

Timber: An Ahead-of-Time Compiler and Serving Infrastructure for Classical Machine Learning Inference

Timber Contributors¹

¹Independent Research

February 28, 2026

Abstract

We present **Timber**, an ahead-of-time (AOT) compiler and serving infrastructure for classical machine learning models. Timber ingests trained model artifacts from five major frameworks—XGBoost, LightGBM, scikit-learn, CatBoost, and ONNX—compiles them through a framework-agnostic intermediate representation (IR) and six domain-specific optimization passes, and emits native inference code targeting C99, WebAssembly, or MISRA-C-compliant output. Timber also provides an Ollama-style workflow: a single `timber load` command compiles and caches a model locally, and `timber serve` exposes it over a REST API with sub-100 μ s warm inference latency. On a breast cancer classification benchmark (50 trees, depth 4, 30 features), Timber achieves **336 \times lower P50 latency** (2 μ s vs. 672 μ s) and **533 \times higher throughput** compared to XGBoost’s Python inference, while maintaining bit-exact numerical agreement ($< 10^{-5}$ maximum absolute error). The generated code requires no runtime dependencies, no dynamic allocation, and no recursion. Timber bridges the gap between training in Python and deploying at native speed—one command to load, one command to serve.

1 Introduction

Classical machine learning models—gradient-boosted trees (GBTs), random forests, and decision tree pipelines—remain the dominant

workhorses of production ML in finance, healthcare, fraud detection, ad tech, and industrial applications [?, ?, ?]. Despite their simplicity relative to deep neural networks, deploying these models in latency-sensitive environments carries significant overhead: Python interpreter startup, framework runtime initialization, memory allocation for intermediate structures, GIL contention under concurrent load, and dependency management complexity.

The deep-learning ecosystem solved its serving problem with optimized runtimes (TensorRT, TVM, Triton). More recently, the LLM ecosystem adopted a radically simple user experience through tools like Ollama [?]: `ollama pull`, `ollama run`. Classical ML has no equivalent. Models trained in XGBoost or scikit-learn are typically deployed behind a Flask endpoint with the entire Python runtime—a 200 MB process serving a 2 MB model.

Timber addresses this gap from two angles:

1. **Compilation.** Timber is an AOT compiler that treats a trained model as a program specification and applies compiler techniques to produce minimal, deterministic, zero-dependency native inference code.
2. **Serving.** Timber provides an Ollama-style CLI and HTTP server: one command to compile and cache a model, one command to serve it over a REST API with sub-100 μ s inference latency.

The key contributions are:

1. A typed, framework-agnostic IR for inference pipelines with support for multi-stage models, ensemble metadata, and vectorization annotations (§??).
2. Front-end parsers for five frameworks: XGBoost, LightGBM, scikit-learn, CatBoost, and ONNX (§??).
3. Six domain-specific optimization passes exploiting tree ensemble structure (§??).
4. Three code generation backends: C99, WebAssembly (WAT), and MISRA-C for safety-critical deployment (§??).
5. A local model store and HTTP inference server providing an Ollama-style developer experience (§??).
6. Differential compilation enabling incremental rebuilds when models are updated (§??).
7. Empirical validation showing 300–500× speedup with bit-exact numerical fidelity (§??).

2 System Design

2.1 Architecture Overview

Timber follows a classical compiler architecture extended with a model store and serving layer:

Artifact $\xrightarrow{\text{parse}}$ IR $\xrightarrow{\text{optimize}}$ IR' $\xrightarrow{\text{emit}}$ C99/WASM $\xrightarrow{\text{compile over sub-models}}$.so $\xrightarrow{\text{HTTP input passthrough}}$ HTTP

The compilation pipeline consists of three phases:

1. **Front-end:** Format-specific parsers convert model artifacts into a canonical IR. Auto-detection selects the appropriate parser based on file extension and content inspection.
2. **Middle-end:** A configurable sequence of six optimization passes transforms the IR, each producing an auditable change log.
3. **Back-end:** Code emitters produce self-contained source code for the target platform (C99, WebAssembly, or MISRA-C).

