

# UTF-8 Plumbing: Byte-level Tokenizers Unavoidably Enable LLMs to Generate Ill-formed UTF-8

Preston Firestone    Shubham Ugare    Gagandeep Singh    Sasa Misailovic  
University of Illinois Urbana-Champaign, USA

## Abstract

Subword tokenization segments input text according to a pre-defined vocabulary to feed it into a language model; the language model, in turn, generates a sequence made from this same vocabulary. The members of the vocabulary can be built of code points or bytes. Using code points means that all members of the vocabulary are valid UTF-8 characters. However, it also requires thousands of initial members to achieve acceptable coverage of inputs. Beginning with bytes, on the contrary, avoids out-of-vocabulary errors with only 256 initial members of the vocabulary, but the members of the vocabulary and sequences of them are not guaranteed to be valid UTF-8. Sequences that are not valid UTF-8 break code that assumes its input to be valid UTF-8. Applications of language models must account for the breakage thereby introduced. In this paper, we formalize tokenization using monoid theory and prove that tokenizers whose vocabularies contain tokens that are ill-formed UTF-8 can always produce sequences that are ill-formed UTF-8. We show formally that attempting to incrementally convert tokens back to a string and interpret the results as UTF-8 gives different results than converting the whole sequence of tokens at once. This formal result predicts real-world bugs: we evaluate mitigations for the problem identified and provide case studies of major foundation models and constrained generation systems.

## 1 Introduction

Large language models (LLMs) compute the probability of a given sequence of tokens drawn with replacement from a finite vocabulary. Tokenization is the process of cutting an input string into tokens such that each token is a member of a predefined vocabulary. Traditionally, tokenization used prior knowledge of the language, such as words, morphemes, or phonemes, to separate the text into useful elements (Grefenstette & Tapanainen, 1994; Palmer, 2000). Contemporary natural language processing has moved to using “subword” tokenization, where the vocabulary is derived automatically from a corpus of texts rather than programmed manually (Mielke et al., 2021).

The text the tokenizer works with is stored in computers as bytes. In order to interpret these bytes as characters meaningful to humans, we must define an encoding scheme mapping between a sequence of bytes and a sequence of characters. The predominant scheme is called UTF-8, regulated by Unicode Technical Committee (2025, §3.4, 3.9.3, 3.10). As of 2025, UTF-8 is used by 98.5% of all websites (W3Techs, 2025) and is required by the W3C for user agents (van Kesteren, 2024, §4.2).

Figure 1 shows two examples of strings as characters, bytes, and tokens, from different languages and models. Contemporary language models’ tokenizers typically work with the bytes on the bottom row of Figure 1 rather than the characters on the top row. Abstracting away the bytes to focus on the characters is an instance of a “leaky” abstraction, a common issue in software engineering: the information that we tried to hide by an abstraction ends up being exposed (often as we need to handle errors) (see Spolsky, 2002; Kiczales et al., 1992; Kiczales, 1991).

अ	ग	ॐ	न	ः	म	ः	ळ	ॐ
⌈E0 A4⌋⌈85⌋	⌈E0 A4⌋⌈97⌋	⌈E0 A5 8D	E0 A4⌋⌈A8⌋	⌈E0 A4 BF	E0 A4⌋⌈AE⌋	⌈E0 A5 80⌋	⌈E0 A4⌋⌈B3⌋	⌈E0 A5 87⌋

(a) Characters and bytes representing the opening of the *Rigveda* (Sanskrit: ऋग्वेद, “अग्निमीळे”, transliterated *agnim īle* (“I praise Agni [fire]”), tokenized according to `c1100k_base`.

Г	р	а	д	г	р	а	д	и	л	а	
⌈D0 93⌋	⌈D1 80	D0 B0	D0 B4⌋	⌈20	D0 B3⌋	⌈D1 80	D0 B0	D0 B4⌋	⌈D0 B8	D0 BB	D0 B0⌋

(b) The first two words of *The Building of Skadar* (Serbian Cyrillic: Зиданье Скадра), “Град градила”, transliterated *Grad gradila* (“The city was built”), tokenized according to Qwen3’s vocabulary.

Figure 1: Examples of UTF-8 encoding. In each subfigure, the top row contains the individual Unicode characters that compose the string (see Unicode Technical Committee, 2025, §2.11), and the bottom row contains the UTF-8 bytes that encode each character in the string; each byte is represented by two hexadecimal numerals. The squiggles  $\wr$  indicate the boundaries between tokens.

Concretely, concatenating tokens made up of bytes does not always result in sequences that can be successfully interpreted as characters. In Figure 1a, the first token,  $\wr E0 A4\wr$ , does not encode any character by itself; only when combined with the second token  $\wr 85\wr$  can the resulting three bytes `E0 A4 85` be interpreted as the character अ. Since language models generate sequences made up of the tokens in their vocabulary rather than generating bytes directly, programs that interact with sequences generated by a language model must handle byte sequences that cannot be interpreted as characters.

**Contributions.** In this paper, we make the following contributions:

- ★ Introduce a formal framework for tokenization based on monoid theory.
- ★ Prove, using our formalism, that vocabularies containing tokens that cannot be interpreted as characters can always produce sequences that cannot be interpreted as characters.
- ★ Show that interpreting bytes as UTF-8 characters is not a homomorphism and formally represent common mitigation strategies in the framework of monoids.
- ★ Classify popular language models according to the type of tokenizer they use.
- ★ Fix failures in constrained generation.

## 2 Background on monoids

To treat tokenizers abstractly, we introduce a formalism based on monoids, common in the literature on combinatorics on words (Sakarovitch, 2009; Lothaire, 1997) but to our knowledge new to natural language processing.

