; BlackGovernor requires its legacy minter constructor argument to be zero.
- the standard optional rewarder rejects the governance token, is bound to one gauge and its approved deployer, and is deployed through a separately attested factory;
- Bribe epoch accounting derives time directly from the shared weekly clock and has no minter dependency.
- Auto-voting operates with a zero rebase-distributor address and skips the legacy claim side effect.
- fresh deployment scripts do not deploy or fund MinterUpgradeable, RewardsDistributor, or SeasonalRewards;
- browser and backend runtimes refuse a disabled, superseded, or non-governance-only deployment configuration;
- the mainnet manifest gate rejects those retired modules, GaugeFactoryCL, and EpochController and requires every retained legacy binding to attest the zero address; and

- the independent RPC attestor reads the policy back from deployed GaugeFactory, GaugeManager, BlackGovernor, and AutoVoterManager.

The retired contracts remain in the repository for historical compatibility, upgrade analysis, and regression testing. Source presence does not make them part of Forge's supported architecture. A deployment that wires or funds them is outside this paper's model and must be rejected rather than presented as Forge.

8.3 What governance voting controls

veBBX voting selects and weights eligible pools for the fee-and-bribe governance process and supports the narrow proposal actions exposed by BlackGovernor. It does not authorize a token budget. An approved pool deployer may voluntarily fund its own gauge reward, and eligible parties may fund an approved bribe, subject to the separate authorization and accounting rules; governance cannot convert unallocated BBX into protocol incentives through the active contracts.

FORGE / 09

9. Vote escrow, staking, and governance

9.1 Vote-escrowed BBX

BBX may be locked into a transferable veNFT for up to four years. Lock duration and amount determine governance weight under the vote-escrow model. Permanent or supermassive positions have explicit accounting treatment rather than being hidden treasury balances.

Votes direct gauge fee and bribe attribution and support the limited proposal actions exposed by Forge governance. Historical checkpoints make proposal voting power a property of the proposal start time, not the current balance at vote or execution time.

9.2 Liquid BBX staking

ForgeStaking allows BBX staking without a fixed withdrawal lock. It distributes actual received protocol-fee tokens using per-token accumulators. No more than 32 reward tokens can be registered, which keeps stake and withdrawal work bounded.

Staked BBX receives a time-weighted governance multiplier. Fresh stake starts at 1× and reaches 4× after three weeks of continuous staking. Adding stake produces a weighted-average start time; any withdrawal resets the remaining stake's timer. Timestamped checkpoints prevent a stake placed after proposal start from influencing that proposal.

The accounted-balance invariant separates staked BBX principal from unclaimed rewards. Fee tokens may arrive by an authorized pull notification or by direct PairFees push; synchronization recognizes only the unaccounted surplus, preventing double distribution.

9.3 Gauges, bribes, and automatic voting

VoterV3 and GaugeManager associate valid ve votes with eligible gauge and bribe accounting. GaugeV2 conserves launch allocations and optional deployer-funded reward balances, while the active factory requires a zero primary reward token. GaugeFactory and GaugeIncentiveFactory bind each program to its approved pool deployer. Bribe contracts associate voluntarily funded external assets with votes without giving the depositor arbitrary control of protocol reserves.

The automatic-voting modules allow users to delegate voting strategy to governed strategy contracts. Auto-voting does not bypass custody, token identity, or checkpoint rules. Strategy dependencies, custody transitions, and manager permissions are explicitly validated.

Automatic voting has no BBX rebase claim. The retained rebase-distributor address is zero in supported deployments, so a bucket votes without a token-reward side effect.

9.4 Governance scope

Forge governance is intentionally narrower than arbitrary execution. BlackGovernor proposals create configured WFLR or non-WFLR pairs through ForgePairGovernance. Pair creation still passes factory identity checks and oracle policy wiring. Incentive-minter calls are not valid proposal targets. Successful execution requires the recorded proposal outcome and matching description hash.

Upgradeable and administrative modules retain real governance risk. A timelock or authorized role can change only exposed parameters or upgrade an upgradeable implementation, but an upgrade can materially change future behavior. Production must therefore publish role owners, delay values, implementations, storage-layout checks, and executed governance actions.

9.5 Permission and token registries

PermissionsRegistry separates named operational roles such as governance, voter administration, gauge administration, bribe administration, fee management, concentrated-pool administration, genesis management, and epoch management. Role lists and memberships are bounded so administrative enumeration cannot grow without limit. Production deployment must move the initial deployer-held registry authorities to the published governance, team, and emergency identities.

TokenHandler is the on-chain eligibility registry used by gauge, bribe, and launch paths. Governance or the narrowly scoped genesis manager may add valid token contracts, remove unsafe tokens, manage connector status, and assign bounded volatility categories. Removing a token also removes stale connector status. This registry is distinct from the browser's approved-token registry: the browser registry determines what the interface presents, while TokenHandler and the contracts determine what protected protocol paths accept.

FORGE / 10

10. Launchpad

The Forge LaunchPad coordinates token fundraising with subsequent liquidity formation. A launch specifies the offered token, funding asset, duration, minimum fill, cap, and any launch-funded project asset. Funds cannot be reused across launch states, and the launch path does not draw from a protocol incentive budget.

At completion, a successful launch can create the intended Forge pair through LaunchManager's explicit factory permission, seed liquidity, and establish the associated gauge. A partially filled or failed launch follows the configured settlement and refund path. Cancellation and refund behavior remain available only in valid states, and transitions are one-way.

The launch system does not certify token quality, team identity, legal status, or future value. Its on-chain guarantee is limited to the configured state machine, custody, allocation, liquidity, and refund rules.

FORGE / 11

11. hpFLR liquid staking

11.1 Economic model

hpFLR is a feeless ERC-20 receipt for deposited FLR. The Vault is the only component that mints; the Vault and the bound RedemptionQueue hold burn authority, with the queue burning only hpFLR already escrowed for a funded claim. A deposit of A FLR mints A hpFLR:

hpFLR minted = FLR received

Principal redemption remains one-for-one. Validator rewards do not increase the principal conversion rate. They become a separate recognized reward liability and are claimed as native FLR or restaked, in which case the reward amount is deposited and the same number of hpFLR units is minted.

This design makes principal, losses, rewards, and redemptions independently visible. It also means hpFLR is not a rebasing token and is not a share whose exchange rate automatically appreciates.

11.2 Token properties

The hpFLR token implementation is immutable. It supports EIP-2612 permit and stores block-number checkpoints for each balance and total supply. Only addresses holding the token's Vault role may mint or burn; the verified deployment limits that role to the Vault and RedemptionQueue, and only the Vault exposes a minting path. Historical lookup rejects the current or a future block so a reward calculation cannot rely on an unsettled state.

11.3 Vault accounting

The Vault tracks:

- liquid FLR reserve;
- pending staking releases;
- last confirmed stake;
- PChainStakeMirror-reported stake;
- recorded loss impairment;
- returned-stake overstatement during mirror lag;
- hpFLR principal liabilities;
- funded but unclaimed redemption allocations; and
- recognized but undistributed validator rewards.

