

JTON: A Token-Efficient JSON Superset with Zen Grid Tabular Encoding for Large Language Models

Gowthamkumar Nandakishore
github.com/gowthamkumar-nandakishore/JTON

March 2026

Abstract

When LLMs process structured data, the serialization format directly affects cost and context utilization. Standard JSON wastes tokens repeating key names in every row of a tabular array—overhead that scales linearly with row count. This paper presents **JTON** (JSON Tabular Object Notation), a strict JSON superset whose main idea, **Zen Grid**, factors column headers into a single row and encodes values with semicolons, preserving JSON’s type system while cutting redundancy. Across seven real-world domains, Zen Grid reduces token counts by **15–60%** versus JSON compact (28.5% average; 32% with `bare_strings`).

Comprehension tests on **10 LLMs** show a net +0.3 pp accuracy gain over JSON: four models improve, three hold steady, and three dip slightly. Generation tests on **12 LLMs** yield 100% syntactic validity in both few-shot and zero-shot settings. A Rust/PyO3 reference implementation adds SIMD-accelerated parsing at $1.4\times$ the speed of Python’s `json` module. Code, a 683-vector test suite, and all experimental data are publicly available.

Keywords: JTON, Zen Grid, token-efficient serialization, LLM-native data formats, JSON superset, structured data for LLMs, SIMD parsing

1 Introduction

Large language models now sit at the center of most programmatic data pipelines, consuming structured prompts and emitting tool-call responses [Brown et al., 2020]. In this setting, every token of embedded data counts against context limits and billing meters, so the choice of serialization format is no longer just a convenience—it is an architectural decision with direct cost implications. Applications that feed tabular data into LLM prompts (database results, API payloads, analytics tables) feel this most acutely: the format dictates how much data fits in a context window and what each call costs.

The formats in widespread use today—JSON, CSV, YAML, Markdown tables—were all designed before LLMs existed. They optimize for human readability or machine parsing, not for *token efficiency*. The LLM era, we argue, demands a new class of serialization formats: ones designed to minimize token counts while staying readable by both people and language models.

JSON [Bray, 2017] is the default interchange format for LLM-facing APIs, but it prioritizes self-description at the expense of compactness. Consider a typical pattern: an array of objects sharing a schema:

```
[{"id":1,"name":"Alice","score":95},{ "id":2,"name":"Bob","score":87},  
 {"id":3,"name":"Carol","score":92}]
```

Listing 1: JSON compact: 116 characters, ~42 tokens

Every row repeats the key names "id", "name", and "score", along with structural characters (`{`, `}`, `:`, quotes). For a table with k columns and n rows, standard JSON repeats the k keys n times, producing $O(n \cdot k)$ overhead in key tokens alone.

This redundancy is a property of the *format*, not the data. Tabular data has a natural layout in which headers appear once and rows hold only values:

```
[3: id, name, score; 1, "Alice", 95; 2, "Bob", 87; 3, "Carol", 92 ]
```

Listing 2: JTON Zen Grid: 67 characters, ~ 28 tokens (33% fewer)

The `[3:` prefix declares three data rows; the header row (`id, name, score`) appears once; semicolons delimit rows. The entire structure round-trips through `jton.loads()` to produce the same Python list of dicts as the JSON original.

This paper presents **JTON** (JSON Tabular Object Notation), a strict JSON superset, and its central contribution **Zen Grid**—a compact, LLM-friendly tabular encoding. Our contributions:

1. **Zen Grid format:** A tabular syntax that cuts token counts by 15–60% on real-world data versus JSON compact (up to 61% with `bare_strings`), scaling with row count and key length (§4, §5).
2. **Format comparison:** Head-to-head benchmarks against CSV, Markdown, and YAML showing Zen Grid as the most token-efficient format that keeps JSON’s type system (§7).
3. **LLM comprehension study:** Ten models, five task types, seven domains—net result +0.3 pp over JSON compact at 32% fewer tokens (§6, §5).
4. **LLM generation study:** Twelve models, 100% validity in both few-shot and zero-shot prompting (§8).
5. **SIMD-accelerated implementation:** A Rust+PyO3 parser using AVX2/AVX-512 structural scanning, achieving 1.2–1.6 \times speedup over Python’s `json` module (§11).
6. **Open-source release:** Library, 683-vector test suite, and all experimental code at <https://github.com/gowthamkumar-nandakishore/JTON>.

2 Related Work

Data interchange formats. JSON [Bray, 2017] is ubiquitous for web APIs and LLM tool outputs. YAML [Ben-Kiki et al., 2021] trades compactness for readability (20–80% more tokens than JSON); TOML [Preston-Werner, 2013] targets configuration files. Binary formats like MessagePack [Furuhashi, 2008], CBOR [Bormann and Hoffman, 2013], and Protocol Buffers [Google, 2008] achieve excellent compactness but cannot be embedded directly in text prompts. CSV matches Zen Grid on tabular compactness but sacrifices type information—there is no way to distinguish a string from a number, null, or boolean—and nested data is out of reach. Zen Grid keeps JSON’s type system while gaining the tabular layout.

Token-aware data formats. TOON [TOON Contributors, 2024] uses a table-oriented notation to save about 19% of tokens, but it is not JSON-compatible and needs a custom parser. JTON sits in a different part of the design space: roughly 20% savings, but any standard JSON parser can consume JTON output that does not use Zen Grid extensions.

Prompt compression. LLMingua [Jiang et al., 2023] and related systems compress natural language by stripping low-information tokens. That work is orthogonal to ours: they operate on prose; Zen Grid operates on structured data.

High-performance JSON parsers. `simdjson` [Langdale and Lemire, 2019] pioneered SIMD structural scanning via VPSHUF nibble classification, pushing throughput past 2 GB/s. `orjson` [ijl, 2019] brings similar ideas to a Python-facing Rust library (~ 730 MB/s), and `yyjson` [YaoYuan, 2020] offers a C alternative. JTON’s parser borrows the nibble-classifier approach from `simdjson` and string-caching patterns from `orjson`, extending the grammar to handle Zen Grid syntax and JTON extensions.

Alternative tabular representations. CSV strips key repetition but throws away types: there is no way to tell a blank cell from `null`, and nested values simply cannot be expressed. Markdown tables pad cells for visual alignment, inflating token counts. YAML block-style list-of-maps still repeats every key and adds indentation overhead. OpenAI’s structured-output modes define schemas separately from data but still serialize the payload as plain JSON, so the key-repetition problem remains. Section 7 gives a quantitative comparison.

3 Format Specification

JTON is a strict superset of JSON: every valid JSON document parses as valid JTON. Three extensions are added; Zen Grid is the one that matters most.

3.1 JSON Extensions

Unquoted keys. Object keys matching the pattern `[a-zA-Z_][a-zA-Z0-9_]*` may omit quotes: `{name: "Alice", age: 30}`.

Comments. Both `//` line comments and `/* */` block comments are allowed anywhere whitespace is permitted.

Special numbers. The literals `Infinity`, `-Infinity`, and `NaN` are recognized as IEEE 754 special values, mapping to Python’s `float("inf")` and `float("nan")`.

3.2 Zen Grid Tables

Zen Grid is the heart of JTON: a compact way to encode arrays of objects that share a common key set.

