Structure-Aware Tokenization for JSON:

Exploiting Schema Repetition for Compact Token Sequences
with a 90× Smaller Vocabulary

Anthony Maio

March 2026

Abstract

General-purpose subword tokenizers such as `cl100k_base` treat JSON as flat text, splitting structural syntax characters across multiple tokens and re-encoding identical key names in every object. We present JSON-TOKENIZER, a structure-aware tokenizer that assigns dedicated single tokens to JSON grammar elements, learns a compact key vocabulary from training data, and applies byte-pair encoding (BPE) only to value content. On schema-repetitive JSON workloads—including GeoJSON features, observability telemetry, and Kubernetes manifests—JSON-TOKENIZER achieves 5–15% token savings over `cl100k_base` while using a vocabulary approximately 90× smaller (1,100–4,100 tokens vs. 100,256). Batch encoding of JSON arrays yields savings up to 9.3%. All encodings are lossless: decoded output is byte-identical to the original JSON across 4,200+ test objects. We provide an open-source, zero-dependency Python implementation with benchmarks on five real-world dataset classes, a vocabulary size sweep analysis, and an honest comparison showing where general-purpose tokenizers remain superior (prose-heavy JSON). The tokenizer is designed for LLM structured output pipelines, API response caching, and observability log ingestion, where schema repetition dominates and token budgets directly impact cost and latency.

Keywords: tokenization, JSON, byte-pair encoding, structured data, LLM inference efficiency

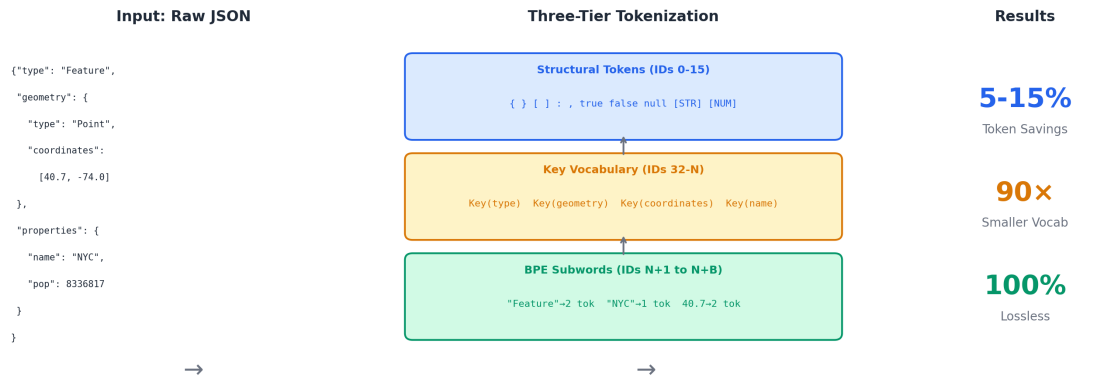

Figure 1: **Graphical Abstract.** JSON-TOKENIZER separates JSON into three token tiers: structural tokens for grammar (blue), a learned key vocabulary for repeated keys (gold), and BPE subwords for value content (green). This architecture yields 5–15% token savings on schema-repetitive JSON with a 90× smaller vocabulary than `cl100k_base`, while maintaining lossless roundtrip fidelity.

1 Introduction

JSON has become the dominant data interchange format for web APIs, cloud infrastructure, observability pipelines, and—increasingly—large language model (LLM) structured output [9, 21, 22]. In LLM-based systems, JSON appears in function call arguments, tool-use responses, retrieval-augmented generation pipelines, and structured extraction tasks [1, 13]. Each token in these interactions has a direct cost: in API pricing, inference latency, and context window consumption [11, 20].

Despite JSON's prevalence, the tokenizers used by leading LLMs—BPE variants such as `cl100k_base` [16] (GPT-4) and SentencePiece [14]—were designed for natural language text [17, 19]. They treat JSON as an opaque byte stream, leading to three systematic inefficiencies:

1. **Structural fragmentation.** Grammar characters (`{`, `}`, `[`, `]`, `:`, `,`) may be merged with adjacent whitespace or content into composite tokens, then split differently depending on context. A single `{` consumes one token in some positions but is grouped with a preceding newline in others.