Its recognized assets are:

$$A = \text{liquidReserve} + \max(\text{lastConfirmedStake} + \text{pendingStaking}, \text{saturatingSub}(\text{mirrorStake}, \text{recordedLoss} + \text{returnedStakeOverstatement}))$$

Where the active RedemptionQueue holds funded claims outside the Vault, live principal liability is total hpFLR supply minus those externally backed queue allocations. Total Vault liability is:

$$L = \text{live hpFLR principal liability} + \text{legacy queue liability} + \text{recognizedRewards}$$

Safety-sensitive actions require:

$$A \geq L$$

Available liquid principal reserve excludes reward earmarks:

$$\text{availableReserve} = \max(\text{liquidReserve} - \text{recognizedRewards}, 0)$$

This prevents validator rewards awaiting distribution from being released as new stake or used twice. A donation can recapitalize the Vault without minting hpFLR or creating a new reward obligation.

11.4 Staking release and return

Only the operations role may release FLR, only to the configured staking destination. The release amount cannot exceed physically held liquid reserve or `availableReserve`, which excludes recognized reward obligations, and the Vault must remain solvent. The current implementation deliberately has no arbitrary per-transaction amount cap or refill delay. Deprecated cap storage remains zero only to preserve the live UUPS layout. A release first becomes pending stake and is confirmed against the stake mirror through explicit accounting transitions.

Stake returns must be reported precisely. Losses are recorded as impairment rather than hidden in an optimistic mirror balance. A divergence circuit breaker can pause new deposits and reward recognition when the expected and observed stake disagree beyond policy, while preserving claims needed to exit existing obligations.

Validator rewards are claimed only through allowlisted reward-manager calls with constrained calldata. The Vault recognizes the measured native balance increase, not an operator-provided number.

11.5 FIFO redemptions

Redemption is a one-way FIFO queue. A holder transfers hpFLR into escrow and receives a position. Requests cannot be cancelled to jump the queue or withdraw escrowed hpFLR after later users have relied on ordering.

The Vault funds requests in bounded batches. The queue holds only the native FLR already allocated to claimable positions; excess value is returned to the Vault in the same transaction. On claim, allocated FLR is paid and the corresponding escrowed hpFLR is burned. Emergency pauses are asymmetric: pausing new deposits or reward recognition does not disable already funded native claims.

11.6 Reward epoch lifecycle

An hpFLR reward epoch progresses through the following stages:

1. canonical epoch closure and accounting recognition;
2. commitment to a future Flare randomness round;
3. selection of the first acceptable secure historical random value in a bounded window;
4. deterministic derivation of distinct snapshot blocks;
5. calculation of effective historical ownership;
6. publication of the complete allocation artifact and Merkle root;
7. an on-chain challenge window; and
8. holder claim or post-deadline rollover.

Claims do not open immediately when a root is posted. The configured challenge window must cover the governance timelock delay plus an independent review buffer. Replacing a root before the first claim restarts the complete challenge window. After the first claim, replacement is blocked so already accepted claims cannot be rewritten.

The EVM contracts cannot infer which external validator reward belongs economically to which hpFLR epoch. Recognizing received rewards into a selected eligible epoch is an operator policy decision. The selected epoch, amount, and resulting artifact are public and bounded, but correct temporal attribution remains part of the disclosed trust model.

11.7 Effective ownership and allocation

For holder i and selected snapshot blocks b1 through bn, the calculator determines effective hpFLR balance after recursively attributing ownership through governed eligible LP, gauge, and vault positions. Structural protocol addresses are excluded from both numerators and eligible supply.

The holder weight is the median across snapshots:

$$w_i = \text{median}(\text{effectiveBalance}(i, b_1), \ldots, \text{effectiveBalance}(i, b_n))$$

If F is the funded reward amount and W is total eligible supply, the allocation is:

$$\text{allocation}_i = \text{floor}(F \times w_i / W)$$

Zero allocations are omitted. The sum of all leaves is allocationTotal and must not exceed F. Integer dust remains unallocated and later enters the rollover pool.

Each Merkle leaf binds holder, epoch identifier, and amount. The complete artifact binds chain ID, finalized calculation block and block hash, relevant contract addresses, calculator version and source commit, snapshot blocks, funded amount, eligible supply, the complete exclusion set, every leaf and proof, root, allocation total, dust, and source metadata. Independent observers can recompute the same artifact from pinned RPC state, optionally against two providers.

11.8 Contract-held hpFLR

RecipientRegistry handles hpFLR owned by contracts that cannot call a claim flow directly. A holder may prove control by self-call or ERC-1271 signature and register a recipient and claimant. Once control is proven, timelock governance cannot later overwrite that holder's registration. Posted epochs pin the registry used for their claims.

EligiblePositionRegistry determines which LP, gauge, or vault structures the calculator may unwrap. Entries are bounded, dependency-aware, and active only across explicit block or epoch intervals. Governance cannot retroactively make an unrelated position eligible for an already finalized snapshot.

11.9 hpFLR roles and trust boundaries

The final deployment has no general owner. A timelock controls upgrades and long-lived configuration. Operations, Guardian, and Root Poster are distinct roles; the deployer renounces final privileges.

- **Operations** can release only available principal to the fixed staking destination and perform constrained reward-manager actions, but cannot mint hpFLR except through funded Vault deposit mechanics.
- **Guardian** can activate temporary asymmetric pauses but cannot redirect assets or permanently change policy.
- **Root Poster** can post a reward allocation within the funded cap. It cannot release principal, but a malicious root can omit, underpay, or misallocate holders unless observers detect and correct it during the challenge window.
- **Timelock governance** can replace a bad pre-claim root, configure modules, and upgrade proxy-based operational contracts. This is a powerful and explicit trust boundary.
- **P-chain custodian/provider** controls assets after release to the external staking environment. EVM contracts cannot cryptographically guarantee external custody or validator performance.
- **PChainStakeMirror** is a trusted accounting oracle. A stale or malicious overreport can overstate backing until the circuit breaker, reconciliation, or governance response catches it.

The root-posting and stake-mirror assumptions are not concealed as decentralization. They are bounded operational responsibilities with public evidence and response paths.

11.10 Current Coston2 validator rehearsal

The live Coston2 Vault is uncapped, reconciled, and has no current stake. A real 1 C2FLR C-to-P-to-C round trip and a separate 1 C2FLR hpFLR mint/release/return/redemption lifecycle passed. A fresh read-only fork of the current live proxies also completed a 50,000 C2FLR mint, release, mirror-confirmed stake-accounting return, existing-queue auto-funding, and exact 50,000 C2FLR redemption. The registry-resolved ValidatorRewardManager is active and allowlisted, and the staking owner has now authorized the Vault as both claim executor and native-reward recipient.

The pending aggregate rehearsal does not divide the funded 50,000 C2FLR among test accounts. That principal holder will mint exactly 50,000 hpFLR. Twelve separately encrypted accounts must each receive 100 C2FLR from the official faucet; conservative transfers then produce twelve distinct hpFLR balances of 94–99 and 101–106, totaling the full 1,200 faucet C2FLR. Operations gas is funded separately. The complete available Vault reserve, including its pre-existing principal, is aligned to the P-chain gwei unit with a sub-gwei non-minting donation and delegated for exactly Coston2's live 14-day minimum. No self-bond validator position is created.

The transaction programs refuse to continue unless all twelve independent faucet balances, exact mints, live verifier limits, validator window, solvency, destination, roles, mirror state, and reward authorizations match the plan. After expiry, the recovery program returns the exact delegated principal and retains the mirror-lag discount until canonical catch-down. A separate holder-claim program binds the published allocation artifact to the live epoch, verifies all thirteen Merkle proofs,

requires distinct allocations for all twelve unequal faucet holders, simulates each claim, and records the exact native-C2FLR distribution and gas delta. The final verification harness then calls the existing FIFO approve, initiate, auto-fund, and claim workflow for every holder's entire balance at the invariant one-to-one principal rate: 50,000 hpFLR returns 50,000 C2FLR to the principal wallet, and each 94–106 hpFLR test balance returns the identical C2FLR amount to its wallet. Rewards are separate and never modify this rate. The detailed live record is `hpflr-main/docs/COSTON2-50000-STAKING-READINESS-2026-08-29.md`.

