TerseTS: A Framework for Time Series Compression

Carlos Enrique Muñiz-Cuza
TU Berlin & BIFOLD, Germany
muniz.cuza@tu-berlin.de

Søren Kejser Jensen
Aalborg University, Denmark
skj@cs.aau.dk

Tom Louis Klein
TU Berlin, Germany
t.klein.1@campus.tu-berlin.de

Sabina Bakhtiiarova
TU Berlin, Germany
s.bakhtiiarova@campus.tu-berlin.de

Matthias Boehm
TU Berlin & BIFOLD, Germany
matthias.boehm@tu-berlin.de

Torben Bach Pedersen
Aalborg University, Denmark
tbp@cs.aau.dk

Abstract

TerseTS is an open-source framework for lossless and lossy time series compression. TerseTS is written in Zig and unifies over a dozen compression methods under one API with C, Rust, Julia, and Python bindings, enabling the consistent use and comparison of these methods. An extensible architecture accommodates diverse algorithmic paradigms and exposes encoding and decoding primitives, enabling the creation of custom compression pipelines. Our demonstration has an interactive dashboard that supports data upload, visualization of reconstructed time series, and comparison of compression trade-offs across different methods. Participants can also design and evaluate custom compression pipelines and compare those with state-of-the-art methods.

Keywords

Time Series Compression, Compression Pipeline, Framework

1 Introduction

Time series are generated at an increasing rate by sensors in manufacturing, energy production, transportation, and IoT. These data streams are often transferred over bandwidth-constrained networks to centralized processing systems. The scale of such data makes compression imperative. However, choosing the right lossy or lossless compressor depends strongly on data characteristics, its intended downstream use, and the desired trade-off between decompression error and compression ratio [3, 5, 17].

Data-dependent Trade-offs: Figure 1 illustrates eight different lossy compression methods and their trade-offs regarding decompression error and compression ratio on two real-life datasets [7]. For Saugeenday, functional approximation methods such as PMC [14] and line simplification methods like VW [20] achieve an excellent trade-off, whereas for Oikolab the same methods perform poorly. This variability highlights that no single compressor universally outperforms the others and that evaluating multiple compression techniques is crucial for good results.

Current Solutions: Existing compression frameworks such as LibPressio [19] provide a common interface and multiple compressors, but target mainly scientific floating-point data with large-scale array storage and dense tensors. In contrast, most time series–specific methods, such as PMC [14], Slide [6], or NeaTS [9], exist only as standalone implementations in different languages, e.g., C++, Rust, or Java. This fragmentation makes their systematic comparison challenging.

Problem Complexity: Providing a unified framework that accommodates heterogeneous time series compression methods is inherently difficult. These algorithms span multiple paradigms, each grounded in distinct mathematical, geometric, and information theory principles. Functional approximation techniques [6, 12, 14] often rely on incremental convex-hull or polygon maintenance, slope intersection tracking, or a dynamic error-bounded fitting criteria. Value-representation methods [8, 15, 16] use low-level bit manipulation, statistical binning, or entropy coding schemes. Line simplification techniques [11, 18, 20] remove points based on geometric or statistically important characteristics. Meanwhile, domain-transformation techniques [1, 10] utilize orthogonal bases such as wavelet transforms and can be challenging to implement. Unifying this diversity of methods in a standardized framework with a single API requires substantial algorithmic and engineering effort, but creates significant value for both researchers and practitioners.

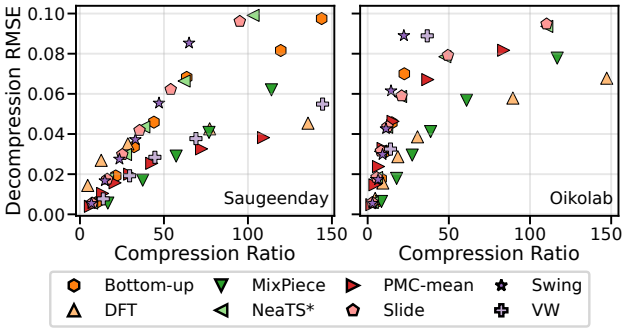


Figure 1: Decompression Error (RMSE) and Compression Ratio Trade-off for Time Series (Saugeenday, Oikolab)[1].