**(Free) monoids.** The basic component of our abstraction is the monoid.

**Definition 1** (Monoid). A monoid is a triple composed of a set of elements, a binary operation on members of that set, and an identity element. We refer to monoids by Greek capital letters ( $\Sigma$ ), their members by subscripted lowercase Latin letters ( $s_0, s_1 \in \Sigma$ ), the binary operation as a subscripted dot ( $\cdot_\Sigma$ ), and the identity element as a subscripted epsilon ( $\epsilon_\Sigma$ ). A **sequence** of members of a monoid is referred to by a lowercase Greek letter ( $\sigma = s_0 \cdot_\Sigma s_1$ ). Subscripts and binary operations are left out when context makes the meaning clear.

Monoids’ binary operation is associative but not commutative. For all members of the monoid,

$$(s_1 \cdot_\Sigma s_2) \cdot_\Sigma s_3 = s_1 \cdot_\Sigma (s_2 \cdot_\Sigma s_3). \quad (1)$$

The identity element is a neutral element for the binary operation, such that for any sequence  $\sigma$

$$\sigma \cdot_\Sigma \epsilon_\Sigma = \epsilon_\Sigma \cdot_\Sigma \sigma = \sigma. \quad (2)$$

Set theoretical operations ( $\subset, \subseteq, =, \dots$ ) are defined between monoids that share a binary operation and an identity element.

An example monoid is a language, where the symbols that make up the language are the set of members  $s_0 \dots$ , concatenation the binary operation  $\cdot_\Sigma$ , and the empty string the identity element  $\epsilon_\Sigma$ . In subsequent sections we will reason about all the sequences that can be formed from a given monoid, which make up the free monoid of a given monoid.

**Definition 2** (Free monoid). The **free monoid** generated by a monoid  $\Sigma$ , denoted  $\Sigma^*$ , is the monoid of all sequences of finite length made of elements from  $\Sigma$ . The free monoid is like the Kleene closure over the set of members of  $\Sigma$ , except the free monoid only contains finite sequences.

In Section 3, we will use the free monoid to describe the set of all sequences that can be generated with a given vocabulary, and in Section 4 we will use a free monoid to describe all byte sequences that can be interpreted as a string.

**Properties of monoids and their free monoids.** We will need to compare the contents of the free monoids of two different monoids that share a binary operation and identity element; this will allow us to make claims about what kinds of sequences are and are not in a given free monoid.

**Lemma 1.** *If some monoid  $\Sigma$  contains a member not in some other free monoid  $\Delta^*$ , where  $\Sigma$  and  $\Delta^*$  share a binary operation and an identity element, then the free monoid  $\Sigma^*$  of the first monoid  $\Sigma$  also contains a members not in the other free monoid  $\Delta^*$ :*

$$\Sigma \not\subseteq \Delta^* \rightarrow \Sigma^* \not\subseteq \Delta^*.$$

*Proof.* Suppose for contradiction that there exist some  $\Sigma$  and  $\Delta$  such that  $\Sigma \not\subseteq \Delta^*$  and  $\Sigma^* \subseteq \Delta^*$ ; the existence of such a pair would be a counterexample to Lemma 1. For the set of members of  $\Sigma$  to not be a subset of  $\Delta^*$ , there must exist some member of  $\Sigma$  that is not in  $\Delta^*$ . That member of  $\Sigma$  is also in  $\Sigma^*$ , but that implies that  $\Sigma^* \not\subseteq \Delta^*$ . This contradiction shows that no counterexample to Lemma 1 exists, thereby proving that the implication holds for all monoids  $\Sigma$  and  $\Delta$  that share a binary operation and an identity element.  $\square$

**Mapping between monoids.** In order to describe the process of tokenization in Section 3, we will need to map sequences back and forth between different monoids. To that end we introduce the abstraction of a stochastic map (Gastaldi et al., 2024).

**Definition 3** (Stochastic map). A **stochastic map**  $\kappa$  from a monoid  $\Sigma$  to a monoid  $\Delta$  is a function from  $\Sigma$  to the set of probability distributions on  $\Delta$ .

Other works analyzing tokenizers (e.g. Zouhar et al., 2023b; Geng et al., 2024; van Antwerpen & Neubeck, 2024) assume that the tokenizer is deterministic or have to cope with the ways it might not be. Following Gastaldi et al. (2024) we use stochastic maps to ensure that our results apply to nondeterministic tokenizers, which might result in more robust models (Geh et al., 2025; Chai et al., 2024a; Hofmann et al., 2022). Similarly, in our notation we do not explicitly depict the probability distribution over tokenizations that  $\tau$  returns, because we do not examine the probability of individual tokenizations, but merely which tokenizations are or are not possible for a given input.

We introduce the property of homomorphism as a restriction on maps:

**Definition 4** (Homomorphism). A map  $\kappa$  from a monoid  $\Sigma$  to  $\Delta$  is a **homomorphism** if and only if, for all  $\sigma, \sigma'$  in  $\Sigma$ ,

$$\kappa(\sigma \cdot_\Sigma \sigma') = \kappa(\sigma) \cdot_\Delta \kappa(\sigma'). \quad (3)$$

We show in Section 3 that tokenizing a string is not a homomorphism and in Section 4 that interpreting bytes as a string is also not a homomorphism. We then discuss real-world mitigations (Section 5) and bugs that arise because tokenizers fail to be homomorphisms (Section 6).

### 3 Tokenization described with monoids

Using the tools we introduced, we can describe a tokenizer as a pair of stochastic maps between two monoids and prove that tokenizing is not a homomorphism.

**Binding and cutting monoids.** We need to describe, in terms of monoids, the process of cutting that a tokenizer does to its input. The system we introduce in this subsection is similar but not identical to the system of Berglund & van der Merwe (2023). See Section 7 for a comparison of our system with theirs.

