GenesisL1 Forest (GL1F)

A Scientific Explanation & Technical Documentation for an On-Chain EVM Gradient-Boosted Tree Studio

Prepared for the GenesisL1 Forest alpha version source release by the Decentralized Science Labs

January 19, 2026

Abstract

GenesisL1 Forest is a browser-only "model studio" that trains gradient-boosted decision tree (GBDT) ensembles locally, serializes them into a compact binary format, deploys them on the GenesisL1 network as ERC-721 "Model NFTs", and supports deterministic on-chain inference through a specialized runtime smart contract. This document is intentionally hybrid: Part I is written in the style of a scientific paper (design goals, algorithmic choices, and formal specifications), and Part II is written as technical documentation (developer workflows, contract APIs, and implementation details). Throughout, we emphasize constraints imposed by Layer-1 execution (gas, determinism, data availability), and we show how GenesisL1 Forest uses code-as-data chunk contracts and fixed-point arithmetic to make on-chain inference practical.

Contents

List of Figures

List of Tables

Part I

Scientific Paper

Chapter 1

Introduction

1.1 What is GenesisL1 Forest?

GenesisL1 Forest ("Forest" for short) is a self-contained web application that runs entirely in the browser and connects to the GenesisL1 network (chain ID 29). It supports:

- training fixed-depth gradient-boosted decision trees (GBDTs) for regression, binary classification, multiclass classification, and multilabel classification;
- serializing trained models into a compact binary format (`GL1F` v1 and v2);
- deploying model bytes to an L1 chain using chunk contracts whose runtime bytecode begins with `GL1C`;
- minting a corresponding ERC-721 token ("Model NFT") holding human-facing metadata; and
- performing deterministic inference on-chain via a dedicated runtime contract.

Terminology

In this document, "forest" refers to an ensemble of decision trees (the classic machine-learning meaning), and GenesisL1 refers to the EVM-compatible Genesis L1 network (chainId 29). Chain parameters and common wallet configuration are publicly listed by community registries.[9]

1.2 Why does this matter? (Importance)

Deploying ML models on-chain is rarely practical with state-of-the-art neural networks, but tree ensembles occupy a sweet spot: inference is a small set of integer comparisons and additions. This makes them attractive for L1 use cases that require:

- **verifiability**: anyone can reproduce inference from public model bytes and public input features;
- **availability**: model parameters remain accessible as long as the chain is accessible;
- **composability**: other contracts can call into the model runtime and build applications on top;

- **market mechanisms**: ownership, licensing, and paid access can be expressed as smart contracts; and
- **education and auditability**: model structure is inspectable, and the runtime behavior is deterministic.

Forest specifically targets the intersection of (i) reproducible inference, (ii) lightweight models, and (iii) a developer experience that does not require local ML toolchains.

1.3 Contributions and scope

This paper/documentation contributes:

1. a formal specification of the `GL1F` binary formats used by Forest;
2. a reference architecture for storing large immutable model blobs as contract bytecode (*code-as-data*), akin to the SSTORE2 pattern;[7]
3. an on-chain inference procedure that supports paywalled usage without relying on spoofable `eth_call` caller addresses;[5, 6]
4. an end-to-end workflow: dataset ingestion → in-browser training → on-chain deployment → inference and marketplace.

Chapter 2

Background

2.1 Gradient boosting decision trees

Gradient boosting constructs an additive model of the form

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \nu \, f_m(x), \tag{2.1}$$

where each f_m is a weak learner (here: a fixed-depth decision tree), ν is a learning rate, and each new tree is fit to the negative gradient of a loss function. The classical formulation and many practical variants are described by Friedman (2001).[1]

In Forest, trees are complete binary trees of a fixed depth d. Each internal node stores a feature index f and a threshold τ; each leaf stores a value. Inference is a deterministic traversal based on comparing the input feature value to the node threshold.

2.2 Why fixed-depth?

Variable-depth trees encode structure compactly, but fixed-depth trees simplify on-chain decoding:

- the number of internal nodes is $2^d - 1$, and leaves 2^d;
- array indexing is arithmetic; no pointers are needed;
- byte offsets become compile-time-like expressions, enabling efficient `EXTCODECOPY`-based reads.

2.3 On-chain constraints

Smart-contract computation is constrained by gas costs, determinism, and limited access to external data. Two constraints are especially relevant:

- **Contract code size limit:** EIP-170 caps runtime code size at 24,576 bytes on Ethereum mainnet and many compatible chains.[4]
- **RPC call semantics:** `eth_call` accepts an optional `from` field, meaning off-chain callers can simulate calls "as if" from arbitrary addresses.[6]

Forest's design uses these constraints as primitives: the code size limit motivates chunking; spoofable `eth_call` motivates signature-gated free inference for paid models.

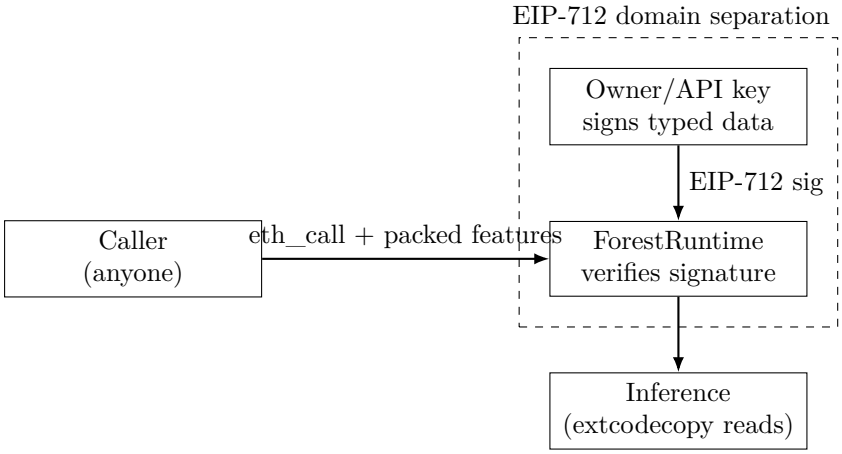

Figure 2.1: Fixed-depth binary tree indexing used by Forest.

Chapter 3

System Architecture

3.1 High-level overview

Forest consists of three layers:

1. **Browser application:** dataset ingestion, model training (WebWorker), visualization, packaging, and deployment transactions.
2. **Smart contracts:** registry (metadata, pricing, access), NFT contract, marketplace, model storage chunks, and on-chain runtime inference.
3. **Genesis L1 network:** an EVM-compatible chain (chainId 29) reachable via JSON-RPC (e.g., `https://rpc.genesisl1.org`).[9]

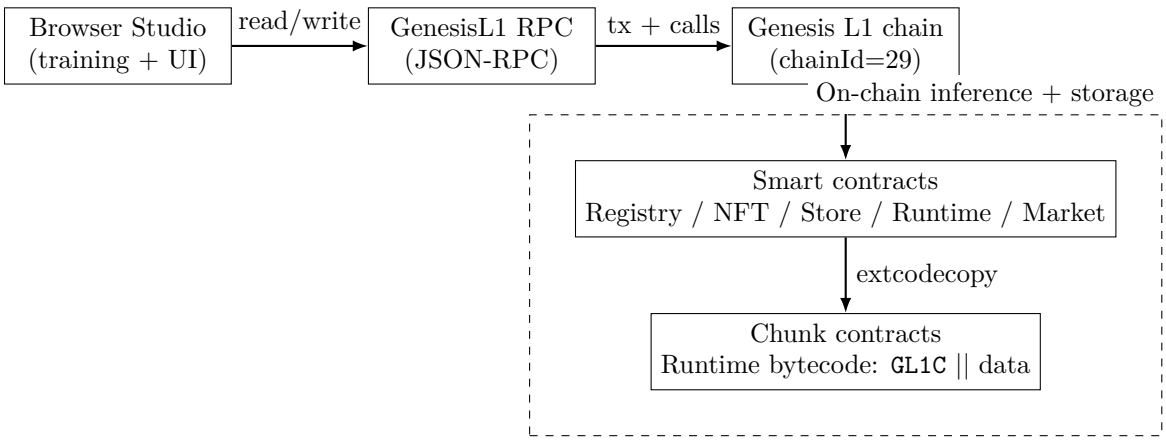

Figure 3.1: GenesisL1 Forest architecture: the browser performs training and deployment, while inference and model availability are enforced on-chain.

3.2 Model lifecycle

A model in Forest typically follows this lifecycle:

1. **Train:** user trains a GBDT on a dataset in the browser.
2. **Serialize:** model parameters are quantized into fixed-point integers and encoded as `GL1F` bytes.
3. **Store bytes:** bytes are split into chunks; each chunk is deployed as a pointer contract (`GL1C` magic) using the on-chain store.
4. **Publish:** a pointer-table contract referencing the chunks is deployed; the registry mints an NFT with metadata and binds it to a model ID.
5. **Infer:** users call the runtime for view or paid inference (depending on pricing mode).
6. **Trade:** the NFT can be listed and sold; ownership affects fee-free privileges and access-key control.

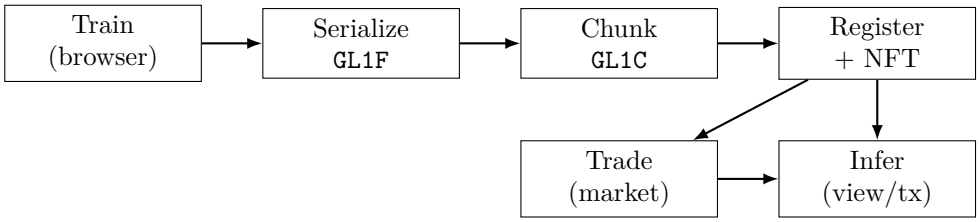

Figure 3.2: Model lifecycle in Forest: from in-browser training to on-chain inference and marketplace trading.

Chapter 4

Binary Formats and Storage

4.1 Chunk contracts (`GL1C`)

Forest stores model bytes on-chain by deploying contracts whose runtime bytecode is:

```
GL1C (4 bytes) || DATA (0..24,572 bytes)
```

This is a variant of the well-known "store-as-code" pattern (often called SSTORE2).[7]

EIP-170 limits runtime bytecode size to 24,576 bytes, motivating the maximum chunk payload size of 24,572 bytes (reserving 4 bytes for the magic prefix).[4]

4.2 Pointer table

To address models larger than a single chunk, Forest stores a *pointer table* as another `GL1C` contract. The table payload is a sequence of 32-byte words, each containing an address in its low 20 bytes (left-padded with zeros). The runtime inference engine reads this table via `EXTCODECOPY` to locate chunk addresses.

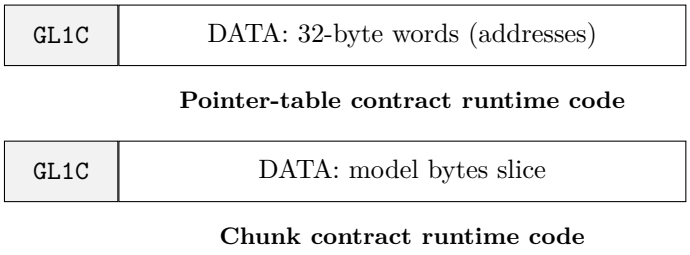

Figure 4.1: Both the pointer table and the chunks are `GL1C` contracts. The runtime reads pointers from the table and then reads model bytes from the chunks.

4.3 Model format `GL1F` v1 (scalar output)

The v1 format encodes regression and binary classification models with a single scalar logit/score output. It begins with a 24-byte header and then stores M trees.

Offset	Field	Notes
0..3	magic	ASCII `GL1F`
4	version	1
6..7	nFeatures	uint16 LE
8..9	depth	uint16 LE
10..13	nTrees	uint32 LE
14..17	baseQ	int32 LE (fixed-point)
18..21	scaleQ	uint32 LE (feature/value scale)
24..	trees	see below

Table 4.1: `GL1F` v1 header. (Reserved bytes omitted for brevity.)

Each tree stores internal nodes followed by leaves. With depth d:

$$\#\text{internal} = 2^d - 1, \tag{4.1}$$

$$\#\text{leaves} = 2^d, \tag{4.2}$$

$$\text{bytes/tree} = (2^d - 1) \cdot 8 + 2^d \cdot 4. \tag{4.3}$$

v1 layout (bytes 0..23 header, then tree blocks)

Figure 4.2: `GL1F` v1 byte layout (schematic).

4.4 Model format `GL1F` v2 (vector output)

The v2 format supports multiclass and multilabel classification. It adds (i) an explicit number of classes/labels and (ii) per-class base logits. Trees are stored class-major: all trees for class 0, then class 1, and so on.

Offset	Field	Notes
0..3	magic	ASCII `GL1F`
4	version	2
6..7	nFeatures	uint16 LE
8..9	depth	uint16 LE
10..13	treesPerClass	uint32 LE
18..21	scaleQ	uint32 LE
22..23	nClasses	uint16 LE
24..	baseLogitsQ	int32 LE $\times$ nClasses
...	trees	class-major blocks

Table 4.2: `GL1F` v2 header fields.

4.5 Why fixed-point (Q) values?

Floating point is not available in Solidity. Forest uses int32 fixed-point representations:

- input features are packed as int32 little-endian values, representing $x \cdot$ `scaleQ`;
- thresholds and leaf values are stored in the same quantized scale;
- the runtime accumulates to an `int256` to avoid overflow across many trees.