The serving layer adds:

4. **Model store:** A local registry (`~/.timber/models/`) that caches compiled artifacts by name.
5. **HTTP server:** An Ollama-style REST API exposing compiled models for inference.

2.2 Intermediate Representation

The Timber IR represents an inference pipeline as an ordered sequence of *stages*, each a typed dataclass:

- **ScalerStage:** Elementwise affine transform $(x - \mu_i)/\sigma_i$ with per-feature means and scales.
 - **TreeEnsembleStage:** A collection of decision trees with split conditions, leaf values, objective type, learning rate, base score, and an `annotations` dictionary for optimization metadata.
 - **EncoderStage:** Categorical-to-numeric transforms (ordinal, one-hot, target encoding).
 - **VotingEnsembleStage:** Weighted aggregation across multiple sub-model IRs.
 - **StackingEnsembleStage:** Meta-learner over sub-model predictions with optional input passthrough.
- Each **Tree** is stored as a flat array of **TreeNode** structures with explicit parent-child indices, feature index, threshold, default-left flag, and leaf value. This representation enables $O(1)$ random access during optimization passes and efficient serialization. The IR supports full JSON round-tripping for reproducibility and debugging.
- The **Objective** enum captures the model’s task type: **REGRESSION**, **BINARY_CLASSIFICATION**, or **MULTICLASS_CLASSIFICATION**. This drives the back-end’s choice of activation function (identity, sigmoid, or softmax) and output dimensionality.

3 Front-End Parsers

Timber supports five model formats through dedicated parsers. All parsers produce the same IR, enabling cross-framework optimization and code generation.

3.1 XGBoost JSON

The XGBoost parser handles the JSON model dump format (version ≥ 2.0), extracting parallel arrays of split indices, thresholds, child pointers, and default-left flags. A critical detail: XGBoost stores `base_score` in probability space for logistic objectives; Timber applies the logit transform $\text{logit}(p) = \ln(p/(1 - p))$ to convert to margin space for correct accumulation.

3.2 LightGBM Text

The LightGBM parser processes the `model.txt` format, parsing tree structures from the text-based array representation with negative-indexed leaf references. Input validation rejects empty files and malformed content with descriptive error messages.

3.3 scikit-learn Pickle

The scikit-learn parser deserializes `.pkl` files and extracts tree structures from fitted estimators. Supported model types include:

- `GradientBoostingClassifier/Regressor`
- `RandomForestClassifier/Regressor`
- `HistGradientBoostingClassifier/Regressor`
- `DecisionTreeClassifier/Regressor`
- Pipeline with `StandardScaler` + tree estimator

For pipelines containing a `StandardScaler`, the parser emits a `ScalerStage` followed by a `TreeEnsembleStage`, preserving the pipeline structure for potential fusion by the optimizer.

3.4 CatBoost JSON

CatBoost uses *oblivious decision trees*—symmetric trees where every node at the same depth uses the same split feature and threshold. The parser converts CatBoost’s JSON format into the Timber IR’s general tree representation, expanding the oblivious structure into explicit internal nodes and leaves. Splits are stored bottom-up in CatBoost’s format; the parser reverses this ordering during tree construction.

3.5 ONNX ML Opset

The ONNX parser handles `TreeEnsembleClassifier` and `TreeEnsembleRegressor` operators from the ONNX ML opset. It extracts the flat node arrays (node IDs, feature IDs, thresholds, modes, tree IDs) and reconstructs the tree structures. This enables Timber to accept models exported from any ONNX-compatible framework.

3.6 Auto-Detection

The `detect_format` function identifies the model format from file extension and content:

- `.json`: Inspects keys (`learner` \rightarrow XGBoost, `oblivious_trees` \rightarrow CatBoost)
- `.txt`, `.model`: LightGBM text format
- `.pkl`, `.pickle`: scikit-learn pickle
- `.onnx`: ONNX protobuf

4 Optimization Passes

Timber implements six optimization passes, each operating on the IR and producing a documented change log for audit purposes. Passes are executed sequentially by the `OptimizerPipeline`, and each reports whether it made changes and the resulting IR.