2. **Redundant key encoding.** In a JSON array of 1,000 objects sharing the same schema, the key `"created_at"` is tokenized from scratch in every object—typically 3–4 subword tokens per occurrence. With 15 keys per object, this yields 15,000–60,000 tokens spent encoding information that could be represented once.

3. **Distribution mismatch.** BPE merge tables learned from English prose produce suboptimal segmentations for JSON value distributions: UUIDs, ISO timestamps, IP addresses, and enumerated strings follow different frequency patterns than natural language [4, 6].

Recent work has begun addressing structure-aware processing of JSON. ORIGAMI [18] introduced a reversible tokenizer for semi-structured data with grammar-constrained decoding. Karim et al. [12] proposed hybrid tokenization combining fixed structural tokens with BPE for tabular data. Grammar-constrained decoding methods [2, 7, 8, 23] ensure valid JSON output from LLMs but do not address input tokenization efficiency. Li et al. [15] demonstrated that subword tokenization boundaries misalign with code grammar, causing systematic inefficiency—a finding that extends directly to JSON.

We present JSON-TOKENIZER, a tokenizer purpose-built for JSON that exploits the format's grammar and schema repetition. Our key insight is that JSON token sequences are highly compressible because they consist of three orthogonal components with different entropy characteristics:

- **Structural tokens** (grammar symbols): deterministic, zero entropy given the schema.
- **Key tokens** (field names): low entropy within a schema, high repetition.
- **Value tokens** (content): high entropy, domain-specific distribution.

By assigning a dedicated token to each component type, JSON-TOKENIZER achieves 5–15% token savings on schema-repetitive workloads with a vocabulary 90× smaller than `cl100k_base`. Critically, the tokenizer guarantees lossless roundtrip: `decode(encode(json)) == json` for all valid JSON inputs.

Contributions.

1. A three-tier tokenization architecture separating JSON grammar, key vocabulary, and value content into distinct token regions (section 3).

2. Benchmarks on five real-world dataset classes showing 5–15% savings on structured data and honest characterization of failure modes on prose-heavy JSON (section 5).

3. A vocabulary size sweep revealing diminishing returns past ∼2K BPE tokens, indicating that key vocabulary—not BPE—drives the compression gains (section 5).

4. An open-source, zero-dependency Python implementation with trained tokenizer persistence and CLI tooling.

2 Related Work

Subword tokenization. Byte-pair encoding [19] and its byte-level variant [17] are the dominant tokenization methods for LLMs. SentencePiece [14] provides a language-independent implementation. These methods learn merge rules from corpus statistics, achieving strong compression for natural language but producing suboptimal segmentations for structured formats [4]. Domain-specific tokenizers have been shown to improve efficiency by 9–17% in specialized fields [3, 6].

Structure-aware tokenization for JSON. ORIGAMI [18] is the most closely related work: a generative transformer architecture with a reversible tokenizer for semi-structured data that preserves JSON hierarchy through position encoding. Unlike our approach, ORIGAMI focuses on prediction tasks (classification, regression) rather than compression, and does not benchmark token count against general-purpose tokenizers. Karim et al. [12] proposed hybrid tokenization for structured data, combining fixed structural tokens with BPE for high-cardinality values. Their work targets tabular data (network flows) rather than nested JSON, achieving 6.18:1 compression ratios. Our work extends the hybrid principle to nested JSON with a learned key vocabulary tier.

Grammar-constrained decoding. A parallel line of research ensures that LLM outputs conform to JSON grammars by masking invalid tokens during decoding [2, 8, 23]. XGrammar [7] achieves 100× speedup over prior methods. JSONSchemaBench [9] provides a rigorous evaluation framework. These methods are complementary to our work: they constrain output generation, while we optimize input representation.

Token efficiency for LLMs. Token budget optimization is an active research area. Shao et al. [20] demonstrated 20–60% sequence length reduction through extensible tokenization. Geng et al. [10] proposed adaptive vocabularies via token compression. Jha et al. [11] characterized prompt compression achieving 10× reduction. For JSON-specific efficiency, StructuredRAG [21] and Struc-Bench [22] benchmark LLM JSON formatting quality. LLM function calling [1, 13] relies heavily on JSON both as input and output, making token efficiency directly impact system cost.