**Definition 5** (Cut monoid). The **cut monoid** of a given monoid is a monoid each of whose members has been prepended and postpended by a squiggle  $\wr$ . The cut monoid of a given monoid  $\Sigma$  is indicated by a superscripted squiggle:

$$\Sigma^\wr = \{\wr\sigma\wr \mid \sigma \in \Sigma\}. \quad (4)$$

Where convenient to do so, we collapse adjacent squiggles  $\wr$  for legibility.

For convenience, we use the notation  $\Sigma^\circ$  to name a finite subset of some monoid  $\Sigma$ ; we call the operator  $\circ$  “bind”. All three operators, cutting ( $\wr$ ), binding ( $\circ$ ), and freeing ( $*$ ), can be applied in succession to a single monoid, which we depict by putting multiple superscripts on the capital Greek letter representing the monoid; the operations are applied in order from left to right.

We next illustrate our notation and define several monoids relevant in the rest of this paper. The monoid  $\Sigma^*$  is all possible finite sequences of members of  $\Sigma$ . The monoid  $\Sigma^{*\wr}$  is the set of all possible finite sequences of members of  $\Sigma$ , each sequence with a squiggle pre- and postpended to it—it represents the set of all possible tokens.  $\Sigma^{*\wr\circ}$  is a finite subset of  $\Sigma^{*\wr}$ —it represents the vocabulary of some tokenizer.  $\Sigma^{*\wr\circ*}$  contains all possible sequences of members of  $\Sigma^{*\wr\circ}$ —it represents all sequences that can be made using the vocabulary of a given tokenizer.

**What is a tokenizer?** Tokenization is cutting the input into tokens. We define it as a pair of mappings between two monoids  $\Sigma^*$  and  $\Sigma^{*\wr\circ*}$  that share a binary operation and identity element:

**Definition 6** (Tokenizer). We call a tokenizer over some free monoid  $\Sigma^*$  and some vocabulary  $\Sigma^{*\wr\circ}$  a pair of stochastic maps  $\tau : \Sigma^* \rightarrow \Sigma^{*\wr\circ*}$  and  $\kappa : \Sigma^{*\wr\circ*} \rightarrow \Sigma^*$ .

The map  $\tau$  cuts its argument such that each token of the cut sequence is a member of some the vocabulary  $\Sigma^{*\wr\circ}$ ;  $\kappa$  joins the cut sequence by removing all the cut operators  $\wr$ .  $\kappa$  is deterministic and works the same way for any vocabulary made from the same initial monoid, whereas the behavior of  $\tau$  depends on its vocabulary. Each tokenizer  $(\tau, \kappa)$  is parameterized by the vocabulary  $\Sigma^{*\wr\circ}$  it is defined over, but we omit this detail in the notation because we will only work with one vocabulary at a time. We will refer to passing some input through  $\tau$  as **tokenizing** and some input through  $\kappa$  as **detokenizing**.<sup>1</sup>

**Tokenizers and homomorphism.** In the paper, we will rely on the following property of tokenizers as we defined them in Definition 6:

**Theorem 1.** *Given a tokenizer  $(\tau, \kappa)$ ,  $\kappa$  is a homomorphism, but  $\tau$  is not.*

*Proof.  $\kappa$  is a homomorphism:* The result of concatenating two cut sequences and removing the cut marks is always identical to removing the cut marks and concatenating them: the cut symbols are removed and the resultant string is identical to its uncut version. We have shown that there are no counter examples to Equation 3 for  $\kappa$ .

*$\tau$  is not homomorphism:* Take some  $s_0 \cdots s_n \in \Sigma^*$  and cut it into some  $\wr s_0 \cdots s_{m-1} \wr s_m \cdot s_{m+1} \wr s_{m+2} \cdots s_n \wr$ . It is impossible to separately cut  $s_0 \cdots s_m$  and  $s_{m+1} \cdots s_n$  such that there is not a cut after  $s_m$  and before  $s_{m+1}$ ; therefore the concatenation of any separate cutting of the two sequences  $s_0 \cdots s_m$  and  $s_{m+1} \cdots s_n$  must include the subsequence  $s_m \wr s_{m+1}$ . Cutting the entire sequence  $s_0 \cdots s_n$  at once allows for sequences that do not contain  $s_m \wr s_{m+1}$ . We show that there exist counterexamples to Equation 3 for  $\tau$ .  $\square$

<sup>1</sup>Though the standard in the literature is to use “decode” for  $\kappa$ , we chose an alternative term “detokenizing” to make it clear that we are speaking of turning tokens into some sequence of members of the uncut monoid, a step we consider separately from the process of interpreting those bytes as a UTF-8 string.

Theorem 1 has already been proved by Geng et al. (2024), though they use a different approach from ours. We reprove the Theorem here to exercise our system and state it in our terms, and extensively use it in Section 4. Theorem 1 applies to any tokenizer that uses cutting, regardless of vocabulary, the order in which cuts are applied, and whether the tokenization is stochastic or not.

**Out-of-vocabulary errors.** Sometimes there does not exist any way to cut a sequence such that the resulting cut sequence is made up exclusively of members of the tokenizer’s vocabulary. Often, however, at least part of the input can be cut into members of the vocabulary. There are three basic ways to deal with sections of the input that cannot be represented in the vocabulary: failing entirely, dropping the unacceptable section, and replacing the section with something else. In terms of our formalism:

1. **Failing entirely:** declaring  $\tau$  to be undefined where its input cannot be mapped to  $\Sigma^{*\iota^*}$ ;
2. **Dropping an unacceptable section:** replacing the sequences between squiggles  $\wr$  that are not in  $\Sigma^{*\iota^*}$  with  $\wr\epsilon_\Sigma\wr$  (this is a special case of the replacement strategy where  $\eta = \epsilon$ );
3. **Replacing the section with other symbols:** mapping the objectionable bytes, according to some scheme, to one or more members of  $\Sigma^{*\iota^*}$  specially set aside for this purpose.<sup>2</sup>

A  $\tau$  that uses any of these mitigations will not be a homomorphism, because none of the mitigations avoids the counterexample given in the proof of Theorem 1. UTF-8 implementations often elect the third option, replacing ill-formed byte sequences with a “replacement character”, U+FFFD ([?](#)).