Chapter 5

On-Chain Inference

5.1 Inference algorithm (scalar)

Inference for v1 models is straightforward: for each tree, traverse from the root to a leaf using comparisons $x_f > \tau$. Then add the leaf value to the accumulator. Pseudocode:

Listing 5.1: Reference pseudocode for scalar inference (v1).

```
acc = baseQ
for t in 0..nTrees-1:
  idx = 0
  for lvl in 0..depth-1:
    (f, thrQ) = node(t, idx)
    xQ = featuresQ[f]
    idx = (idx*2 + 2) if xQ > thrQ else (idx*2 + 1)
  leafIndex = idx - (2^depth - 1)
  acc += leaf(t, leafIndex)
return acc
```

5.2 View inference versus paid inference

Forest supports pricing modes:

- mode 0: free; mode 1: tips (fee optional); mode 2: paid-required.

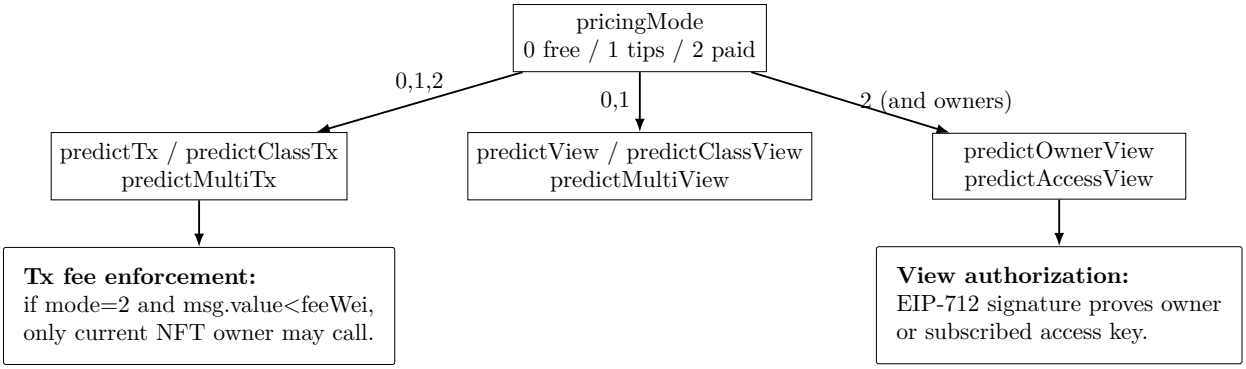

Figure 5.1: Which inference entrypoints are valid under each pricing mode.

A key subtlety is that off-chain read calls can spoof the caller address: `eth_call` includes an optional `from` field.[6] Therefore, *paid-required* models must not allow free view inference gated solely by `msg.sender`. Forest uses two mechanisms:

1. an on-chain transaction path (`predictTx`) that enforces payment unless the current NFT owner calls it;
2. an EIP-712 signature-gated view path (`predictOwnerView` / `predictAccessView`) that proves authorization without relying on the spoofable call sender.[5]

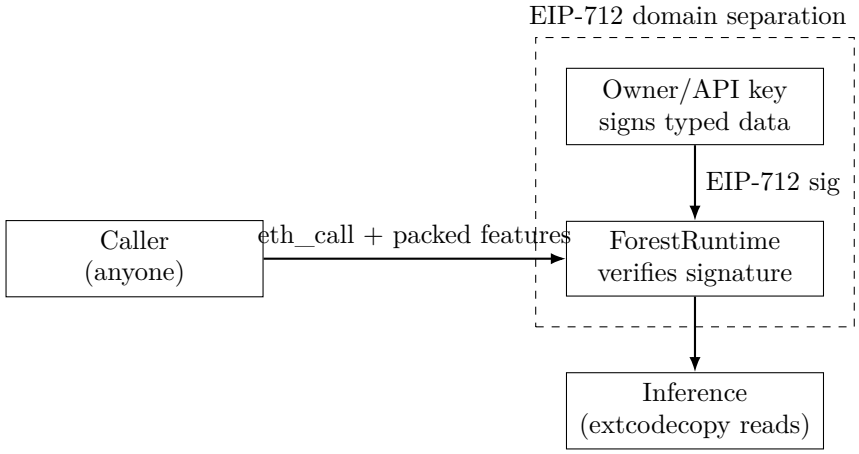

Figure 5.2: Signature-gated view inference prevents fee bypass via spoofed `eth_call` callers.

5.3 Complexity and gas intuition

For scalar v1 models, inference is $O(M \cdot d)$ comparisons plus $O(M \cdot d)$ byte reads for thresholds and feature indices. Reads are performed from contract bytecode via `EXTCODECOPY`, avoiding `SLOAD`. This can be substantially cheaper for medium-sized blobs than storing the entire model in standard storage.[7]

Chapter 6

Governance, Licensing, and Search

6.1 On-chain Terms and license

The registry records an active Terms-of-Service version and an active license identifier; deployments must explicitly accept both. The default license is Creative Commons Attribution-ShareAlike 4.0 (CC BY-SA 4.0).[8]

6.2 Title-word search index

Forest maintains a simple on-chain inverted index mapping word hashes to token IDs. Search performs an AND query by intersecting membership sets (optimized by scanning the first word list). This supports discovery without requiring off-chain indexing infrastructure.

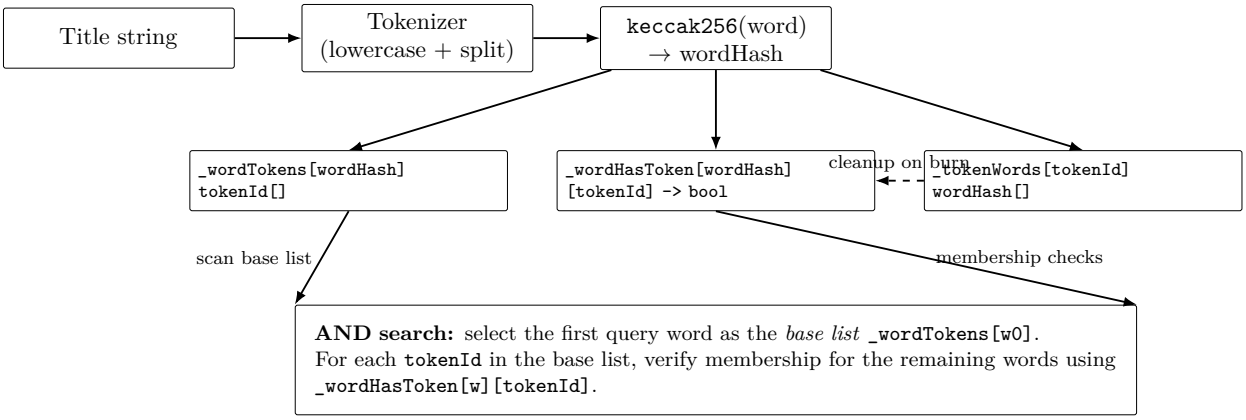

Figure 6.1: On-chain title-word index used for discovery without off-chain indexing.

Chapter 7

Limitations and Future Work

7.1 Limitations

- **Model class:** Only fixed-depth GBDTs are supported. This is intentional to keep inference deterministic and compact.
- **Feature scaling:** Users must supply or accept a scaling factor `scaleQ` to avoid int32 overflow.
- **Privacy:** Inputs provided to on-chain transactions are public. Signature-gated view inference can keep inputs off-chain, but off-chain callers still reveal inputs to their RPC provider.
- **Expressiveness:** Trees are trained in-browser with a pragmatic histogram split search; they are not intended to compete with highly optimized libraries like XGBoost or LightGBM.[2, 3]

7.2 Possible extensions

- support for feature normalization metadata and standardized "model cards";
- optional Merkle commitments to datasets and training hyperparameters;
- more granular access-control primitives and royalty standards.

Part II

Technical Documentation

Chapter 8

Quickstart

8.1 Local development

The repository is a static web application and must be served over HTTP (not `file://`) so module imports work.

Listing 8.1: Serve locally using Python.

```
python3 -m http.server 8080
# open http://localhost:8080/forest.html
```

8.2 Network configuration

Forest targets Genesis L1 (EVM, chainId 29). Public registries list the chain ID and common RPC endpoints.[9]

Chapter 9

End-to-End Workflow

9.1 Dataset ingestion

Forest can ingest CSV files and infer label schemas for several tasks. Typical steps:

1. upload CSV; choose feature columns; choose label column(s);
2. perform numeric conversion and basic cleaning;
3. split into train/val/test with seeded shuffle; stratify for single-label classification.

9.2 Training

Training runs in a WebWorker and supports:

- regression with squared loss;
- binary classification with logistic loss;
- multiclass classification with softmax cross-entropy;
- multilabel classification with independent sigmoids.

Practical detail: scale selection

The create page chooses `scaleQ` to preserve precision while ensuring quantized values fit comfortably in int32, with a safety headroom below 2,147,483,647.

9.3 Serialization and deployment

Deployment consists of $N + 2$ transactions for a model split into N chunks:

1. N transactions: write each model chunk to `ModelStore.write` (deploying `GL1C` pointer contracts);
2. 1 transaction: write the pointer-table (32-byte pointers) as another `GL1C` contract;
3. 1 transaction: call `ModelRegistry.registerModel` to mint an NFT and store runtime parameters.

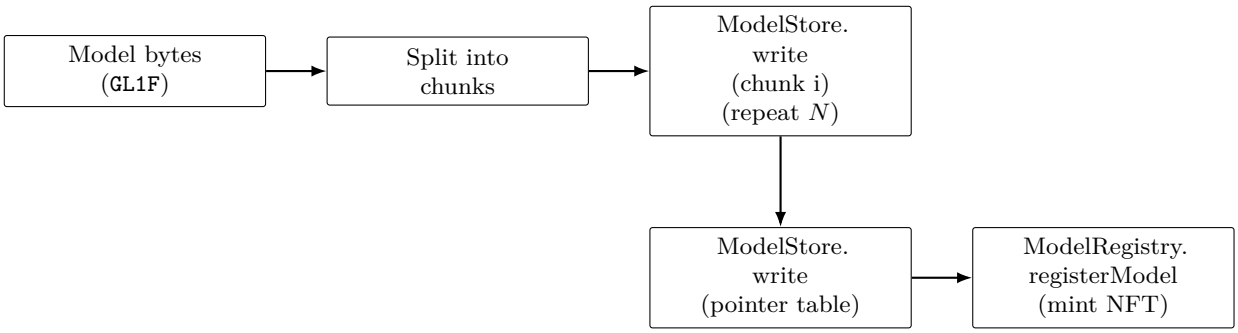

Figure 9.1: Deployment pipeline used by the Create page.

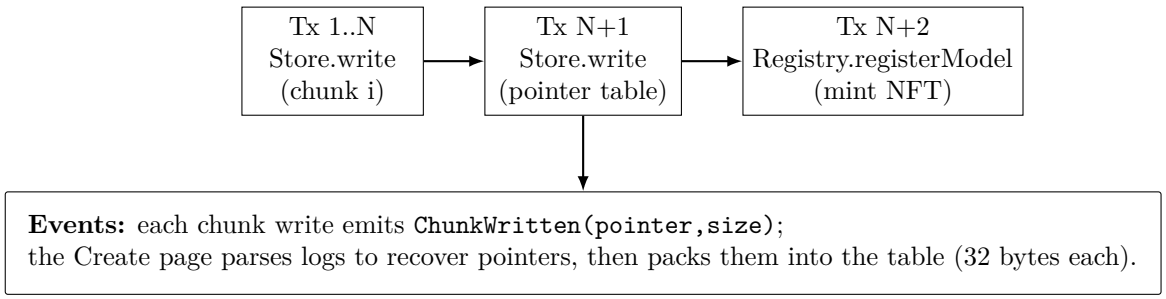

Figure 9.2: Transaction-level deployment sequence.

Chapter 10

Contract Reference

This chapter summarizes the contract suite. Full source listings are included in the appendices.

10.1 ModelStore

Purpose: store immutable byte chunks as runtime code (`GL1C` || DATA). The `write` function deploys a minimal pointer contract whose runtime code is exactly the stored bytes, respecting the EIP-170 size limit.[4]

10.2 ModelRegistry

Purpose: register models, bind them to NFTs, manage pricing and inference enablement, manage access plans, and maintain a title-word search index.

10.3 ForestRuntime

Purpose: read model bytes from chunk contracts and perform deterministic inference. Supports scalar (v1), vector (v2), multiclass argmax helpers, paid tx inference, and signature-gated view inference for owners and subscribed API keys.

10.4 ModelNFT and Marketplace

ModelNFT: ERC-721 token with on-chain metadata (title, description, icon, packed feature schema). **Marketplace:** optional listing and purchase contract for Model NFTs.

Chapter 11

Frontend Module Map

Forest is intentionally "no build step" for local use: ES modules load directly in the browser.

- `src/create_page.js`: dataset, training orchestration, deployment.
- `src/train_worker.js`: core training implementation.
- `src/local_infer.js`: local model decoding and inference.
- `src/eth.js`: provider + wallet state utilities.
- `src/abis.js`: contract ABIs.

Chapter 12

Suite-Level Architecture and Navigation

This chapter expands the original whitepaper with a suite-oriented perspective: rather than viewing GenesisL1–Forestas a set of contracts and a model format only, we describe the full end-user and developer workflow as implemented by the reference web application contained in `genesis_forest_suite_debug_plotly_blue_v5.zip`. The goal is to document (i) what each UI tab does, (ii) which on-chain and off-chain components it touches, and (iii) how training, heuristic search, and feature scoring are realized in a reproducible, deterministic way.