TerseTS Solution: TerseTS provides a unified, modular framework for time series compression and decompression and exposes all methods through the same API. It contains a heterogeneous collection of lossy and lossless methods, ranging from geometric, functional, and statistical techniques to bit-level encoders. TerseTS is implemented in the Zig language for high performance and simple installation. It currently has C, Rust, Julia, and Python bindings. The parameters of the compression methods (e.g., absolute or relative error bounds) are declared using a standardized JSON-based format. This design simplifies the API and integration of new methods with different parameters. Furthermore, TerseTS exposes the internal representation of the compressed series, allowing users to construct multi-stage compression pipelines dynamically, where the output of one stage (e.g., functional approximation) can be passed to subsequent ones (e.g., coefficient quantization or timestamp delta-encoding).

Contributions: This paper introduces TerseTS[2] together with an interactive dashboard that supports visual exploration of decompressed representations, compression trade-offs, and user-defined pipelines. The main technical contributions are:

[1] NeaTS* represents the lossy compression phase of NeaTS [9].
[2] TerseTS Open-Source Repository: https://github.com/cmcuza/TerseTS.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.74

Table 1: Summary of Compression Methods in TerseTS.

Category	Methods	Description
Functional Approximation	*PMC-mean/mid*, *Swing*, *Swing-disc*, *Slide*, *Convex-abc*, *SimPiece*, *MixPiece*, *NeaTS** [4, 6, 9, 13]	Approximates the time series using low-order piecewise functions with bounded error per segment. Supported basis functions include linear, exponential, square-root, quadratic, and power functions.
Line Simplification	*Bottom-up*, *Sliding-window*, *Visvalingam-Whyatt* [11, 20]	Reduces the time series to a subset of representative points while preserving geometric fidelity or bounded aggregate error (e.g., RMSE).
Value Representation	*RLE*, *Bit-QT*, *Serf-QT*, *PWLH*, *DELTA*, *PWCH* [2, 15, 16]	Encodes values compactly using quantization, histograms, dictionaries, or lossless encoding schemes.

- **A unified compression framework** integrating over a dozen lossy and lossless time series compression methods under a single, standardized, easy-to-use API that enables systematic benchmarking and experimentation.
- **An interactive dashboard** allowing users to upload their own data, and visualize its decompressed representation, key performance indicators, compression trade-offs, and design custom compression pipelines.

2 Framework Overview

In this section, we provide a brief overview of TerseTS in terms of its general architecture as well as its API and language bindings.

2.1 Framework Architecture

TerseTS is a lightweight open-source framework written in Zig that exposes a small C API for language bindings. It has no dependencies for simple installation. The API defines two main data structures, `UncompressedValues` (double arrays) and `CompressedValues` (byte arrays), and provides the `compress()` and `decompress()` functions that operate on them. In addition to `UncompressedValues`, `compress()` must be passed a method identifier and a JSON string specifying the method parameters. Beyond basic compression, TerseTS exposes two functions `extract()` and `rebuild()` that decode and reassemble the internal compressed representation of the methods, respectively.

Implementation and Portability: TerseTS is implemented in the Zig language due to its performance, portability, ease of installation, and seamless interoperability with C. Compared to C++, Zig provides more explicit memory management, a cross-platform build system, and simpler cross-compilation. Compared to Rust, Zig provides similar safety guarantees without a complex ownership model and simpler cross-compilation. Installing Zig only requires downloading a ZIP file, extracting it, and updating `PATH`. Thus, Zig is suited for building lightweight, dependency-free native libraries. Building TerseTS requires only running `zig build` in the repository. The bindings are designed to be easy to use for users of that language, i.e., a header for C, Julia `.jl` files for Julia, a Cargo crate for Rust, and a package for Python.