## 4 UTF-8 and monoids

**The UTF-8 encoding scheme.** In the UTF-8 encoding scheme (Unicode Technical Committee, 2025, §3.9.3), each character is paired with some byte sequence one to four bytes long; a single such byte sequence is called a UTF-8 **code unit**. Descriptions of the UTF-8 code units are found in Table 1; any sequence of bytes made up exclusively of concatenated, non-overlapping UTF-8 code units is a **well-formed** UTF-8 string. See Appendix B for more details about Unicode and UTF-8. The rest of this section will discuss UTF-8 using the monoid and tokenizer abstractions we introduced earlier. We begin by phrasing the encoding of characters as bytes and the interpretation of bytes as characters in terms of monoids.

First byte	Second byte	Third byte	Fourth byte
00..7F			
C2..DF	80..BF		
E0	A0..BF	80..BF	
E1..EC	80..BF	80..BF	
ED	80..9F	80..BF	
EE..EF	80..BF	80..BF	
F0	90..BF	80..BF	80..BF
F1..F3	80..BF	80..BF	80..BF
F4	80..8F	80..BF	80..BF

Table 1: Well-formed UTF-8 byte sequences (Unicode Technical Committee, 2025, §3.9.3). A range such as 80..BF should be read as an inclusive range from 80 through BF.

**Definition 7 (UTF-8).** We refer to the monoid of UTF-8 code units as  $Y$ , which is the  $B^{*\iota^*}$  specified in Table 1. The sequences of  $B^*$  that are not in  $Y^*$  are ill-formed UTF-8 sequences; the members of  $B^*$  that are in  $Y^*$  are well-formed.

Not all byte sequences are well-formed UTF-8; see Appendix B for examples. The existence of byte sequences that are ill-formed in UTF-8 motivates Theorem 2, which follows immediately from Lemma 1 and Definition 7.

**Theorem 2.** Any vocabulary  $B^{*\iota^*}$  that contains a byte sequence  $\beta$  that is ill-formed in UTF-8 ( $\beta \notin Y^*$ ) will be able to generate ill-formed sequences:

$$B^{*\iota^*} \not\subseteq Y^* \rightarrow B^{*\iota^*} \not\subseteq Y^*.$$


<sup>2</sup>See Unicode Technical Committee (2025, §3.9.6) and van Kesteren (2024, §4.1) for the standard algorithm for this replacement.

Due to Theorem 2, any language model’s vocabulary that contains byte tokens ill-formed in UTF-8 can generate token sequences that are ill-formed in UTF-8.

**Enforcing UTF-8 breaks homomorphism in tokenizers.** In Section 6 we will show impacts of assuming that the output of a language model is well-formed UTF-8. To prove that the problems we discuss are inevitable, we introduce the following theorem.


**Theorem 3.** *Interpreting a byte sequence as UTF-8 is not a homomorphism, but serializing a sequence of well-formed UTF-8 byte sequences is a homomorphism.*

*Proof.* Mapping between bytes and well-formed UTF-8 sequences is like tokenization, as in Definition 6. Mapping from well-formed UTF-8 sequences to a sequence of bytes means simply concatenating those bytes (Unicode Technical Committee, 2025, §3.10). Joining the UTF-8 code units into a byte string is naturally like a detokenizer  $\kappa$  from  $Y^*$  (i.e.  $B^{*\iota^*}$ ) to  $B^*$ . Mapping from a sequence of bytes to a well-formed UTF-8 sequence requires splitting that byte sequence into consecutive, adjacent UTF-8 code units. This is like a  $\tau$  from  $B^*$  to  $Y^*$  (i.e. to  $B^{*\iota^*}$ ). Theorem 3 follows directly by application of Theorem 1.  $\square$

Theorem 3 applies regardless of which of the strategies of Section 3 is used. In the next section, we will assume that decoding applies the third strategy discussed in Section 3, and we will refer to the member of  $Y$  set aside for ill-formed byte sequences as  $\eta$ . The number of  $\eta$ s used to replace ill-formed bytes in the sequence and their identity is variable. In Section 6 we will see examples where more than one distinct replacement character exists. See Unicode Technical Committee (2025, §3.9.6) for the standard algorithm used for replacement when decoding UTF-8. The process assumes a single  $\eta$ , the character U+FFFD “”, known as a *replacement character*. The number of  $\eta$ s introduced to suture any ill-formed sections of the byte sequence will not be relevant to the results of the next section, only that such a unique character exists.

## 5 Incrementally detokenizing ill-formed UTF-8 sequences

It is common for language model applications to stream or otherwise incrementally generate tokens, decoding and displaying the decoded tokens opportunistically. Interactive systems, for example, can display the intermediate steps of the generation to the user; hosted language models on a remote server might send tokens to the client as soon as they become available, rather than wait for generation to complete before sending anything downstream.

As we have shown in Section 4, certain combinations of tokens can be ill-formed UTF-8, causing a process that attempts to decode them to apply one of the coping strategies of Subsection 3. The decoding processes used in deployed tokenizers use the third strategy described at the end of Section 3: “replacing the [ill-formed] section with other symbols”, in particular they use the symbol U+FFFD  (“replacement character”); our formalism calls it  $\eta$ .