12.1 Page map and responsibilities

The reference suite is a static web application with multiple top-level pages ("tabs" in the navigation bar). Each page is a single HTML document that imports a dedicated JavaScript module under `src/`. The navigation bar is rendered consistently across pages via `src/ui_nav.js`, which also centralizes wallet connection state and exposes a unified "system configuration" (RPC endpoint and contract addresses).

While each page is independent, they share:

- a **system configuration** (RPC URL + contract addresses) stored in browser local storage;
- a **wallet state** (address, chainId) broadcast through a custom DOM event;
- a **debug dock** (collapsible log console) implemented by `src/debug_dock.js`;
- common utilities (formatting, unit conversion, task labeling, feature metadata packing) in `src/common.js`.

12.2 System configuration model

All pages interpret "the chain" through a JSON system object loaded by `loadSystem()` from `localStorage`. The configuration includes:

- `rpc`: RPC endpoint used for *read* calls (`eth_call`);
- `store`: `ModelStore` contract address (chunk writes);

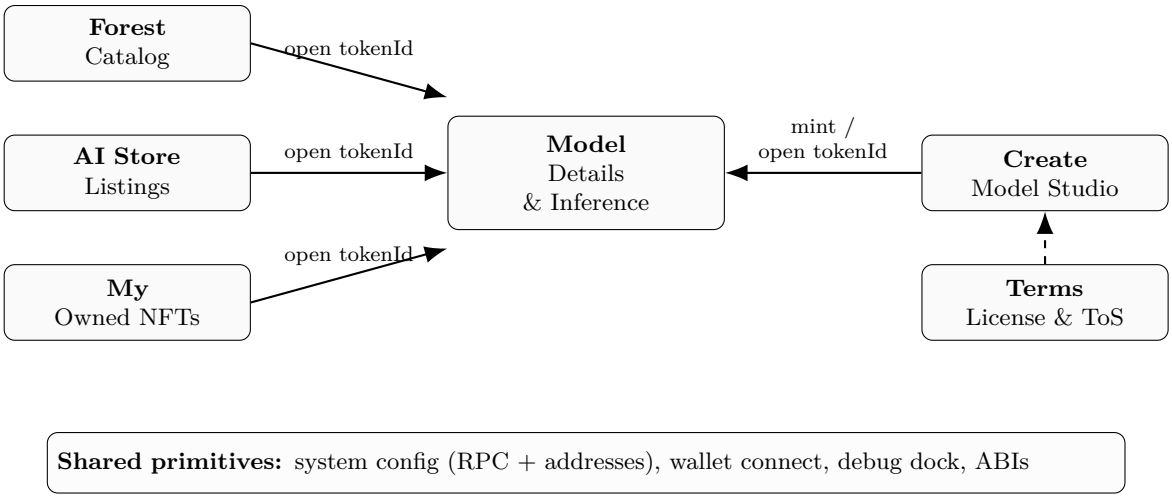

Figure 12.1: Page map of the GenesisL1 Forest reference suite (shipped pages). "Model" is a detail page reachable from catalog/search pages and from Create after minting. The dashed arrow indicates that the active license and ToS version shown in Terms gates Create actions.

- `registry`: `ModelRegistry` address (model registration and metadata);
- `nft`: `ModelNFT` address (ERC-721 ownership and icon/features);
- `runtime`: `ForestRuntime` address (on-chain inference engine);
- `market`: `ModelMarketplace` address (listing/buying).

A key design choice is that the suite prefers the configured `rpc` for calls (reliability and consistent gas limits for `eth_call`), but uses the browser wallet provider for signed transactions. This separation reduces failures due to wallet providers that restrict large `eth_call` payloads.

12.3 Wallet event bus and cross-page consistency

The wallet module (`src/eth.js`) exports a small reactive state. When the wallet connects or the chain changes, the suite dispatches the DOM event `genesis_wallet_changed`. Pages subscribe to this event to refresh UI and enable/disable actions.

> **Operational implication.** Pages never assume the user stays connected: every "write" action re-checks the current wallet state (address and chainId) and fails early with actionable messages.

12.4 Debug dock

The debug dock is a suite-wide observability component. It is intentionally simple—a text buffer plus controls to collapse, clear, and copy logs. The dock also shows a coarse "state" label (e.g., `idle`, `training`) and a connection summary (wallet address + chainId). Because it is a pure client-side console, it is safe to run in untrusted contexts and can be embedded in static hosting.

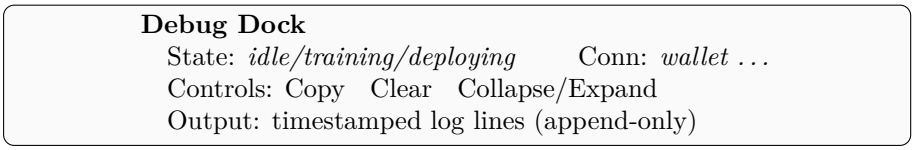

Figure 12.2: Conceptual structure of the debug dock. The implementation is intentionally browser-compatibility focused: it uses `navigator.clipboard` when available and falls back to `document.execCommand("copy")` otherwise.

Chapter 13

Forest Tab: The Model Catalog

The **Forest** tab (`forest.html` + `src/forest_page.js`) is the discovery layer for all registered models. It provides a lightweight catalog UI with paging, keyword search, and optional filters by NFT owner and model creator.

13.1 User-facing behavior

At a high level, the page renders a grid of model cards. Each card shows:

- model title and short description;
- task type (regression / binary / multiclass / multilabel);
- structural parameters (#features, #trees, depth, scale);
- pricing status (disabled/free/tips/paid) and fee;
- an icon (128×128 PNG) stored on-chain via `ModelNFT`.

Selecting a card navigates to the **Model** page with a `tokenId` query parameter.

13.2 Data plane: what contracts are queried

Forest is read-only. It queries:

1. **Registry** (`ModelRegistry`) for the authoritative model summary (inference enable flag, pricing mode, fee, structural stats).
2. **NFT** (`ModelNFT`) for icon bytes and user-facing metadata.
3. **Market** (`ModelMarketplace`) optionally, to display current listing prices.

Because the icon is stored as bytes, the suite converts it to a PNG blob and uses an object URL. This avoids needing external image hosting and keeps the catalog purely on-chain.

13.3 Search and indexing

The primary search mechanism is a word-hash index in the registry (see Figure 6.1). The Forest UI tokenizes the search box into lowercase words of length ≥ 2 and computes `keccak256`hashes. The query is then executed via `registry.searchTitleWords(wordHashes, page, pageSize)`.

> **Design trade-off.** Word-hash search is robust to on-chain storage constraints: it avoids storing large inverted indices, and it allows prefix-free matching by exact word equality. However, it is not a semantic search: synonyms and typos are not matched unless the title explicitly contains the queried word.

Forest additionally supports filtering by **owner address** and **creator address**. Owner filtering uses the enumerable ERC-721 interface; creator filtering requires scanning because creator is stored in the registry model struct and is not enumerable by creator in constant time.

13.4 Paging and performance considerations

Catalog pages are rendered in fixed-size batches (default 25). For large filters (e.g., creator scans), the UI enforces a scan cap to avoid locking the browser. This is a deliberate stance: the reference suite is a static web app and must remain responsive even on low-end devices.

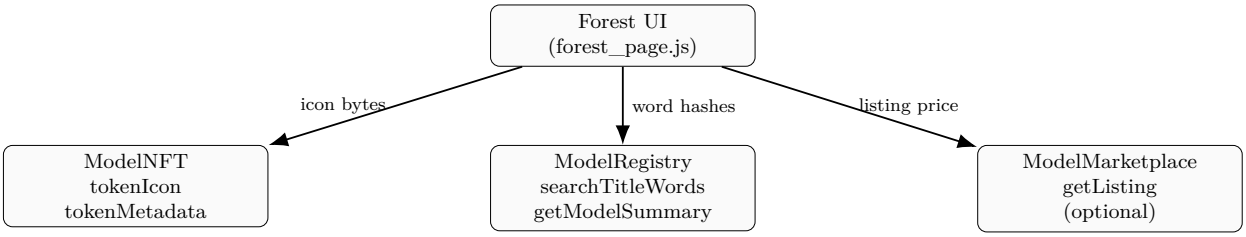

Figure 13.1: Read-path for rendering a page of Forest catalog cards. The UI merges registry summaries with NFT metadata and (optionally) marketplace listing state.

Chapter 14

AI Store Tab: Marketplace Surface

The **AI Store** tab (`aistore.html` + `src/market_page.js`) is a specialized view over the same underlying registry: it shows only models currently listed for sale in the marketplace contract.

14.1 Listing semantics

A listing is not a registry attribute; it lives in the marketplace contract. This separation is intentional:

- registry remains the *canonical* model catalog and runtime configuration;
- marketplace contains optional liquidity features (list, delist, buy) and can be replaced/upgraded independently;
- models can exist without ever being listed.

14.2 Query strategies

The AI Store view supports keyword search similar to Forest. When no query is provided, it simply pages through the marketplace's listing list. When a query is provided, it first asks the registry for matching token IDs via word-hash search, and then filters that set by checking listing status in the marketplace.

> **Engineering note.** Search in a listing-only view is expensive if implemented as "scan all listings and filter by title". The suite instead uses registry search as the first-stage filter.

14.3 Buy flow

Buying is a single transaction `market.buy(tokenId)` with `msg.value` equal to the listing price. Ownership transfers to the buyer as an ERC-721 transfer. After the transaction is mined, the UI refreshes listing status and owner address.

Chapter 15

Model Tab: Inference, Pricing, and Access Control

The **Model** page (`model.html` + `src/model_page.js`) is the operational core of the suite. It supports:

- loading model metadata and runtime parameters from chain;
- performing inference via *view* calls or paid *transactions* depending on pricing mode;
- managing owner settings (pricing, recipient, inference enable);
- marketplace actions (list/unlist/buy);
- managing API access keys and subscription plans for paid models.

15.1 Model identity: tokenId vs modelId

The suite uses two identifiers:

- **tokenId**: ERC-721 token identifier (human-facing, used in URLs).
- **modelId**: `keccak256`hash of the serialized model bytes (content-addressed, used by the runtime and registry for storage binding).

The Model page loads by `tokenId` and then asks the registry for the corresponding `modelId` and storage pointers (table pointer, chunking parameters).

15.2 Feature packing and quantization

On-chain inference consumes a packed feature vector `packedFeaturesQ`.

- Each feature is quantized as $q_i = \text{round}(x_i \cdot \text{scaleQ})$.
- Each q_i is clamped to signed 32-bit range and encoded little-endian.
- The packed byte array is $4 \cdot n_{\text{features}}$ bytes long.

This is the same quantization used in the model format described in Section 4.5. The suite maintains consistency by always reading `scaleQ` from the registry summary before packing.

15.3 Inference modes and pricing

The suite implements three pricing modes (Figure 5.1) and selects an execution path accordingly.

15.3.1 Mode 0: free view inference

When pricing mode is `0` (Free), any user can call the runtime view function (e.g., `predictView`) to obtain a result without a transaction.

15.3.2 Mode 1: tips

When pricing mode is `1` (Tips), view inference remains enabled. Users may optionally send a transaction with `msg.value` as a tip; the runtime emits an event containing the inference output (because transactions cannot return values to the UI).

15.3.3 Mode 2: paid required

When pricing mode is `2` (Paid required), public view inference would allow fee bypass (because `eth_call` cannot enforce payment). The suite therefore distinguishes three paid-required paths:

1. **Paid transaction inference** (anyone): `predictTx` with `msg.value` $\geq$ fee.
2. **Owner-signed view inference** (NFT owner): `predictOwnerView` with an EIP-712 signature from the NFT owner address.
3. **Access-key view inference** (subscribers): `predictAccessView` with an EIP-712 signature from an *API key* whose access expiry is stored in the registry.

15.4 API access keys and subscription plans

A distinctive feature of FORESTis that paid-required models can be queried via view calls by parties holding an *API key*. In the suite, an API key is a regular Ethereum keypair (address + private key). The address is written to the registry with an expiry block number.

15.4.1 Owner API key

During minting (Create tab), the deployer specifies an **owner access key address**. The registry sets this key's expiry to the maximum value, granting perpetual access. The private key is never placed on-chain; it must be stored securely by the model owner.

15.4.2 Subscriber access keys

Model owners can publish paid access plans. Users buy a plan by paying on-chain, which extends their access key expiry. The Model page supports:

- generating an API keypair (locally) and storing the private key in the user's possession;
- buying a plan for that key (transaction);
- using the key to sign an EIP-712 `AccessView` message (local signing) and calling `predictAccessView` (view call).