Implemented Methods: Table 1 summarizes the set of lossy and lossless compression algorithms implemented in TerseTS. The framework currently contains more than a dozen techniques covering the three families: functional approximation, line simplification, and value representation [18]. Each method is implemented from scratch in Zig, following the algorithmic definitions and descriptions in the original papers. Whenever possible, methods are decomposed into their fundamental compression primitives, enabling fine-grained comparison and flexible recombination at the logical level rather than as fixed compression pipelines. Future work will extend TerseTS with domain-transformation methods based on wavelet and Fourier bases [3]. Figure 2 illustrates the reconstructed time series produced by several representative lossy compressors, highlighting their qualitative differences even on these very short time series.

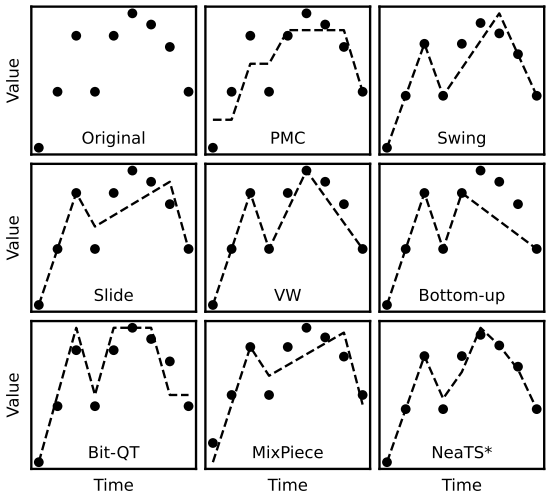


Figure 2: Decompressed Representation Obtained by Different Lossy Compressors Implemented in TerseTS.

Decoupled Logical and Physical Compression: Whenever a method's algorithmic core is capable of reducing a time series size, TerseTS explicitly separates the logical compression from the physical encoding. For instance, dictionary encoding could further compress any functional approximation method listed in Table 1, but embedding such encoders directly would restrict users from exploring alternative schemes better suited to their data. In these cases, TerseTS outputs a unified intermediate representation where each element is stored as an 8-byte value, enabling users to design pipelines with `extract()` and `rebuild()` that perform high-level structural compression and then physical encodings. This standardized interface promotes fair comparison across different methods and supports experimentation before committing to low-level optimizations. For methods inherently operating at the physical level, such as Serf-QT [16], TerseTS implements their complete compression pipeline.

2.2 API

TerseTS can currently be used from Zig, C, Rust, Julia, and Python, while providing the same API for all languages. Compression is performed by calling `compress()` with the uncompressed values, a method identifier, and method-specific parameters encoded as JSON strings. These parameters follow predefined configuration schemas that depend on the selected compression method, such as absolute or relative error bounds, histogram bin counts, aggregate error constraints, or other method-specific objectives. All configurations are parsed and validated by the framework, and invalid or incompatible parameters produce errors. Decompression is performed via `decompress()`. Memory is manually freed in Zig, ensuring safe cross-language operations. Listing 1 and Listing 2 show the C and Julia bindings in use.

```c
#include "tersets.h"
int main(void) {
  double data[3] = {2.0, 3.0, 2.5};
  struct UncompressedValues u = {data, 3};
```

```c
  struct CompressedValues c;
  struct UncompressedValues d;
  // Configuration for compression.
  enum Method method = SwingFilter;
  const char *cfg = "{\"abs_error_bound\":0.1}";
  // Compress and decompress the data.
  compress(u, &c, method, cfg);
  decompress(c, &d);
  // Free memory.
  freeCompressedValues(&c);
  freeUncompressedValues(&d);
  return 0;
}
```

Listing 1: Compression and Decompression in C.

```julia
include("TerseTS.jl")
data = [2.0, 3.0, 2.5]
# Configuration for compression.
method = TerseTS.SwingFilter
cfg = "{\"abs_error_bound\": 0.1}"
# Compress and decompress the data.
c = TerseTS.compress(data, method, cfg)
d = TerseTS.decompress(c)
```

Listing 2: Compression and Decompression in Julia.

Python API: The Python bindings are very similar to the Julia bindings due to the languages' similarity. As in Julia, memory management is automatic, in contrast to the explicit deallocation required by the C bindings. The Python bindings support zero-copy passing of `NumPy` arrays to Zig when possible, reducing overhead and improving execution time. Method parameters can be provided either as Python dictionaries or as JSON strings, using the same configuration keys as in the C and Julia bindings.