4.1 Pass 1: Dead Leaf Elimination

Leaves whose absolute contribution is below a configurable threshold ϵ relative to the maximum leaf value are pruned:

$$\text{prune if } |v_\ell| < \epsilon \cdot \max_{\ell'} |v_{\ell'}| \quad (1)$$

When both children of an internal node are pruned, the node collapses to a leaf with the average value. This reduces code size and inference-time branch traversals.

4.2 Pass 2: Constant Feature Detection

When both children of an internal node are leaves with identical values, the split is redundant and the node is folded into a single leaf. This occurs when a feature provides no discriminative power at a particular node—common in models trained with early stopping.

4.3 Pass 3: Threshold Quantization

For each feature, all split thresholds across all trees are analyzed to determine the minimum precision required:

- **int8:** All thresholds are integers in $[-128, 127]$.
- **int16:** All thresholds are integers in $[-32768, 32767]$.
- **float16:** Fewer than 4 significant decimal digits.
- **float32:** Default.

This metadata is stored in the IR annotations and is available to SIMD-specialized backends.

4.4 Pass 4: Frequency-Ordered Branch Sorting

Given calibration data, each internal node’s left-/right branch frequency is counted. If the less-taken branch is currently the fall-through path, the children are swapped (with threshold adjustment) to improve branch prediction and instruction cache utilization on pipelined architectures.

4.5 Pass 5: Pipeline Fusion

When a **ScalerStage** (with per-feature mean μ_i and scale σ_i) precedes a **TreeEnsembleStage**, the scaler is absorbed into the tree thresholds:

$$\theta'_i = \theta_i \cdot \sigma_i + \mu_i \quad (2)$$

This eliminates the runtime scaling step entirely, reducing the pipeline to a single stage with zero overhead for preprocessing.

4.6 Pass 6: Vectorization Analysis

The vectorization pass analyzes the tree ensemble to identify opportunities for SIMD-batched inference. For each tree, it computes:

- **Depth profile:** Maximum and average depth across all trees.
- **Feature access order:** The sequence of feature indices accessed along root-to-leaf paths.
- **Feature frequency:** How often each feature appears in split nodes across the ensemble.
- **Structure groups:** Sets of trees with identical topology (same depth, same branching pattern), which can be evaluated in lockstep with SIMD instructions.

The pass produces a **VectorizationHint** stored in the ensemble’s **annotations** field, including a recommended batch tile size. Trees with uniform shallow depth (≤ 3) receive larger tile recommendations, as their traversal maps well to SIMD gather/scatter patterns.

5 Code Generation

Timber supports three code generation backends, each targeting a different deployment scenario.

5.1 C99 Emitter (Primary)

The primary back-end produces five files: **model.h** (public API with ABI version), **model_data.c** (static const arrays), **model.c** (inference logic), **CMakeLists.txt**, and **Makefile**.

5.1.1 Design Guarantees

- **No dynamic allocation:** All data is `static const`; the context struct is statically allocated.
- **No recursion:** Tree traversal uses iterative `while` loops with bounded iteration count equal to tree depth.
- **Double-precision accumulation:** Tree output sums use `double` before casting¹ to `float` at output, matching framework²-native precision.³
- **NaN handling:** Missing values follow the⁴ `default_left` path, matching XGBoost⁵/LightGBM semantics.⁶
- **Thread safety:** The context is read⁸-only after initialization; concurrent⁹ `timber_infer` calls are safe.
- **ABI versioning:** `TIMBER_ABI_VERSION` enables runtime compatibility checks.
- **Error codes:** All functions return typed error codes (`TIMBER_OK`, `TIMBER_ERR_NULL`, etc.) with `timber_strerror()` for human-readable messages.

5.1.2 Runtime Logging

The generated code includes an optional logging callback:

Listing 1: Runtime logging API

```
1 typedef void (*timber_log_fn)(
2     int level, const char* msg);
3 void timber_set_log_callback(
4     timber_log_fn fn);
5 const char* timber_strerror(int code
6     );
```

Levels follow syslog convention: 0 = error, 1 = warn, 2 = info, 3 = debug. When no callback is set, logging is a no-op with zero overhead.

5.1.3 Multi-class Classification

For multi-class objectives, the emitter generates an inline softmax with double-precision accumulation:

$$p_k = \frac{\exp(s_k - \max_j s_j)}{\sum_j \exp(s_j - \max_j s_j)} \quad (3)$$

where s_k is the accumulated score for class k . The max-subtraction ensures numerical stability.