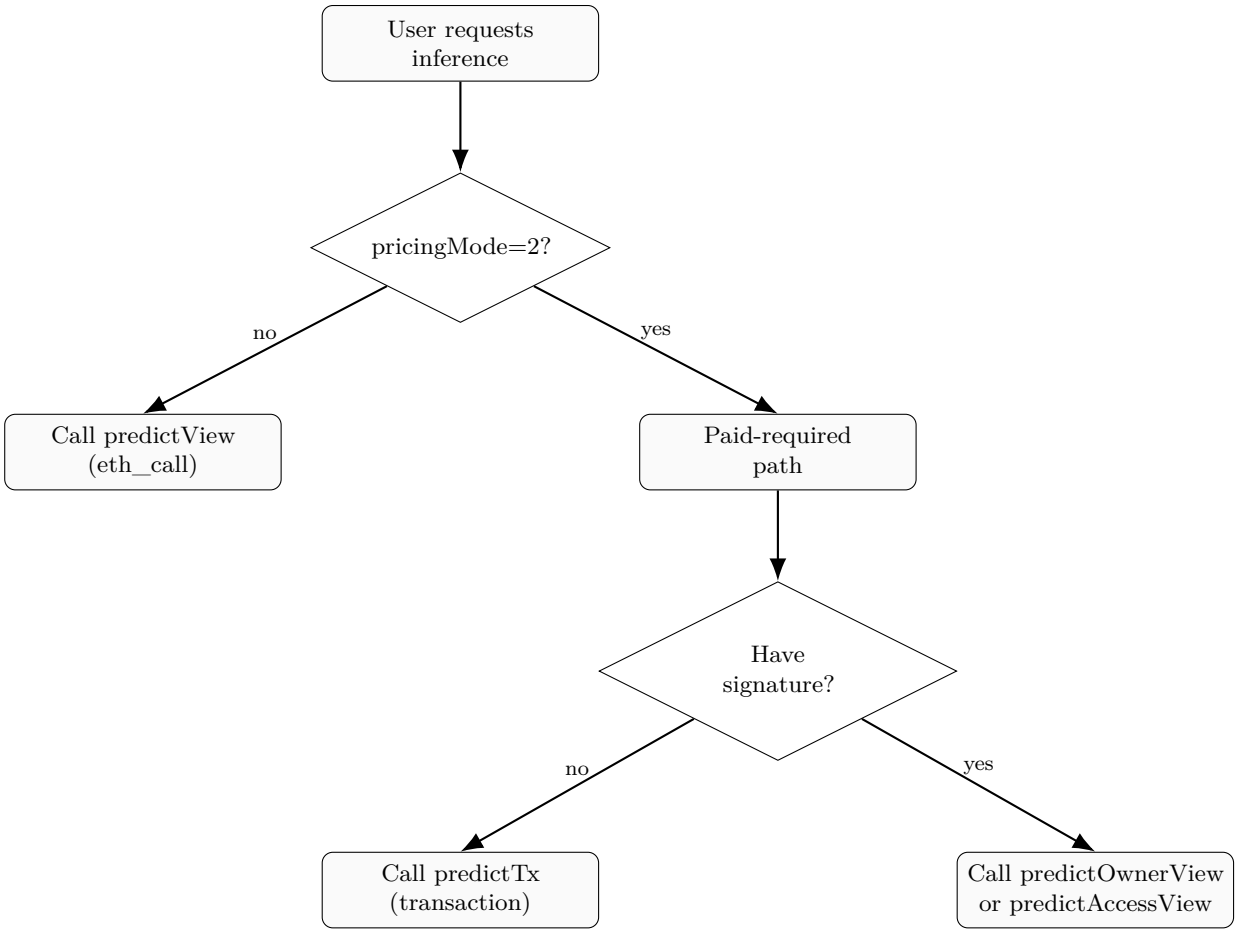

Figure 15.1: Decision tree for inference execution in the Model tab. Paid-required models permit view inference only via signatures.

15.5 Owner settings and lifecycle actions

If the connected wallet is the current NFT owner, the Model page enables additional controls:

- toggle inference enabled/disabled;
- change pricing mode, fee, and recipient;
- set (rotate) the owner API access key address;
- list or unlist the NFT in the marketplace;
- burn and delete the model (permanent removal).

The last action, burn+delete, is intentionally irreversible: it removes the model record from the registry and burns the NFT. This is relevant for moderation and for owners who wish to deprecate models.

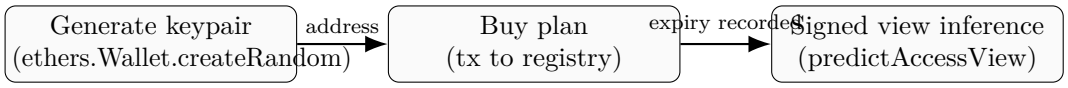

Figure 15.2: Access-key workflow for paid-required models: a keypair is generated locally, recorded on-chain with an expiry, and then used to authorize view inference through EIP-712 signatures.

Chapter 16

My Tab: Portfolio View

The **My** page (`my.html` + `src/my_page.js`) lists models owned by the connected wallet. It is a convenience view that builds on ERC-721 enumerability:

1. query `balanceOf(owner)`;
2. iterate `tokenOfOwnerByIndex(owner,i)`;
3. for each tokenId, query registry summary and NFT metadata;
4. render a card grid identical in style to Forest.

The page is intentionally minimal: it performs no writes and does not expose listing or inference actions directly (those are done in the Model page).

Chapter 17

Create Tab: Model Studio

The **Create** tab (`create.html` + `src/create_page.js`) is a browser-native training environment for FORESTmodels. It consists of four sub-tabs:

1. **Dataset**: upload and inspect data; define task, label(s), features, and imbalance strategy.
2. **Training**: set hyperparameters; run training; optionally run heuristic search.
3. **Local preview**: inspect metrics and curves; compute feature scores; run local predictions.
4. **Mint**: choose metadata and pricing; deploy model bytes on-chain (chunking + register).

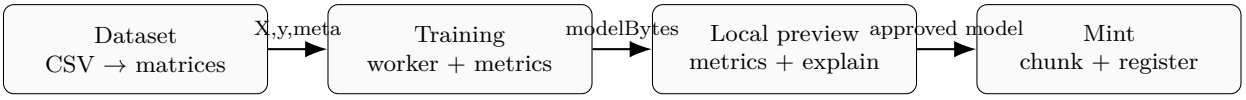

Figure 17.1: Create tab high-level pipeline. Dataset construction is prerequisite to training; training produces serialized model bytes; preview performs interpretability and sanity checks; mint writes bytes to chain and registers the NFT.

17.1 Dataset sub-tab

17.1.1 CSV parsing and type inference

The suite accepts a CSV file as input and parses it in-browser using `src/csv_parse.js`. Parsing features:

- supports quoted fields and escaped quotes;
- treats empty fields as missing;
- limits row count to a safety threshold (to avoid memory explosion in the browser);
- preserves raw strings initially; numeric conversion happens later.

After parsing, the user selects:

- a task type (regression / binary / multiclass / multilabel);
- label column(s) according to the task;
- a set of feature columns.

17.1.2 Label encoding per task

Dataset encoding differs by task:

Regression The label column must be numeric. Rows with non-finite label or feature values are dropped. The label vector is a float array.

Binary classification The label column is categorical. The user chooses which label value corresponds to class 0 (negative) and class 1 (positive). Rows with other label values are dropped.

Multiclass classification The label column is categorical. The user selects $K \geq 2$ allowed label values and orders them; the order defines the integer class mapping $0..K-1$. Other labels are dropped.

Multilabel classification The user selects $L \geq 2$ label columns, each expected to be parseable as $\{0,1\}$ (accepting common textual forms like `true/false`, `yes/no`). Rows where any selected label is missing/invalid are dropped.

17.1.3 Feature selection and exclusion rules

The Create UI lists all CSV columns with checkboxes. The selected label column (or selected label columns for multilabel) are automatically excluded from the feature set to prevent leakage.

17.1.4 Split preview and determinism

The suite uses deterministic shuffling based on a user-provided seed. This ensures that training results are reproducible across browsers, provided that floating-point behavior is stable.

For binary and multiclass tasks, the suite optionally performs **stratified splitting** so that class proportions are similar across train/validation/test partitions. Stratification is disabled for multilabel tasks because multi-dimensional label stratification is nontrivial and can be misleading.

17.1.5 Class imbalance handling

Class imbalance handling is a *training-only* mechanism: it influences gradients/hessians and therefore learned splits and leaf values, but does not change the on-chain model format.

Three modes are provided:

- **None**: weights are all 1.
- **Auto**: weights are computed as the inverse frequency (binary: $w_c = N/(2\,n_c)$; multiclass: $w_c = N/(K\,n_c)$; multilabel: per-label positive weights based on `neg/pos` ratio).
- **Manual**: the user supplies weights directly.

Two additional controls refine behavior:

- **cap**: an upper bound on weights to avoid exploding updates on extremely rare classes.
- **normalize**: rescales weights so the average weight (approximate) is 1.

17.1.6 Data Galaxy: 3D distribution and PCA

The Dataset tab includes a 3D scatter visualization to help detect gross issues (separability, outliers, label leakage). Two modes are supported:

1. **Raw feature triplet**: plot any three selected features.
2. **PCA-3 projection**: compute the first three principal components on a sampled subset of rows and plot in that latent space.

PCA is computed in-browser without external numeric libraries. The implementation:

- samples up to a user-defined number of valid rows;
- standardizes features to zero mean and unit variance;
- uses repeated power iteration with Gram–Schmidt orthonormalization to estimate the top three eigenvectors of the covariance matrix.

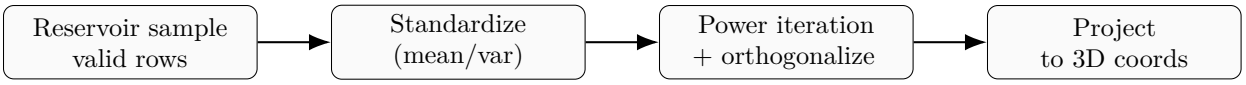

Figure 17.2: PCA-3 pipeline used by the Create tab's 3D visualization. The implementation avoids heavyweight numerical dependencies and is designed to remain responsive by yielding to the browser event loop during long operations.

17.2 Training sub-tab

17.2.1 Exposed hyperparameters

The Training sub-tab exposes a pragmatic set of hyperparameters that directly influence:

- model quality (bias/variance trade-offs);
- model size (bytes stored on-chain);
- inference cost (time/gas proportional to $n_{\text{trees}} \cdot \text{depth}$).

A detailed reference table is provided in Chapter 18.

17.2.2 Model size estimate and on-chain constraints

Before training, the suite estimates the serialized model size using the same formula as the on-chain format specification (Appendix, Chapters 12+). It enforces two constraints:

1. **Absolute byte limit**: `SIZE_LIMIT = 15,000,000`. This is a suite-level guardrail to prevent deploying extremely large models.
2. **Tree-count bound for v2**: registry stores `nTrees` as `uint16`. For multiclass/multilabel models (format v2), total trees equals `treesPerClass` × `nClasses` and must fit in 65535.

If the user selects parameters that violate constraints, the suite automatically clamps them (reducing trees in steps of 25, then reducing depth) and keeps UI controls in sync.

17.2.3 Training worker: isolation and responsiveness

Training is executed in a Web Worker (`src/train_worker.js`) to keep the UI responsive. The main thread sends:

- numeric feature matrix X as a flat Float64Array;
- label vector(s) y (Float64Array for regression; Uint8Array/Int32Array encodings for classification);
- task metadata (#features, #classes/labels, feature min/range);
- hyperparameters and imbalance settings.

The worker posts back:

- progress updates every tree (training/val metrics); and
- a final `done` message containing `modelBytes` and a `meta` summary.

17.2.4 Learning-rate schedules

Besides a constant learning rate, two schedules are implemented:

Plateau schedule If validation loss does not improve for a specified number of trees, multiply the learning rate by $(1 - \text{dropPct}/100)$, but do not go below `minLR`.

Piecewise schedule The user specifies explicit ranges of tree indices and learning rates (e.g., "1-100 0.1"). Ranges are 1-indexed and inclusive.

These schedules are applied inside the worker as a function of tree index and observed validation metric.

17.2.5 Early stopping and final refit

When early stopping is enabled, training monitors validation loss and records the best iteration. The suite optionally performs a **final refit** stage:

1. train on train/val split with early stopping to select `bestIter`;
2. retrain from scratch on train+val for a fixed tree budget `bestIter`, with early stopping disabled.

This mirrors common practice in boosting workflows: validation is used for model selection, then the selected configuration is fit on the largest available non-test dataset.

17.2.6 Heuristic hyperparameter search

The Training tab can run a lightweight heuristic search (random mutation around a pivot configuration) to find improved hyperparameters without leaving the browser. The implementation and distributions are detailed in Chapter 19.

17.3 Local preview sub-tab

The Local preview tab is a safety and interpretability gate before minting. It:

- decodes the trained model bytes locally using the same format definitions as the runtime;
- displays best train/val/test metrics reported by the worker;
- plots the per-tree metric curves;
- computes feature scores (split usage + permutation importance);
- offers a per-row prediction playground to compare predicted vs actual labels.

17.4 Mint sub-tab

Minting connects the browser-trained model to the on-chain system. It consists of:

1. selecting metadata (name, description, 128×128 PNG icon);
2. selecting pricing mode and fee recipient;
3. generating (or supplying) an owner API access key;
4. agreeing to the currently-active Terms and license;
5. writing model bytes to the `ModelStore` as chunks;
6. writing the pointer-table contract;
7. registering the model in the registry and minting the NFT.

The suite estimates required deploy value via `requiredDeployFeeWei(totalBytes)` and separately reminds the user that gas is paid in addition to that deploy value.

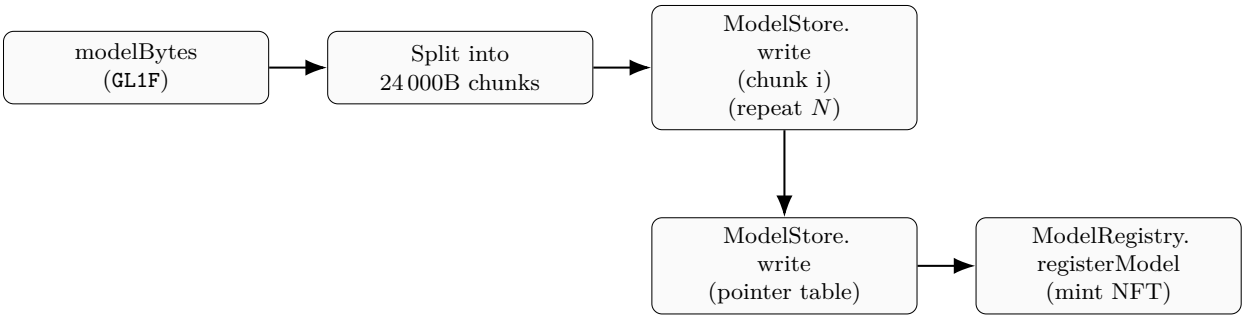

Figure 17.3: Minting pipeline: model bytes are chunked and written via `ModelStore`, then the pointer-table pointer and metadata are registered in `ModelRegistry`.