Rust API: The Rust bindings for `TerseTS` are provided as a Rust crate. Thus, they are built using `cargo` and can easily be added to other Rust projects. The build process integrates the Zig toolchain via a custom `build.rs` script. Memory management is automated through Rust's ownership and lifetime system. Thus, memory is automatically deallocated, unlike in Zig and C.

Compression Pipelines: `TerseTS` supports the composition of multiple compression stages into pipelines, allowing complex compression strategies to be built. Pipelines are constructed using the functions `extract()` and `rebuild()`, which expose and reassemble intermediate compressed representations. Each stage specifies a compression method together with its configuration and operates independently of the internal logic of the preceding stages, enabling fully modular pipeline design. Listing 3 shows a Python example that combines functional approximation, coefficient quantization, and run-length index encoding into a pipeline.

```python
import tersets
u = [2.0, 3.0, 2.5]
# Prepare compression pipeline configuration.
pipeline = [
    {"method": tersets.Method.SwingFilter,
    "configuration": {"abs_error_bound": 0.1}},
    {"method": tersets.Method.SerfQT,
    "configuration": {"abs_error_bound": 0.01}},
    {"method": tersets.Method.RunLengthEncoding,
    "configuration": {}}]
# Compress with the first method.
c1 = tersets.compress(u, **pipeline[0])
# Extract coefficients and indices.
idx, cfc = tersets.extract(c1)
# Compress both parts.
c2 = tersets.compress(cfc, **pipeline[1])
c3 = tersets.compress(idx, **pipeline[2])
# Compressed representation = c2 + c3.
# Decompression phase.
d_cfc = tersets.decompress(c2)
d_idx = tersets.decompress(c3)
d_c1 = tersets.rebuild(d_idx, d_cfc,
                       pipeline[0]["method"])
d = tersets.decompress(d_c1)
```

Listing 3: Custom Compression Pipeline in Python.

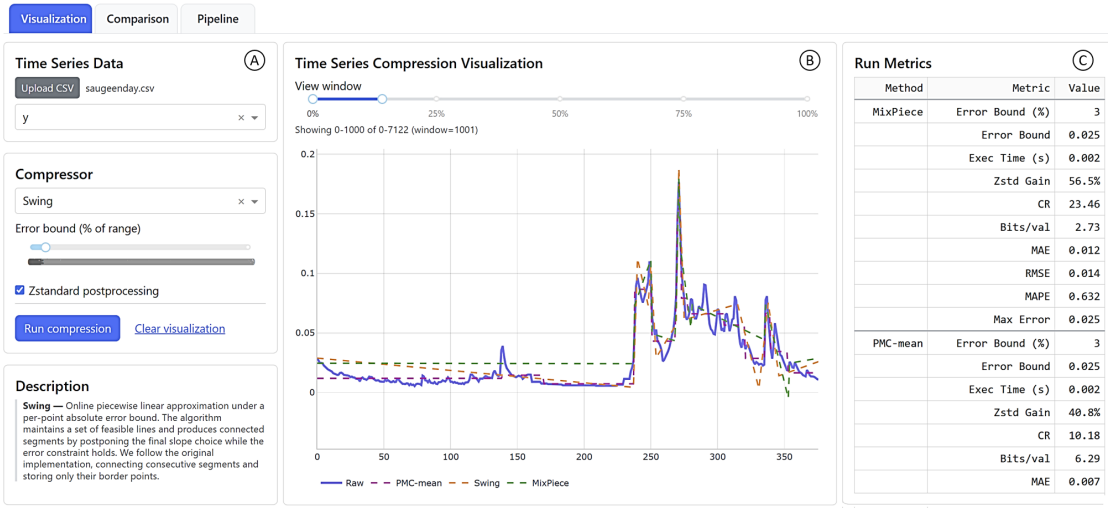
3 Demonstration

Our live demonstration features an interactive dashboard split into three major components: Visualization, Comparison, and Pipeline. Each component follows the same layout composed of three panels: (A) a *left panel* for uploading datasets (multiple will be provided), selecting compression methods, and adjusting parameters; (B) a *central panel* displaying the main visualization or analysis results; and (C) a *right panel* presenting detailed statistics and metrics related to the compression methods. Figure 3 illustrates a running example of the dashboard layout with the Visualization component active and the *central panels* of the Comparison and Pipeline components.