FORGE / 12

12. WFLR delegation and hpFLR composition

WFLR delegation and hpFLR validator rewards are independent yield sources.

For WFLR pairs, DataProviderRegistry stores approved providers and WFLRDelegationManager applies bounded provider weights. Keeper operations synchronize pair delegation, claim canonical epochs, and forward exactly measured reward assets. Provider removal also clears lifecycle state so old provider configuration cannot remain silently active.

For hpFLR, validator rewards arise from the provider's external staking operation and enter the Vault as measured native FLR. The RewardDistributor allocates them across eligible historical hpFLR ownership.

An hpFLR/WFLR or hpFLR-containing liquidity position may therefore participate in several economic flows, but each flow retains its own accounting. No component may count the same FLR simultaneously as hpFLR principal, an undistributed hpFLR reward, an AMM reserve, and a delegation reward.

FORGE / 13

13. Opt-in shielded transfers and selective disclosure

13.1 Scope and terminology

Forge privacy is an explicit, opt-in transfer path through a dedicated `ForgeShieldedPool`. It is not embedded in every Forge AMM pool and does not make an AMM swap, liquidity deposit, hpFLR action, governance action, gauge position, vault position, launch, or delegation private. Those ordinary transactions remain public on chain. A user enters the shielded system only by submitting a deposit to the shielded-pool contract and exits it by submitting a valid zero-knowledge withdrawal.

The privacy claim is deliberately narrow: a valid withdrawal proves that the spender knows an unspent note contained in a recognized Merkle root without revealing which public deposit created that note. The system obscures the **deposit-to-withdrawal link**. It does not hide the deposited asset, deposit amount, deposit transaction, deposit sender, block or timestamp; nor does it hide the withdrawal asset, amount, transaction, recipient, relayer, fee, root, proof, nullifier hash, block, or timestamp. It also cannot hide wallet-provider telemetry, IP/network metadata, browser state, or subsequent public transactions. “Shielded transfer” is therefore more accurate than “private transaction.”

13.2 Note creation and local custody

The browser generates two distinct, nonzero canonical BN254 field elements using a cryptographic random source:

- `secret`, the private membership witness; and
- `nullifier`, the private preimage used to derive a one-time spend marker.

The strict note plaintext contains only `secret`, `nullifier`, token address, integer amount, creation timestamp, checksum-normalized depositor, and an optional payment reference. Unsupported fields are rejected by the adapter. The browser derives:

```
noteCommitment = MiMC(nullifier, secret)
```

It also predicts the amount-and-token-bound leaf:

```
leaf = MiMC(noteCommitment, amount, uint160(token))
```

The note witness is not sent to a Forge or ProofRails server. Before requesting a wallet transaction, the browser creates a password-encrypted local backup containing the note, its final commitment, its exact compliance-plaintext hash, and a generated X25519 user view-key pair. The backup is stored locally and can be exported as an encrypted JSON file. The password must be at least twelve characters. Forge never stores the password and cannot recover or reset it. Users must retain an offline copy of the encrypted backup and a separate secure backup of the password. Loss of either item means the user can no longer construct the withdrawal witness; the platform has no recovery copy of the user's plaintext note or password.

Two independent ciphertexts are created with authenticated X25519/XSalsa20-Poly1305 boxes:

1. a user ciphertext decryptable with the view secret stored inside the password-protected backup;

2. a compliance ciphertext decryptable with the deployment's separately held compliance key.

Each ciphertext includes an ephemeral public key and nonce. Ciphertext is public data and is permanently observable on chain; confidentiality depends on the corresponding private key and the security of the browser and backup password.

13.3 Deposit execution and amount binding

The deposit call identifies the token, amount, base note commitment, user ciphertext, and compliance ciphertext. Native FLR requires `msg.value` to equal the amount exactly. ERC-20 deposits measure the shielded pool's actual balance delta and reject fee-on-transfer behavior. Only assets with a nonzero configured minimum are enabled, and the amount must meet that minimum and remain inside the circuit field.

The contract—not the browser—derives the final Merkle leaf from the supplied base commitment and the token and amount actually received. This prevents a client from depositing one value while creating a proof-bearing commitment for another. Duplicate commitments are rejected. The final leaf is inserted into a depth-20 incremental MiMC Merkle tree, token liabilities are increased, and events publish the commitment, leaf index, token, amount, timestamp, and both ciphertexts. The `ForgeNoteRegistry` records the same token, amount, and ciphertext state and accepts writes only from its bound shielded pool.

The incremental tree keeps immutable zero-subtree values separate from mutable insertion frontiers. This distinction is security-critical after the first leaf: treating an already-filled subtree as the next empty right sibling produces a root that a standards-compatible witness builder cannot reconstruct. The replacement Coston2 deployment includes this correction, and its root was independently rebuilt from multiple live deposit events.

13.4 Merkle membership and confirmation policy

To withdraw, the browser scans confirmed `Deposit` events beginning at the attested deployment block in bounded RPC ranges. It reconstructs leaves in their emitted order, checks that the note's commitment exists at the expected index, computes its twenty sibling elements and direction bits, and derives the candidate root. The root must match a root retained by the deployed pool. Events that have not reached the configured confirmation count are not included.

The contract retains a bounded history of recent roots. A proof against an unknown or stale root is rejected. This permits withdrawals after subsequent deposits while bounding storage. An unavailable RPC, missing event, mismatched leaf, inconsistent index, or unrecognized root stops proof creation; the client does not guess a path.

13.5 Local Groth16 proof generation and withdrawal

The withdrawal circuit receives the secret, nullifier, Merkle siblings, and direction bits as private inputs. Its public inputs are ordered and bound as:

1. Merkle root;
2. nullifier hash;
3. recipient address;
4. amount;
5. token address;
6. relayer address; and
7. relayer fee.

The nullifier hash is derived as `MiMC(nullifier, 0)`. Publishing it does not reveal the nullifier preimage, but it gives the contract a deterministic one-time spend identifier. The circuit reconstructs `MiMC(nullifier, secret)`, binds that result to the public amount and token, follows the private Merkle path, and proves that the resulting root equals the public root.

Proof generation occurs in the browser with same-origin WASM and proving-key artifacts. Before use, the client hashes those bytes and compares them with the artifact hashes bound into the attested deployment and publication configuration. It does not fetch arbitrary proving material from a user-supplied origin. The resulting Groth16 proof reveals no Merkle path or note preimage.

The contract rejects a previously used nullifier hash, a field-invalid value, an unknown root, a zero recipient, a fee greater than the note amount, or a fee without a relayer. It verifies the proof against the exact public-input order, marks the nullifier spent before transferring value, decrements the corresponding token liability, and pays the public recipient and optional relayer. The withdrawal transaction is public. Spending the withdrawn asset in a normal Forge swap is also public and may create a new observable correlation point.

13.6 What determines practical privacy

Zero-knowledge validity does not create a useful anonymity set by itself. Practical unlinkability depends on how many economically plausible deposits could correspond to a withdrawal. Exact or unusual amounts, immediate timing, a pool with few notes, wallet reuse, the same fee/relayer pattern, or recognizable follow-on activity can make a linkage statistically likely even though the proof does not reveal it. Fixed denominations, delayed withdrawal, independent relaying, and larger sets can improve privacy, but the current interface does not promise those conditions.