Chapter 18

Hyperparameter and Specification Reference

This chapter consolidates the training hyperparameters exposed by the Create tab, links them to the worker implementation, and highlights interactions with on-chain constraints.

18.1 Primary hyperparameters (Create → Training)

Name	UI control	Range	Default	Meaning / implementation notes
Number of trees	`treesNum`	10–5000 (v1)	250	Boosting rounds for v1 (regression/binary). For v2 (multiclass/multilabel), interpreted as *trees per class/label*; total trees = `trees` $\times K$. Clamped for size and for `uint16` total-tree bound.
Depth	`depthNum`	1–12	4	Maximum depth of each tree. The model stores a full binary tree of depth `depth`; missing splits are represented by forced nodes (threshold = INT32_MAX) so that inference is constant-time per tree.
Learning rate	`lrNum`	0.001–1	0.05	Step size multiplier applied to leaf updates. In worker, regression uses $\Delta = \eta\,\overline{r}$; classification uses Newton step $\Delta = -\eta\,G/(H+\lambda)$.
Min leaf samples	`minLeafNum`	1–1000	10	Minimum number of training samples required in each child of a split; if violated, the node becomes forced (no further splits).

Name	UI control	Range	Default	Meaning / implementation notes
Bins	`binsNum`	8–512	32	Histogram bin count used for approximate split search. Higher bins increase training time and (slightly) improve split resolution; does not affect model size because thresholds are stored as int32 regardless of bins.
Binning mode	`binningMode`	{linear, quantile}	linear	Linear mode uses uniform bins between feature min and max. Quantile mode precomputes per-feature thresholds from a sample and bins by empirical quantiles.
Seed	`seedNum`	1–2,147,483,647	42	Seed for deterministic shuffling, feature sampling, and heuristic search PRNG. The worker uses xorshift32 to generate reproducible pseudo-randomness.
Train split	`trainSplitNum`	50%–90%	70%	Fraction of usable rows assigned to training. Remaining rows are split between validation and test.
Validation split	`valSplitNum`	5%–40%	20%	Fraction of usable rows assigned to validation. Test split is implied: $1 - \texttt{train} - \texttt{val}$.
Early stopping	`earlyStopOn`	on/off	on	If enabled, stop when validation metric has not improved for `patience` trees. Best iteration is recorded.
Patience	`patienceNum`	1–500	25	Number of non-improving trees tolerated before early stopping triggers.
LR schedule	`lrSchedMode`	none/plateau/piecewise	none	Optional schedule that modifies learning rate over time. Plateau schedule reacts to validation metric stagnation; piecewise schedule uses explicit tree-index ranges.
Refit train+val	`refitOn`	on/off	off	If enabled *and* early stopping is on, the suite performs a second training pass on train+val using the selected best tree budget.
Heuristic search	`heuristicSearchOn`	on/off	off	Enables multiple training rounds with mutated hyperparameters, tracking best validation metric.
Search rounds	`heuristicSearchRounds`	1–1000	10	Maximum number of heuristic-search candidate rounds.

18.2 Imbalance-handling parameters

Imbalance parameters appear on the Dataset tab when a classification task is selected.

Name	UI control	Default	Meaning / implementation notes
Mode	`imbMode`	none	none/auto/manual. Auto computes inverse-frequency weights; manual exposes per-class (or per-label) inputs.
Cap	`imbCap`	20	Upper bound on weights. Prevents extreme gradients when classes are extremely rare.
Normalize	`imbNormalize`	on	Rescales weights so the weighted average is approximately 1. Helps keep learning-rate interpretation stable.
Stratify split	`imbStratify`	on	Only for binary/multiclass. Uses label-stratified splits to keep class proportions stable across train/-val/test.
Manual weights	dynamic	1	Binary: `w0,w1`. Multiclass: per-class. Multilabel: per-label positive weights (pos_weight style).

18.3 Fixed internal parameters

Some important training choices are fixed in code for simplicity and determinism:

- **Column sampling**: at each split, the worker samples $\lceil \sqrt{n_{\text{features}}} \rceil$ candidate features.
- **Regularization**: classification uses $\lambda = 1$ in the Newton leaf formula and split gain.
- **Model topology**: each tree is stored as a complete binary tree of depth `depth`; forced nodes encode early termination.
- **Quantile threshold sampling**: quantile binning uses a fixed sample budget (default 50k rows) from the (shuffled) training set.

Chapter 19

Heuristic Search: Auto-Tuning in the Browser

The heuristic search facility in the Create tab is designed for "good enough" tuning without requiring external services. It behaves as a bounded random search with memory:

1. start from the user's base hyperparameters;
2. train a candidate model and evaluate its validation metric;
3. maintain the best candidate so far;
4. generate the next candidate by mutating either the current best (75% probability) or the original base configuration (25% probability);
5. repeat for a fixed number of rounds or until stopped.

19.1 Candidate generation distribution

Candidate generation uses a deterministic xorshift32 PRNG seeded from the user-provided seed. Parameters are perturbed multiplicatively (trees, learning rate, minLeaf) and additively (depth), then clamped to UI bounds and to on-chain size limits.

Listing 19.1: Excerpt of heuristic candidate generation (Create tab).

```
// pivot = bestParams with prob 0.75, else baseParams
const treesFactor = 2 ** ((rand01()-0.5) * 1.4);      // ~[0.62..1.62]
let trees = roundTo25(pivot.trees * treesFactor);

aStep = round((rand01()-0.5) * 4);                    // [-2..2]
let depth = clamp(pivot.depth + aStep);

const lrFactor = 10 ** ((rand01()-0.5) * 0.8);        // ~[0.40..2.51]
let lr = clamp(pivot.lr * lrFactor);

const mlFactor = 2 ** ((rand01()-0.5) * 2.0);         // ~[0.5..2]
let minLeaf = clamp(round(pivot.minLeaf * mlFactor));

// plateau schedule params are optionally perturbed
// ...

```

```
// final clamp: enforce size limit and uint16 tree bounds
const cl = clampForSize(trees, depth, task, nClasses);
return { ...pivot, trees: cl.trees, depth: cl.depth, lr, minLeaf, ... };
```

19.2 Objective and selection

The search compares candidates using `bestValMetric` reported by the worker:

- regression: validation MSE (lower is better);
- classification: validation log loss (lower is better).

Accuracy is reported for diagnostics but is not the primary objective. This is intentional: log loss is smoother and more sensitive to probability calibration than accuracy.

19.3 Search history table and reproducibility

Each round appends a row to a "Search history" table containing:

- round number;
- selected hyperparameters;
- best validation metric and whether it improved the incumbent.

Because randomness is seeded deterministically and the worker's training loop is deterministic given the shuffled splits, the entire search is reproducible if run with the same dataset, same seed, and same browser floating-point behavior.

19.4 Stopping and failure handling

A "Stop" control terminates the active worker and aborts the search loop. The suite is careful to unwind pending Promises so that UI state returns to a consistent idle mode.

Chapter 20

Feature Scoring and Interpretability

The Create tab implements feature scoring to help users understand and validate a model before minting. The goal is not to provide a full interpretability suite, but to supply two complementary signals:

1. **Split usage**: how often a feature is used in learned (non-forced) internal nodes.
2. **Permutation importance**: how much the test metric degrades when a feature is randomly permuted.

20.1 Split usage counts

For a fixed-depth tree representation, many internal nodes may be "forced" (no learned split). These are encoded by `thr = INT32_MAX`. The feature scoring logic ignores forced nodes when counting split usage.

Split usage is fast to compute: for each tree and each internal node, increment `count[feat]` if the node is not forced.

20.2 Permutation importance on a budget

Permutation importance is computed on a sampled subset of the test split to keep the UI responsive. The procedure is:

1. select a sample of test rows (cap at 1024 by default, further reduced adaptively);
2. compute baseline predictions and baseline metric;
3. for each feature j:
   (a) copy the feature matrix subset;
   (b) permute column j (shuffle indices);
   (c) recompute predictions and metric;
   (d) record Δloss (increase in loss) and Δacc (drop in accuracy, if applicable).

The suite adaptively chooses the test-sample size based on an estimated compute budget proportional to $(n_{\text{features}} \cdot n_{\text{trees}} \cdot \text{depth})$.

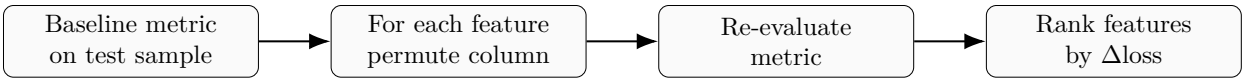

Figure 20.1: Permutation importance pipeline as implemented in the Create tab. The output is a feature ranking table shown in Local preview.

20.3 Interpreting feature scores

- A high split count suggests that the feature frequently yields useful partitions, but it does not directly quantify contribution magnitude.
- A high permutation Δloss indicates that the feature contains information that the model relies on (possibly redundantly with other features).
- Negative Δloss (loss improves when permuted) can occur due to sampling noise or because the feature introduces spurious correlations.

20.4 Limitations

Permutation importance is a post-hoc diagnostic. It is sensitive to correlated features: if two features are strongly correlated, permuting one may have limited effect because the other retains similar information. The suite therefore presents split usage alongside permutation scores to provide a second perspective.

Chapter 21

Terms, License, and Legal State

This chapter documents the suite page that surfaces the protocol's active license and Terms of Service (ToS), and explains how these values are used by the Create flow. The shipped suite is intentionally *read-only* with respect to legal state: it can display the current parameters, but it does not include any privileged deployment, moderation, or governance console.

21.1 Terms tab

The Terms page (`terms.html` + `src/terms_page.js`) is read-only and displays:

- the currently active license (ID, name, URL);
- deploy and listing fees;
- the current ToS version and full ToS text stored on-chain.

This is important for Create: minting requires the user to agree to the active license and ToS version, and the mint transaction records those IDs.

21.2 Debug dock revisited

All pages include the debug dock described in Section 12. For Create, the dock is particularly valuable because the end-to-end flow involves multiple sequential transactions (chunk writes, pointer-table deployment, registry registration, and optional listing) interleaved with long-running worker computations (training, evaluation, feature scoring).

Chapter 22

Engineering Notes and Edge Cases

This chapter collects practical engineering considerations that arise from implementing an on-chain GBDT suite in the browser.

22.1 Determinism and reproducibility

The suite is designed to be deterministic given (dataset, selected columns, seed, hyperparameters):

- dataset splits are generated via seeded shuffling;
- feature subsampling uses a deterministic xorshift PRNG;
- quantile thresholds are computed from a deterministic prefix sample.

However, full cross-platform bitwise determinism is not guaranteed: JavaScript floating-point behavior is standardized, but subtle differences in optimization, math library implementations, and worker scheduling can produce small numeric differences. The model format's quantization helps stabilize results by rounding thresholds and leaf values to int32.

22.2 Numeric stability and scale selection

The suite chooses `scaleQ` to maximize precision without overflow. It sets

$$\text{scaleQ} = \min\left(10^6,\ \left\lfloor \frac{2{,}147{,}480{,}000}{\max|x|} \right\rfloor,\ \left\lfloor \frac{2{,}147{,}480{,}000}{\max|y|} \right\rfloor\right)$$

for regression, and omits the y bound for classification.

22.3 Performance characteristics

Training time scales approximately with:

$$\mathcal{O}(n_{\text{trees}} \cdot 2^{\text{depth}} \cdot n_{\text{rows}})$$

modulated by histogram binning and feature subsampling. Inference time (both local and on-chain) scales with $\mathcal{O}(n_{\text{trees}} \cdot \text{depth})$.

22.4 Security notes: signatures and deadlines

Signature-based view inference (OwnerView and AccessView) uses:

- EIP-712 domain separation (chainId + verifying contract);
- a short deadline (unix timestamp) to reduce replay window;
- a hash of packed features (`keccak256`of packed bytes) rather than raw feature bytes in typed data, keeping the signature payload small.

These measures ensure that a signature cannot be replayed on other chains or other runtime deployments, and limit the time window in which it is valid.

Part III

Appendices

Appendix A

Binary Specification (Normative)

A.1 Packed feature vectors

For a model with n features, the runtime expects a byte array of length $4n$. Feature i is encoded as little-endian int32 at offset $4i$ and represents $\lfloor x_i \cdot \texttt{scaleQ} \rceil$.

A.2 Tree block (v1 and v2)

Let d be depth, $P = 2^d$, $I = P - 1$. Then for each tree:

- internal nodes: I records, each 8 bytes: `u16 feature` + `i32 thresholdQ` + `u16 reserved`.
- leaves: P records, each 4 bytes: `i32 leafQ`.

Appendix B

Selected Source Excerpts (Informative)

> **Note**
>
> This appendix includes *selected* excerpts from the bundled source tree under `genesis/`. The full source remains in the project folder, but is not typeset here in full to keep the document to a manageable length.