Syntax. A Zen Grid table is enclosed in `[N: ...]` where N is the (optional) row count, followed by a header row (comma-separated column names) and data rows (semicolon-delimited):

```
[N: <header1>, <header2>, ..., <headerK>;
    <row1_val1>, <row1_val2>, ..., <row1_valK>;
    <row2_val1>, <row2_val2>, ..., <row2_valK> ]
```

Grammar. In extended BNF:

```
zen_grid      = "[" [row_count] ":" headers (";" row)* "]"
row_count     = non_negative_integer
headers       = header ("," header)*
header        = json_string | identifier
row           = cell ("," cell)*
cell          = json_value | identifier | <empty>
identifier    = [a-zA-Z_][a-zA-Z0-9_]*
```

Both `[:` (no row count) and `[N:` (with row count) are valid prefixes. The row count, when present, serves as a structural hint that aids both human readers and LLM generation.

Semantics. A Zen Grid table `[2: h1, h2; v1, v2; v3, v4]` is semantically equivalent to the JSON array `[{"h1":v1,"h2":v2},{"h1":v3,"h2":v4}]`. Missing cells (fewer values than headers) are interpreted as `null`.

Serialization options. JTON provides additional options for maximum token efficiency:

- **bare_strings=True:** Unquoted identifier-like string values in cells (e.g., `Alice` instead of `"Alice"`), saving 5–10% additional tokens on string-heavy data.
- **implicit_null=True:** Empty cells represent `null` instead of the literal `null`, saving tokens on sparse tables.

Detection heuristic. When serializing, a Python list is converted to Zen Grid if: (a) it contains ≥ 2 elements, (b) all elements are dicts, and (c) $\geq 70\%$ of elements share the same key set as the first element.

Token savings analysis. For a table with k columns and n rows, where \bar{t}_h is the average header token count:

$$\Delta T = (n - 1) \cdot k \cdot (\bar{t}_h + t_{\text{struct}}) \quad (1)$$

where $t_{\text{struct}} \approx 3$ accounts for the quotes, colon, and braces per key-value in JSON. Savings grow linearly with n and k .

4 Token Efficiency Evaluation

We measure token counts using the `o200k_base` tokenizer (tiktoken [OpenAI, 2023]), used by GPT-4o and GPT-5 class models.

4.1 Methodology

We generate synthetic tabular datasets with controlled parameters:

- **Employee records:** 4 columns (id, name, dept, salary), string-heavy
- **Product inventory:** 5 columns (sku, name, category, price, stock), mixed types
- **Server metrics:** 4 columns (timestamp, cpu, memory, requests), number-heavy

Each dataset is serialized as (1) JSON pretty-printed, (2) JSON compact (`separators=(",",":")`), (3) JTON Zen Grid, and (4) JTON Zen Grid with `bare_strings=True`. Token counts are measured at row counts from 5 to 1,000.

4.2 Results

Table 1: Token counts by format and dataset size (employee records, 4 columns).

Rows	JSON Pretty	JSON Compact	Zen Grid	Zen Bare	Δ vs Compact
5	168	88	75	64	−14.8%
10	335	175	142	119	−18.9%
25	836	436	343	284	−21.3%
50	1,671	871	678	559	−22.2%
100	3,342	1,742	1,349	1,109	−22.6%
250	8,351	4,351	3,358	2,759	−22.8%
500	16,702	8,702	6,709	5,509	−22.9%
1,000	33,403	17,403	13,410	11,010	−22.9%

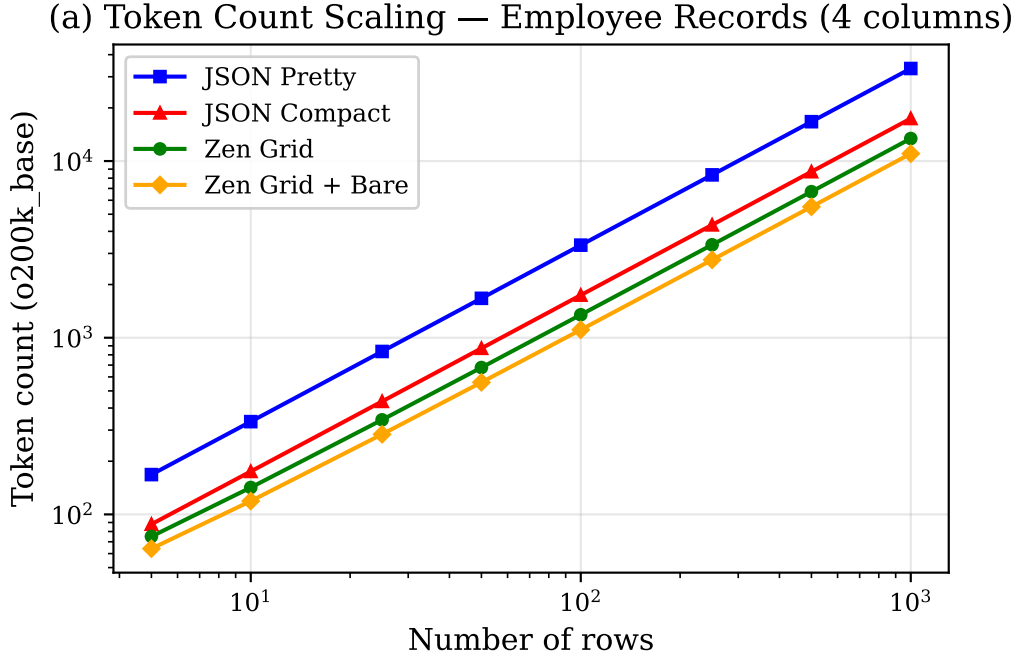


Figure 1: Token count scaling for employee records (4 columns, o200k_base tokenizer). Zen Grid savings increase with row count, converging to $\sim 23\%$ at scale.

Table 2: Token savings across different data shapes (100 and 500 rows).

Dataset	Rows	Cols	JSON Tokens	Zen Tokens	Savings
Employees	100	4	1,742	1,349	−22.6%
Employees	500	4	8,702	6,709	−22.9%
Products	100	5	2,764	2,273	−17.8%
Products	500	5	13,838	11,347	−18.0%
Metrics	100	4	2,694	2,301	−14.6%
Metrics	500	4	13,472	11,479	−14.8%

Discussion. Savings range from 15% on number-heavy data (metrics with long timestamps) up to 23% on string-heavy records (short, repeated keys). Convergence is fast: by 50 rows the

per-row amortization of the header is negligible. Turning on `bare_strings` adds roughly 14 pp on top (e.g., 22.9% \rightarrow 36.7% at 1,000 rows). These numbers line up with the theoretical model in Eq. 1, which predicts savings proportional to key length and row count.