The contract publishes the deposit sender and withdrawal recipient through transactions and events. Using the same address at both ends may defeat the user's objective. A malicious or compromised client can exfiltrate the witness before encryption. A compromised backup password or user view key reveals the note. A compromised compliance key can decrypt the compliance ciphertext without waiting for the official committee workflow. The threshold workflow governs authorized disclosure and evidence creation; cryptography cannot force a stolen decryption key to follow policy.

13.7 Threshold selective disclosure

Official disclosure is coordinated by `ForgeComplianceOracle` and the bound note registry. The committee does not approve a reserialized object. It approves `keccak256` of the **exact UTF-8 JSON bytes** encrypted at deposit. Once the configured distinct-signer threshold approves the same note and exact plaintext hash, a disclosure event can be emitted. Removing a signer invalidates that signer's old approvals. The signer set, threshold, registry, and authorized adapter recorder are checked against the published deployment before the browser enables shielded writes.

Disclosure authority cannot spend a note. Only knowledge of the note witness and a valid Groth16 proof can authorize withdrawal. Conversely, fund withdrawal does not require committee approval; the compliance path is evidence and disclosure policy, not custody authority.

The Forge adapter receives the disclosure event, decrypts the compliance ciphertext in memory, and compares its bytes with the committee-approved hash before parsing. It then validates the strict schema, independently reconstructs the base and amount-bound commitments, locates the canonical deposit transaction, verifies depositor, token, amount, and timestamp, and checks the nullifier's spent state when applicable. Only after all checks pass does it project already-public settlement fields to ProofRails. Secret, nullifier preimage, user view key, and payment-reference plaintext are not included in the ProofRails settlement projection.

13.8 ProofRails evidence and anchoring

ProofRails reconciles the projected transaction against Coston2, produces its deterministic signed evidence archive, and submits the complete archive hash to the separately deployed `EvidenceAnchor`. After confirmation, the adapter verifies receipt content, archive signature, trusted signer, anchor bytecode and role policy, transaction-bearing anchor provenance, trusted anchor sender, and confirmation count. It then records the receipt and anchor facts through the authorized recorder function of `ForgeComplianceOracle`.

An oracle record does not reveal the note witness and does not make the transfer legally compliant by itself. It proves that the configured disclosure and evidence pipeline accepted a particular reconciled public settlement projection and anchored exact evidence bytes under the selected trust policy.

13.9 Current Coston2 deployment and publication status

The original Coston2 privacy deployment is quarantined and is not referenced by the application. The replacement graph deployed on 29 August 2026 is:

COMPONENT	COSTON2 ADDRESS
MiMCSponge	`0x67aC516184D790879fff15aF03262435b7b9E8C0`
Groth16 verifier	`0xC57DCE04777E8B2747C953735C62d087905E3e5D`
ForgeNoteRegistry	`0xb009B8080BeB872a59211De2E9196AA40711eb03`
ForgeShieldedPool	`0x9aC33e41491483600d23a254CE0A85d7F1351E1C`
ForgeComplianceOracle	`0xa992402A7ba41cDD347D9bc258ee67926Edcf419`
ProofRails EvidenceAnchor	`0x67A9e7Bda7F185B5E7C31FC246c5e2CCb42bCA28`

The runtime was attested at Coston2 block 34,643,608. Three full live rehearsals completed deposit, local proof generation, withdrawal, threshold disclosure, ProofRails settlement reconciliation, signed bundle verification, EvidenceAnchor confirmation, and Forge oracle write-back. A deliberate multi-deposit regression independently reconstructed the same root held by the contract after two deposits. The publication gate binds the deployment manifest, runtime attestation, circuit hashes, role configuration, and most recent complete rehearsal before enabling the Privacy tab. The live Cloudflare application exposes the path as **Opt-in shielded transfers** and labels all other tabs **Public on-chain**.

The long-running ProofRails API, worker, PostgreSQL, Redis/RQ, and Forge adapter are a separate service plane. They were exercised locally for the rehearsal and are not hosted by Cloudflare Pages. If that service plane is offline, user-held valid notes remain withdrawable because withdrawal is enforced by the shielded contract and proof verifier, not by ProofRails. New official disclosure evidence and oracle write-back wait for service recovery.

13.10 Mainnet release boundary

The Coston2 proving setup is a test artifact, not a production ceremony. Shielded transfers are not approved for Flare-mainnet value. Mainnet publication remains blocked until the exact release has:

- a final reviewed circuit and public-input binding;
- a documented multiparty phase-two trusted setup;

- reproducibly verified proving and verification artifacts;
- final client backup, recovery, relayer, disclosure, and operational policies;
- production-separated compliance, signer, adapter, anchor, and service credentials;
- continuously hosted and monitored evidence services where official disclosure is offered; and
- a dedicated internal exact-commit privacy review, deployment attestation, multi-deposit audit, and complete mainnet-candidate rehearsal.

FORGE / 14

14. ProofRails verifiable evidence

14.1 Purpose

ProofRails records a cited Flare or Coston2 settlement, reconciles it against canonical chain data, creates ISO 20022-style XML and supporting records, signs a deterministic archive, and anchors the archive's SHA-256 hash on Flare.

The integrated Coston2 `EvidenceAnchor` is deployed at `0x67A9e7Bda7F185B5E7C31FC246c5e2CCb42bCA28`. It completed the three current shielded-transfer rehearsals with transaction-bearing anchor provenance and subsequent Forge compliance-oracle write-back. Cloudflare Pages does not host ProofRails: the FastAPI service, worker, PostgreSQL, Redis/RQ, signing, anchoring, and Forge adapter processes are a separately operated service plane.

The integrated implementation retains the project's hardened v1.3-compatible private API. The current upstream ProofRails v2.2 surface includes materially different registration, public-artifact, inventory, and facilitator behavior. Those changes are a migration requiring a separate product and threat-model decision; they are not silently imported. The official hosted documentation and repository were rechecked on 28 August 2026; they continue to describe a broader receipts, public verification, delivery-evidence, and x402 service than the fixed-inventory private integration defined here.

14.2 Fixed evidence inventory

Each evidence archive contains exactly six files:

FILE	PURPOSE
pain001.xml	ISO 20022-style payment initiation representation
receipt.json	Canonical receipt and reconciled settlement fields
tip.json	Associated tip or project evidence fields
manifest.json	Canonical inventory, lengths, and SHA-256 hashes of the three data files
manifest.sig	Ed25519 signature over the canonical manifest
public_key.pem	Public key corresponding to the archive signature

The ZIP is deterministic: names are sorted, metadata is fixed, permissions are fixed, and files are stored without transformation that would make archive identity dependent on runtime behavior.

14.3 Settlement reconciliation

Production evidence cannot be signed merely because a client supplies a transaction hash. The service verifies chain, transaction success, required confirmations, asset identity, sender, receiver, and exact amount.

For native FLR, sender, receiver, and transaction value must match exactly. For ERC-20 settlement, the configured token must emit exactly one matching Transfer for the modeled receipt. Because the current receipt identity does not include log index, a transaction can create at most one receipt globally. Project references are unique, and an idempotent retry must reproduce the original evidence fields.