B.1 Solidity: `ModelStore.sol` (full; short)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

/// @notice Stores arbitrary byte chunks on-chain by deploying a tiny pointer contract
///         whose runtime bytecode is: MAGIC(4 bytes) || DATA.
///
/// Design goals:
/// - On-chain writes avoid SSTORE-heavy storage (cheaper for medium blobs).
/// - Reads are easy off-chain via eth_getCode / extcodecopy.
/// - Compatible with evmVersion=istanbul (no post-Istanbul opcodes required).
contract ModelStore {
    // 0x47 0x4c 0x31 0x43 = "GL1C" (GenesisL1 Chunk)
    uint32 public constant MAGIC = 0x474c3143;

    event ChunkWritten(address indexed pointer, uint256 size);

    /// @notice Deploy a new pointer contract containing (MAGIC || data) as its runtime bytecode.
    /// @dev Runtime code size must respect EIP-170 (24,576 bytes). We reserve 4 bytes for MAGIC.
    function write(bytes calldata data) external returns (address pointer) {
        uint256 dlen = data.length;
        require(dlen <= 24_572, "CHUNK_TOO_LARGE");

        uint256 rlen = dlen + 4; // runtime length
        bytes memory init = new bytes(14 + rlen);

        assembly ("memory-safe") {
            let p := add(init, 32)

            // Minimal init-code:
```

```
            // PUSH2 rlen
            // PUSH1 0x0e
            // PUSH1 0x00
            // CODECOPY
            // PUSH2 rlen
            // PUSH1 0x00
            // RETURN
            mstore8(p, 0x61)
            mstore8(add(p, 1), shr(8, rlen))
            mstore8(add(p, 2), and(rlen, 0xff))
            mstore8(add(p, 3), 0x60)
            mstore8(add(p, 4), 0x0e)
            mstore8(add(p, 5), 0x60)
            mstore8(add(p, 6), 0x00)
            mstore8(add(p, 7), 0x39)
            mstore8(add(p, 8), 0x61)
            mstore8(add(p, 9), shr(8, rlen))
            mstore8(add(p, 10), and(rlen, 0xff))
            mstore8(add(p, 11), 0x60)
            mstore8(add(p, 12), 0x00)
            mstore8(add(p, 13), 0xf3)

            // Runtime start
            let r := add(p, 14)

            // MAGIC (4 bytes)
            mstore(r, shl(224, 0x474c3143))

            // DATA
            calldatacopy(add(r, 4), data.offset, dlen)

            // CREATE
            pointer := create(0, p, mload(init))
        }

        require(pointer != address(0), "CREATE_FAIL");
        emit ChunkWritten(pointer, dlen);
    }
}
```

B.2 Solidity: `ModelRegistry.sol` (registration + access keys)

B.2.1 Title-word AND search (for the Store UI)

```
        uint16 nFeatures,
        uint16 nTrees,
        uint16 depth,
        int32 baseQ,
        uint32 scaleQ,
        bool inferenceEnabled,
        uint8 pricingMode,
        uint256 feeWei,
        address feeRecipient
    ) {
        Model storage m = models[modelId];
        require(m.exists && m.active, "NF");
```

```
        return (m.tablePtr, m.chunkSize, m.numChunks, m.totalBytes, m.nFeatures,
            m.nTrees, m.depth, m.baseQ, m.scaleQ, m.inferenceEnabled,
            m.pricingMode, m.feeWei, m.feeRecipient);
    }

    // AND search on words (exact hash match), paginated over the first word list.
    function searchTitleWords(bytes32[] calldata words, uint256 cursor, uint256
        limit) external view returns (uint256[] memory tokenIds, uint256
        nextCursor) {
        if (words.length == 0) return (new uint256[](0), 0);

        uint256[] storage baseList = _wordTokens[words[0]];
        uint256 n = baseList.length;
        if (cursor >= n) return (new uint256[](0), 0);

        uint256[] memory tmp = new uint256[](limit);
        uint256 found = 0;
        uint256 i = cursor;

        for (; i < n && found < limit; i++) {
            uint256 tid = baseList[i];
            bytes32 mid = modelIdByTokenId[tid];
            if (mid == bytes32(0)) continue;
            Model storage m = models[mid];
            if (!m.exists || !m.active) continue;

            bool ok = true;
            for (uint256 w = 1; w < words.length; w++) {
                if (!_wordHasToken[words[w]][tid]) { ok = false; break; }
            }
            if (!ok) continue;

            tmp[found++] = tid;
        }

        tokenIds = new uint256[](found);
        for (uint256 k = 0; k < found; k++) tokenIds[k] = tmp[k];
        nextCursor = (i >= n) ? 0 : i;
    }
```

B.2.2 `registerModel`: mint NFT, bind bytes, enforce Terms + License

```
    function registerModel(
        bytes32 modelId,
        address tablePtr,
        uint32 chunkSize,
        uint32 numChunks,
        uint32 totalBytes,
        uint16 nFeatures,
        uint16 nTrees,
        uint16 depth,
        int32 baseQ,
        uint32 scaleQ,
        string calldata title_,
        string calldata description_,
        bytes calldata iconPng32,
        string calldata featuresPacked,
```

```
        bytes32[] calldata titleWordHashes,
        uint8 pricingMode,
        uint256 feeWei,
        address recipient,
        uint32 tosVersionAccepted_,
        uint32 licenseIdAccepted_,
        address ownerKey
    ) external payable returns (uint256 tokenId) {
        require(address(modelNFT) != address(0), "NFT_NOT_SET");
        require(modelId != bytes32(0), "MIDO");
        require(!models[modelId].exists, "EXISTS");
        uint256 requiredFee = requiredDeployFeeWei(totalBytes);
        require(msg.value == requiredFee, "DEPLOY_FEE");

        require(tosVersionAccepted_ == tosVersion, "TOS");
        require(licenseIdAccepted_ == activeLicenseId, "LIC");
        require(ownerKey != address(0), "OWNER_KEY");

        require(bytes(title_).length > 0, "TITLE");
        require(bytes(description_).length > 0, "DESC");
        require(iconPng32.length > 0, "ICON");
        require(numChunks > 0, "NO_CHUNKS");
        require(chunkSize > 0, "CHUNKO");

        // fee rules
        if (pricingMode == 0) {
            feeWei = 0;
        } else {
            require(feeWei > 0, "FEE_ZERO");
        }
        if (recipient == address(0)) recipient = msg.sender;

        // mint NFT
        tokenId = modelNFT.mintTo(msg.sender, title_, description_, iconPng32,
            featuresPacked);
        // Grant the model owner a perpetual API access key.
        accessExpiry[modelId][ownerKey] = type(uint64).max;
        emit OwnerAccessKeySet(modelId, ownerKey, type(uint64).max);

        Model storage m = models[modelId];
        m.exists = true;
        m.active = true;
        m.modelId = modelId;

        m.tablePtr = tablePtr;
        m.chunkSize = chunkSize;
        m.numChunks = numChunks;
        m.totalBytes = totalBytes;

        m.nFeatures = nFeatures;
        m.nTrees = nTrees;
        m.depth = depth;
        m.baseQ = baseQ;
        m.scaleQ = scaleQ;

        m.inferenceEnabled = true;
        m.pricingMode = pricingMode;
        m.feeWei = feeWei;
        m.feeRecipient = recipient;
```

```
74
75          m.creator = msg.sender;
76          m.tosVersionAccepted = tosVersionAccepted_;
77          m.licenseIdAccepted = licenseIdAccepted_;
78          m.tokenId = tokenId;
79
80          modelIdByTokenId[tokenId] = modelId;
81          tokenIdByModelId[modelId] = tokenId;
82
83          // title index
84          if (titleWordHashes.length > 0) {
85              bytes32[] storage arr = _tokenWords[tokenId];
86              for (uint256 i = 0; i < titleWordHashes.length; i++) {
87                  bytes32 wh = titleWordHashes[i];
88                  if (wh == bytes32(0)) continue;
89                  if (_wordHasToken[wh][tokenId]) continue;
90                  _wordHasToken[wh][tokenId] = true;
91                  _wordTokens[wh].push(tokenId);
92                  arr.push(wh);
93              }
94          }
95
96          // forward deploy fee to owner
97          if (requiredFee > 0) {
98              (bool ok,) = owner.call{value: requiredFee}("");
99              require(ok, "FEE_SEND");
100         }
101
102         emit ModelRegistered(tokenId, modelId, msg.sender);
103     }
```

B.2.3 Subscription access keys (paid-required models)

```
1     // ===== API Access Key Plans =====
2
3     function createAccessPlan(bytes32 modelId, uint32 durationBlocks, uint256
          priceWei, bool active) external returns (uint8 planId) {
4         _requireTokenOwnerByModelId(modelId);
5         require(models[modelId].pricingMode == 2, "MODE");
6         require(durationBlocks > 0, "DUR0");
7         planId = accessPlanCount[modelId] + 1;
8         require(planId != 0, "PLAN_OVERFLOW"); // uint8 overflow
9         accessPlanCount[modelId] = planId;
10        _accessPlans[modelId][planId] = AccessPlan({durationBlocks:
              durationBlocks, priceWei: priceWei, active: active});
11        emit AccessPlanSet(modelId, planId, durationBlocks, priceWei, active);
12    }
13
14    function setAccessPlan(bytes32 modelId, uint8 planId, uint32 durationBlocks,
          uint256 priceWei, bool active) external {
15        _requireTokenOwnerByModelId(modelId);
16        require(models[modelId].pricingMode == 2, "MODE");
17        require(planId > 0 && planId <= accessPlanCount[modelId], "PLAN_ID");
18        require(durationBlocks > 0, "DUR0");
19        _accessPlans[modelId][planId] = AccessPlan({durationBlocks:
              durationBlocks, priceWei: priceWei, active: active});
20        emit AccessPlanSet(modelId, planId, durationBlocks, priceWei, active);
```

```
    }

    function getAccessPlan(bytes32 modelId, uint8 planId) external view returns
        (uint32 durationBlocks, uint256 priceWei, bool active) {
        AccessPlan memory p = _accessPlans[modelId][planId];
        return (p.durationBlocks, p.priceWei, p.active);
    }

    function buyAccess(bytes32 modelId, uint8 planId, address key) external
        payable returns (uint64 newExpiry) {
        require(key != address(0), "KEY0");
        Model storage m = models[modelId];
        require(m.exists && m.active, "NF");
        require(m.pricingMode == 2, "MODE");

        AccessPlan memory p = _accessPlans[modelId][planId];
        require(p.active, "PLAN_OFF");
        require(msg.value == p.priceWei, "PRICE");

        uint64 cur = accessExpiry[modelId][key];
        uint64 start = cur > uint64(block.number) ? cur : uint64(block.number);
        newExpiry = start + uint64(p.durationBlocks);
        accessExpiry[modelId][key] = newExpiry;

        // payout to current owner / recipient
        address payTo = m.feeRecipient;
        if (payTo == address(0)) {
            payTo = modelNFT.ownerOf(m.tokenId);
        }
        if (msg.value > 0) {
            (bool ok,) = payTo.call{value: msg.value}("");
            require(ok, "PAY_FAIL");
        }

        emit AccessPurchased(modelId, msg.sender, key, planId, newExpiry);
    }

    function setOwnerAccessKey(bytes32 modelId, address key) external {
        _requireTokenOwnerByModelId(modelId);
        require(models[modelId].pricingMode == 2, "MODE");
        require(key != address(0), "KEY0");
        accessExpiry[modelId][key] = type(uint64).max;
        emit OwnerAccessKeySet(modelId, key, type(uint64).max);
    }

    function revokeAccessKey(bytes32 modelId, address key) external {
        _requireTokenOwnerByModelId(modelId);
        require(models[modelId].pricingMode == 2, "MODE");
        require(key != address(0), "KEY0");
        accessExpiry[modelId][key] = 0;
        emit AccessRevoked(modelId, key);
    }
```

B.3 Solidity: `ForestRuntime.sol` (view gating + chunk reads)

B.3.1 View inference and fee-gating rationale

```
contract ForestRuntime {
    bytes4 internal constant CHUNK_MAGIC = 0x474c3143; // "GL1C"
    IModelRegistryRuntime public immutable registry;
    // ---- EIP-712: owner-gated view inference (no-tx) ----
    // We cannot safely allow free view inference for paid models based on msg.sender alone,
    // because eth_call can spoof the caller address. Instead, the current NFT owner signs
    // an EIP-712 message and anyone can relay it in a read-call.
    bytes32 private constant _EIP712DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");
    bytes32 private constant _NAME_HASH = keccak256(bytes("GenesisL1 Forest"));
    bytes32 private constant _VERSION_HASH = keccak256(bytes("1"));
    bytes32 private constant _OWNER_VIEW_TYPEHASH = keccak256("OwnerView(bytes32 modelId,bytes32 packedHash,uint256 deadline)");
    bytes32 private constant _ACCESS_VIEW_TYPEHASH = keccak256("AccessView(bytes32 modelId,bytes32 packedHash,uint256 deadline)");


    event Inference(bytes32 indexed modelId, address indexed caller, int256 scoreQ, uint256 valueWei);
    event InferenceClass(bytes32 indexed modelId, address indexed caller, uint16 classIndex, int256 bestScoreQ, uint256 valueWei);
    // Vector-output inference (model format v2): returns logitsQ per label/class.
    // NOTE: We emit int256[] to preserve the full accumulator range (can exceed int32 for many trees).
    event InferenceMulti(bytes32 indexed modelId, address indexed caller, int256[] logitsQ, uint256 valueWei);

    constructor(address registryAddr) {
        require(registryAddr != address(0), "REG0");
        registry = IModelRegistryRuntime(registryAddr);
    }