If the user attempts to interpret the byte tokens as UTF-8 as the tokens come in, then concatenate, the result will be different from what the user would have gotten had they waited for all the tokens before interpreting them all together. This problem was detected in serving engines for language models, which host language models and serve them through an API locally or over the network: these engines failed to generate correct text, because they incorrectly assumed that the conversion from bytes to well-formed UTF-8 was a homomorphism. This problem and a common mitigation strategy were introduced to Hugging Face TGI by a user’s issue (Hugging Face, 2025; 0x1997, 2023) and percolated thence to vLLM (Kwon et al., 2023; Yard1, 2023), OpenLLM (bentoml, 2025; jeffwang0516, 2023), and SGLang (Zheng et al., 2024; hnyls2002, 2024), all of which faced similar issues for the same reason. See Appendix C for a Python code from the serving engines.

---

**Algorithm 1** Incrementally generate using byte-level tokenizer.

---

**Require:**

- $\tau : B^* \rightarrow Y^*$ . Tokenize a sequence of bytes to a sequence of UTF-8 code units, replacing ill-formed bytes with  $\eta$ .
- $\kappa : B^{*\circ*} \rightarrow B^*$ . Detokenize from a sequence of byte tokens in some vocabulary  $B^{*\circ}$  to a sequence of bytes.
- $LM : B^{*\circ*} \rightarrow B^{*\circ}$ . A language model that takes in a sequence of tokens and returns a new token.
- $i$ : stateful integer initialized to 0.
- $j$ : stateful integer initialized to 0.

```

1: function ADVANCETOKEN( $\beta : B^{*\circ*}$ )
2:    $v : Y^* \leftarrow \tau(\kappa(\beta[i : j]))$ 
3:    $v' : Y^* \leftarrow \tau(\kappa(\beta[i : \text{LENGTH}(\beta)]))$ 
4:   if  $\text{LENGTH}(v') > \text{LENGTH}(v) \wedge$ 
5:      $v'[\text{LENGTH}(v') - 1] \neq \eta$  then
6:      $i \leftarrow j$ 
7:      $j \leftarrow \text{LENGTH}(\beta)$ 
8:     return  $v'[\text{LENGTH}(v) : \text{LENGTH}(v')]$ 
9:   else
10:  return  $\epsilon_Y$ 
1: function GENERATE
2:    $\beta : B^{*\circ*} \leftarrow \epsilon_B$ 
3:    $v : Y^* \leftarrow \epsilon_Y$ 
4:   loop
5:      $\beta \leftarrow \beta \cdot_B LM(\beta)$ 
6:      $v \leftarrow v \cdot_Y \text{ADVANCETOKEN}(v, \beta)$ 
7:     EMIT( $\beta, v$ )

```

---

application of these operations without destructively decoding the byte tokens such that further tokens cannot be successfully appended to the sequence.

Algorithm 1 cannot make concatenating UTF-8 tokens a homomorphism (consistent with Theorems 1 and 3). If one attempted to concatenate two sequences in  $Y^*$  that came out of the function GENERATE, one would always end up with a sequence in  $Y^*$ . But if the sequences in  $B^{*\circ*}$  that Algorithm 1 detokenizes and decodes as UTF-8 are not well-formed UTF-8, then the outputs of Algorithm 1 will remove or mangle the information the ill-formed bytes contained. For example, the text of Figure 1a contains twelve tokens. Decoding the first five with Algorithm 1 produces the UTF-8 code units “अग”; decoding the remaining seven produces “?मीळे”: concatenating produces “अग?मीळे”, not the expected “अग्नमीळे”. Dropping the ill-formed bytes, strategy 2 from Section 3, will not restore homomorphism: “अग” · “मीळे” is “अगमीळे”, not “अग्नमीळे”. Algorithm 1 is not always necessary: none of the Cyrillic characters in Figure 1b are split across more than one token, while several of the Devanagari characters Figure 1a are. Detokenizing and decoding this Cyrillic sequence is a homomorphism as a special case, because all of its tokens are well-formed; correctly handling the Devanagari sequence, on the other hand, requires special handling.

The information lost when decoding an ill-formed byte sequence (whether failing, dropping, or replacing) is always gone. Concatenating the decoded tokens will therefore produce different results from decoding the concatenated tokens wherever the split between tokens is not aligned with the split between code units. Algorithm 1 decodes the intermediate results of generation while correctly appending new tokens to those generated so far. The local variable  $v$  in the function GENERATE always contains the code units generated so

Algorithm 1 formalizes this mitigation strategy using the framework of monoids. The function ADVANCETOKEN opportunistically interprets byte tokens as UTF-8 sequences when the end of the sequence’s final token is aligned with the edge between two code points. Each time it is called, ADVANCETOKEN receives all the tokens generated so far, with the new token at the end of the sequence. If there is a new well-formed code unit at the end of the sequence of tokens, ADVANCETOKEN updates  $i$  and  $j$  to point to the last two positions in the token sequence where the token and code unit boundary lined up and returns the new successfully-decoded code units. The algorithm ignores the tokens before the  $i^{\text{th}}$  token, because appending bytes to well-formed UTF-8 cannot change the previous code units: well-formed UTF-8 code units do not overlap.

