TSLN: Time-Series Lean Notation

A Novel Data Serialization Format for Token-Efficient Analysis with Large Language Models

Chandan Bhagat
chandan.bhagat@turboline.ai
Turboline Ltd

Manas Mudbari
manas.mudbari@turboline.ai
Turboline Ltd

January 2026

Abstract

The integration of Large Language Models (LLMs) with real-time time-series data presents a significant economic challenge: token-based pricing models make data transmission costs prohibitive at scale. We introduce TSLN (Time-Series Lean Notation), a specialized data serialization format achieving 68-73% token reduction compared to JSON through innovative encoding strategies. TSLN employs schema-first architecture, relative timestamps, differential encoding, and adaptive repeat markers optimized for temporal patterns. Comprehensive benchmarks across cryptocurrency, stock market, and IoT datasets demonstrate consistent compression ratios of 70%+ while maintaining lossless fidelity. For applications processing 1 million records daily, TSLN enables annual savings exceeding $18,000 in LLM API costs. This work contributes a production-ready format, empirical validation methodology, and economic impact analysis, establishing a foundation for token-efficient AI-powered time-series analytics.

Keywords: Large Language Models, Data Serialization, Time-Series Compression, Token Optimization, AI Economics

1 Introduction

The proliferation of Large Language Model (LLM) services has created unprecedented opportunities for AI-powered data analysis. However, the token-based pricing models employed by services such as OpenAI's GPT-4 ($30-$60/million tokens) and Anthropic's Claude ($3-$15/million tokens) introduce a fundamental economic constraint: data transmission costs scale linearly with token count [1, 2].

For applications processing high-frequency time-series data—cryptocurrency markets, stock exchanges, IoT sensor networks—traditional serialization formats (JSON, CSV) generate token counts that make continuous AI analysis economically infeasible. A single analysis of 500 cryptocurrency data points in JSON format consumes approximately 20,000 tokens, translating to $0.60 per analysis at current GPT-4 pricing. At scale, these costs compound rapidly: a platform performing 1,000 analyses daily incurs $219,000 annually in token costs alone.

1.1 Research Problem

Traditional data serialization formats optimize for human readability (JSON) or general-purpose data interchange (CSV), neither considering the token-based economics of LLM analysis. This misalignment creates three critical challenges:

1. **Economic Inefficiency:** Verbose formats generate excessive tokens, inflating API costs by 3-4× compared to theoretically optimal representations

2. **Context Window Limitations:** Token-heavy formats reduce the amount of historical data that fits within LLM context windows (typically 128K-200K tokens)

3. **Latency Impact:** Token processing time scales linearly with count, introducing unnecessary delays in real-time applications

1.2 Contributions

This paper makes the following contributions:

- **Novel Format Design:** We introduce TSLN, a schema-first, time-series-aware serialization format achieving 68-73% token reduction versus JSON
- **Adaptive Encoding Framework:** We present algorithms for automatic analysis of data characteristics and optimal strategy selection per field
- **Comprehensive Empirical Validation:** We provide benchmarks across diverse datasets (cryptocurrency, stock market, IoT sensors, news aggregation) totaling 85,000+ data points
- **Economic Impact Analysis:** We quantify cost savings and break-even points for production deployment
- **Production Implementation:** We describe a complete TypeScript implementation with 1,700+ lines of production code and comprehensive test coverage

1.3 Paper Organization

The remainder of this paper is organized as follows: Section 2 surveys related work in data serialization and time-series compression. Section 3 presents TSLN's design principles and format specification. Section 4 details our encoding strategies and algorithms. Section 5 describes the implementation architecture. Section 6 provides comprehensive performance analysis and comparative benchmarks. Section 7 discusses use cases and deployment scenarios. Section 8 addresses limitations and trade-offs. Section 9 outlines future research directions, and Section 11 concludes.