14.4 Verification model

Full bundle verification checks:

- bounded download and uncompressed archive size;
- exact six-file inventory, unique safe paths, and no path traversal;
- manifest version, canonical semantics, declared lengths, and SHA-256 values;
- consistency across XML, receipt, tip, and manifest;
- safe XML parsing and the configured schema policy;
- Ed25519 signature validity;
- signer fingerprint against a separately configured trust list;
- the SHA-256 hash of the complete ZIP;
- the pinned EvidenceAnchor contract, chain, runtime code, historical event, and minimum confirmations; and
- anchor sender against a separately configured trusted-sender list.

The public key inside an archive proves only that the archive is internally self-consistent. It does not establish who owns that key. Trust comes from a fingerprint obtained through a separate authenticated channel. Likewise, a matching on-chain hash proves that bytes were anchored no later than the confirmed event; it does not prove that every business statement in those bytes is true.

The combined result is verification_passed. A matches_onchain result alone is insufficient. Hash-only verification cannot inspect archive contents or signature semantics.

14.5 EvidenceAnchor

EvidenceAnchor rejects a zero bundle hash and records a bundle hash only once. An initial owner and operator are set explicitly; the deployer gains no implicit production privilege merely by broadcasting deployment. The owner manages operator authorization and transfers ownership through a two-step process.

Production startup pins chain ID, deployed runtime code hash, current owner, authorized operator, minimum confirmations, and trusted anchor senders. A contract at the expected address is not trusted if its bytecode or role state differs.

14.6 Service topology

The supported production topology separates:

- TLS termination and edge rate limiting;
- FastAPI request handling;
- PostgreSQL durable receipt state;
- authenticated TLS Redis and Redis Queue;

- one or more isolated workers;
- durable evidence and statement storage; and
- worker-only Ed25519 and EVM signing keys.

API processes receive the public signing key, not private signing material. Worker jobs serialize both a receipt-specific lock and a shared anchor-wallet lock, preventing duplicate processing and cross-process nonce collisions. Pending jobs survive API process restarts; bounded retries distinguish transient failure from terminal failure.

Production admin and API-key pepper secrets are accepted only from mode-restricted files. A worker process rejects HTTP even if it is accidentally launched through ASGI, so a key-bearing worker does not become a second API surface. Readiness checks exercise PostgreSQL and configured Redis rather than reporting healthy from process liveness alone. Settlement-provider outages return retryable 503 responses and do not create a receipt; they are not mislabeled as an invalid transaction.

Artifacts, status, statements, file access, and event streams are project-scoped. Public API-key minting is disabled by default. Callback messages are canonical, domain-separated Ed25519 messages binding the timestamp and body. A consumer must verify freshness and fingerprint, then reconcile the callback with the authenticated receipt API before changing business state.

14.7 ISO status

The generated payment message follows the pain.001.001.09 structure, but it is described as **ISO 20022-style** unless official, licensed, pinned schema validation is configured and succeeds. FLR is not an ISO 4217 fiat currency code. An anchor and a valid signature do not turn style-compatible project evidence into bank certification.

FORGE / 15

15. Application and operations architecture

The Forge application opens on a formal platform overview that distinguishes public on-chain modules from the opt-in shielded-transfer path and links each explainer directly to its executable tab. Expanded descriptions state transaction flow, enforcement boundary, current deployment status, and retained trust assumptions. The functional application exposes hpFLR, swap, shielded transfers, liquidity, lock and staking, governance, earn, vault, launch, delegation, and evidence flows. Static configuration is schema-validated and tied to deployment and rehearsal manifests. Dynamic content is rendered through safe DOM paths or constrained values.

The Forge backend accepts only known token and route types, applies security headers, disables caching of sensitive responses, bounds request and response bodies, and rate-limits relevant operations. Outbound ProofRails access is scoped to configured HTTPS destinations and tenant credentials.

Keepers automate bounded public or authorized maintenance: delegation-reward claims, delegation synchronization, gauge fee collection, fee synchronization, and evidence jobs. They do not trigger a BBX mint or incentive epoch. Keepers are not a substitute for protocol liveness monitoring. Every keeper action must be safe to retry, constrained by contract state, and observable through logs or receipts.

FORGE / 16

16. Security architecture

16.1 Assets to protect

The main protected assets are:

- AMM reserves and LP fee claims;
- staked BBX principal and multi-token rewards;
- gauge-held LP principal, bribe liabilities, deployer-funded pool reward balances, and launch funds;
- hpFLR liquid and externally staked principal;
- funded redemption and validator-reward liabilities;
- governance voting integrity and upgrade authority;
- privacy note witnesses and disclosure keys;
- ProofRails signer and anchor keys;
- tenant-scoped evidence and API credentials; and
- exact source, deployment, and configuration provenance.

16.2 Adversary classes

The design considers ordinary malicious users, flash-liquidity borrowers, hostile token contracts, compromised frontends, malicious route providers, stale or manipulated data, unauthorized role holders, compromised operators, dishonest root posters, malicious archive suppliers, tenant-crossing API clients, SSRF targets, queue duplication, dependency compromise, and deployment error.

It does not assume that external P-chain custody, the stake mirror, root-poster operation, governance keys, or production cloud systems are cryptographically trustless. Those are controlled trust boundaries.

16.3 Core invariants

The implementation and regression suites are organized around invariants including:

1. hpFLR minting is exactly one-for-one with received FLR.
2. Only the Vault can mint or burn hpFLR.
3. Vault recognized assets do not fall below its modeled liabilities after safety-sensitive transitions.
4. Funded queue FLR corresponds to claimable FIFO allocations; claims burn matching escrowed hpFLR.
5. Reward claims cannot exceed the posted allocation total or funded epoch amount.
6. Historical reward weight derives from finalized checkpoints and committed randomness.
7. AMM post-fee reserves preserve the selected invariant.

8. Pair fees, staking fees, bribes, and third-party gauge rewards cannot be claimed more than once or by later ownership.
9. Aggregated routes cannot call arbitrary targets, break token continuity, or leave active execution assets stranded.
10. Governance voting power is read at the relevant historical timestamp.
11. Supported GaugeFactory and GaugeManager deployments cannot configure or distribute a protocol primary reward, and every retained governance-token minter or rebase-distributor binding is zero.
12. Evidence verification trusts neither a bundled public key nor an anchor event without independent policy.
13. A ProofRails receipt cannot cite an unreconciled or differently modeled settlement in production.
14. Production deployment cannot proceed from untracked, dirty, or different source than the internally approved commit.
15. A shielded leaf binds the note witness to the token and amount actually deposited; withdrawal requires a known root and an unused nullifier hash.
16. Enabling the shielded interface requires the deployment, runtime attestation, signer/recorder policy, circuit hashes, and completed ProofRails rehearsal to agree.

16.4 Defense in depth

Controls are layered rather than treated as substitutes:

- a user minimum output and pair invariant protect ordinary AMM execution;
- pair-level FTSO output ceilings protect direct calls;
- aggregator prechecks and postchecks protect routed execution;
- factory authorization protects pair identity;
- deployment verification protects live wiring;
- frontend checks explain policy without becoming the policy.

The same pattern appears in hpFLR: caps constrain operational release, solvency checks constrain accounting, the stake mirror supplies observed state, divergence pauses constrain anomalies, and public reconciliation detects incorrect operation. ProofRails similarly combines settlement reconciliation, deterministic content, signature, independently trusted signer identity, complete archive hashing, anchor contract identity, trusted operator identity, and confirmations.

16.5 Pausing and liveness

Emergency controls are asymmetric. Stopping new risk should not automatically stop users from receiving already funded claims. hpFLR deposit and reward-recognition pauses leave native claims live. Forge's factory can pause swaps while fee claims and other safe accounting paths remain separately governed. ProofRails can stop writes when production trust configuration is invalid while retaining read-only verification where safe.