The outer function, GENERATE, shows how to exercise ADVANCETOKEN to emit the incrementally produced text as well-formed code units are completed. Both the byte tokens and text generated so far are stored in the local variables  $\beta$  and  $v$ , which enables the generating process to output text as it is generated without losing information about the exact bytes the model generated. Note the binary operations on lines 5 and 6 of the function GENERATE in Algorithm 1: the first concatenates bytes and the second UTF-8 code units. The purpose of Algorithm 1 is to enable the simultaneous

far; if the final code unit is the filler character  $\eta$  ( $\text{?}$ ), then  $v$  does not contain the final code unit. Because it appends in the tokens’ monoid  $B^{*\circ}$  as well as in the code units’ monoid  $Y$  (see lines 5 and 6 of the function `GENERATE`), Algorithm 1 is a partial fix for certain cases, but as we have seen, it has inherent limitations.

## 6 Sealing the leaks in UTF-8 decoding

Deployed tokenizers usually concatenate byte tokens and interpret them as if they were UTF-8; when the process reaches an ill-formed subsequence, it has the choices outlined in Section 3: failure, dropping, and replacement. Theorem 3 shows that if a vocabulary contains byte tokens that are ill-formed UTF-8, it will be able to produce sequences that are ill-formed UTF-8. We next classify the tokenizers of common foundation models and discuss coping with the impacts of Theorem 3 in constrained generation.

**Tokenization strategies of foundation models.** In Table 2, we classify several popular language models according to the style of tokenizer they use. We find two broad categories: byte-level, and byte-fallback. Byte-level tokenizers impose no constraints on the formation of tokens, which may or may not be valid in UTF-8. Since all non-ASCII characters are represented by more than one byte in UTF-8, any script that uses a character other than the 128 in ASCII could have its characters split across multiple tokens by a byte-level tokenizer.

Byte-fallback tokenizers require that all tokens in the vocabulary be valid in UTF-8, with the exception of 256 (or 243, excluding the 13 bytes that never appear in any UTF-8 code unit) tokens, one for each byte. These extra tokens are used to represent parts of the input that cannot otherwise be represented by tokens in the tokenizer’s vocabulary. Figure 2 gives an example of such a tokenization, where each of the tokens contains a valid UTF-8 string, except for the three single-byte tokens used to encode the rare character  $\text{⦿}$  (“multiocular o”, used in a single manuscript from the 15th century). Theorem 2 applies to both strategies equally, because the single-byte tokens are already ill-formed and so a sufficient condition for the theorem.

м	ного	⦿	чит	ї	й
<code>\D0 BC\</code>	<code>\D0 BD D0 BE D0 B3 D0 BE\</code>	<code>\EA \99 \AE\</code>	<code>\D1 87 D0 B8 D1 82\</code>	<code>\D1 97\</code>	<code>\D0 B9\</code>

Figure 2: Characters and bytes representing the Old Church Slavonic word  $\text{многo⦿читїй}$ , transliterated *mnogoočitii* (“many-eyed”), tokenized according to the vocabulary of Gemma-3.

Eight out of ten of the model families classified in Table 2 retain the same style of tokenization throughout their generations. Llama changes from SentencePiece (Kudo & Richardson, 2018) to a vocabulary derived from OpenAI’s `tiktoken` (OpenAI, 2025) for its third generation (Grattafiori et al., 2024). This change of dependencies resulted in a switch from byte-fallback (since SentencePiece enforces UTF-8 validity on its tokens by default, the only mitigation it offers for covering out-of-vocabulary inputs is byte-fallback) to byte-level. Phi-3, unlike the other Phi-series models, used Llama 2’s vocabulary; Phi-{1, 1.5} use vocabu-

Model	Tokenizer Type
OpenAI since GPT-2 (Radford et al., 2019; Brown et al., 2020; OpenAI et al., 2024)	Byte-level
Qwen, Qwen2.5, Qwen3 (Bai et al., 2023; Yang et al., 2025b;a)	Byte-level
Llama 1, 2 (Touvron et al., 2023a;b)	Byte-fallback
Llama 3 (Grattafiori et al., 2024)	Byte-level
Mistral, Mixtral (Jiang et al., 2023; 2024)	Byte-fallback
Gemma 1, 2, 3 (Mesnard et al., 2024; Gemma Team et al., 2024; 2025)	Byte-fallback
OLMo, OLMo 2 (Groeneveld et al., 2024; Team OLMo et al., 2025)	Byte-level
Phi-1, 1.5, 2, 4 (Gunasekar et al., 2023; Li et al., 2023; Javaheripi & Bubeck; Microsoft et al., 2025)	Byte-level
Phi-3 (Abdin et al., 2024)	Byte-fallback
Cohere R and R+ (Cohere, 2024)	Byte-level
Stable LM 1, 2 (Stability-AI; Bellagente et al., 2024)	Byte-level
CodeGen, CodeGen2 (Nijkamp et al., 2023b;a)	Byte-level

Table 2: Foundation models and the tokenizers they use.

aries from CodeGen; Phi-2’s documentation does not describe the tokenization in detail, but manual inspection of the publicly-available vocabulary reveals it to be byte-level; and Phi-4 uses one of OpenAI’s vocabularies from `tiktoken` (OpenAI, 2025), all of which are byte-level. As we discuss in Section 7, ablations are seldom performed on tokenizers, and none of the models in Table 2 were tested with more than one tokenizer. We presume that the reason for skipping ablations is the excessive cost of training multiple models of the same scale and architecture but with distinct tokenizations (and embeddings).

**Existing constrained generation systems and non-homomorphic tokenizers.** Constrained generation techniques are used to restrict language model outputs to adhere to specified rules. We examined various grammar-constrained generation systems and tested them to discover their behavior during partial generation decoding (Scholak et al., 2021; Poesia et al., 2022; Willard & Louf, 2023; Ugare et al., 2025b;a; Banerjee et al., 2025; Loula et al., 2025; Suresh et al., 2025). Among popular grammar-constrained generation tools, we found that Synchromesh (Poesia et al., 2022) and SynCode (Ugare et al., 2025b) encountered issues when grammars included non-ASCII characters such as emojis or mathematical symbols such as ‘ $\forall$ ’. Both tools use character-based parsers rather than byte-based ones, which created this vulnerability.