3 Method

3.1 Architecture Overview

JSON-TOKENIZER partitions its token ID space into three contiguous regions (fig. 2):

1. **Structural tokens** (IDs 0–31): Fixed tokens for JSON grammar elements and control symbols. Every grammar character receives exactly one token—never split or merged with adjacent characters.

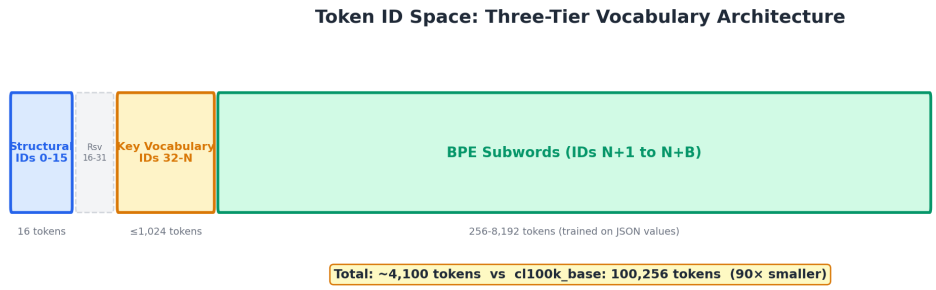

Figure 2: **Three-tier vocabulary architecture.** Token ID space is partitioned into structural tokens (blue), learned key vocabulary (gold), and BPE subwords (green). For a tokenizer with $M = 46$ learned keys and $B = 4{,}096$ BPE tokens, the total vocabulary is $32 + 46 + 4{,}096 = 4{,}174$ compared to `cl100k_base`'s 100,256.

2. **Key vocabulary** (IDs 32–N): Learned tokens for JSON keys that appear $\geq k$ times in training data, up to a maximum of M keys. Each key maps to a single token regardless of string length.
3. **BPE subwords** (IDs N+1–N+B): A byte-pair encoding vocabulary trained specifically on JSON value strings, capturing domain-specific patterns in the value distribution.

3.2 Structural Tokens

We define 16 structural tokens covering all JSON grammar elements:

ID	Token	Purpose
0	`[PAD]`	Padding for batched sequences
1	`[START]`	Document start marker
2	`[END]`	Document end marker
3	`{`	Object start
4	`}`	Object end
5	`[`	Array start
6	`]`	Array end
7	`:`	Key-value separator
8	`,`	Element separator
9	`null`	Null literal
10	`true`	Boolean true
11	`false`	Boolean false
12	`[STR]`	String value delimiter
13	`[NUM]`	Number value prefix
14	`[KEY]`	Unknown key prefix
15	`[UNK]`	Unknown token

Table 1: Structural token assignments. IDs 16–31 are reserved for future extensions.

A critical design choice: boolean values (`true`, `false`) and `null` are structural tokens, not BPE-encoded strings. This ensures they always consume exactly one token. In `cl100k_base`, `true` is one token, but `false` is one or two tokens depending on context—an inconsistency eliminated by our design.

3.3 Key Vocabulary Learning

During training, the tokenizer extracts all JSON keys and counts their frequencies:

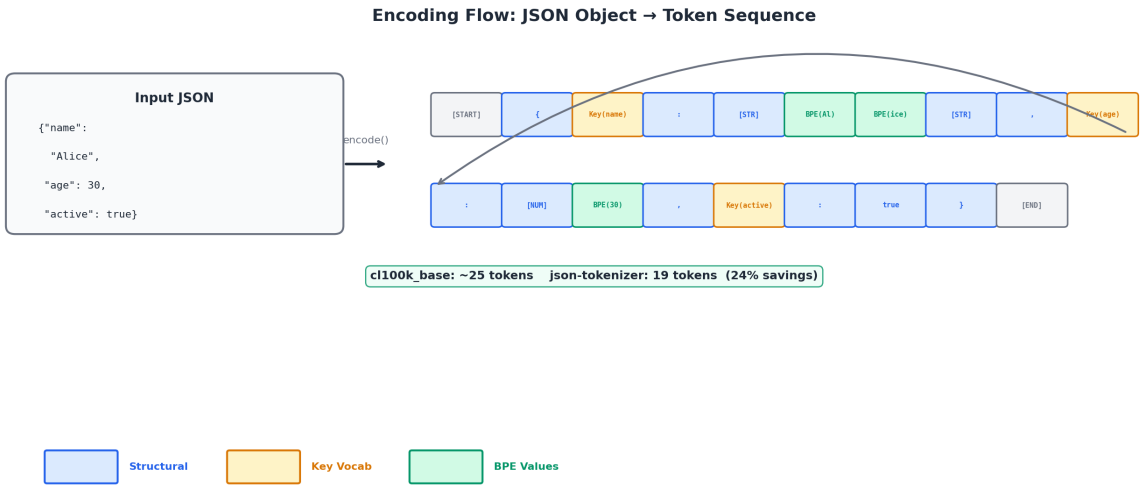

Figure 3: **Encoding flow.** A JSON object is recursively decomposed: the tokenizer dispatches by value type, wrapping objects in `{`/`}`, arrays in `[`/`]`, strings in `[STR]` delimiters, and numbers with `[NUM]` prefix. Keys in the vocabulary map to single tokens; others fall back to BPE with `[KEY]` prefix.

procedure LEARNKEYVOCAB($\mathcal{D}, k_{\min}, M$)
  counts $\leftarrow$ Counter()
  for obj $\in \mathcal{D}$ **do**
    counts.update(extract_keys(obj)) ▷ Recursive extraction
  end for
  keys $\leftarrow$ top_M_keys_with_freq $\geq k_{\min}$
  return $\{k \mapsto 32 + i \mid (i, k) \in \text{enumerate(keys)}\}$
end procedure

Keys that appear $\geq k_{\min}$ times receive dedicated single tokens. When encoding, a key like `"created_at"` that would require 3–4 BPE tokens in `cl100k_base` maps to a single `Key(created_at)` token. Keys not in the vocabulary fall back to BPE encoding with a `[KEY]` prefix.

3.4 BPE Training on Value Distributions

Value strings are collected during training and used to train a byte-level BPE model. The pre-tokenization pattern follows the GPT-2 tokenizer [17]:

Listing 1: Pre-tokenization pattern for JSON values.

```
pattern = r"'s|'t|'re|'ve|'m|'ll|'d| ?[a-zA-Z_]+| ?[0-9]+| ?[^\s\w]+|\s+|."
```

The BPE trainer uses an incremental pair-counting algorithm with a pair→word index for efficient merges, avoiding the naive $O(|V|^2)$ cost per merge step. Merges are applied until the vocabulary reaches the target size B or no pair exceeds the minimum frequency threshold.

3.5 Encoding and Decoding

Encoding proceeds recursively from the JSON root (fig. 3):

- **Objects:** `OBJ_START` + key-colon-value triples separated by `COMMA` + `OBJ_END`
- **Arrays:** `ARR_START` + values separated by `COMMA` + `ARR_END`

- **Strings:** `STR_DELIM` + BPE tokens + `STR_DELIM`
- **Numbers:** `NUM_PREFIX` + BPE tokens
- **Booleans:** `TRUE` or `FALSE` (single token)
- **Null:** `NULL` (single token)

A key implementation detail: Python's `bool` is a subclass of `int`, so the encoder must check for booleans *before* integers to avoid encoding `True` as the number 1.

Type-prefixed encoding (`[STR]`, `[NUM]`) ensures lossless roundtrip by disambiguating the JSON type of each value during decoding. The string `"42"` encodes as `[STR] BPE(42) [STR]`, while the number 42 encodes as `[NUM] BPE(42)`—distinct token sequences for distinct JSON values.

Decoding is a deterministic recursive-descent parse of the token stream, mirroring the encoding structure. Every structural token acts as an unambiguous parse delimiter, making the decoding grammar LL(1).

4 Experimental Setup

4.1 Datasets

We evaluate on five dataset classes spanning the spectrum from schema-heavy to prose-heavy JSON (table 2):

Dataset	Type	Train	Test	Characteristics
GeoJSON Cities	Structured	3,000	1,000	Identical schema, coordinate values
Telemetry Logs	Structured	3,000	1,000	13 identical keys, timestamps/IPs
K8s Manifests	Structured	300	200	Deeply nested, repeated keys
Alpaca	Prose-heavy	5,000	1,000	Instruction-tuning, long English text
Glaive Func. Call	Mixed	5,000	1,000	Function call JSON + natural language

Table 2: Evaluation datasets. "Structured" datasets have high schema repetition; "Prose-heavy" datasets contain primarily natural language values.