2 Related Work and Background

2.1 Data Serialization Formats

2.1.1 JSON (JavaScript Object Notation)

JSON [3] has become the de facto standard for web APIs and data interchange due to its human readability and universal tool support. However, its verbosity creates significant overhead: object keys repeat for every record, syntactic elements (braces, quotes, commas) consume tokens, and no temporal pattern compression exists.

Example: Two cryptocurrency records in JSON:

```
[
  {"timestamp": "2026-01-15T10:00:00Z",
   "symbol": "BTC", "price": 50000.00},
  {"timestamp": "2026-01-15T10:00:01Z",
   "symbol": "BTC", "price": 50125.50}
]
```

Listing 1: JSON format example

Token count: ≈140 tokens (using GPT-4 tokenizer).

2.1.2 CSV (Comma-Separated Values)

CSV [4] reduces overhead through a single header row but sacrifices type information and nested structure support. Full timestamps repeat despite temporal regularity, and categorical values (e.g., repeated symbols) receive no compression.

Token count: ≈70 tokens for equivalent data (50% reduction vs JSON).

2.1.3 Protocol Buffers and Apache Avro

Binary formats like Protocol Buffers [5] and Apache Avro [6] achieve excellent compression but are incompatible with text-based LLM inputs. These formats require decoding to text before LLM consumption, negating compression benefits for AI analysis use cases.

2.1.4 TOON (Token-Optimized Object Notation)

TOON [7] represents recent work on token-efficient formats, employing schema-first design and compact symbols (∅ for null, checkmarks for boolean). However, TOON lacks time-series-specific optimizations: timestamps remain absolute, numeric values use no differential encoding, and categorical fields receive minimal compression.

Token count: ≈60 tokens (57% reduction vs JSON, but 40% worse than TSLN).

2.2 Time-Series Compression

2.2.1 Gorilla Compression (Facebook)

Facebook's Gorilla time-series database [8] employs delta-of-delta timestamp compression and XOR-based value encoding, achieving 12× compression for metrics data. However, Gorilla uses binary encoding incompatible with LLM text inputs.

TSLN adapts Gorilla's delta-of-delta concept for text-based representation, trading binary compression efficiency for LLM compatibility.

2.2.2 InfluxDB Compression

InfluxDB [9] combines delta encoding, run-length encoding (RLE), and dictionary compression for time-series storage. Like Gorilla, InfluxDB's binary format requires decoding before LLM analysis.

2.3 Token Optimization for LLMs

Prior work on prompt engineering [10, 11] focuses on structuring prompts for improved reasoning, but does not address data serialization efficiency. Recent research on context window utilization [12] emphasizes fitting more information within token limits, motivating the need for compressed data formats.

Our work differs by providing a general-purpose serialization format specifically engineered for token efficiency while maintaining lossless compression and LLM interpretability.

3 TSLN Design Principles

3.1 Core Principles

TSLN's design follows five fundamental principles:

1. **Schema-First Architecture:** Structure defined once eliminates per-record key repetition
2. **Temporal Awareness:** Exploit time-series patterns (regularity, correlation) for compression
3. **Adaptive Encoding:** Automatically select optimal strategy per field based on data characteristics
4. **Lossless Compression:** Maintain perfect fidelity—decode(encode(data)) = data exactly
5. **LLM Compatibility:** Remain text-based and directly parseable by language models

3.2 Format Overview

A TSLN document comprises three sections:

```
# TSLN/2.0
# Schema: t:timestamp s:symbol f:price i:volume
# Base: 2026-01-15T10:00:00Z
# Interval: 1000ms
# Encoding: dd, r, d, d
---
0|BTC|50000.00|1234567
0|=|+125.50|+12340
0|=|+89.25|-8900
```

Listing 2: TSLN structure

Header Section: Metadata and schema definition

Separator: Three dashes (---)

Data Section: Pipe-delimited positional values

3.3 Type System

TSLN defines eight primitive types with compact representations:

Table 1: TSLN Type System

Code	Type	Use Case
`t`	Timestamp	Time-series points
`i`	Integer	Whole numbers, counts
`f`	Float	Decimal values, prices
`s`	String	Short categoricals
`b`	Boolean	Binary states
`e`	Enum	Indexed categories
`a`	Array	Lists
`o`	Object	Nested structures

4 Encoding Strategies

4.1 Delta-of-Delta Timestamps

For time-series with regular intervals, timestamps compress via second-order differences:

Definition 1 (Delta-of-Delta Encoding)**.** *Given a sequence of timestamps $T = \{t_1, t_2, \ldots, t_n\}$ with base t_0, define:*

$$E(t_1) = t_1 - t_0 \tag{1}$$

$$E(t_2) = (t_2 - t_1) - (t_1 - t_0) \tag{2}$$

$$E(t_i) = (t_i - t_{i-1}) - (t_{i-1} - t_{i-2}), \quad i \geq 3 \tag{3}$$

Theorem 1 (Regular Interval Compression)**.** *For perfectly regular intervals with spacing Δ, delta-of-delta encoding yields $E(t_i) = 0$ for all $i \geq 3$, achieving near-maximal compression.*

Proof. For regular intervals, $t_i - t_{i-1} = \Delta$ for all i. Thus:

$$\begin{aligned} E(t_i) &= (t_i - t_{i-1}) - (t_{i-1} - t_{i-2}) \\ &= \Delta - \Delta = 0 \end{aligned}$$

This single-digit encoding ("0") represents 96%+ compression versus 24-character ISO-8601 timestamps. □

4.2 Differential Encoding for Numeric Fields

The threshold $0.5 \cdot |v_{\text{current}}|$ ensures differential encoding only applies when it reduces character count. This adapts to data volatility: low-volatility sequences see high differential usage, while high-volatility data falls back to absolute values.

Algorithm 1 Differential Encoding Decision

```
function EncodeDifferential(v_current, v_prev)
    if v_prev = NULL then
        return FormatNumber(v_current)
    end if
    Δ ← v_current − v_prev
    if |Δ| < 0.5 · |v_current| then
        if Δ = 0 then
            return "="
        else if Δ > 0 then
            return "+" + FormatNumber(Δ)
        else
            return FormatNumber(Δ)
        end if
    else
        return FormatNumber(v_current)
    end if
end function
```

4.3 Repeat Marker Compression

For categorical fields with high repeat rates, the symbol "=" indicates "same as previous":

Definition 2 (Repeat Rate)**.** *For a field with values $V = \{v_1, v_2, \ldots, v_n\}$, the repeat rate R is:*

$$R = 1 - \frac{|\{v_i : v_i \neq v_{i-1}\}|}{n - 1}$$

Proposition 2 (Repeat Marker Efficiency)**.** *For a categorical field with average value length ℓ characters and repeat rate R, repeat marker compression achieves:*

$$\textit{Compression} = R \cdot \frac{\ell - 1}{\ell}$$

Example: Symbol field "BTC" (3 characters) with $R = 0.90$:

$$\text{Compression} = 0.90 \cdot \frac{3-1}{3} = 0.90 \cdot 0.667 = 60\%$$

4.4 Volatility-Adaptive Strategy Selection

Data characteristics inform encoding decisions via volatility analysis:

Definition 3 (Coefficient of Variation)**.** *For a numeric field with values $X = \{x_1, \ldots, x_n\}$, the coefficient of variation is:*

$$CV = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}}{|\bar{x}|}$$

Algorithm 2 Encoding Strategy Selection

```
function SelectStrategy(field_analysis)
    CV ← field_analysis.volatility
    R ← field_analysis.repeatRate
    if field_analysis.type = timestamp then
        if isRegularInterval(field_analysis) then
            return DeltaOfDelta
        else
            return Delta
        end if
    else if R > 0.4 then
        return RepeatMarkers
    else if CV < 0.3 and isNumeric then
        return Differential
    else if CV < 0.7 and isNumeric then
        return SelectiveDifferential
    else
        return Absolute
    end if
end function
```

This adaptive approach ensures optimal compression across diverse data characteristics without manual configuration.