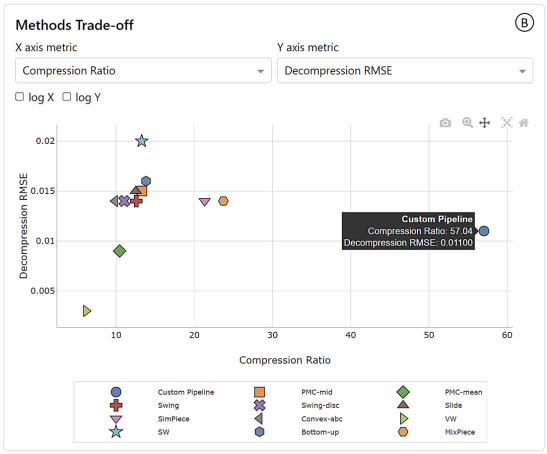
Visualization Component: This component allows inspection of decompressed time series. In (A), users can select a compression method, set the error bound (as a percentage of the data range), and optionally enable Zstandard postprocessing. In (B), users can then visualize the reconstructed series alongside the original one. Figure 3a shows an example using the first 1,000 points of the Saugeenday dataset [7] compressed with Swing [6], PMC-mean [14], and MixPiece [13] with the same error bound. Complementing the visualization, (C) reports key metrics such as execution time, decompression error, and compression ratio for a quantitative comparison.

Comparison Component: This component aggregates results from multiple runs and configurations across all selected compression methods. Users can choose which metrics to compare on the X and Y axes (e.g., RMSE vs. compression ratio or execution time vs. bits-per-value) and switch between linear or logarithmic scales. Interactive markers and legends make it easy to identify the trade-offs between methods and identify Pareto-optimal compressors under different error bounds. Figure 3b shows the (B) panel visualizing a comparison between implemented methods and a user-defined custom pipeline in terms of compression ratio (X-axis) and decompression RMSE (Y-axis).

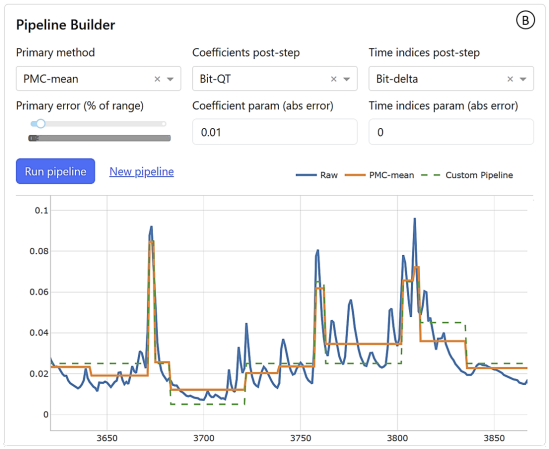
Pipeline Component: This component allows participants to evaluate custom multi-stage compression pipelines. Pipelines consist of a primary compressor followed by optional post-processing steps for encoding coefficients and indices. Figure 3c shows panel (B) with an example of a compression pipeline consisting of PMC-mean as the main logical compressor, followed by a Bit-Packed Quantization (Bit-QT) of coefficients and a Bit-Packed Delta Encoding (Bit-delta) of indices. Users can adjust parameters for each stage, run the full pipeline, and visualize the resulting reconstruction and performance metrics side by side with existing methods. A comparison table also reports key performance indicators like execution time and the contribution of each compressor of the pipeline.



(a) TerseTS Demonstration Dashboard Overview with the Visualization Component Active.



(b) Central Panel of the Comparison Component.



(c) Central Panel of the Pipeline Component.

Figure 3: Overview of the demonstration dashboard. Participants interact with TerseTS to (a) visualize different compression methods, (b) compare their trade-offs on selected metrics, and (c) construct custom compression pipelines.