    // Read-call inference.
    //
    // IMPORTANT: If a model is configured as "paid required" (mode=2), this function reverts.
    // Paid inference must be performed through predictTx() so fees can be enforced.
    function predictView(bytes32 modelId, bytes calldata packedFeaturesQ) external view returns (int256 scoreQ) {
        // Read the model settings to block free inference for pay-required models.
        // We destructure all 13 fields for cross-solc stability (no blank tuple slots).
        (
            address _tablePtr,
            uint32 _chunkSize,
            uint32 _numChunks,
            uint32 _totalBytes,
            uint16 _nFeatures,
            uint16 _nTrees,
            uint16 _depth,
            int32 _baseQ,
            uint32 _scaleQ,
```

```
            bool enabled,
            uint8 mode,
            uint256 _feeWei,
            address _recipient
        ) = registry.getModelRuntime(modelId);

        // Silence unused-variable warnings.
        _sinkA(_tablePtr);
        _sinkU(_chunkSize);
        _sinkU(_numChunks);
        _sinkU(_totalBytes);
        _sinkU(_nFeatures);
        _sinkU(_nTrees);
        _sinkU(_depth);
        _sinkI(_baseQ);
        _sinkU(_scaleQ);
        _sinkU(_feeWei);
        _sinkA(_recipient);

        require(enabled, "INF_DISABLED");
        require(mode != 2, "PAID_ONLY");

        scoreQ = _predict(modelId, packedFeaturesQ);
    }

    // Read-call inference for multiclass classification (model format v2).
    //
    // IMPORTANT: If a model is configured as "paid required" (mode=2), this function reverts.
    // Paid inference must be performed through predictClassTx() so fees can be enforced.
    function predictClassView(bytes32 modelId, bytes calldata packedFeaturesQ)
        external
        view
        returns (uint16 classIndex, int256 bestScoreQ)
    {
        (
            address tablePtr,
            uint32 chunkSize,
            uint32 numChunks,
            uint32 totalBytes,
            uint16 nFeatures,
            uint16 _nTrees,
            uint16 _depth,
            int32 _baseQ,
            uint32 _scaleQ,
            bool enabled,
            uint8 mode,
            uint256 _feeWei,
            address _recipient
        ) = registry.getModelRuntime(modelId);

        // Silence unused-variable warnings.
        _sinkU(_nTrees);
        _sinkU(_depth);
        _sinkI(_baseQ);
        _sinkU(_scaleQ);
        _sinkU(_feeWei);
        _sinkA(_recipient);
```

```
100
101         require(enabled, "INF_DISABLED");
102         require(mode != 2, "PAID_ONLY");
103
104         (classIndex, bestScoreQ) = _predictClassFromChunks(modelId,
                packedFeaturesQ, tablePtr, chunkSize, numChunks, totalBytes,
                nFeatures);
105     }
106
107     // Read-call inference for vector-output v2 models (multiclass/multilabel).
108     // For multilabel classification, the caller should apply
            sigmoid(logitQ/scaleQ) per label.
109     //
110     // IMPORTANT: If a model is configured as "paid required" (mode=2), this
            function reverts.
111     // Paid inference must be performed through predictMultiTx() so fees can be
            enforced.
112     function predictMultiView(bytes32 modelId, bytes calldata packedFeaturesQ)
113         external
114         view
115         returns (int256[] memory logitsQ)
116     {
117         (
118             address tablePtr,
119             uint32 chunkSize,
120             uint32 numChunks,
121             uint32 totalBytes,
122             uint16 nFeatures,
123             uint16 _nTrees,
124             uint16 _depth,
125             int32 _baseQ,
126             uint32 _scaleQ,
127             bool enabled,
128             uint8 mode,
129             uint256 _feeWei,
130             address _recipient
131         ) = registry.getModelRuntime(modelId);
132
133         // Silence unused-variable warnings.
134         _sinkU(_nTrees);
135         _sinkU(_depth);
136         _sinkI(_baseQ);
137         _sinkU(_scaleQ);
138         _sinkU(_feeWei);
139         _sinkA(_recipient);
140
141         require(enabled, "INF_DISABLED");
142         require(mode != 2, "PAID_ONLY");
143
144         logitsQ = _predictMultiFromChunks(modelId, packedFeaturesQ, tablePtr,
                chunkSize, numChunks, totalBytes, nFeatures);
145     }
```

B.3.2 Chunk addressing and cross-chunk reads via EXTCODECOPY

```
1
```

```
    function _chunkPtrAt(address tablePtr, uint256 chunkIdx) internal view
        returns (address ptr) {
        // read 32-byte slot from table runtime code at offset 4 + chunkIdx*32
        uint256 src = 4 + chunkIdx * 32;
        bytes32 word;
        assembly ("memory-safe") {
            let p := mload(0x40)
            extcodecopy(tablePtr, p, src, 32)
            word := mload(p)
        }
        ptr = address(uint160(uint256(word)));
        require(ptr != address(0), "BAD_PTR");
        _requireChunkMagic(ptr, "CHUNK_CODE");
    }

    function _readBytes(address tablePtr, uint32 chunkSize, uint256 off, uint256
        n) internal view returns (bytes32 outWord) {
        // reads up to 32 bytes starting at off, returns in lowest bytes of
            outWord
        require(n > 0 && n <= 32, "READN");
        uint256 csz = uint256(chunkSize);

        uint256 chunkIdx = off / csz;
        uint256 inChunk = off % csz;

        address ptr = _chunkPtrAt(tablePtr, chunkIdx);

        // if within one chunk
        if (inChunk + n <= csz) {
            assembly ("memory-safe") {
                let p := mload(0x40)
                extcodecopy(ptr, p, add(4, inChunk), n)
                outWord := mload(p)
            }
        } else {
            // boundary: read first part then second part
            uint256 n1 = csz - inChunk;
            uint256 n2 = n - n1;

            bytes memory tmp = new bytes(n);
            assembly ("memory-safe") {
                extcodecopy(ptr, add(tmp, 32), add(4, inChunk), n1)
            }
            address ptr2 = _chunkPtrAt(tablePtr, chunkIdx + 1);
            assembly ("memory-safe") {
                extcodecopy(ptr2, add(add(tmp, 32), n1), 4, n2)
                outWord := mload(add(tmp, 32))
            }
        }
    }

    function _readU16Model(address tablePtr, uint32 chunkSize, uint256 off)
        internal view returns (uint16 v) {
        bytes32 w = _readBytes(tablePtr, chunkSize, off, 2);
        uint256 b0 = uint8(bytes1(w));
        uint256 b1 = uint8(bytes1(w << 8));
        v = uint16(b0 | (b1 << 8));
    }

```

```
57      function _readU8Model(address tablePtr, uint32 chunkSize, uint256 off)
            internal view returns (uint8 v) {
58          bytes32 w = _readBytes(tablePtr, chunkSize, off, 1);
59          v = uint8(bytes1(w));
60      }
61
62      function _readU32Model(address tablePtr, uint32 chunkSize, uint256 off)
            internal view returns (uint32 v) {
63          bytes32 w = _readBytes(tablePtr, chunkSize, off, 4);
64          uint256 b0 = uint8(bytes1(w));
65          uint256 b1 = uint8(bytes1(w << 8));
66          uint256 b2 = uint8(bytes1(w << 16));
67          uint256 b3 = uint8(bytes1(w << 24));
68          v = uint32(b0 | (b1 << 8) | (b2 << 16) | (b3 << 24));
69      }
70
71      function _readI32Model(address tablePtr, uint32 chunkSize, uint256 off)
            internal view returns (int32 v) {
72          bytes32 w = _readBytes(tablePtr, chunkSize, off, 4);
73          uint256 b0 = uint8(bytes1(w));
74          uint256 b1 = uint8(bytes1(w << 8));
75          uint256 b2 = uint8(bytes1(w << 16));
76          uint256 b3 = uint8(bytes1(w << 24));
77          uint32 u = uint32(b0 | (b1 << 8) | (b2 << 16) | (b3 << 24));
78          v = int32(int256(uint256(u)));
79      }
80
81
82      // ---- EIP-712 helpers ----
83      function _domainSeparatorV4() internal view returns (bytes32) {
84          return keccak256(abi.encode(
85              _EIP712DOMAIN_TYPEHASH,
86              _NAME_HASH,
87              _VERSION_HASH,
88              block.chainid,
89              address(this)
90          ));
91      }
92
93      function _hashTypedDataV4(bytes32 structHash) internal view returns (bytes32)
            {
94          return keccak256(abi.encodePacked(hex"1901", _domainSeparatorV4(),
                structHash));
95      }
96
97      function _recover(bytes32 digest, bytes calldata sig) internal pure returns
            (address) {
98          if (sig.length != 65) return address(0);
99          bytes32 r;
100         bytes32 s;
```

B.4 JavaScript: `train_worker.js` (model serialization + a tree builder)

B.4.1 Binary formats `GL1F` v1 and v2 (serialization)

```
function serializeModel({ nFeatures, depth, nTrees, baseQ, scaleQ, trees }) {
  const pow = 1 << depth;
  const internal = pow - 1;
  const perTree = internal * 8 + pow * 4;
  const totalBytes = 24 + nTrees * perTree;

  const out = new Uint8Array(totalBytes);
  const dv = new DataView(out.buffer);

  out[0] = "G".charCodeAt(0);
  out[1] = "L".charCodeAt(0);
  out[2] = "1".charCodeAt(0);
  out[3] = "F".charCodeAt(0);
  out[4] = 1;
  out[5] = 0;

  dv.setUint16(6, nFeatures, true);
  dv.setUint16(8, depth, true);
  dv.setUint32(10, nTrees, true);
  dv.setInt32(14, baseQ, true);
  dv.setUint32(18, scaleQ, true);
  out[22] = 0;
  out[23] = 0;

  let off = 24;
  for (let t = 0; t < nTrees; t++) {
    const tr = trees[t];
    const feat = tr.feat;
    const thr = tr.thr;
    const leaf = tr.leaf;
    for (let i = 0; i < internal; i++) {
      dv.setUint16(off, feat[i], true); off += 2;
      dv.setInt32(off, thr[i], true); off += 4;
      dv.setUint16(off, 0, true); off += 2;
    }
    for (let i = 0; i < pow; i++) {
      dv.setInt32(off, leaf[i], true); off += 4;
    }
  }

  return out;
}

function serializeModelV2({ nFeatures, depth, nClasses, treesPerClass,
    baseLogitsQ, scaleQ, treesByClass }) {
  const pow = 1 << depth;
  const internal = pow - 1;
  const perTree = internal * 8 + pow * 4;

  const headerSize = 24 + nClasses * 4;
  const totalTrees = treesPerClass * nClasses;
  const totalBytes = headerSize + totalTrees * perTree;

  const out = new Uint8Array(totalBytes);
  const dv = new DataView(out.buffer);

  out[0] = "G".charCodeAt(0);
  out[1] = "L".charCodeAt(0);
```

```
  out[2] = "1".charCodeAt(0);
  out[3] = "F".charCodeAt(0);
  out[4] = 2; // version
  out[5] = 0;

  dv.setUint16(6, nFeatures, true);
  dv.setUint16(8, depth, true);
  dv.setUint32(10, treesPerClass, true);
  dv.setInt32(14, 0, true); // reserved
  dv.setUint32(18, scaleQ, true);
  dv.setUint16(22, nClasses, true);

  // base logits
  let off = 24;
  for (let k = 0; k < nClasses; k++) {
    dv.setInt32(off, baseLogitsQ[k] | 0, true);
    off += 4;
  }

  // Trees: class-major (all trees for class0, then class1, ...)
  for (let k = 0; k < nClasses; k++) {
    const clsTrees = treesByClass[k] || [];
    for (let t = 0; t < treesPerClass; t++) {
      const tr = clsTrees[t];
      const feat = tr.feat;
      const thr = tr.thr;
      const leaf = tr.leaf;
      for (let i = 0; i < internal; i++) {
        dv.setUint16(off, feat[i], true); off += 2;
        dv.setInt32(off, thr[i], true); off += 4;
        dv.setUint16(off, 0, true); off += 2;
      }
      for (let i = 0; i < pow; i++) {
        dv.setInt32(off, leaf[i], true); off += 4;
      }
    }
  }

  return out;
}
```

B.4.2 Regression tree builder (histogram/quantile thresholding)