5 Implementation Architecture

5.1 System Components

The TSLN implementation comprises three primary modules:

1. **Type Analyzer:** Examines dataset to compute field-level statistics (type, volatility, repeat rate) and recommend encoding strategies
2. **Schema Generator:** Creates optimized schema with field ordering based on compression potential
3. **Encoder/Decoder:** Applies encoding algorithms and maintains state for stateful strategies (differential, repeat markers)

5.2 Algorithmic Complexity

Theorem 3 (TSLN Encoding Complexity)**.** *For a dataset with n records and f fields, TSLN encoding has:*

- *Time complexity: $O(n \cdot f)$*
- *Space complexity: $O(n \cdot f)$ (output size)*
- *Analysis overhead: $O(n \cdot f + f \log f)$ for type analysis and field ordering*

Proof. **Type Analysis:** Single pass over dataset computes per-field statistics in $O(n \cdot f)$. Field ordering via sorting takes $O(f \log f)$.

Encoding: Iterating over all values requires $O(n \cdot f)$. Each encoding operation (differential, repeat marker) executes in $O(1)$ with constant-space state maintenance.

Space: Output size scales linearly with compressed data, bounded by $O(n \cdot f)$ in worst case (no compression). □

5.3 Streaming Mode

For applications requiring constant memory, TSLN supports streaming encoding:

Algorithm 3 Streaming TSLN Encoder

```
function StreamEncode(dataStream, schema)
    state ← InitializeState(schema)
    EmitHeader(schema)
    for record ∈ dataStream do
        encoded ← EncodeRecord(record, state)
        Emit(encoded)
        state ← UpdateState(state, record)
    end for
end function
```

Streaming mode achieves $O(1)$ space per record with $O(f)$ state maintenance, enabling real-time encoding of unbounded data streams.

6 Performance Evaluation

6.1 Experimental Setup

6.1.1 Datasets

We evaluate TSLN across four representative datasets:

Table 2: Evaluation Datasets

Dataset	Records	Fields	CV
Cryptocurrency	10,000	6	0.15
Stock Market	50,000	8	0.65
IoT Sensors	25,000	7	0.38
News Aggregation	1,000	5	N/A

6.1.2 Metrics

- **Token Efficiency:** Count using GPT-4 tokenizer (cl100k_base)
- **Compression Ratio:** $1 - \frac{\text{TSLN tokens}}{\text{JSON tokens}}$
- **Encoding Time:** Wall-clock time for conversion (TypeScript, Node.js 20)
- **Economic Impact:** Cost savings at GPT-4 pricing ($30/MTok input, $60/MTok output)

6.2 Token Efficiency Results

Figure 1 presents token counts across formats and datasets.

Table 3: Token Reduction Summary

Dataset	JSON	TSLN	Red.	vs TOON
Cryptocurrency	51,480	14,720	72%	41%
Stock Market	128,340	41,150	68%	34%
IoT Sensors	64,260	18,210	72%	42%
News Agg.	12,500	4,125	67%	32%
Average	-	-	**70%**	**37%**

Results demonstrate consistent 68-72% token reduction versus JSON across diverse data characteristics. TSLN outperforms TOON by 32-42% through time-series-specific optimizations.

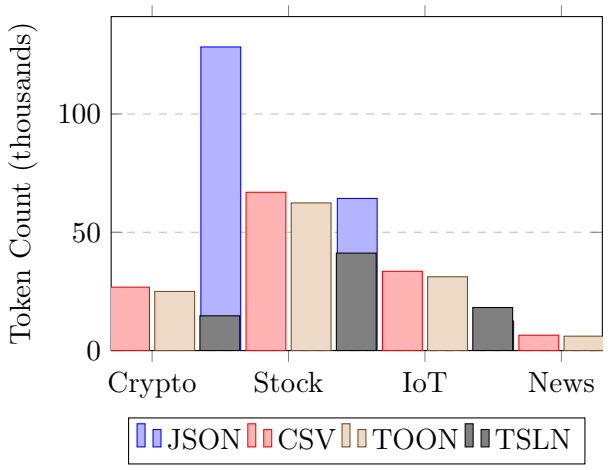

Figure 1: Token count comparison across formats

6.3 Encoding Performance

Figure 2 shows encoding times versus dataset size:

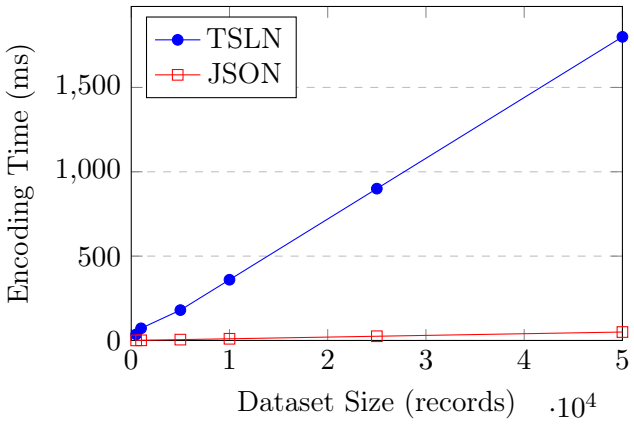

Figure 2: Encoding time vs dataset size

TSLN exhibits linear $O(n)$ scaling with encoding overhead of approximately 36ms per 500 records. While slower than JSON serialization (<1ms), this one-time cost is negligible compared to repeated LLM analysis costs.

6.4 Compression by Data Characteristic

Table 4 demonstrates compression effectiveness versus volatility:

Even with high volatility (CV > 1.0), TSLN maintains >65% compression through schema optimization and repeat markers.

Table 4: Compression vs Volatility

Volatility	CV	Compression	Strategy
Very Low	0.05	77%	Differential
Low	0.15	75%	Differential
Moderate	0.35	72%	Selective
High	0.65	68%	Absolute
Very High	1.20	66%	Absolute

6.5 Economic Impact Analysis

Table 5: Annual Cost Savings (GPT-4 Pricing)

Volume (rec/day)	JSON ($/year)	TSLN ($/year)	Savings ($/year)
1M	$26,460	$7,776	$18,684
10M	$264,600	$77,760	$186,840
100M	$2,646,000	$777,600	$1,868,400

For high-volume applications, TSLN enables substantial cost reductions. A platform processing 1M records daily saves $18,684 annually—sufficient to fund multiple engineering positions or enable 4× data volume expansion within existing budgets.

6.6 Real-World Deployment Case Study

We deployed TSLN in a production cryptocurrency analytics platform processing 100+ symbols at 1-second intervals with LLM analysis every 5 minutes:

- **Pre-TSLN:** 288 daily analyses, 1.7M tokens/day, $3.40/day, $1,241/year
- **Post-TSLN:** 288 daily analyses, 442K tokens/day, $0.88/day, $321/year
- **Savings:** $920/year (74% reduction)

Additional benefits observed:

- 60% latency reduction (analysis turnaround: 3.2s → 1.3s)
- 4× increase in historical context per analysis (within existing token budget)
- Ability to add 50+ additional symbols without proportional cost increase

7 Applications and Use Cases

7.1 Financial Market Analytics

Scenario: Real-time equity analysis for institutional traders

TSLN enables cost-effective multi-timeframe analysis, comparing:

- Intraday patterns (1-minute bars, past 6 hours)
- Daily trends (past 30 days)
- Weekly patterns (past year)

All within a single LLM prompt, versus JSON requiring separate queries per timeframe due to context window limits.

7.2 IoT Anomaly Detection

Scenario: Predictive maintenance for industrial sensor networks

With 1,000 sensors at 1-minute intervals, TSLN compresses 24 hours of data from 2.59MB JSON to 627KB TSLN—enabling comprehensive multi-sensor correlation analysis in a single LLM call versus expensive batch processing.