The way to fix this problem is to constrain generation at the level of bytes rather than characters. Attempting to constrain at the level of characters fails where the bytes of the token are ill-formed UTF-8. For example, the character  $\forall$ , used in Lean (Moura & Ullrich, 2021), might be tokenized `␣E2 88 ␣80␣`. If the model has generated the first token of the character, then the constrained generation algorithm must permit the second token as a continuation, even though neither of the tokens is well-formed. Constraining at the level of bytes rather than characters by using bytes as the input alphabet for lexers, DFAs, and so forth repairs this problem.

For the v0.3.0 release of SynCode, we replaced the previous character-level finite state machines with byte-level ones. To evaluate SynCode’s ability to handle non-ASCII Unicode characters, we conducted an experiment using an emoji generation task. We selected a subset of the TweetEval emoji dataset Barbieri et al. (2020), filtering for three common emoji classes. The task required the model to generate exactly one emoji character in response to a given tweet, adhering to a constrained grammar specification (see listing 4 in Appendix E). We evaluated this task across 100 examples from the TweetEval test set on two versions of SynCode: v0.2.0 which used character-level constraints, and the current version which implements byte-level constraints. Performance improvements are reported in Table 3: v0.2.0 crashed on all examples, whereas v0.3.0 successfully generated emojis for all examples.

Metric	SynCode v0.2.0	SynCode v0.3.0
Accuracy	0%	62%
Crash Rate	100%	0%

Table 3: Performance comparison between SynCode versions on emoji generation task

## 7 Related work

We direct the reader to Appendix A for more related work.

**Formal approaches to tokenization.** Following Sakarovitch (2009) and Lothaire (1997), we treat languages as monoids. Monoids have found use in the analysis of formal languages (Yang & Wu, 2023) but hardly at all in the study of natural languages. The work of Joachim Lambek on type grammar has led him to study natural language in terms of ordered monoids (Lambek, 1997; 2007), but his work came to our attention too late to significantly impact our work here.

**UTF-8 challenges in tokenization.** Rahman et al. (2024, §II.D) discuss the challenges of UTF-8, both because it encodes characters as sequences of varying length, and many characters are encoded by more than one byte. The latter challenge cannot be avoided unless one wants to limit one’s character space to 256 unique characters, but the former issue is

unique to UTF-8 and distinguishes it from other encoding schemes, UTF-16 and UTF-32, which encode all code points in two and four bytes respectively.

**Ablations.** To our knowledge, ablations on the relative efficacy of byte-fallback and byte-level tokenization have not been performed. Dagan et al. (2024) ablate on datasets, pre-tokenization schemes, and vocabulary size; Ali et al. (2024) ablate on datasets, tokenization algorithms (BPE v. Unigram) and implementations (Hugging Face v. SentencePiece). None directly address the tradeoffs between byte-level and byte-fallback tokenization.

**Glitch tokens.** “Glitch tokens” are under-trained tokens in the model’s vocabulary that can cause erratic behavior (Land & Bartolo, 2024; Geiping et al., 2024). They are distinct from the ill-formed UTF-8 sequences we discuss. Land & Bartolo (2024) discuss what they call “partial UTF-8” tokens, defined as “tokens representing byte sequences that cannot be converted to Unicode characters [decoded, in our terms] as they contain only part of the full UTF-8 encoding for a character”. This is a special case of our ill-formed tokens: ill-formed byte sequences need not contain any part of a UTF-8 encoding form. Ill-formed tokens are explicitly excluded from Land & Bartolo (2024)’s experiments because they “are not suitable for building verification prompts”, presumably because these prompts must be well-formed UTF-8 to work with existing interfaces (e.g. Hugging Face’s).

The byte-fallback tokens (see Section 6) that encode the bytes of ASCII characters are a source of glitchy tokens (see Geiping et al., 2024, Figure 23, in the Appendix). Geiping et al. (2024) exclude byte-fallback tokens from the in the main body of the paper that reports the glitchiest tokens in the Llama 2 vocabulary, because the tokens that represent the bytes of the ASCII characters cannot be the result of ordinary tokenization.

**Improbable bigrams.** Jang et al. (2024) examine pairs of tokens made up of tokens that are not well-formed UTF-8 but that, when concatenated, make a well-formed byte sequence. These tokens enable make attacks similar to those of Land & Bartolo (2024); Geiping et al. (2024), with the difference that Jang et al. (2024) examine pairs of tokens, whereas Land & Bartolo (2024); Geiping et al. (2024) examine individual tokens. These bigrams are well-formed and distinct from our examination of ill-formed sequences.

**Superscripted squiggles.** We borrow our notation of squiggles  $\wr$  for the division between tokens from Berglund & van der Merwe (2023), though we slightly redefine the notation of a superscripted squiggle: our cutting operation  $\wr$  describes the set of all possible tokens made from a given monoid, whereas the  $\Sigma^\wr$  of Berglund & van der Merwe (2023) is “the set of all tokenizations constructed from strings in  $\Sigma^*$ ”. Their  $\Sigma^\wr$  is equivalent to our  $\Sigma^{*\wr}$ . Also, we add binding ( $\circ$ ) and introduce the application of multiple superscripts to a single base monoid.

## 8 Conclusions and future work

The paper shows that UTF-8, tokens, and strings are leaky abstractions when generating text using language models. Rather than burying the issue, practitioners should accustom themselves to not expecting the inputs or outputs of their LLM system to be well-formed UTF-8. Implementers of systems and applications for language models should test their implementations on non-ASCII characters and ensure that they behave properly when an input or generated sequence is ill-formed UTF-8. Authors should be more precise when specifying the tokenization scheme their model uses when describing its architecture. We leave for future work surveying more foundation models, testing language model applications, expanding Algorithm 1 for concatenation in both directions rather than just appending (though we note that it behaves correctly when appending an arbitrarily large number of new tokens), and experimenting with multiple constrained generation algorithms.