```
function buildTreeRegression({
  X, nRows, nFeatures, trainSamples, residual,
  featMin, featRange, depth, minLeaf, lr, scaleQ, rng,
  bins = 32, binning = "linear", qThr = null
}) {
  // Split-candidate histogram binning is training-only.
  // On-chain format (tree structure + quantized thresholds/leaves) stays
      unchanged.
  const BINS = Math.max(8, bins | 0);
  const isQuantile = String(binning || "").toLowerCase() === "quantile";

  const pow = 1 << depth;
  const internal = pow - 1;

```

```
  const featU16 = new Uint16Array(internal);
  const thrI32 = new Int32Array(internal);
  const leafI32 = new Int32Array(pow);

  function fillForced(nodeIdx, level, leafValQ) {
    if (level === depth) {
      leafI32[nodeIdx - internal] = leafValQ;
      return;
    }
    featU16[nodeIdx] = 0;
    thrI32[nodeIdx] = INT32_MAX;
    fillForced(nodeIdx * 2 + 1, level + 1, leafValQ);
    fillForced(nodeIdx * 2 + 2, level + 1, leafValQ);
  }

  function computeLeafQ(samples) {
    const m = meanResidual(residual, samples);
    const v = lr * m;
    return clampI32(Math.round(v * scaleQ));
  }

  function nodeSplit(nodeIdx, level, samples) {
    if (stopFlag) return;

    if (samples.length === 0) { fillForced(nodeIdx, level, 0); return; }
    if (level === depth) { leafI32[nodeIdx - internal] = computeLeafQ(samples);
        return; }
    if (samples.length < 2 * minLeaf) { fillForced(nodeIdx, level,
        computeLeafQ(samples)); return; }

    const colsample = Math.max(1, Math.round(Math.sqrt(nFeatures)));
    const feats = sampleFeatures(nFeatures, colsample, rng);

    let bestF = -1;
    let bestThrQ = 0;
    let bestSSE = Infinity;

    const cnt = new Int32Array(BINS);
    const sum = new Float64Array(BINS);
    const sum2 = new Float64Array(BINS);

    for (let fi = 0; fi < feats.length; fi++) {
      const f = feats[fi];
      const range = featRange[f];
      if (!(range > 0)) continue;

      const thrArr = isQuantile ? (qThr ? qThr[f] : null) : null;
      if (isQuantile) {
        if (!thrArr || (thrArr.length | 0) !== (BINS - 1)) continue;
      }

      cnt.fill(0); sum.fill(0); sum2.fill(0);

      const minF = featMin[f];
      const inv = 1 / range;

      let totalCount = 0;
      let totalSum = 0;
      let totalSum2 = 0;
```

```
      for (let i = 0; i < samples.length; i++) {
        const r = samples[i];
        const x = X[r * nFeatures + f];
        const rr = residual[r];

        let b = 0;
        if (isQuantile) {
          // Lower-bound: first threshold >= x. Returns [0..BINS-1].
          let lo = 0, hi = thrArr.length;
          while (lo < hi) {
            const mid = (lo + hi) >> 1;
            if (x <= thrArr[mid]) hi = mid;
            else lo = mid + 1;
          }
          b = lo;
        } else {
          b = Math.floor(((x - minF) * inv) * BINS);
          if (b < 0) b = 0;
          else if (b >= BINS) b = BINS - 1;
        }

        cnt[b] += 1;
        sum[b] += rr;
        sum2[b] += rr * rr;

        totalCount += 1;
        totalSum += rr;
        totalSum2 += rr * rr;
      }

      if (totalCount < 2 * minLeaf) continue;

      let leftCount = 0;
      let leftSum = 0;
      let leftSum2 = 0;

      for (let b = 0; b < BINS - 1; b++) {
        leftCount += cnt[b];
        leftSum += sum[b];
        leftSum2 += sum2[b];

        const rightCount = totalCount - leftCount;
        if (leftCount < minLeaf || rightCount < minLeaf) continue;

        const rightSum = totalSum - leftSum;
        const rightSum2 = totalSum2 - leftSum2;

        const leftSSE = leftSum2 - (leftSum * leftSum) / leftCount;
        const rightSSE = rightSum2 - (rightSum * rightSum) / rightCount;
        const sse = leftSSE + rightSSE;

        if (sse < bestSSE) {
          bestSSE = sse;
          bestF = f;
          const thrF = isQuantile ? thrArr[b] : (minF + range * ((b + 1) / BINS));
          bestThrQ = clampI32(Math.round(thrF * scaleQ));
        }
      }
```

```
    }

    if (bestF < 0) { fillForced(nodeIdx, level, computeLeafQ(samples)); return; }

    const left = [];
    const right = [];
    for (let i = 0; i < samples.length; i++) {
      const r = samples[i];
      const x = X[r * nFeatures + bestF];
      const xQ = clampI32(Math.round(x * scaleQ));
      if (xQ > bestThrQ) right.push(r);
      else left.push(r);
    }

    if (left.length < minLeaf || right.length < minLeaf) { fillForced(nodeIdx,
        level, computeLeafQ(samples)); return; }

    featU16[nodeIdx] = bestF;
    thrI32[nodeIdx] = bestThrQ;

    nodeSplit(nodeIdx * 2 + 1, level + 1, left);
    nodeSplit(nodeIdx * 2 + 2, level + 1, right);
  }

  nodeSplit(0, 0, Array.from(trainSamples));
  return { feat: featU16, thr: thrI32, leaf: leafI32 };
}
```

B.5 JavaScript: `create_page.js` (deployment chunking and pointer-table creation)

```
      if (!trained?.bytes?.length) throw new Error("Train a model first");
      if (!datasetNumeric?.featureNames?.length) throw new Error("Feature labels
          missing");
      if (!iconBytes?.length) throw new Error("Upload icon");
      if (!ownerKeyAddr?.value) throw new Error("Generate owner API key");
      if (!ownerKeySaved?.checked) throw new Error("Confirm you saved the owner
          API key private key");
      if (!(agreeTos.checked && agreeLicense.checked)) throw new Error("Agree to
          Terms and License");

      const title = metaName.value.trim();
      const desc = metaDesc.value.trim();
      if (title.length < 3 || desc.length < 8) throw new Error("Provide name +
          description");
      const words = titleWordHashes(title);
      if (!words.length) throw new Error("Title should include at least one word
          (2+ chars)");

      const mode = Number(pricingMode.value);
      const feeEth = clamp(pricingFee.value || "0", 0.001, 1);
      let feeWei = 0n;
      if (mode === 0) feeWei = 0n;
      else feeWei = ethToWei(String(feeEth));

      const { signer } = await getSignerProvider();
```

```
      const signerAddr = await signer.getAddress();

      const store = new ethers.Contract(mustAddr(sys.store), ABI_STORE, signer);
      const registry = new ethers.Contract(mustAddr(sys.registry), ABI_REGISTRY,
          signer);

      // Model bytes
      const bytes = trained.bytes;
      const total = bytes.length;

      // Read chain settings via the dedicated RPC (more reliable than wallet
          provider for eth_call).
      const rprov = getReadProvider(sys.rpc);
      const regRead = new ethers.Contract(mustAddr(sys.registry), ABI_REGISTRY,
          rprov);

      let deployFeeWei = 0n;
      let sizeFeeWeiPerByte = 0n;
      let requiredFeeWei = 0n;
      let licId = 0;
      let tosVer = 0;
      try { deployFeeWei = BigInt(await regRead.deployFeeWei()); } catch {}
      try { sizeFeeWeiPerByte = BigInt(await regRead.sizeFeeWeiPerByte()); }
          catch {}
      try {
        requiredFeeWei = BigInt(await regRead.requiredDeployFeeWei(total));
      } catch {
        requiredFeeWei = deployFeeWei + (sizeFeeWeiPerByte * BigInt(total));
      }
      try { licId = Number(await regRead.activeLicenseId()); } catch {}
      try { tosVer = Number(await regRead.tosVersion()); } catch {}

      dlog(`[${nowTs()}] Deploy fee (base): ${weiToEth(deployFeeWei)} L1`);
      dlog(`[${nowTs()}] Size fee: ${weiToEth(sizeFeeWeiPerByte)} L1 per byte`);
      dlog(`[${nowTs()}] Required deploy value: ${weiToEth(requiredFeeWei)} L1`);
      dlog(`[${nowTs()}] Active licenseId=${licId} tosVersion=${tosVer}`);

      let recipient = signerAddr;
      if (pricingRecipient.value.trim()) recipient =
          mustAddr(pricingRecipient.value.trim());

      // chunking (fixed)
      const chunkSize = CHUNK_SIZE;
      const numChunks = Math.ceil(total / chunkSize);

      dlog(`[${nowTs()}] Chunking: total=${total} chunkSize=${chunkSize} (fixed)
          chunks=${numChunks}`);

      const iface = new ethers.Interface(ABI_STORE);
      const ptrs = [];

      for (let i=0;i<numChunks;i++){
        const start = i*chunkSize;
        const end = Math.min(total, start+chunkSize);
        const chunk = bytes.slice(start,end);
        dlog(`[${nowTs()}] Chunk ${i+1}/${numChunks}: store.write(${chunk.length}
            bytes)`);
        const tx = await store.write(chunk, { gasLimit: 30_000_000 });
        dlog(`  tx.hash ${tx.hash}`);
```

```
        const rcpt = await tx.wait();
        dlog(`  mined status=${rcpt.status}
            gasUsed=${rcpt.gasUsed?.toString?.()||"?"}`);
        if (rcpt.status !== 1) throw new Error("chunk write reverted");

        let ptr = null;
        for (const lg of rcpt.logs) {
          try {
            const pl = iface.parseLog(lg);
            if (pl?.name === "ChunkWritten") { ptr = pl.args.pointer; break; }
          } catch {}
        }
        if (!ptr) throw new Error("ChunkWritten not found");
        ptrs.push(ptr);
        dlog(`  chunk pointer: ${ptr}`);
      }

      // pointer table chunk: 32 bytes each pointer
      const table = new Uint8Array(32*numChunks);
      for (let i=0;i<numChunks;i++){
        const addr = ethers.getAddress(ptrs[i]);
        const ab = ethers.getBytes(addr);
        table.set(ab, i*32 + 12);
      }
      dlog(`[${nowTs()}] Writing pointer-table: ${table.length} bytes`);
      const ttx = await store.write(table, { gasLimit: 30_000_000 });
      dlog(`  table tx.hash ${ttx.hash}`);
      const trc = await ttx.wait();
      dlog(`  table mined status=${trc.status}
          gasUsed=${trc.gasUsed?.toString?.()||"?"}`);
      if (trc.status !== 1) throw new Error("table write reverted");

      let tablePtr = null;
      for (const lg of trc.logs) {
        try {
          const pl = iface.parseLog(lg);
          if (pl?.name === "ChunkWritten") { tablePtr = pl.args.pointer; break; }
        } catch {}
      }
      if (!tablePtr) throw new Error("table ChunkWritten not found");
      dlog(`[${nowTs()}] Pointer-table pointer: ${tablePtr}`);

      // Register
      const modelId = trained.modelId;
      let labelsForNft = null;
      let labelNamesForNft = null;
      if (selectedTask === "binary_classification" && datasetNumeric?.classes) {
        // Class labels for binary classification.
        labelsForNft = [datasetNumeric.classes[0], datasetNumeric.classes[1]];
      } else if (selectedTask === "multiclass_classification" &&
          Array.isArray(datasetNumeric?.classes)) {
        // Class labels for multiclass classification.
        labelsForNft = datasetNumeric.classes;
      } else if (selectedTask === "multilabel_classification" &&
          Array.isArray(datasetNumeric?.labelNames)) {
        // Multilabel:
        // - `labelNames` are the output label names (one per selected label
            column)
        // - `labels` are the binary class labels (defaults to 0/1)
```

```
        labelNamesForNft = datasetNumeric.labelNames;
        labelsForNft = ["0", "1"];
      }
      const featuresPacked = packNftFeatures({
        task: selectedTask,
        // Regression/binary/multiclass store the single label column name.
            Multilabel stores labelNames instead.
        labelName: (selectedTask === "multilabel_classification") ?
            "(multilabel)" : datasetNumeric.labelName,
        labels: labelsForNft,
        labelNames: labelNamesForNft,
        featureNames: datasetNumeric.featureNames
      });

      const depth = trained.decoded.depth;
      const nTrees = trained.decoded.nTrees;
      const nFeatures = trained.decoded.nFeatures;
```

Appendix C

Reproducibility Checklist

- **Determinism:** keep training seeds, dataset hashes, and serialization version (`GL1F` v1/v2) alongside the model NFT metadata.
- **Quantization:** record `scaleQ` and confirm no int32 overflow for feature magnitudes (Sec. 4.5).
- **Bytes integrity:** store and publish `keccak256(modelBytes)` and (optionally) the ordered list of chunk pointers.
- **Contract addresses:** pin the deployed addresses of `ModelRegistry`, `ModelNFT`, `ForestRuntime`, and `ModelStore`.
- **Inference parity:** test N random feature vectors: local JS inference vs on-chain `predictView`/`predictTx` must match bit-for-bit.
- **Pricing modes:** if using paid-required mode, test: (a) fee enforcement on tx, (b) owner EIP-712 view path, and (c) access-key view path.
- **License + Terms:** record the accepted `tosVersion` and `licenseId` at registration time.

Appendix D

References

Bibliography

[1] J. H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, 29(5):1189–1232, 2001. DOI: 10.1214/aos/1013203451. `https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy-function-approximation-A-gradient-boosting-machine/10.1214/aos/1013203451.full`.

[2] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD*, 2016. DOI: 10.1145/2939672.2939785. `https://arxiv.org/abs/1603.02754`.

[3] G. Ke et al. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *NeurIPS*, 2017. `https://proceedings.neurips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf`.

[4] Ethereum Improvement Proposals. EIP-170: Contract code size limit. `https://eips.ethereum.org/EIPS/eip-170`.

[5] Ethereum Improvement Proposals. EIP-712: Typed structured data hashing and signing. `https://eips.ethereum.org/EIPS/eip-712`.

[6] Ethereum Improvement Proposals. EIP-1474: Remote procedure call specification (`eth_call` supports an optional `from`). `https://eips.ethereum.org/EIPS/eip-1474`.

[7] 0xSequence. SSTORE2: cheaper storage in contract bytecode and reads via `EXTCODECOPY`. `https://github.com/0xsequence/sstore2`.

[8] Creative Commons. Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) deed. `https://creativecommons.org/licenses/by-sa/4.0/deed.en`.

[9] Chainlist. Genesis L1 (chainId 29) network parameters. `https://chainlist.org/chain/29`.