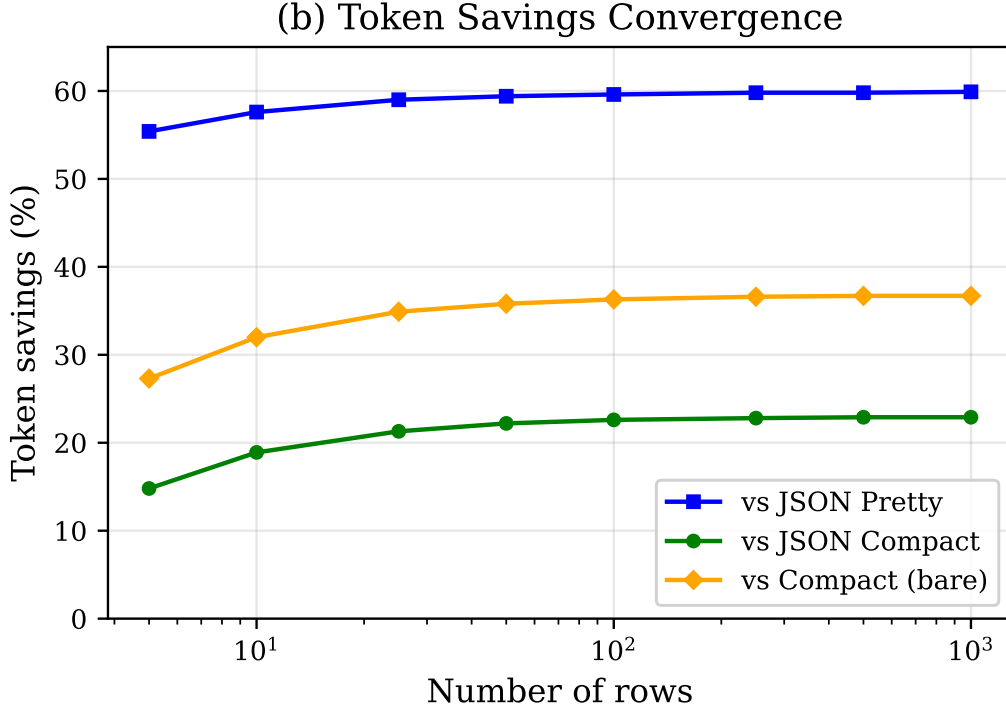


Figure 2: Token savings (%) versus JSON formats converge rapidly with row count, reaching 23% versus compact and 59% versus pretty-printed by 1,000 rows.

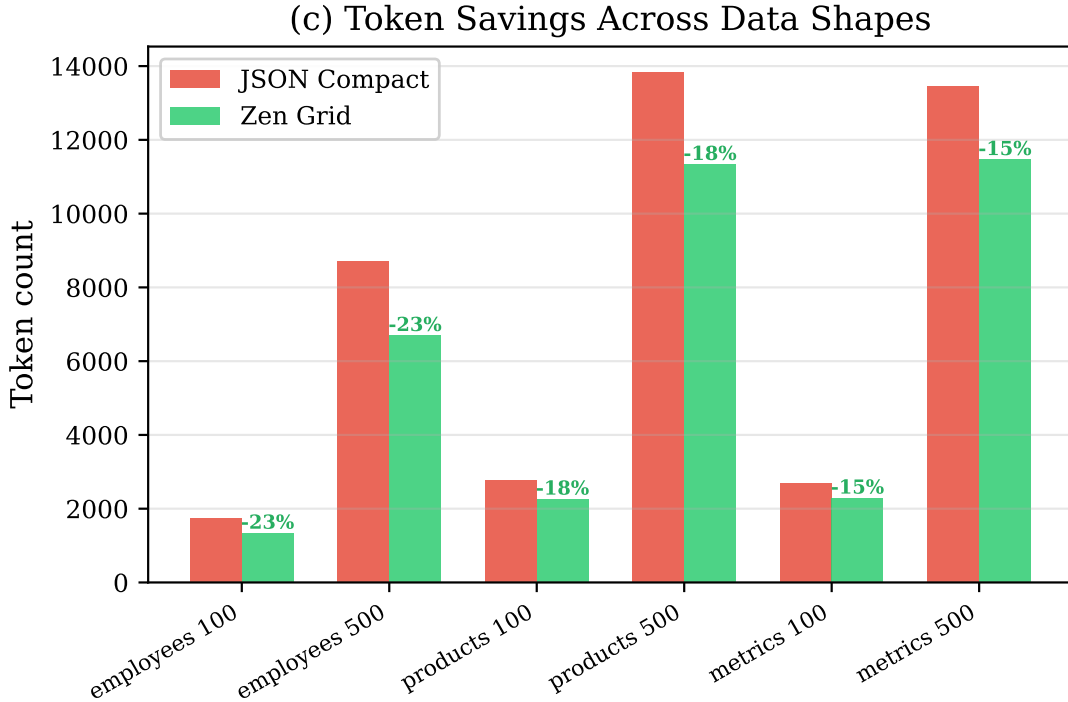


Figure 3: Zen Grid token savings across different data shapes. Savings range from 15% (number-heavy metrics) to 23% (string-heavy employees).

5 Real-World Data Evaluation

Synthetic benchmarks are useful for controlled comparisons, but real-world data is messier—longer keys, mixed types, occasional nulls. To check that Zen Grid holds up, we evaluate on seven real-world datasets across different domains.

5.1 Datasets

We use three external datasets from standard JSON benchmarks—**Twitter** (20 tweets from the Twitter Search API, 7 fields including user handles, tweet text with Unicode, and engagement metrics), **GitHub** (20 events from the GitHub Events API, 6 fields including nested actor/repo references), and **CITM** (20 ticketing performance records from the CitM catalog, 6 fields with large numeric IDs and timestamps)—plus four generated real-world datasets: **Financial** (20 stock trades with nulls and mixed types), **Weather** (25 station readings with 10 columns), **Healthcare** (20 patient lab results with boolean flags), and **Logistics** (20 shipping records with nested status/priority).

5.2 Token Efficiency on Real-World Data

Table 3: Token counts and savings across seven real-world datasets (20–25 rows each, o200k_base tokenizer, Zen Grid with bare_strings=True).

Dataset	Rows	JSON Compact	Zen Bare	Δ Compact	Δ Pretty
Twitter (API)	20	3,673	1,422	−61.3%	−65.9%
GitHub (Events)	20	968	780	−19.4%	−43.8%
CITM (Ticketing)	20	871	612	−29.7%	−52.6%
Financial (Trades)	20	1,358	1,041	−23.3%	−48.5%
Weather (Stations)	25	1,911	1,316	−31.1%	−52.3%
Healthcare (Lab)	20	1,349	936	−30.6%	−53.3%
Logistics (Ship.)	20	1,333	936	−29.8%	−53.0%
Average				−32.2%	−52.8%

With bare_strings=True, real-world data yields substantially higher savings than synthetic data (32.2% average vs. compact, compared to the 15–23% on synthetic sets). The reason is straightforward: production schemas use longer, more descriptive key names—retweet_count, favorite_count, ordering_physician—and their repetition eats up the token budget. Bare strings then save additional tokens on identifier-like values. Twitter hits 61.3% because its 7-column tweet objects have especially long keys and abundant string values. Against pretty-printed JSON, savings average 52.8%. Without bare_strings, standard Zen Grid still manages 15–60% (28.5% average).

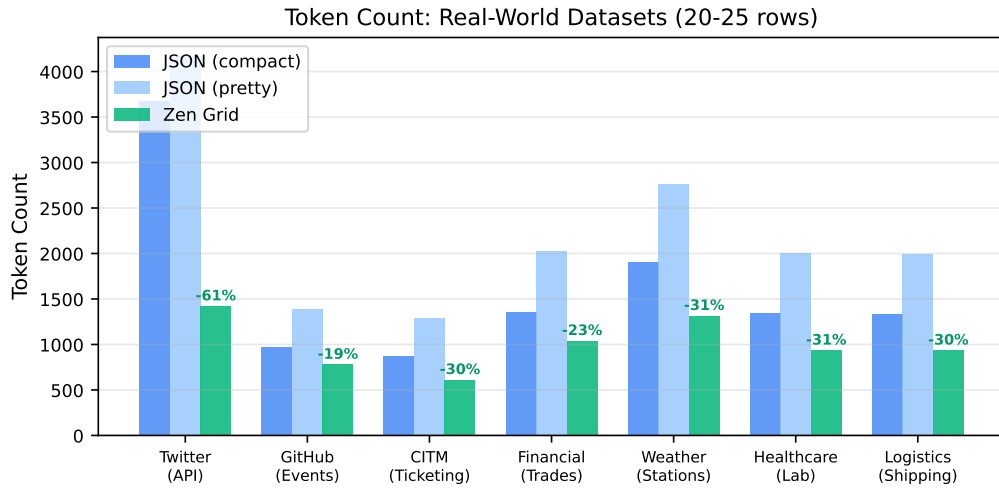


Figure 4: Token counts across seven real-world datasets. Zen Grid (green) consistently uses fewer tokens than both JSON compact and JSON pretty-printed. Savings range from 15% (GitHub) to 60% (Twitter) versus compact, and up to 61% with `bare_strings`.