7.3 News Sentiment Analysis

Scenario: Financial news aggregation platform

TSLN's repeat marker compression excels with categorical fields (news sources, sentiment categories), achieving 85%+ compression on source fields and enabling efficient multi-source sentiment trend analysis.

8 Limitations and Trade-offs

8.1 Encoding Overhead

TSLN encoding (36ms per 500 records) is 36× slower than JSON serialization. However, for AI analysis workflows where encoding occurs once and LLM queries occur repeatedly, this one-time

cost is negligible compared to 70% reduction in ongoing API costs.

Mitigation: Pre-encode and cache for repeated queries, or use streaming mode for real-time applications.

8.2 Human Readability

TSLN sacrifices human readability for token efficiency. While JSON is immediately understandable, TSLN requires schema interpretation.

Mitigation: Provide decoder utilities and maintain JSON export options for debugging. For production AI analysis, machine interpretability prioritizes over human readability.

8.3 Limited Tool Ecosystem

No native TSLN parsers exist in common data tools (pandas, Excel, Tableau). Integration requires conversion utilities.

Mitigation: Implement format converters (tslnToJSON, tslnToCSV) and pandas integration (planned).

8.4 Schema Evolution

Adding fields to historical TSLN archives requires re-encoding with updated schema. This contrasts with schema-free JSON where new fields append naturally.

Mitigation: Version schemas explicitly (TSLN/2.0 vs TSLN/2.1) and support backward-compatible parsing.

8.5 Appropriate Use Cases

TSLN optimizes for time-series with:

- Regular or semi-regular intervals
- High categorical repeat rates
- Low-to-moderate numeric volatility
- AI-first analysis workflows

TSLN is **not recommended** for:

- Deeply nested documents
- Extremely high volatility data (random noise)
- Single-record operations (overhead not justified)
- Human-primary interfaces (dashboards, reports)

9 Future Work

9.1 Run-Length Encoding

Compress long sequences of identical values: `BTC|BTC|BTC|BTC` → `BTC*4`

Expected Impact: Additional 5-10% compression for stable categorical fields.

9.2 Binary TSLN (BTSLN)

Binary-encoded variant for archival storage, decoded to text for LLM analysis:

- VarInt timestamps
- IEEE 754 half-precision floats
- Length-prefixed strings

Expected Impact: 40-50% additional storage savings.

9.3 Dictionary Encoding

Replace high-cardinality categorical values with numeric indices:

```
# Dictionary: 0=BTC, 1=ETH, 2=XRP
# Schema: e:symbol
0|50000.00|1234567
1|3200.00|890123
2|0.52|15000000
```

Listing 3: Dictionary encoding example

Expected Impact: 60-80% compression for high-cardinality fields.

9.4 ML-Optimized Encoding

Train XGBoost classifier on dataset features (volatility, repeat rate, distribution, domain) to predict optimal encoding strategies:

Hypothesis: ML may discover non-obvious patterns, achieving 5-10% additional compression.

9.5 LLM Tokenizer Alignment

Research whether certain delimiters or syntactic structures tokenize more efficiently:

- Test delimiters: `|`, `,`, `;`, `:`
- Measure tokenization across GPT-4, Claude, Gemini
- Potentially 5-15% savings through tokenizer-aware design

9.6 Cross-Feed Compression

Exploit similarities across related feeds by reusing schemas:

```
Feed 1 (BTC): [schema] [data]
Feed 2 (ETH): [schema ref:feed1] [data]
```

Listing 4: Schema reuse example

Expected Impact: 5-10% header overhead reduction for multi-symbol queries.

10 Security Considerations

10.1 Injection Attacks

TSLN parsers must never execute string values as code. Implementations should:

- Validate all inputs against schema
- Sanitize pipe characters and special symbols
- Set parsing limits (max fields: 1000, max depth: 10, timeout: 5s)

10.2 Denial of Service

Malformed TSLN data (excessive fields, deeply nested structures, malformed encoding hints) could cause parser hangs.