No pause design guarantees liveness if Flare itself halts, an external custodian refuses to return funds, governance keys are unavailable, infrastructure is destroyed, or a required oracle never recovers. These scenarios require documented incident and recovery procedures.

FORGE / 17

17. Privilege and control matrix

AUTHORITY	MAY DO	CANNOT DO BY THAT AUTHORITY ALONE
Forge governance / timelock	Execute exposed proposals, manage governed configuration, authorize reviewed upgrades	Bypass contract caps or spend unrelated hpFLR/ProofRails assets
Pair fee manager	Set bounded fee policy, handlers, oracle policy, and factory wiring	Exceed fee caps or create an identity-mismatched pair
Gauge / voter managers	Create fee-only gauges, account votes, route fees, and manage eligible bribes	Configure a protocol primary reward, mint the governance token, or drain user principal
Approved pool incentive deployer	Create, own, fund, renew, and disclose an external-token rewarder for its assigned gauge	Configure another pool, use the governance token, spend protocol assets, or block LP withdrawal
hpFLR Ops	Release available principal to the configured staking destination, report returns, call allowed reward managers	Change destination, release reward liabilities or unavailable principal, upgrade modules, or post reward roots
hpFLR Guardian	Activate bounded emergency pauses	Redirect funds, change governance, or disable all existing claims permanently
hpFLR Root Poster	Post reproducible roots within funded amounts	Release principal, overfund an epoch, or upgrade the system
hpFLR timelock	Configure and upgrade operational modules, correct a pre-claim root	Alter an immutable hpFLR token implementation in place
P-chain provider	Operate externally released stake	Mint hpFLR or alter EVM checkpoints directly
ProofRails API project key	Create and read records for its project	Read another project's evidence or operate anchor keys
ProofRails worker	Sign evidence and submit authorized anchors	Change on-chain owner policy without owner authority
EvidenceAnchor owner	Manage operators and two-step ownership	Rewrite an already anchored hash or validate archive content
EvidenceAnchor operator	Anchor new nonzero hashes	Transfer ownership or make a hash semantically true

Production must assign these roles to distinct keys or multisig/timelock identities where the threat model requires separation. The same address must not silently accumulate mutually checking roles.

FORGE / 18

18. Assurance and verification record

The current remediated source underwent a full internal source, architecture, dependency, deployment, and regression review. As of this paper's date, the assessment records zero open actionable Critical, High, Medium, or Low implementation findings identified by that review.

Representative verified results are:

PROJECT	VERIFICATION RESULT
Forge contracts	113 focused Hardhat tests pass, including a circuit-compatible multi-deposit Merkle regression
Forge application/tooling	17 Node files, three adapter files, Pages/Functions builds, public browser interaction smoke, and production audit pass
hpFLR contracts	251 Solidity tests across 32 suites pass on the last pinned receipt and in the current local revalidation
hpFLR invariants	Five reward and six Vault stateful invariants run at 128,000 calls each
hpFLR tooling	Standalone Node operations suites plus both distributor application suites pass
Live hpFLR runtime	Current module upgrade passed on a live-state fork and Coston2; the resulting deployment manifest passes 145/145 checks
ProofRails service	50 Python tests, Python bytecode compilation, Solidity compilation, Compose runtime rehearsal, and Node audits pass
Live Forge runtime	4 pools, 4 gauges, 3 vaults, 3 venues, 12 oracle policies, 8 protected probes, 36 UI bindings, and 71 read probes pass
Live privacy runtime	Replacement graph attested; three full rehearsals pass; two-deposit event reconstruction equals the on-chain root
Contract size	Current deployable implementations retain EIP-170 margin; RewardDistributor is the largest hpFLR implementation at 22,461 bytes
Secret scan	No embedded private-key or seed material was detected in active source

One no-fix upstream elliptic advisory remains only in the Forge development dependency graph. It is absent from production and key-bearing runtime paths and is tracked as a contained dependency-scanner exception, not an application vulnerability.

This evidence is not a mainnet approval and is not a guarantee that no unknown defect exists. The exact hpFLR pinned-tool gate must rerun after the source is frozen; the locally available Forge 1.1.0 is deliberately rejected by the 1.5.1 release policy. The ProofRails production image and Python dependency scan, hpFLR fork tests, hpFLR Anvil smoke, and live Coston2 lifecycle rehearsals remain environment-specific release evidence. The detailed assessment is in SECURITY-ASSESSMENT-2026-08-27.md.

FORGE / 19

19. Mainnet release discipline

Mainnet is a controlled transition from exact source to exact deployed state. It is not a network flag changed on a test deployment.

19.1 Source and build provenance

- Freeze a clean, tracked commit for Forge, hpFLR, and ProofRails.
- Run all pinned compilers, tests, invariants, storage-layout checks, size gates, formatters, and dependency checks against that commit.
- Record build inputs, compiler versions, library links, implementation bytecode, proxy initializers, container image digest, and artifact hashes.
- Perform an internal exact-commit release review and retain the signed or hashed approval evidence.

19.2 Fresh rehearsal

- Deploy the exact release candidate freshly to Coston2.
- Do not treat older Coston2 addresses as proof for remediated bytecode.
- Exercise deposits, redemptions, reward epochs, swaps, routing, fee claims, gauges, launches, delegation harvest, evidence generation, verification, role transfer, pause recovery, and upgrade rehearsal.
- Verify both expected privileges and expected absence of privileges.

Disposable Coston2 keys may continue to be used for test-only rehearsals because they protect no production value. They must never be promoted to mainnet. Mainnet requires newly generated, independently held deployer, governance, guardian, operations, root-poster, ProofRails signer, anchor operator, owner, database, and API credentials as applicable.

19.3 Live configuration verification

- Resolve Flare system contracts and canonical epoch parameters from current authoritative sources.
- Confirm all implementation addresses, proxy admins, owners, timelocks, delays, caps, token feed IDs, staleness settings, oracle policies, pair creators, delegation providers, reward managers, eligible positions, exclusions, anchor bytecode, and trusted signer fingerprints.
- Confirm deployer privilege renunciation and pairwise separation of checking roles.
- Pin ProofRails source commit to an immutable image digest and the deployed EvidenceAnchor attestation.

19.4 Operational readiness

- Establish monitoring for solvency, mirror divergence, pauses, pending stake, redemption depth, reward-root deadlines, root replacements, abnormal FTSO failures, keeper failures, governance proposals, upgrades, anchor backlog, and failed evidence jobs.
- Test backups and restoration for PostgreSQL, Redis durability where required, artifact storage, configuration evidence, and public allocation artifacts.
- Publish incident response and communication procedures.
- Open mainnet functionality incrementally, with value-bearing privacy disabled.

The project does not require an external audit as a release condition. Its release model requires comprehensive internal exact-commit review, reproducible evidence, rehearsals, and fail-closed gates. This is a statement of the chosen assurance process, not a claim that internal review eliminates all risk.

FORGE / 20

20. Risks and limitations

20.1 Smart-contract risk

An unknown defect may exist in contract logic, compiler behavior, linked libraries, proxy configuration, or integration assumptions. Upgradeability can repair defects but introduces governance risk. Immutable components reduce upgrade risk but cannot be patched in place.

20.2 Market and liquidity risk

AMM users face price impact, impermanent loss, volatile collateral value, fee-on-transfer or nonstandard-token behavior where permitted, oracle divergence, and insufficient route liquidity. FTSO protection is a deviation guard, not a guarantee of best execution or an insurance policy.