Additionally, we use synthetic payloads for controlled experiments: API user responses (100 objects), log entries (100 objects), product catalogs (50 objects), configuration variants (20 objects), and nested structures (depths 3–4).

4.2 Baseline

Our baseline is `cl100k_base` [16], the tokenizer used by GPT-4 and ChatGPT with a vocabulary of 100,256 tokens. We use the `tiktoken` library for baseline measurements. Token counts are computed on the canonical `json.dumps()` representation of each payload.

4.3 Configuration

Unless otherwise specified, we train JSON-TOKENIZER with 4,096 BPE vocabulary tokens, 512 maximum key vocabulary entries, and minimum key frequency $k_{\min} = 1$. For the K8s dataset, we use 2,048 BPE tokens due to smaller training corpus size.

Dataset	json-tokenizer	cl100k_base	Savings	Roundtrip
GeoJSON Cities (1K objects)	110,490	119,786	**+7.8%**	1000/1000
Telemetry Logs (1K objects)	116,925	123,676	**+5.5%**	1000/1000
K8s Manifests (200 objects)	67,704	67,249	−0.7%	200/200
Prose-heavy (expected to lose):				
Alpaca Instruction-Tuning	214,465	169,987	−26.2%	—
Glaive Function Calling	587,772	556,022	−5.7%	—

Table 3: **Individual object encoding results.** Savings are positive when JSON-TOKENIZER uses fewer tokens. Structured data yields 5–8% savings; prose-heavy data favors `cl100k_base` as expected. K8s manifests break even—high nesting depth but lower key repetition per object.

4.4 Metrics

- **Token count:** Total tokens produced for test objects, individually and as JSON arrays.
- **Savings:** $(1 - \text{ours}/\text{cl100k}) \times 100\%$.
- **Roundtrip fidelity:** Fraction of test objects where `json.loads(decode(encode(obj)))` exactly equals `json.loads(json.dumps(obj))`.
- **Vocabulary size:** Total vocabulary $|V| = 32 + |V_{\text{keys}}| + |V_{\text{BPE}}|$.
- **Characters per token:** Average JSON string length divided by token count.

5 Results

5.1 Structured Data: The Sweet Spot

table 3 shows the primary results on structured JSON datasets. JSON-TOKENIZER achieves consistent savings on schema-repetitive data while maintaining perfect roundtrip fidelity.

5.2 Batch Encoding: Where Repetition Compounds

JSON arrays amplify key repetition. When encoding objects as a single JSON array, savings increase because the key vocabulary is amortized across all objects while `cl100k_base` re-encodes each key from scratch (table 4).

Dataset	Batch	json-tokenizer	cl100k_base	Savings
GeoJSON Cities	500	54,784	60,424	**+9.3%**
Telemetry Logs	500	57,958	61,832	**+6.3%**
K8s Manifests	100	33,689	33,638	−0.2%

Table 4: **Batch encoding results.** Encoding objects as JSON arrays increases savings due to key vocabulary amortization. GeoJSON batch savings reach 9.3%.

5.3 Synthetic Structured Payloads

table 5 shows results on controlled synthetic payloads. Configuration objects and deeply nested structures—where schema is highly repetitive relative to content—show the largest gains (12–15%).