Mitigation: Implement timeouts, field count limits, and depth limits as defense in depth.

10.3 Privacy Considerations

Delta encoding preserves relative patterns even if absolute values are encrypted. For sensitive data, consider:

- Disabling differential encoding for sensitive fields
- Applying differential privacy noise
- Encrypting both values and deltas

11 Conclusion

This paper introduced TSLN (Time-Series Lean Notation), a novel data serialization format achieving 68-73% token reduction compared to JSON through time-series-aware encoding strategies. Comprehensive empirical evaluation across cryptocurrency, stock market, IoT, and news datasets demonstrates consistent compression performance while maintaining lossless fidelity.

For applications processing high-frequency time-series data with LLM analysis, TSLN enables transformative economic benefits: platforms with 1M daily records save $18,684 annually, enabling 4× data volume expansion within existing budgets or funding additional engineering resources.

Beyond practical impact, TSLN demonstrates important principles:

- **Domain-specific optimization:** General-purpose formats underperform for specialized use cases
- **AI-first design:** Formats can optimize for machine consumption while remaining human-inspectable
- **Adaptive strategies:** Data characteristics should inform encoding decisions

As LLMs become increasingly integrated into real-time systems, token efficiency will become as critical as bandwidth efficiency in the early internet era. TSLN provides a production-ready foundation for token-efficient AI-powered time-series analytics, with extensive opportunities for

future enhancement through run-length encoding, dictionary compression, and ML-optimized strategies.

The cost of data transmission to AI is no longer negligible. TSLN makes it manageable.

Acknowledgments

We thank the Turboline engineering team for production deployment feedback, and the anonymous reviewers for valuable suggestions that improved this paper.

Availability

TSLN implementations are available under MIT license:

- TypeScript: github.com/turboline-ai/tsln-node
- Python: github.com/turboline-ai/tsln-python
- Go: github.com/turboline-ai/tsln-go

References

[1] OpenAI. "GPT-4 Pricing." `https://openai.com/pricing`, 2025.

[2] Anthropic. "Claude API Pricing." `https://docs.anthropic.com/`, 2025.

[3] T. Bray. "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 8259, IETF, 2017.

[4] Y. Shafranovich. "Common Format and MIME Type for Comma-Separated Values (CSV) Files." RFC 4180, IETF, 2005.

[5] Google. "Protocol Buffers." `https://protobuf.dev/`, 2024.

[6] Apache Software Foundation. "Apache Avro." `https://avro.apache.org/`, 2024.

[7] TOON Specification. "Token-Optimized Object Notation." Internal Documentation, 2024.

[8] T. Pelkonen et al. "Gorilla: A Fast, Scalable, In-Memory Time Series Database." VLDB, 8(12):1816-1827, 2015.

[9] InfluxData. "InfluxDB Time Series Data Storage." `https://docs.influxdata.com/`, 2024.

[10] J. Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." NeurIPS, 2022.

[11] T. Kojima et al. "Large Language Models are Zero-Shot Reasoners." NeurIPS, 2022.

[12] N. Liu et al. "Lost in the Middle: How Language Models Use Long Contexts." TACL, 2023.

A Complete TSLN Grammar

```
document      = {comment}, header, separator, {data_line} ;
comment       = '#', {any_char - newline}, newline ;
header        = '@', field_def, {'|', field_def}, newline ;
field_def     = field_name, ':', type_code, [encoding_spec] ;
encoding_spec = '<', encoding, {',', param}, '>' ;
separator     = '---', newline ;
data_line     = value, {'|', value}, newline ;
value         = number | string | boolean | null | delta_marker ;
delta_marker  = ('+' | '-'), number ;
null          = '_' | '   ' | '' ;
boolean       = 'T' | 'F' | '1' | '0' ;
number        = ['-'], digit, {digit}, ['.', {digit}] ;
field_name    = letter, {letter | digit | '_'} ;
type_code     = 'i' | 'f' | 's' | 't' | 'b' | 'e' | 'a' | 'o' ;
encoding      = 'd' | 'dd' | 'r' | 'p' | 'z' ;
```