5.1.4 Generated API

Listing 2: Public C API

```
int timber_init(TimberCtx** ctx);
void timber_free(TimberCtx* ctx);
int timber_infer(const float* inputs
    ,
    int n_samples, float* outputs,
    const TimberCtx* ctx);
int timber_infer_single(
    const float inputs[N_FEATURES],
    float outputs[N_OUTPUTS],
    const TimberCtx* ctx);
```

5.2 WebAssembly Emitter

The WASM back-end produces WebAssembly Text Format (WAT) and JavaScript bindings for browser and edge deployment. The emitter generates:

- A `model.wat` file with linear memory layout for tree data, a `$traverse_tree` function, and the main `$timber_infer_single` export.
- A `timber_model.js` file providing `loadTimberModel()` which instantiates the WASM module, writes inputs to linear memory, invokes inference, and reads outputs—exposing a simple `predict(features)` API.

For binary classification, the emitter includes a sigmoid activation ($\sigma(x) = 1/(1 + e^{-x})$) implemented via an `$exp_neg` helper in WAT.

5.3 MISRA-C Compliance Mode

For safety-critical deployment (automotive, medical devices, avionics), Timber provides a MISRA-C:2012 compliant emitter. The `MisraCEmitter` wraps the standard C99 emitter and applies post-processing transformations:

- Unsigned integer suffix enforcement (Rule 10.1)
- No compiler extensions (Rule 1.1)
- No redefinition of standard identifiers (Rule 21.1)
- Compliance marker in generated headers

A built-in compliance checker validates the generated code against seven MISRA rules and produces a `MisraReport` with violation and warning counts.

6 Serving Infrastructure

Inspired by Ollama’s developer experience for LLMs, Timber provides an end-to-end workflow for classical ML models: load once, serve anywhere.

6.1 Model Store

The model store (`~/.timber/models/`) is a local registry that caches compiled artifacts. When the user runs `timber load model.json --name fraud-detector`, Timber:

1. Auto-detects the model format.
2. Parses the artifact into the IR.
3. Runs the full optimization pipeline (6 passes).
4. Emits C99 source code.
5. Compiles a shared library (`.so/.dylib`) via `gcc`.
6. Stores everything under `~/.timber/models/fraud-detector/` with a `model_info.json` registry entry.

Subsequent `timber serve fraud-detector` calls load the pre-compiled library instantly—no recompilation.

6.2 CLI Interface

Listing 3: Ollama-style CLI

```
1 # Load and compile a model
2 timber load model.json --name my-
   model
3
4 # List cached models
5 timber list
6
7 # Serve over HTTP (port 11434)
8 timber serve my-model
9
10 # Remove a cached model
11 timber remove my-model
```

6.3 HTTP API

The server exposes an Ollama-compatible REST API:

Table 1: HTTP API endpoints

Endpoint	Method	Description
<code>/api/predict</code>	POST	Run inference
<code>/api/generate</code>	POST	Alias (Ollama compat)
<code>/api/models</code>	GET	List loaded models
<code>/api/model/:id</code>	GET	Model metadata
<code>/api/health</code>	GET	Health check

The predict endpoint accepts JSON:

Listing 4: Inference request

```
1 curl http://localhost:11434/api/
   predict \
2   -d '{"model": "fraud-detector",
3      "inputs": [[1.0, 2.0, ...]]}'
```

and returns:

Listing 5: Inference response

```
1 {"model": "fraud-detector",
2  "outputs": [0.97],
3  "n_samples": 1,
4  "latency_us": 91.0,
5  "done": true}
```

The server handles the HTTP envelope in Python; the actual inference call goes directly to the compiled C shared library via `ctypes`. This

architecture means Python is never in the inference hot path—it only parses JSON and copies buffers.

6.4 Serving Latency

On a MacBook Pro (Apple M-series), the end-to-end HTTP round-trip for a 50-tree breast cancer model measured $91\mu\text{s}$ warm latency at the `/api/predict` endpoint (measured via `curl`). The C inference itself is $\sim 2\mu\text{s}$; the remainder is HTTP parsing and JSON serialization.