20.3 Oracle and Flare-system risk

Protected operations depend on fresh, correctly mapped Flare data and stable interfaces. Incorrect feeds, delayed updates, network upgrades, or a fee-bearing interface change may halt guarded operations or admit incorrect references if deployment policy is wrong.

20.4 hpFLR custody and solvency risk

FLR released to P-chain operations leaves direct EVM custody. The provider may lose access, suffer slashing or operational loss, delay return, or report state incorrectly. The PChainStakeMirror may be stale or dishonest. Caps, accounting, impairment, and pause controls limit or expose risk but do not make external custody trustless.

One-to-one redemption is an accounting obligation, not a guarantee of immediate liquidity. A redemption may wait in FIFO order until sufficient FLR returns to the Vault.

20.5 Root-poster and data-availability risk

A malicious or unavailable Root Poster can delay a reward epoch or publish an incorrect allocation within the funded cap. The design relies on complete artifact publication, independent reproduction, challenge time, and timelock correction before claims. If observers do not verify or data is unavailable, an incorrect root may become operationally final after the first claim.

20.6 Governance risk

Concentrated voting positions, compromised governance keys, voter apathy, bribery, malicious upgrades, or parameter misuse can damage the protocol. Timelocks give observers time; they do not guarantee that observers act.

20.7 Infrastructure and key risk

Backends, workers, RPC providers, databases, queues, storage, DNS, TLS, callbacks, and signing infrastructure may fail or be compromised. Production separation and pinning narrow the effect of a compromise but cannot eliminate operational security.

20.8 Proof and evidence limitations

An on-chain anchor proves existence of exact bytes no later than an event under the selected confirmation policy. A signature proves control of a key. Settlement reconciliation proves modeled transfer facts. None independently proves the legality, commercial purpose, tax treatment, source of funds, human identity, or truth of arbitrary metadata.

20.9 Privacy limitations

The privacy subsystem is enabled only for the Coston2 rehearsal and uses a test proving setup. It hides the deterministic Merkle membership path and deposit-to-withdrawal link, not the public token, amount, sender/recipient transactions, relayer, fee, root, nullifier hash, block timing, RPC use, IP address, wallet telemetry, or later asset movement. A unique amount, small anonymity set, short delay, address reuse, or recognizable follow-on behavior may make correlation likely.

User privacy depends on correct browser code, cryptographic randomness, same-origin artifact integrity, backup confidentiality, password strength, and uncompromised user and compliance keys. Loss of the user's witness backup can make funds practically unspendable. Compromise of the user view key reveals the user's note. Compromise of the compliance key can reveal compliance plaintext outside the official threshold workflow. Threshold authorization constrains sanctioned disclosure and evidence creation; it cannot prevent misuse of a stolen decryption key.

20.10 Regulatory and tax uncertainty

Tokens, staking, swaps, incentives, launches, privacy technology, and payment evidence may be regulated differently across jurisdictions. Users and deployers remain responsible for applicable legal, tax, reporting, sanctions, consumer-protection, and licensing obligations.

FORGE / 21

21. Development roadmap

The roadmap is organized by evidence, not arbitrary dates.

Phase A — remediated release candidate

- Preserve zero known actionable findings from the internal assessment.
- Freeze clean commits and complete all local and CI quality gates.
- Finalize documentation, parameter registry, operational runbooks, and deployment manifests.

Phase B — fresh Coston2 rehearsal

- Core Forge and the replacement privacy graph are deployed and runtime-attested on Coston2.
- The current web candidate is published, live-browser tested, and bound to the privacy rehearsal.
- The hpFLR native C/P round trip, small C-chain lifecycle, cap-removal upgrade, and canonical validator-reward claim authorization are complete; current modules are live and pass 145/145 independent deployment checks.
- Fund the twelve independent faucet holders, then complete the prepared aggregate hpFLR minimum-duration delegation, mirror, real validator reward, return, and FIFO redemption lifecycle.
- Freeze the exact candidate commit and repeat the passing runtime evidence against that immutable source identity before selecting a mainnet candidate.

Phase C — non-privacy Flare launch candidate

- Generate distinct production credentials and custody arrangements.
- Verify live Flare dependencies and finalize governed parameters.
- Deploy, attest, transfer roles, renounce deployer access, and complete a low-value end-to-end rehearsal.
- Open functionality incrementally under monitoring and incident controls.

Phase D — operational decentralization

- Broaden independent reward-artifact reproduction.
- Improve public monitoring of root, solvency, mirror, gauge, governance, and anchor state.
- Move eligible authorities to appropriately separated timelocks and multisigs.
- Evaluate reductions in single-operator dependencies where Flare and custody architecture permit.

Phase E — optional privacy production work

- Preserve the completed Coston2 atomic deployment, multi-deposit audit, and full ProofRails rehearsal as test evidence rather than treating it as a production setup.
- Freeze the final circuit.
- Complete and verify a multiparty setup.
- Finish client, relayer, recovery, compliance, and disclosure policy.
- Conduct a dedicated internal exact-commit privacy review and fresh rehearsal; and
- enable only after its separate release gate passes.

No roadmap phase promises deployment, token value, yield, or regulatory acceptance.

FORGE / 22

22. Parameter classification

22.1 Current protocol constants and hard bounds

PARAMETER	CURRENT IMPLEMENTATION
BBX supply behavior	Fixed at constructor deployment; no subsequent mint role
Forge gauge primary reward token	Disabled; active factories require address(0)
Optional pool rewarder	One deployer-owned external token per gauge; deployer-selected funded stream duration
Maximum ve lock	4 years
BBX staking vote multiplier	1× to 4×
Multiplier ramp	3 weeks
Forge swap fee maximum	0.25% of input
Protocol fee-share maximum	20% of applicable swap fee remainder
Referral fee-share maximum	12% of swap fee
Staking fee-share maximum	30% of applicable fee remainder
Aggregator maximum hops	4
Aggregator maximum split routes	4
FTSO feed batch maximum	64
FTSO deviation maximum	5,000 basis points
FTSO freshness configuration range	90–3,600 seconds
Forge staking reward-token maximum	32
hpFLR staking release	Uncapped by time/amount; fixed destination, role, availability, and solvency constrained
ProofRails evidence inventory	Exactly 6 files

22.2 Implementation defaults or candidate deployment values

PARAMETER	CURRENT VALUE	CLASSIFICATION
Stable-pair swap fee	0.04%	Factory initialization default; governed within cap

PARAMETER	CURRENT VALUE	CLASSIFICATION
Volatile-pair swap fee	0.18%	Factory initialization default; governed within cap
Protocol fee share	15%	Factory initialization default; governed within cap
Referral fee share	0%	Factory initialization default; governed within cap
Staking fee share	0% default; 25% in Coston2 script	Governed within cap
FTSO freshness	300 seconds	Oracle default; governed within allowed range
LP delegation reward stream	7 days	Current implementation policy
Total BBX at Coston2 deployment	1,000,000,000 BBX	Candidate deployment configuration

hpFLR challenge windows, claim windows, snapshot counts, divergence thresholds, eligible positions, and ProofRails confirmation policies are deployment inputs subject to contract bounds and release verification. Staking releases are not amount- or time-capped: the authorized Ops account can move the full available reserve only to the fixed timelock-configured custody destination. These trust parameters and custody assignments must be published with the production manifest rather than inferred from examples.