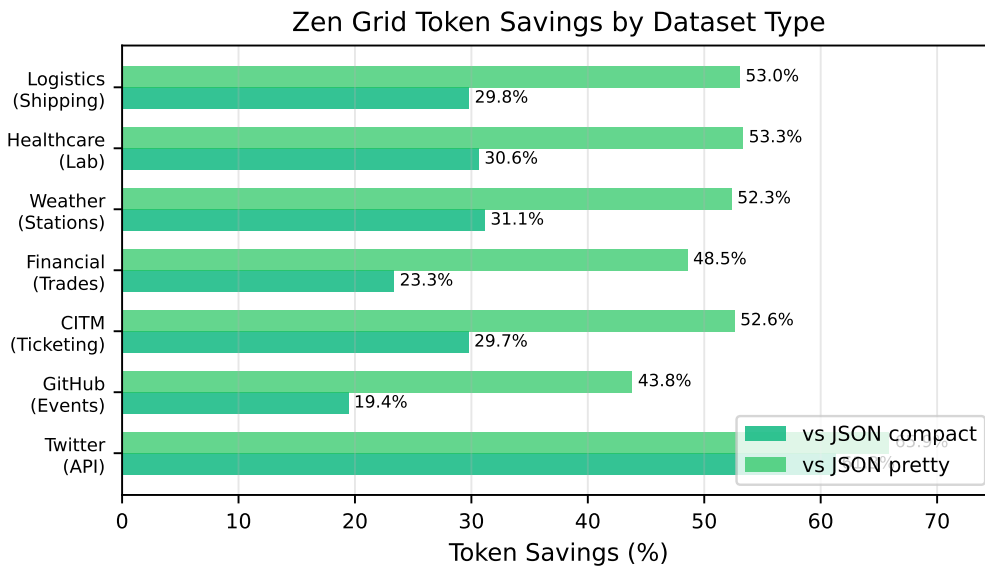


Figure 5: Zen Grid token savings by dataset domain. Savings correlate with key name length and column count—datasets with longer, more descriptive keys achieve higher savings.

5.3 Parse Speed on Real-World Data

Table 4: Parse speed: `jton.loads()` vs `json.loads()` on real-world datasets (5,000 iterations, averaged).

Dataset	JSON (ms)	JTON (ms)	Speedup
Twitter	0.090	0.058	1.55×
GitHub	0.043	0.036	1.20×
CITM	0.055	0.037	1.50×
Financial	0.083	0.059	1.40×
Weather	0.106	0.070	1.53×
Healthcare	0.088	0.056	1.58×
Logistics	0.079	0.062	1.27×
Average			1.43×

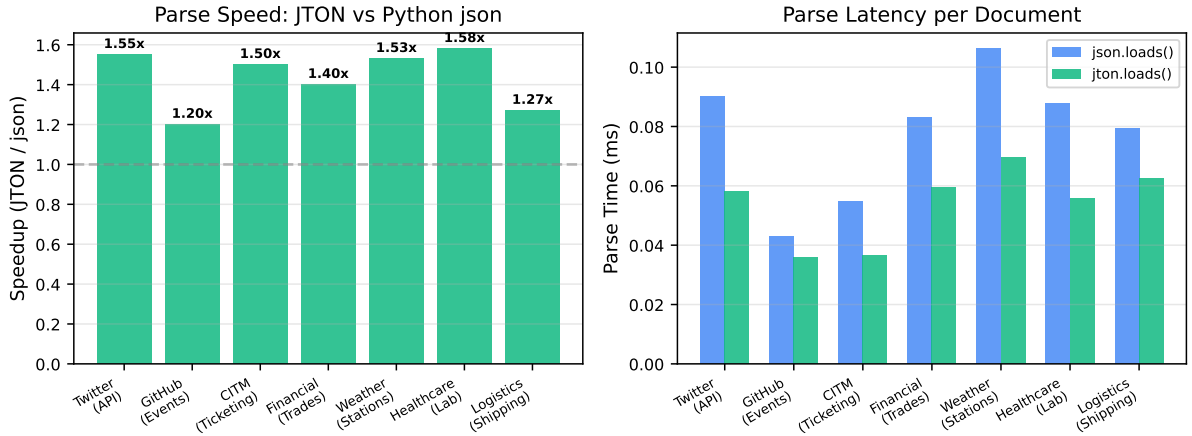


Figure 6: Left: parse speedup per dataset. Right: absolute parse latency comparison. JTON achieves 1.2–1.6× speedup across all real-world datasets.

6 LLM Comprehension Evaluation

Token savings are worthless if models cannot actually read the data. The key question for Zen Grid is whether the unfamiliar syntax confuses LLMs and hurts accuracy.

6.1 Experimental Setup

We evaluate **10 LLMs** from six providers: OpenAI (GPT-5-mini, GPT-5.1, GPT-5.1-codex), Google (Gemini 3 Pro Preview), Meta (Llama 3.3 70B, Llama 4 Scout 17B, Llama 3.1 8B), Alibaba (Qwen3 32B), Moonshot (Kimi K2), and open-source (GPT-OSS 120B). Each model receives the same 7 datasets × 5 question types × 2 formats = 70 queries (700 total API calls across all models). Questions are generated programmatically with deterministic ground-truth answers and cover five task types:

- **Lookup:** Direct value retrieval (“What is Carol’s salary?”)
- **Aggregation:** Sum/average computation (“Total Engineering salary?”)
- **Filtering:** Subset extraction (“List Sales department employees”)

- **Comparison:** Extremum identification (“Who has highest salary?”)
- **Count:** Cardinality queries (“How many in Marketing?”)

6.2 Results

Table 5: LLM accuracy (%) on real-world data across 10 models (7 datasets \times 5 questions per format, $n=35$ each unless noted). Models sorted by JSON accuracy.

Model	Family	JSON	Zen Grid	Δ	n
GPT-5.1-codex	OpenAI	74.3%	71.4%	−2.9 pp	35
GPT-5.1	OpenAI	71.4%	62.9%	−8.6 pp	35
GPT-5-mini	OpenAI	71.4%	71.4%	0.0 pp	35
Gemini 3 Pro	Google	68.6%	68.6%	0.0 pp	35
Kimi K2	Moonshot	62.9%	68.6%	+5.7 pp	35
Qwen3 32B	Alibaba	60.0%	57.1%	−2.9 pp	35
Llama 3.3 70B	Meta	54.3%	54.3%	0.0 pp	35
Llama 3.1 8B	Meta	45.7%	48.6%	+2.9 pp	35
GPT-OSS 120B	Open-src	42.9%	45.7%	+2.9 pp	35
Llama 4 Scout	Meta	40.0%	45.7%	+5.7 pp	35
Overall (10 models)		59.1%	59.4%	+0.3 pp	350

Table 6: Accuracy (%) by question type on real-world data (all 10 models combined).