7 Differential Compilation

In production, models are retrained frequently. Recompiling the entire model on every update is wasteful when only a few trees have changed. Timber’s differential compilation module addresses this:

1. **Tree hashing:** Each tree is assigned a content hash based on its structure (node IDs, feature indices, thresholds, leaf values, child pointers).
2. **Diff computation:** Given two IR instances (old and new), Timber identifies added, removed, modified, and unchanged trees by comparing hash sets.
3. **Incremental assembly:** The new IR is annotated with diff metadata, enabling downstream tooling to perform selective recompilation.

This is particularly valuable in model monitoring pipelines where a model is retrained hourly on fresh data—typically only 5–20% of trees change between versions.

8 Ensemble Composition

Timber supports composing multiple compiled models into higher-order ensembles:

- **Voting ensembles:** Weighted soft or hard vote across N sub-models. Each sub-model is a complete Timber IR; the

`VotingEnsembleStage` aggregates their predictions with configurable weights.

- **Stacking ensembles:** A meta-learner (itself a Timber IR) trained on the outputs of N base models. The `StackingEnsembleStage` supports optional passthrough of original features.

These compositions are first-class IR stages, enabling the full optimization pipeline to operate on the ensemble structure.

9 Evaluation

9.1 Experimental Setup

We evaluate Timber on the UCI Breast Cancer Wisconsin dataset (569 samples, 30 features) using an XGBoost binary classifier (50 trees, max depth 4, learning rate 0.1). The compiled model targets x86_64 with `gcc -O3 -std=c99`. The Python baseline uses XGBoost 2.0+ with `booster.predict()`. All benchmarks use 1,000 warmup iterations followed by 10,000 timed iterations, reporting P50/P95/P99 latency and throughput.

9.2 Single-Sample Latency

Table 2: Single-sample inference latency (μs)

Method	P50	P95	P99	Tput
XGBoost Python	672	2,776	13,468	840/s
Timber C99	2	2	3	447,868/s
Speedup	336×	1,388×	4,489×	533×

9.3 Batch Inference

9.4 Numerical Accuracy

The key to achieving bit-exact results is twofold: (1) converting XGBoost’s probability-space `base_score` to logit space via $\text{logit}(p) = \ln(p/(1-p))$, and (2) using `double` accumulation for the tree output sum before the final sigmoid/-softmax and cast to `float`.

Table 3: Batch inference latency (μ s, batch size = 10)

Method	P50	P95	P99	Tpu
XGBoost Python	521	1,305	13,151	11,311/3
Timber C99	15	39	163	475,195/1
Speedup	35\times	33\times	81\times	42\times

Table 4: Numerical accuracy vs. XGBoost Python (114 test samples)

Metric	Value
Max absolute error	$< 10^{-5}$
Mean absolute error	$< 10^{-6}$
Divergent samples ($> 10^{-5}$)	0 / 114
Classification agreement	100%

9.5 Tail Latency

Timber’s P99 latency (3μ s) is $4,489\times$ lower than XGBoost Python’s P99 ($13,468\mu$ s). The Python runtime’s garbage collection, GIL, and DMatrix construction create extreme tail latency; Timber’s deterministic code path eliminates these sources entirely.

9.6 Artifact Size

For the 50-tree breast cancer model:

Table 5: Deployment artifact comparison

Artifact	Size
Timber compiled .so	48 KB
Timber C99 source	58 KB
XGBoost Python runtime	~ 50 MB

The compiled artifact is approximately **1,000 \times smaller** than the Python runtime required to serve the same model.