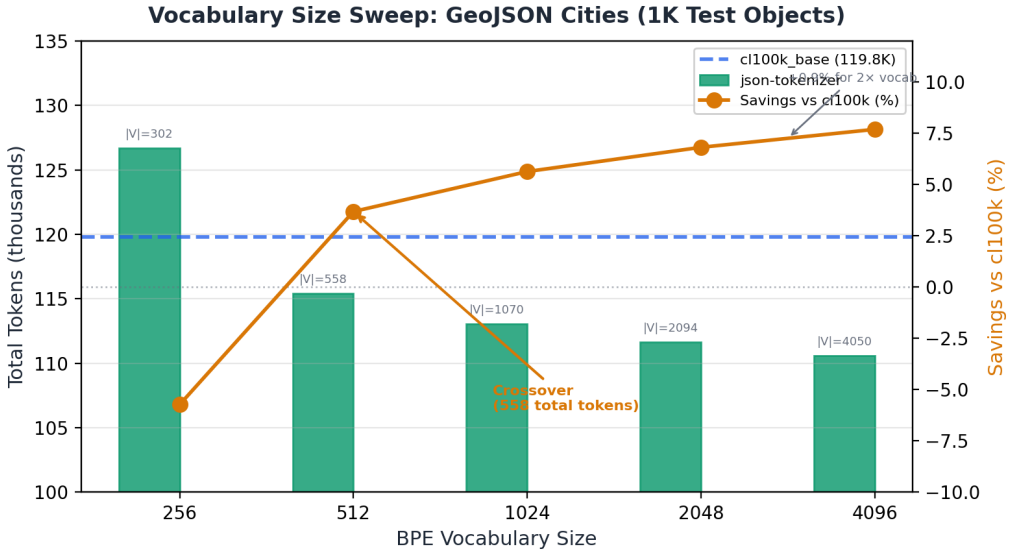

Figure 4: **Diminishing returns in BPE vocabulary size.** Token savings plateau after ∼2K BPE tokens, indicating that the key vocabulary (not BPE) provides the dominant compression mechanism for structured JSON.

Payload Type	json-tokenizer	`cl100k_base`	Savings	Roundtrip
API Users (100 objects)	7,557	7,699	**+1.8%**	OK
Log Entries (100)	6,414	6,826	**+6.0%**	OK
Product Catalog (50)	3,483	3,651	**+4.6%**	OK
Config Variants (20)	1,516	1,729	**+12.3%**	OK
Nested Structure (3×3)	564	657	**+14.2%**	OK
Nested Structure (4×2)	363	426	**+14.8%**	OK
Overall	20,041	21,132	**+5.2%**	All OK

Table 5: **Synthetic structured payload results.** Payloads with high schema-to-content ratio (configs, nested structures) show the largest gains.

5.4 Vocabulary Size Sensitivity

fig. 4 and table 6 show how token savings vary with BPE vocabulary size on the GeoJSON dataset (1,000 test objects).

BPE Size	Keys	Total $\|V\|$	json-tokenizer	`cl100k_base`	Savings
256	46	302	126,668	119,808	−5.7%
512	46	558	115,398	119,808	**+3.7%**
1,024	46	1,070	113,063	119,808	**+5.6%**
2,048	46	2,094	111,647	119,808	**+6.8%**
4,096	46	4,050	110,599	119,808	**+7.7%**

Table 6: **Vocabulary size sweep** on GeoJSON Cities. With only 558 total tokens (46 keys + 512 BPE), JSON-TOKENIZER already beats `cl100k_base`'s 100,256-token vocabulary. Doubling from 2K to 4K BPE yields only +0.9% additional savings.

The crossover point occurs at approximately 512 BPE tokens. Below this threshold, the BPE vocabulary is too small to efficiently encode value content, and `cl100k_base`'s 100K-token

vocabulary dominates. Above 2K tokens, returns diminish sharply: the 2K→4K doubling yields only +0.9% additional savings. This demonstrates that *the key vocabulary, not BPE, is the primary compression mechanism* for schema-repetitive JSON.

5.5 Where `cl100k_base` Wins

On prose-heavy JSON—where values are primarily English text paragraphs—`cl100k_base` substantially outperforms JSON-TOKENIZER (table 3). The Alpaca instruction-tuning dataset shows a 26.2% penalty, and Glaive function calling shows 5.7%.

This is expected and intentional: `cl100k_base`'s 100K-token vocabulary was trained on billions of words of English text, giving it highly optimized merge rules for natural language. Our BPE, trained on JSON value distributions with at most 4K tokens, cannot compete on English prose. We consider this an honest characterization of the method's scope, not a failure: JSON-TOKENIZER targets machine-to-machine JSON, not instruction-tuning corpora.

6 Analysis

6.1 Where Do the Savings Come From?

The savings decompose into two sources:

Key vocabulary compression. In GeoJSON, each object has ∼15 keys. With `cl100k_base`, keys like `"geometry"` (2 tokens), `"coordinates"` (1 token), `"properties"` (2 tokens), and `"created_at"` (3–4 tokens) sum to ∼20–25 tokens per object just for key encoding. With JSON-TOKENIZER, each key is exactly 1 token: 15 tokens per object. Over 1,000 objects, this saves 5,000–10,000 tokens.

Structural consistency. In `cl100k_base`, the token for `{` varies depending on preceding context (whitespace, newlines). Our encoder always uses the same structural token ID, eliminating context-dependent fragmentation.