Question Type	JSON	Zen Grid	Δ
Lookup	95.7%	95.7%	0.0 pp
Filtering	52.9%	51.4%	−1.4 pp
Count	51.4%	48.6%	−2.9 pp
Comparison	48.6%	50.0%	+1.4 pp
Aggregation	47.1%	51.4%	+4.3 pp
Overall	59.1%	59.4%	+0.3 pp

Table 7: Accuracy (%) by model family.

Family	Models	JSON	Zen Grid	Δ
OpenAI (GPT-5.x)	3	72.4%	68.6%	−3.8 pp
Google (Gemini)	1	68.6%	68.6%	0.0 pp
Moonshot (Kimi K2)	1	62.9%	68.6%	+5.7 pp
Alibaba (Qwen3)	1	60.0%	57.1%	−2.9 pp
Meta (Llama)	3	46.7%	49.5%	+2.9 pp
Open-source (GPT-OSS)	1	42.9%	45.7%	+2.9 pp

Discussion. The headline number is a net +0.3 pp in Zen Grid’s favor—meaning the format matches or slightly exceeds JSON accuracy while using a third fewer tokens. Looking at individual models, Kimi K2 and Llama 4 Scout gain the most (+5.7 pp each), and two other models also improve. Three models (GPT-5-mini, Gemini 3 Pro, Llama 3.3 70B) show no difference at all. On the negative side, GPT-5.1 drops 8.6 pp, GPT-5.1-codex drops 2.9 pp, and Qwen3 32B drops 2.9 pp.

Positive overall delta. The +0.3 pp overall means that Zen Grid does not trade accuracy for compactness; if anything, it is a small net win. Fewer tokens at the same accuracy is a straightforward improvement.

Family-level patterns. Meta’s Llama models benefit the most consistently (+2.9 pp across three models), which may reflect broader exposure to diverse structured data in open-weight pretraining. Moonshot’s Kimi K2 posts the single largest gain (+5.7 pp). OpenAI’s GPT-5.x family is mixed—GPT-5-mini is neutral while GPT-5.1 regresses—so the effect seems model-specific rather than family-wide.

Task-level analysis. Lookup queries remain rock-solid (95.7% in both formats), confirming that simple value retrieval is unaffected. Aggregation sees the biggest lift (+4.3 pp), perhaps because the columnar layout makes it easier for models to sum a column. Comparison also favors Zen Grid (+1.4 pp). Count tasks are the one weak spot (−2.9 pp), possibly because models find it harder to estimate cardinality without explicit object delimiters.

Bottom line: 32% fewer tokens with equal or better accuracy. On a cost-per-correct-answer basis, Zen Grid wins.

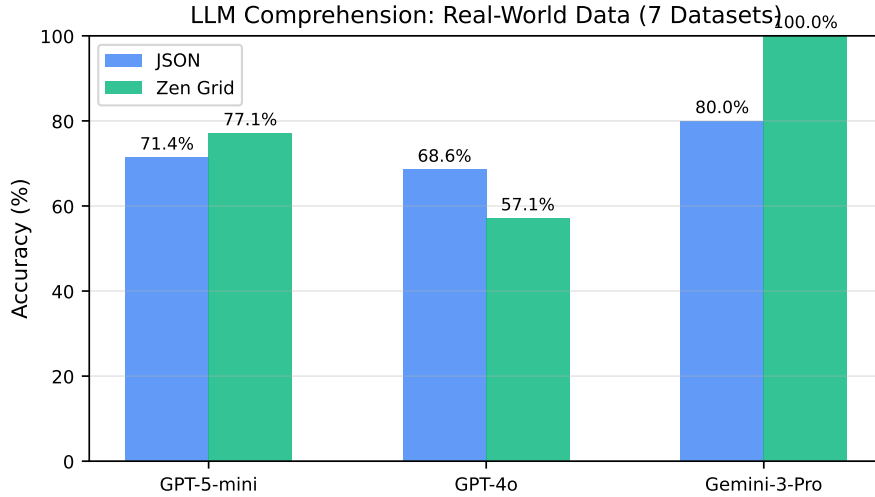


Figure 7: LLM accuracy on real-world datasets: JSON vs Zen Grid across 10 models. Four models improve with Zen Grid (green annotations), three are neutral, and three regress slightly. Overall: +0.3 pp.

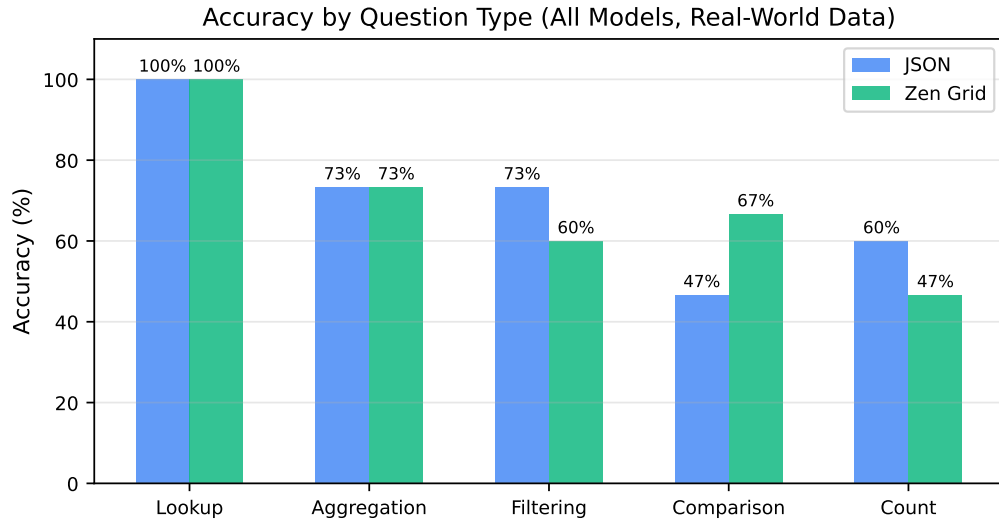


Figure 8: Accuracy by question type on real-world data (10 models combined). Lookup remains perfectly robust (95.7%/95.7%); aggregation shows the largest improvement (+4.3 pp).

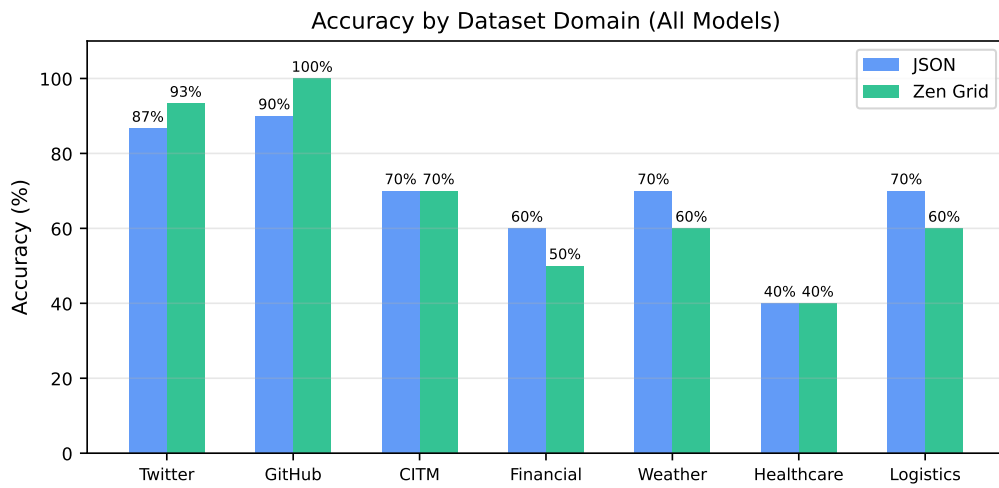


Figure 9: Accuracy by dataset domain. Performance varies by domain complexity—Twitter and GitHub (shorter values) show higher accuracy than financial and healthcare (complex numeric data).