FORGE / 23

23. Contract and component map

Forge core

- **BBX:** fixed-supply governance-only token.
- **VotingEscrow:** time-locked veNFT voting positions.
- **ForgeStaking:** liquid BBX staking, multi-token fee rewards, and historical vote multipliers.
- **BlackGovernor:** proposal lifecycle and historical vote accounting.
- **VoterV3 / GaugeManager:** gauge voting, fee/bribe attribution, lifecycle management, and enforcement of the disabled-primary-reward policy.
- **GaugeV2 / Bribes / GaugeFactory / GaugeIncentiveFactory:** fee-only LP custody, fee routing, external bribes, and optional external rewards owned and funded by each pool's approved deployer.
- **Pair / PairFees / PairFactory / PairGenerator:** stable and volatile AMM creation, reserves, and segregated fees.
- **RouterV2 / GlobalRouter / ForgeDexAggregator:** liquidity and typed swap execution.
- **Algebra compatibility modules:** optional concentrated routing, custom pool deployment, pool provenance, and concentrated gauges; not active merely because source is present.
- **PermissionsRegistry / TokenHandler:** bounded protocol roles and on-chain token, connector, and volatility eligibility.
- **FTSOPriceOracle:** Flare feed mapping, freshness, normalization, and execution-band policy.
- **ForgeLPVault / FTSOYieldDistributor:** ERC-4626 LP custody and streamed WFLR delegation rewards.
- **WFLRDelegationManager / DataProviderRegistry:** bounded provider delegation and reward harvesting.
- **LaunchFactory / LaunchManager / LaunchPool / PriceAuction:** launch state, funding, liquidity, and price formation.
- **AutoVoter / AutoVotingEscrow managers and strategies:** delegated governance automation.
- **MiMCSponge / Groth16Verifier:** circuit-compatible commitment hashing and verification of the seven withdrawal public inputs.
- **ForgeShieldedPool / MerkleTreeWithHistory:** opt-in deposits, incremental commitment tree, known-root history, nullifier spending, and native/ERC-20 withdrawal accounting.
- **ForgeNoteRegistry / ForgeComplianceOracle:** bound ciphertext registry, threshold exact-byte disclosure approvals, recorder authorization, and immutable evidence-receipt write-back.
- **Browser privacy runtime / Forge ProofRails adapter:** local witness creation, encrypted backup, hash-pinned proof generation, confirmed event reconstruction, disclosure verification, and public-settlement projection.
- **Retired compatibility modules:** MinterUpgradeable, RewardsDistributor, SeasonalRewards, GaugeCL, GaugeFactoryCL, and EpochController remain in source for historical analysis but are not active deployment components.

hpFLR

- **HpFLR:** immutable feeless token, permit, and historical checkpoints.
- **Vault:** deposits, minting, solvency, staking release and return, loss, and reward recognition.
- **RedemptionQueue:** one-way FIFO escrow, funding, claim, and burn.
- **RewardDistributor:** canonical epochs, committed randomness, Merkle roots, claims, restaking, and rollover.
- **RecipientRegistry:** contract-holder claimant and recipient control.
- **EligiblePositionRegistry:** governed recursive attribution of LP, gauge, and vault ownership.
- **HpTimelock:** delayed governance and upgrade control.

ProofRails

- **FastAPI service:** authenticated receipt, artifact, verification, statement, and administration API.
- **PostgreSQL:** durable project-scoped receipt and identity state.
- **Redis/RQ:** durable job transport, locks, recovery, and event coordination.
- **Worker:** settlement reconciliation, artifact construction, signing, and anchoring.
- **EvidenceAnchor:** owner/operator-controlled one-time bundle-hash anchoring.

FORGE / 24

24. Glossary

- **AMM:** automated market maker.
- **BBX:** fixed-supply Forge governance-only token.
- **Basis point:** one hundredth of one percent; 10,000 basis points equal 100 percent.
- **Challenge window:** time between hpFLR root publication and the opening of claims.
- **Coston2:** Flare's test network used for rehearsal; its assets and keys are not mainnet assets or keys.
- **FTSOv2:** Flare Time Series Oracle version 2.
- **Gauge:** contract holding eligible LP positions, routing pair fees, and optionally integrating an external-token rewarder owned and funded by that pool's approved deployer.
- **hpFLR:** one-to-one FLR principal receipt with separately distributed validator rewards.
- **LP:** liquidity provider or liquidity-position token.
- **Merkle root:** compact commitment supporting inclusion proofs for a set of allocations.
- **P-chain:** Flare's staking environment outside ordinary EVM contract custody.
- **ProofRails:** integrated signed evidence and on-chain anchoring service.
- **Root Poster:** hpFLR role authorized to publish a reproducible reward allocation commitment.
- **Shielded transfer:** an opt-in deposit and zero-knowledge withdrawal whose specific linkage is hidden while token, amount, transactions, timing, and withdrawal recipient remain public.
- **Nullifier hash:** public one-time spend identifier derived from a private note nullifier.
- **veNFT:** transferable NFT representing a vote-escrowed BBX position.
- **WFLR:** wrapped native FLR used by EVM contracts and Flare delegation.

FORGE / 25

25. References

Repository specifications:

- Forge README
- Forge deployment and wiring requirements
- Forge current status
- hpFLR README
- hpFLR reward fairness specification
- hpFLR privileged-role threat model
- hpFLR handover and deployment guide
- ProofRails implementation specification
- ProofRails security and deployment policy
- ProofRails API documentation
- Comprehensive internal security assessment

External technical references:

- Flare developer documentation: https://dev.flare.network/
- Flare FTSO documentation: https://dev.flare.network/ftso/
- ProofRails documentation: https://docs.proofrails.com/
- ProofRails evidence and trust model: https://docs.proofrails.com/architecture/evidence-model
- Current upstream ProofRails v2.2 repository: https://github.com/proofrails/middleware-ISO-v2.2-and-x402
- ISO 20022 overview: https://www.iso20022.org/
- ERC-20: https://eips.ethereum.org/EIPS/eip-20
- ERC-2612 permit: https://eips.ethereum.org/EIPS/eip-2612
- ERC-4626 tokenized vaults: https://eips.ethereum.org/EIPS/eip-4626
- ERC-1271 contract signatures: https://eips.ethereum.org/EIPS/eip-1271

FORGE PROTOCOL / TECHNICAL PAPER

Conclusion

Forge combines exchange liquidity, governance-directed fee attribution, Flare-native yield, principal-preserving liquid staking, opt-in shielded transfers, and verifiable settlement evidence without treating them as one undifferentiated pool of trust.

The AMM enforces stable or volatile invariants and separates reserves from fee claims. The router and aggregator constrain execution to typed, continuous routes and can bind protected swaps to fresh FTSOv2 references. BBX governance uses fixed supply and historical voting power without a protocol incentive stream. hpFLR keeps principal one-for-one while distributing validator rewards through committed historical ownership. The shielded pool hides a note's specific deposit-to-withdrawal link while keeping its narrower public-data and key-management limitations explicit. ProofRails binds exact reconciled evidence bytes to an independently trusted signer and confirmed on-chain anchor.

The remaining risks are explicit: contracts can contain unknown defects; governance and upgrades are powerful; P-chain custody and stake-mirror accuracy are trusted; reward roots require active verification; infrastructure and keys must be operated securely; and an evidence anchor proves bytes, not every real-world assertion. Mainnet readiness therefore depends on the exact release evidence and operational controls, not on this paper alone.

Forge Protocol

Flare-native liquidity, liquid staking, opt-in shielded transfers, governance, and verifiable settlement evidence.

Implementation-aligned draft v1.3 — 30 August 2026