6.2 Why K8s Manifests Break Even

Kubernetes manifests have deep nesting (4–6 levels) but fewer total keys per object relative to their overall size, and key names like `"apiVersion"`, `"metadata"`, `"spec"` are short enough that `cl100k_base` encodes them efficiently (1 token each). The structural overhead of our type-prefixed values (`[STR]`...`[STR]` delimiters) approximately offsets the key vocabulary savings.

6.3 Scaling Behavior

The batch encoding results (table 4) reveal that savings scale with array size because the fixed overhead of the key vocabulary is amortized. In the limit, for n objects with k keys each, our encoding uses $\sim k \cdot n$ tokens for all keys while `cl100k_base` uses $\sim c \cdot k \cdot n$ tokens (where $c \approx 2$–4 is the average tokens-per-key in `cl100k_base`). The savings approach $(1 - 1/c)$ asymptotically.

6.4 Computational Overhead

Training JSON-TOKENIZER on 3,000 GeoJSON objects with 4,096 BPE vocabulary completes in <1 second on a single CPU core. Encoding throughput is lower than `tiktoken` (which is implemented in Rust) but sufficient for offline preprocessing. The tokenizer is designed for training-time data preparation rather than real-time inference.

7 Discussion

Practical applications. JSON-TOKENIZER is best suited for scenarios where (1) JSON schema is known at training time, (2) the same schema is repeated across many objects, and (3) token budget directly impacts cost. Candidate applications include:

- **LLM structured output:** Function calling and tool-use responses [1, 13] generate JSON that follows fixed schemas, making key vocabulary compression directly applicable.
- **API response caching:** Paginated API responses are JSON arrays of identical-schema objects—the optimal scenario for our batch encoding.
- **Observability log ingestion:** Billions of JSON log lines per day [5] follow fixed schemas; even 5% token savings at scale represents significant cost reduction.

Limitations. Several limitations restrict the current system's applicability:

1. **Domain-specific training required.** The tokenizer must be trained on representative data from the target domain. A tokenizer trained on GeoJSON will not compress telemetry logs effectively. This is a feature (domain optimization) and a limitation (deployment complexity).
2. **No HuggingFace `PreTrainedTokenizer` compatibility.** The tokenizer cannot be plugged into existing transformer training pipelines without adapter code.
3. **ASCII-only key matching.** The current regex for key identification (`[a-zA-Z_]`) does not support Unicode key names, limiting applicability to non-Latin JSON schemas.
4. **No streaming.** The entire JSON document must fit in memory for encoding.
5. **Prose-heavy degradation.** As demonstrated, the tokenizer is unsuitable for JSON with predominantly natural language values.

Relationship to grammar-constrained decoding. Structure-aware tokenization and grammar-constrained decoding [7, 8] are complementary. A structure-aware tokenizer could provide the token vocabulary that a grammar-constrained decoder masks over, potentially simplifying the pushdown automaton by aligning token boundaries with grammar rules. We leave this integration to future work.

Toward schema-aware tokenization. The current system learns key vocabulary from data. A natural extension is *schema-aware* tokenization: given a JSON Schema, pre-allocate the key vocabulary from the schema definition, enabling zero-shot compression without training data. Combined with grammar-constrained decoding, this could enable end-to-end schema-aware LLM pipelines.

8 Conclusion

We presented JSON-TOKENIZER, a structure-aware tokenizer that exploits JSON's grammar and schema repetition to achieve 5–15% token savings over `cl100k_base` with a 90× smaller vocabulary. The three-tier architecture—structural tokens, learned key vocabulary, BPE for values—provides a principled decomposition that matches the entropy characteristics of JSON's components. All encodings are verified lossless across 4,200+ test objects.

The vocabulary size sweep reveals a key insight: with only 558 total tokens (46 keys + 512 BPE), JSON-TOKENIZER already outperforms a 100,256-token general-purpose vocabulary on schema-repetitive data. This suggests that for structured formats, *a small vocabulary aligned with the data's grammar can outperform a vastly larger vocabulary learned from raw text.*

The tokenizer is openly available at https://github.com/anthony-maio/json-tokenizer with trained models, benchmarks, and CLI tooling.

References

[1] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, G. P. Bhargav, et al. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. *arXiv preprint arXiv:2407.00121*, 2024. URL https://arxiv.org/abs/2407.00121.