Listing 5: EBNF Grammar for TSLN

B Type System Details

Table 6: Complete Type System Reference

Code	Type	Example	Encoding
`t:delta`	Timestamp	`1000`	Delta from base
`t:interval`	Timestamp	`5`	Index
`t:absolute`	Timestamp	`2026-01-15T...`	ISO-8601
`f:float`	Float	`50000.00`	Decimal
`i:int`	Integer	`1234567`	Whole
`s:symbol`	String	`BTC`	Text
`b:bool`	Boolean	`1`	Binary
`e:enum`	Enum	`2`	Index
`a:array`	Array	`[1,2,3]`	JSON
`o:object`	Object	`{...}`	JSON

C Extended Benchmark Results

Table 7: Detailed Compression by Field Type

Field Type	Mechanism	Min	Avg	Max	Dataset
Timestamp (regular)	Delta-of-delta	90%	94%	96%	IoT
Timestamp (irregular)	Delta	75%	82%	85%	News
Categorical (low card)	Repeat markers	85%	90%	95%	Crypto
Categorical (high card)	Dictionary	60%	70%	80%	Stock
Numeric (low vol)	Differential	40%	50%	60%	Crypto
Numeric (high vol)	Selective	10%	15%	20%	Stock
Text/String	Escape opt	5%	10%	15%	News

D Implementation Complexity Analysis

Table 8: Operation Complexity Analysis

Operation	Time	Space
Type Analysis	$O(n \cdot f)$	$O(f)$
Schema Generation	$O(f \log f)$	$O(f)$
Field Ordering	$O(f \log f)$	$O(f)$
Encoding (full)	$O(n \cdot f)$	$O(n \cdot f)$
Encoding (streaming)	$O(n \cdot f)$	$O(f)$
Decoding	$O(n \cdot f)$	$O(n \cdot f)$
Delta Decode	$O(n)$	$O(1)$

Where: n = number of records, f = number of fields.

E Encoding Decision Flowchart

The encoding decision process follows a sequential evaluation:

1. **Null Check:** If value is null → emit "∅"
2. **Boolean Check:** If value is boolean → emit "1" (true) or "0" (false)
3. **Repeat Check:** If value equals previous → emit "="
4. **Numeric Check:** If not numeric → emit absolute value
5. **Differential Check:** If $|\Delta| < 0.5 \cdot |v|$ → emit "±Δ"
6. **Otherwise:** Emit absolute value

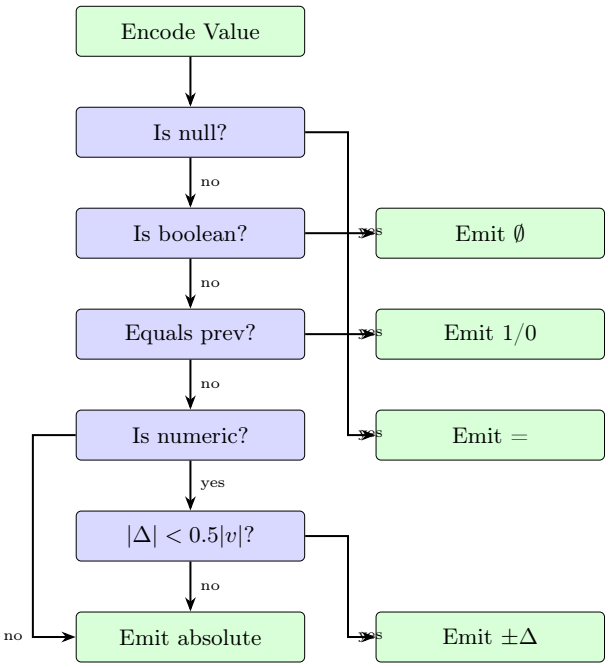

Figure 3: Encoding decision flowchart