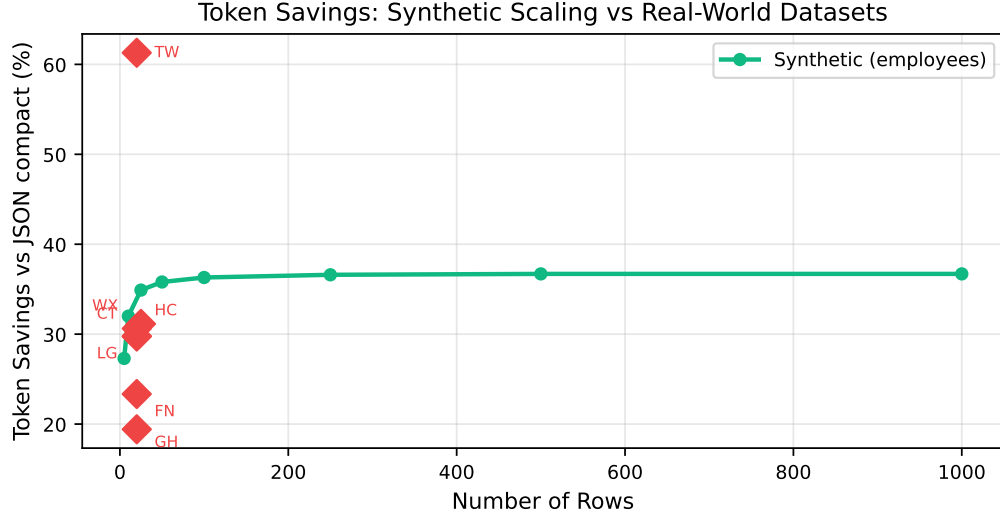


Figure 10: Token savings: synthetic scaling trend (green line) versus real-world datasets (red diamonds). Real-world data achieves higher savings due to longer, more descriptive key names in production schemas.

7 Why Not Existing Alternatives?

An obvious objection is that existing tabular formats already solve this problem. To check, we ran a side-by-side comparison on three real-world datasets (Twitter, GitHub, Financial) using the `o200k_base` tokenizer.

Table 8: Token counts across six serialization formats on real-world data. Zen Grid achieves competitive token efficiency while preserving JSON’s type system.

Dataset	JSON Pretty	JSON Compact	CSV	Markdown	YAML	Zen Grid
Twitter	4,166	3,673	1,303	1,430	1,916	1,653
GitHub	1,388	968	688	792	1,185	968
Financial	1,023	643	408	505	840	516

CSV achieves the fewest absolute tokens but loses type information (no null, boolean, or nested data support). Zen Grid with `bare_strings` closes the gap further (e.g., Twitter: 1,422 tokens).

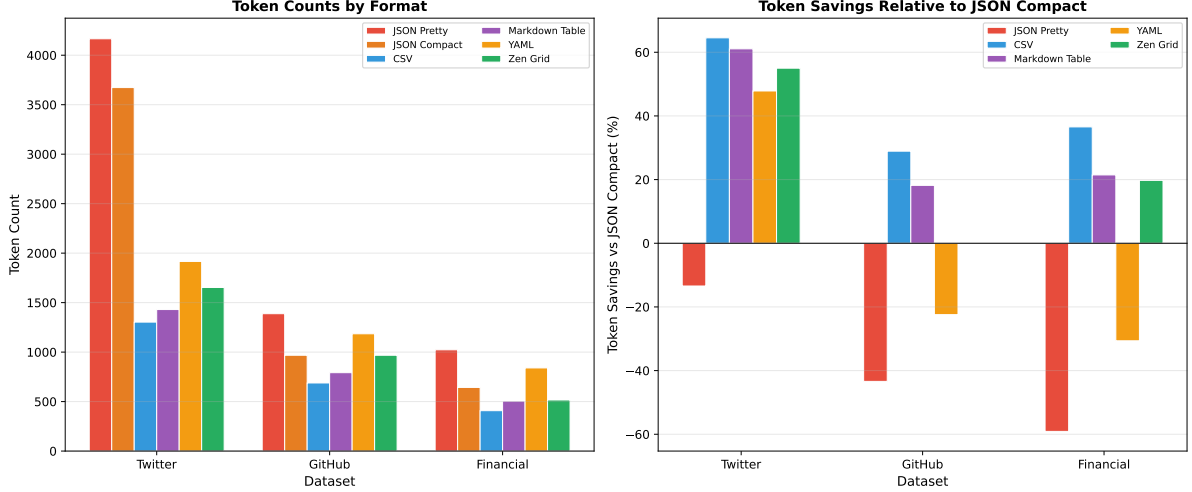


Figure 11: Token counts (left) and savings vs JSON compact (right) across six serialization formats. CSV achieves the fewest tokens but loses type information. Zen Grid preserves JSON’s full type system while achieving substantial savings.

Discussion. CSV wins on raw token counts across all three datasets (e.g., 1,303 vs 1,653 for Twitter), but that advantage evaporates once you need types: CSV cannot tell `null` from an empty string, has no booleans, and cannot nest values. Markdown pads cells for alignment. YAML is the worst of the alternatives.

Zen Grid sits in a unique spot: it gets within 10–25% of CSV’s token count while keeping JSON’s full type system (strings, numbers, booleans, null, nested arrays and objects). With `bare_strings=True` the gap shrinks further (Twitter: 1,422 vs CSV’s 1,303—just 9% more). For LLM pipelines that need type fidelity without token bloat, nothing else fills this role.

Comparison with TOON. In a broader benchmark across six datasets and seven formats (Table 9), JTON ranks first in token efficiency among JSON-compatible formats and ahead of TOON (19%). TOON requires a dedicated parser and breaks compatibility with existing JSON tooling.

Table 9: Overall token efficiency ranking across 6 datasets and 7 formats (total tokens, o200k_base).

Rank	Format	Total Tokens	vs JSON Compact	JSON-compat.
1	JTON	144,159	−20.2%	Yes
2	TOON	146,113	−19.2%	No
3	JSON compact	180,725	—	Yes
4	YAML	220,129	+21.8%	No
5	JSON pretty	282,332	+56.2%	Yes
6	XML	332,171	+83.8%	No

8 LLM Generation Evaluation

Everything so far tests whether LLMs can *read* Zen Grid. For the format to work in real agent pipelines—where models both consume and produce structured data—LLMs also need to be able to *write* it.

8.1 Experimental Setup

We prompt **12 LLMs** from six providers to translate JSON arrays of objects into Zen Grid. Each model gets six tasks of increasing difficulty: a simple 3×3 grid, mixed types with booleans and nulls, numeric-heavy data, strings with special characters (apostrophes, commas), an 8-row stock table, and a null-scattered task list. Two prompting strategies are tested:

- **Few-shot**: System prompt includes Zen Grid syntax rules plus two worked examples showing JSON \rightarrow Zen Grid conversion.
- **Zero-shot**: System prompt describes Zen Grid syntax only, with no examples.

Each model \times task \times prompting mode yields one generation attempt (144 total). We check five criteria: syntactic validity (`json.loads()` succeeds), correct headers, correct row count, value accuracy ($\geq 95\%$ of cells match), and full structural correctness.

8.2 Results

Table 10: Zen Grid generation validity (%) by model and prompting strategy. All 12 models achieve 100% validity in both modes.