Acknowledgments

We gratefully acknowledge funding from the German Federal Ministry of Research, Technology and Space (under grant BIFOLD25B).

References

[1] Mohammed Abo-Zahhad. 2011. ECG signal compression using discrete wavelet transform. *Discrete wavelet transforms-theory and application* (2011).
[2] Chiranjeeb Buragohain, Nisheeth Shrivastava, and Subhash Suri. 2007. Space Efficient Streaming Algorithms for the Maximum Error Histogram. In *ICDE*. 1026–1035. doi:10.1109/ICDE.2007.368961
[3] Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *ACM Comput. Surv.* 55, 10 (2023). doi:10.1145/3560814
[4] M. Dalai and R. Leonardi. 2006. Approximations of One-Dimensional Digital Signals Under the L_{infty} Norm. *IEEE Trans. Signal Process.* 54, 8 (2006), 3111–3124. doi:10.1109/TSP.2006.875394
[5] Sheng Di and et. al. 2025. A Survey on Error-Bounded Lossy Compression for Scientific Datasets. *ACM Comput. Surv.* 57, 11 (2025), 287:1–287:38. https://doi.org/10.1145/3733104
[6] Hazem Elmeleegy et al. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *PVLDB* 2, 1 (2009), 145–156. doi:10.14778/1687627.1687645
[7] Rakshitha Godahewa and et. al. 2021. Monash Time Series Forecasting Archive. In *NeurIPS*, Vol. 1. https://datasets-benchmarks-proceedings.neurips.cc/paper/2021.
[8] R.M. Gray and D.L. Neuhoff. 1998. Quantization. *Trans. on Inf. Theory* (1998).
[9] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series with Random Access. In *ICDE*. 1579–1592. doi:10.1109/ICDE65448.2025.00122
[10] S Edward Hawkins III and Edward Hugo Darlington. 2012. *Algorithm for compressing time-series data.* Technical Report. NASA Tech Briefs.
[11] E. Keogh, S. Chu, D. Hart, and M. Pazzani. 2001. An online algorithm for segmenting time series. In *ICDM*. 289–296. doi:10.1109/ICDM.2001.989531
[12] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-Piece: Highly Accurate Piecewise Linear Approximation through Similar Segment Merging. In *VLDB*, Vol. 16. 1910–1922. doi:10.14778/3594512.3594521
[13] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2024. Flexible grouping of linear segments for highly accurate lossy compression of time series data. *VLDB J.* 33, 5 (2024). doi:10.1007/S00778-024-00862-Z
[14] I. Lazaridis and S. Mehrotra. 2003. Capturing sensor-generated time series with quality guarantees. In *ICDE*. 429–440. doi:10.1109/ICDE.2003.1260811
[15] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp.* 46, 6 (2016).
[16] Ruiyuan Li and et. al. 2025. *Serf*: Streaming Error-Bounded Floating-Point Compression. *SIGMOD* 3, 3 (2025), 216:1–216:27. doi:10.1145/3725353
[17] Carlos Enrique Muñiz-Cuza and et. al. 2024. Evaluating the Impact of Error-Bounded Lossy Compression on Time Series Forecasting. In *EDBT*. OpenProceedings.org, 650–663. doi:10.48786/EDBT.2024.56
[18] Carlos Enrique Muñiz-Cuza, Matthias Boehm, and Torben Bach Pedersen. 2026. CAMEO: Autocorrelation-Preserving Line Simplification for Lossy Time Series Compression. In *EDBT*. OpenProceedings.org, 15–28. doi:10.48786/EDBT.2026.02
[19] Robert Underwood, Victoriana Malvoso, Jon C Calhoun, Sheng Di, and Franck Cappello. 2021. Productive and Performant Generic Lossy Data Compression with LibPressio. In *DRBSD@SC*. doi:10.1109/ICDE65448.2025.00122
[20] M. Visvalingam and J. D. Whyatt. 1993. Line generalisation by repeated elimination of points. *The Cartographic Journal* 30, 1 (1993), 46–51. doi:10.1179/000870493786962263