9.7 Framework Coverage

We validate correct parsing and compilation across all five supported frameworks:

Table 6: Test coverage by component (144 tests total)

Component	Tests
XGBoost parser	7
LightGBM parser	(via rigorous)
scikit-learn parser	7
CatBoost parser	7
WASM emitter	5
Vectorization pass	6
Diff compilation	6
MISRA-C compliance	4
Stacking/voting ensembles	4
Model store + CLI	18
Auto-detect + formats	4
Multi-class softmax	5
Fuzz testing (robustness)	17
Rigorous end-to-end	13
Total	144

10 Production Considerations

10.1 Drop-in Predictor

Timber provides a Python `ctypes` wrapper (TimberPredictor) that serves as a drop-in replacement for framework `predict()` methods:

Listing 6: Drop-in prediction

```

1 from timber.runtime import
   TimberPredictor
2
3 pred = TimberPredictor.from_model("
   model.json")
4 y = pred.predict(X) # numpy in/out

```

10.2 Audit Trail

Every compilation produces a deterministic JSON audit report containing: input artifact SHA-256 hash, compiler version, optimization pass log with per-pass timing, output file hashes, model summary, and target specification. This supports regulatory review in financial and healthcare applications.

10.3 Limitations

- The WASM emitter produces WAT text; binary encoding (`.wasm`) requires an external tool (e.g., `wat2wasm`).
- Inter-tree optimizations (tree merging, ensemble distillation) are not yet implemented.
- The CatBoost parser supports oblivious trees only; non-oblivious CatBoost models require conversion.
- The HTTP server uses Python’s `http.server`; production deployments at scale should front this with a reverse proxy.
- SIMD code generation from vectorization hints is analyzed but not yet emitted; the current code relies on the C compiler’s auto-vectorization.

11 Related Work

Treelite [?] compiles tree ensembles to C but uses monolithic code generation without an intermediate representation or optimization passes. It supports XGBoost and LightGBM but not scikit-learn pipelines or CatBoost.

Hummingbird [?] converts tree models to tensor computations, running inference through PyTorch or ONNX Runtime. This enables GPU acceleration but introduces significant runtime dependencies.

lleaves [?] uses LLVM to JIT-compile LightGBM models. While achieving native performance, it is framework-specific and requires the LLVM toolchain at deployment time.

ONNX Runtime [?] provides generic ML inference with the TreeEnsemble ML operator, but incurs dynamic dispatch overhead and requires the full runtime.

Ollama [?] pioneered the “load and serve” developer experience for LLMs. Timber adapts this UX paradigm for classical ML: `timber load` replaces `ollama pull`, and `timber serve` replaces `ollama run`.

Timber differentiates by providing: (a) a framework-agnostic IR enabling cross-framework optimization, (b) six domain-specific optimizer passes, (c) three code generation backends (C99, WASM, MISRA-C), (d) an Ollama-style serving workflow, (e) differential compilation for incremental updates, and (f) deterministic audit trails for regulated environments.

12 Conclusion

Timber demonstrates that applying classical compiler techniques to trained ML models yields dramatic inference performance improvements—300–500× over Python-based inference—with bit-exact numerical fidelity. By combining AOT compilation with an Ollama-style developer experience, Timber makes native-speed inference as simple as:

```
1 pip install timber
2 timber load model.json --name my-
  model
3 timber serve my-model
```

The generated code is zero-allocation, zero-dependency C99 suitable for the most constrained deployment targets: embedded systems, edge devices, latency-critical microservices, and regulated environments requiring full auditability. With support for five input frameworks, three output targets, and a comprehensive optimization pipeline, Timber bridges the gap between training in Python and deploying at native speed.

References

- [1] H. Cho et al., “Treelite: Toolbox for decision tree deployment,” 2018. <https://github.com/dmlc/treelite>
- [2] S. Nakandala et al., “A Tensor Compiler for Unified Machine Learning Prediction Serving,” OSDI, 2020.
- [3] S. Boehm, “lleaves: Compiled LightGBM Models,” 2021. <https://github.com/siboehm/lleaves>

- [4] Microsoft, “ONNX Runtime: Cross-platform, high performance ML inferencing and training accelerator,” 2019. <https://onnxruntime.ai>
- [5] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” KDD, 2016.
- [6] G. Ke et al., “LightGBM: A Highly Efficient Gradient Boosting Decision Tree,” NeurIPS, 2017.
- [7] L. Prokhorenkova et al., “CatBoost: Unbiased boosting with categorical features,” NeurIPS, 2018.
- [8] Ollama, “Get up and running with large language models,” 2023. <https://ollama.ai>
- [9] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” JMLR, vol. 12, pp. 2825–2830, 2011.