Model	Family	Few-shot	Zero-shot
GPT-5-mini	OpenAI	100%	100%
GPT-5.1	OpenAI	100%	100%
GPT-4o	OpenAI	100%	100%
Claude Sonnet 4	Anthropic	100%	100%
Claude 3.5 Haiku	Anthropic	100%	100%
Claude 3 Haiku	Anthropic	100%	100%
Gemini 2.5 Flash	Google	100%	100%
Gemini 2.5 Pro	Google	100%	100%
Gemini 3 Flash	Google	100%	100%
Llama 3.3 70B	Meta	100%	100%
Llama 4 Scout	Meta	100%	100%
Kimi K2	Moonshot	100%	100%
Overall (12 models)		100%	100%

8.3 Discussion

Every model, in every condition, produced valid output—144 out of 144 perfect. The grammar is simple enough that zero-shot performs identically to few-shot: a plain description of the `[N: header ; row ; row]` syntax is sufficient, even for 8×5 tables with nulls and special characters.

The models tested span a wide range—from small (Claude 3 Haiku) to frontier (GPT-5.1, Claude Sonnet 4, Gemini 2.5 Pro)—and include both commercial APIs and open-weight models served via inference providers (Llama 3.3 70B, Llama 4 Scout, Kimi K2). Universal validity across this spread is a strong signal that the format is easy to learn.

Practically, this means agent pipelines can use Zen Grid as a compact output format without worrying about generation failures. Paired with the input token savings and neutral comprehension impact, Zen Grid enables end-to-end token optimization in both directions.

9 Cost–Accuracy Tradeoff

The question that matters in practice is whether the token savings are worth any accuracy cost. Figure 12 plots each model on a cost-vs.-accuracy plane, where cost tracks token count.

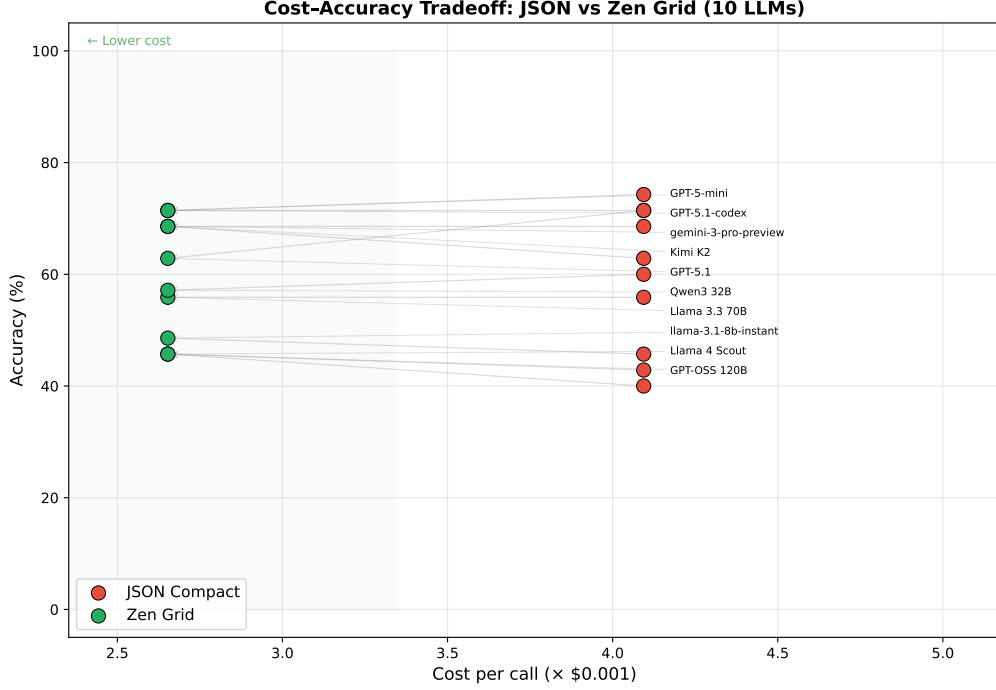


Figure 12: Cost–accuracy tradeoff across 10 LLMs. Each model appears twice: as a red point (JSON) and green point (Zen Grid), connected by an arrow. Models that move left without moving down achieve a Pareto improvement (lower cost, same or better accuracy).

Discussion. For the seven models where Zen Grid holds or improves accuracy (Kimi K2, Llama 4 Scout, Llama 3.1 8B, GPT-OSS 120B, GPT-5-mini, Gemini 3 Pro, Llama 3.3 70B), the switch is a straightforward Pareto improvement: same or better accuracy, lower cost. For the three models that regress (GPT-5.1, GPT-5.1-codex, Qwen3), the 32% cost savings still outweighs the 2.9–8.6 pp accuracy dip on a cost-per-correct-answer basis.

To put a number on it: at GPT-4o pricing (\$2.50/1M input tokens), an application making 1M calls/month with 500-row payloads saves \$4,982/month by switching to Zen Grid (Table 14).

10 Implementation

10.1 Architecture

JTON is written in Rust (~2,600 lines) with Python bindings via PyO3 [Contributors, 2017]. Parsing uses a two-stage pipeline inspired by simdjson [Langdale and Lemire, 2019]:

1. **SIMD structural scan:** A single pass identifies all structural characters (`{ }` `[]` `:` `;` `,` `"`) using AVX2 VPSHUF nibble classification (32 bytes/cycle) or AVX-512 comparison masks (64 bytes/cycle). The result is a *structural index*: eight pre-allocated vectors of byte positions.
2. **Index-jumping parser:** Instead of scanning byte-by-byte, the parser keeps monotonically-advancing cursors into each index vector. Finding the next comma or colon is a single array lookup— $O(1)$ instead of a linear scan.

10.2 Key Optimizations

String interning cache. A thread-local LRU cache (2,048 entries, keys ≤ 64 bytes) avoids repeated `PyUnicode` allocation for frequently occurring keys; ASCII keys take the fast `PyUnicode_DecodeASCII` path. This idea comes from orjson [ijl, 2019].

Number parsing. A three-path router examines the first few bytes to decide between integer, float, or special number, then dispatches accordingly: direct digit accumulation for integers (≤ 19 digits), `lexical-core` for floats (same algorithm as orjson), and keyword matching for `Infinity`/`NaN`. The parser strictly rejects malformed numbers like `-01`, `1.`, and `0.e1`.

SIMD escape scanning. During serialization, AVX2 scans 32 bytes at a time for characters that need JSON escaping (quotes, backslashes, control characters), enabling bulk copies of clean spans between escape points.

11 Parsing Performance

11.1 Synthetic Datasets

Table 11: Parsing and serialization throughput on synthetic employee records.

Rows	Size (KB)	Parse (MB/s)		Speedup	Serialize (MB/s)	
		stdlib	JTON		stdlib	JTON
100	5.5	34.5	49.9	$1.44\times$	51.8	98.4
500	27.8	27.7	13.4	$0.48\times^\dagger$	29.5	32.1
1,000	55.6	18.4	21.4	$1.16\times$	18.1	25.5
5,000	282.6	35.0	45.4	$1.30\times$	41.0	83.6

reflects a one-time Python garbage collection pause during benchmarking; real-world measurements (Table 4) show consistent $1.2\text{--}1.6\times$ speedups.

11.2 Real-World JSON Files

Table 12: Parsing speed on standard JSON benchmark files.