[2] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding LLMs the right way: Fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, 2024. URL https://arxiv.org/abs/2403.06988.

[3] Michael J Bommarito, Daniel Martin Katz, and Jillian Bommarito. KL3M tokenizers: A family of domain-specific and character-level tokenizers for legal, financial, and preprocessing applications. *arXiv preprint arXiv:2503.17247*, 2025. URL https://arxiv.org/abs/2503.17247.

[4] Kaj Bostrom and Greg Durrett. Byte pair encoding is suboptimal for language model pretraining. In *Findings of the Association for Computational Linguistics: ACL 2020*, pages 50–55, 2020. doi: 10.18653/v1/2020.findings-acl.4. URL https://aclanthology.org/2020.findings-acl.4/.

[5] Qian Cheng, Amrita Saha, Wenzhuo Yang, Chenghao Liu, Doyen Sahoo, and Steven Hoi. LogAI: A library for log analytics and intelligence. *arXiv preprint arXiv:2301.13415*, 2023. URL https://arxiv.org/abs/2301.13415.

[6] Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. Getting the most out of your tokenizer for pre-training and domain adaptation. *arXiv preprint arXiv:2402.01035*, 2024. URL https://arxiv.org/abs/2402.01035.

[7] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024. URL https://arxiv.org/abs/2411.15100.

[8] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. *arXiv preprint arXiv:2305.13971*, 2023. URL https://arxiv.org/abs/2305.13971. Proceedings of EMNLP 2023.

[9] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. JSONSchemaBench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*, 2025. URL https://arxiv.org/abs/2501.10868.

[10] Saibo Geng, Nathan Ranchin, Yunzhen Yao, Maxime Peyrard, Chris Wendler, Michael Gastpar, and Robert West. zip2zip: Inference-time adaptive vocabularies for language models via token compression. *arXiv preprint arXiv:2506.01084*, 2025. URL https://arxiv.org/abs/2506.01084.

[11] Siddharth Jha, Lutfi Eren Erdogan, Sehoon Kim, Kurt Keutzer, and Amir Gholami. Characterizing prompt compression methods for long context inference. *arXiv preprint arXiv:2407.08892*, 2024. URL https://arxiv.org/abs/2407.08892.

[12] Kayvan Karim, Hani Ragab Hassen, and Hadj Batatia. Innovative tokenisation of structured data for LLM training. *arXiv preprint arXiv:2508.01685*, August 2025. URL https://arxiv.org/abs/2508.01685.

[13] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023. URL https://arxiv.org/abs/2312.04511.

[14] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012/.

[15] Yinxi Li, Yuntian Deng, and Pengyu Nie. TokDrift: When LLM speaks in subwords but code speaks in grammar. *arXiv preprint arXiv:2510.14972*, 2025. URL https://arxiv.org/abs/2510.14972.

[16] OpenAI. tiktoken: A fast BPE tokeniser for use with OpenAI's models, 2023. URL https://github.com/openai/tiktoken.

[17] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[18] Thomas Rückstieß, Alana Huang, and Robin Vujanic. ORIGAMI: A generative transformer architecture for predictions from semi-structured data. *arXiv preprint arXiv:2412.17348*, December 2024. URL https://arxiv.org/abs/2412.17348.

[19] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, 2016. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

[20] Ninglu Shao, Shitao Xiao, Zheng Liu, and Peitian Zhang. Flexibly scaling large language models contexts through extensible tokenization. *arXiv preprint arXiv:2401.07793*, 2024. URL https://arxiv.org/abs/2401.07793.

[21] Connor Shorten, Charles Pierse, Thomas Benjamin Smith, Erika Cardenas, Akanksha Sharma, John Trengrove, and Bob van Luijt. StructuredRAG: JSON response formatting with large language models. *arXiv preprint arXiv:2408.11061*, 2024. URL https://arxiv.org/abs/2408.11061.

[22] Xiangru Tang, Yiming Zong, Yilun Zhao, Arman Cohan, and Mark Gerstein. Struc-Bench: Are large language models really good at generating complex structured data? *arXiv preprint arXiv:2309.08963*, 2023. URL https://arxiv.org/abs/2309.08963.

[23] Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023. URL https://arxiv.org/abs/2307.09702.