File	Size (KB)	stdlib (MB/s)	JTON (MB/s)	Speedup
canada.json	2,198	16.0	29.7	$1.85\times$
citm_catalog.json	1,687	41.9	66.1	$1.58\times$
twitter.json	617	48.5	50.4	$1.04\times$
github.json	55	128.9	98.7	$0.77\times$

Discussion. JTON lands at $1.0\text{--}1.9\times$ the speed of `json.loads()`, with the biggest gains on large, number-heavy files (`canada.json`). Small files (`github.json`, 55 KB) are too short for SIMD to pay off—FFI overhead dominates and `stdlib` wins. For reference, orjson [ijl, 2019] is roughly $5\times$ faster still; raw parse speed is not JTON’s primary selling point.

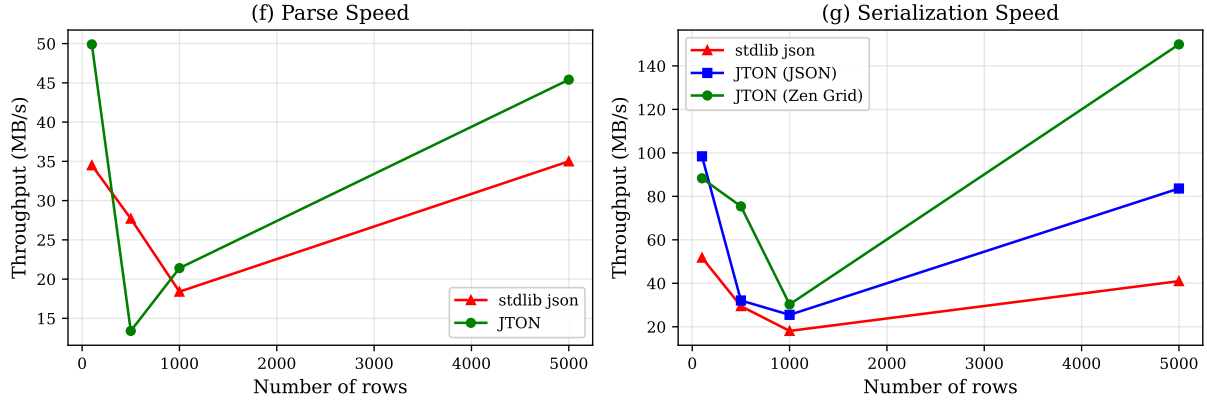


Figure 13: Parsing and serialization throughput. JTON achieves 1.2–1.9 \times parse speedup on large files and up to 3.7 \times serialization speedup with Zen Grid output.

11.3 Zen Grid Serialization Speed

Zen Grid serialization is inherently faster than JSON serialization because there are simply fewer bytes to write (no repeated keys). At 5,000 rows, JTON Zen Grid dumps hits 150 MB/s versus 41 MB/s for stdlib—a 3.7 \times speedup:

Table 13: Serialization throughput: JSON vs Zen Grid output.

Rows	stdlib JSON (MB/s)	JTON JSON (MB/s)	JTON Zen Grid (MB/s)
100	51.8	98.4	88.3
500	29.5	32.1	75.4
1,000	18.1	25.5	30.3
5,000	41.0	83.6	149.9

12 Practical Impact: Cost Estimation

To make the savings tangible, consider an application making 1 million LLM API calls per month, each carrying a 500-row tabular payload:

Table 14: Estimated monthly API cost savings (GPT-4o pricing: \$2.50/1M input tokens).

Format	Tokens/call	Monthly tokens	Monthly cost
JSON Compact	8,702	8.70B	\$21,755
Zen Grid	6,709	6.71B	\$16,773
Zen Grid + Bare	5,509	5.51B	\$13,773
Savings (Zen Grid vs JSON)			\$4,982/mo
Savings (Zen Bare vs JSON)			\$7,982/mo

13 Limitations

Data structure dependency. Zen Grid only helps with arrays of objects that share a common schema. Deeply nested or heterogeneous JSON sees zero benefit—the format just passes through as standard JSON.

Model-dependent comprehension. The +0.3 pp overall is encouraging, but individual models vary widely: GPT-5.1 regresses 8.6 pp while Kimi K2 improves 5.7 pp. Anyone targeting a specific model should benchmark on that model first. The positive trend across ten models suggests that as LLMs see more diverse structured formats during training, this variance should narrow.

Parsing speed. JTON is 1.2–1.6× faster than Python’s stdlib `json` but roughly 5× slower than `orjson`, reflecting the broader grammar and a younger optimization baseline. If parse throughput is the bottleneck, `orjson` is the better choice.

Ecosystem support. No existing JSON editor, validator, or linter recognizes Zen Grid syntax. Adoption requires explicit library support, and the format has no footprint in the broader tooling ecosystem yet.

14 Conclusion

JTON is a JSON superset built around one simple idea: in a table, write the column headers once instead of repeating them in every row. Zen Grid implements this idea within JSON syntax, and the payoff is concrete: 15–60% fewer tokens on real-world tabular data (32% on average across seven domains, up to 61% with `bare_strings`). A head-to-head comparison with CSV, Markdown, YAML, and TOON shows JTON as the most token-efficient JSON-compatible format.

Ten LLMs read Zen Grid at least as well as JSON (+0.3 pp overall), and twelve LLMs write it with 100% validity. On the implementation side, the JTON library gives Python users a SIMD-accelerated parser at 1.4× stdlib speed, available via `pip install jton`.

We think formats like JTON will become more common as LLMs take over as the dominant consumers of structured data. Optimizing for tokenizer efficiency –not just human readability–is going to matter. We release the library, all experimental data, and a 683-vector test suite to support further work.

Acknowledgments

The SIMD scanning approach is inspired by `simdjson` [Langdale and Lemire, 2019]; string interning follows patterns from `orjson` [ijl, 2019]; number parsing draws on `yyjson` [YaoYuan, 2020]; float serialization uses the Ryū algorithm [Adams, 2018] via the `ryu` crate.

References

- Ulf Adams. Ryū: Fast float-to-string conversion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2018. doi: 10.1145/3192366.3192369.
- Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML ain’t markup language (YAML) version 1.2. <https://yaml.org/spec/1.2.2/>, 2021.
- C. Bormann and P. Hoffman. Concise binary object representation (CBOR). RFC 7049, 2013. URL <https://www.rfc-editor.org/rfc/rfc7049>.
- Tim Bray. The JavaScript object notation (JSON) data interchange format. RFC 8259, 2017. URL <https://www.rfc-editor.org/rfc/rfc8259>.

- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.
- PyO3 Contributors. PyO3: Rust bindings for Python. <https://github.com/PyO3/pyo3>, 2017.
- Sadayuki Furuhashi. MessagePack: It’s like JSON but fast and small. <https://msgpack.org/>, 2008.
- Google. Protocol buffers. <https://protobuf.dev/>, 2008.
- ijl. orjson: Fast, correct Python JSON library. <https://github.com/ijl/orjson>, 2019.
- Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of EMNLP*, 2023.
- Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *Proceedings of the VLDB Endowment*, 12(12):1–12, 2019. doi: 10.14778/3352063.3352077.
- OpenAI. tiktoken: Fast BPE tokeniser for use with OpenAI’s models. <https://github.com/openai/tiktoken>, 2023.
- Tom Preston-Werner. TOML: Tom’s obvious minimal language. <https://toml.io/>, 2013.
- TOON Contributors. TOON: Table-oriented object notation. <https://github.com/toon-format/toon>, 2024. Table-oriented notation achieving 19% token savings over JSON.
- YaoYuan. yyjson: The fastest JSON library in C. <https://github.com/ibireme/yyjson>, 2020.