All humans have the right to interact with computers in their own language and to be able to use computers to generate and manipulate text, with equal regard to all languages. This paper works toward that goal by clarifying some common issues present with large language models. It has addressed a leak in one of the abstractions used to work with language, but the general problem of how to process all natural languages uniformly remains open.

## Acknowledgments

We thank the anonymous reviewers for their comments. This research was supported in part by NSF Grants No. CCF-1846354, CCF-2313028, CCF-2238079, CCF-2316233, CNS-2148583, and an Amazon Research Award.

## Ethics statement

Davis & Suignard (2014) describes security issues that face systems that interact with Unicode as closely as language model applications and infrastructure must. They describe non-visual exploits such as buffer overflows during encoding or decoding, text comparison, ill-formed input bytes, including several exploits that are particular to UTF-8. Davis & Suignard (2014)'s visual exploits are based on visual spoofs, visually mistakable strings: these are two or more different sequences of code points that appear the same to the user (see Unicode Technical Committee (2025, §3.11) and Davis et al. (2024)). These would be tokenized differently by all tokenizers, because they are not the same byte or code point sequence and so cannot be represented by the same tokens.

Visual spoofs can be used for attacks similar to those recently studied by Geh et al. (2025), where models generated radically different responses to varying tokenizations of the same prompt; in some cases Geh et al. (2025) broke safety and alignment restrictions trained into models. A visual spoof would circumvent a mitigation Geh et al. (2025) suggest for their attack: providers of language models as a service could prohibit users from tokenizing their own texts and require them to submit well-formed UTF-8 in order to avoid adversarial tokenizations. Visual spoofs could produce the same effect of adversarial tokenization without the user needing to have direct control over the tokenization process.

When discussing how to decode UTF-8 sequences, the Unicode standard offers the following foreboding words: “[s]ilently ignoring ill-formed sequences is strongly discouraged because joining text from before and after the ill-formed sequence can cause the resulting text to take a new meaning. This result would be especially dangerous in the context of textual formats that carry embedded program code” (Unicode Technical Committee, 2025, C10). This warning is particularly apt for processes executing code that a language model generated.

## References

- 0x1997. Missing and garbled characters when streaming unicode text. GitHub Issue, 2023. URL <https://github.com/huggingface/text-generation-inference/issues/333>.
- Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL <https://arxiv.org/abs/2404.14219>.
- Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. Do all languages cost the same? tokenization in the era of commercial language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9904–9923, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.614. URL <https://aclanthology.org/2023.emnlp-main.614/>.
- Mehdi Ali, Michael Fromm, Klaudia Thellmann, Richard Rutmann, Max Lübbering, Johannes Leveling, Katrin Klug, Jan Ebert, Niclas Doll, Jasper Buschhoff, Charvi Jain, Alexander Weber, Lena Jurkschat, Hammam Abdelwahab, Chelsea John, Pedro Ortiz Suarez, Malte Ostendorff, Samuel Weinbach, Rafet Sifa, Stefan Kesselheim, and Nicolas Flores-Herr. Tokenizer choice for LLM training: Negligible or crucial? In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 3907–3924, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.247. URL <https://aclanthology.org/2024.findings-naacl.247/>.
- Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023. URL <https://arxiv.org/abs/2309.16609>.
- Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Crane: Reasoning with constrained llm generation. In *Forty-Second International Conference on Machine Learning*, 2025. URL <https://icml.cc/virtual/2025/poster/43624>.
- Francesco Barbieri, Jose Camacho-Collados, Leonardo Neves, and Luis Espinosa-Anke. Tweeteval: Unified benchmark and comparative evaluation for tweet classification, 2020. URL <https://arxiv.org/abs/2010.12421>.
- Marco Bellagente, Jonathan Tow, Dakota Mahan, Duy Phung, Maksym Zhuravinskyi, Reshith Adithyan, James Baicoianu, Ben Brooks, Nathan Cooper, Ashish Datta, Meng Lee, Emad Mostaque, Michael Pieler, Nikhil Pinnaparju, Paulo Rocha, Harry Saini, Hannah Teufel, Niccolo Zanichelli, and Carlos Riquelme. Stable lm 2 1.6b technical report, 2024. URL <https://arxiv.org/abs/2402.17834>.
- bentoml. OpenLLM. GitHub Repository, 2025. URL <https://github.com/bentoml/OpenLLM>.
- Martin Berglund and Brink van der Merwe. Formalizing bpe tokenization. *Electronic Proceedings in Theoretical Computer Science*, 388:16–27, September 2023. ISSN 2075-2180. doi: 10.4204/eptcs.388.4. URL <http://dx.doi.org/10.4204/EPTCS.388.4>.
- Kaj Bostrom and Greg Durrett. Byte pair encoding is suboptimal for language model pre-training. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for*

- Computational Linguistics: EMNLP 2020*, pp. 4617–4624, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.414. URL <https://aclanthology.org/2020.findings-emnlp.414/>.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL <https://arxiv.org/abs/2005.14165>.
- Yekun Chai, Yewei Fang, Qiwei Peng, and Xuhong Li. Tokenization falling short: On subword robustness in large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 1582–1599, Miami, Florida, USA, November 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.86. URL <https://aclanthology.org/2024.findings-emnlp.86/>.
- Yekun Chai, Qingyi Liu, Jingwu Xiao, Shuohuan Wang, Yu Sun, and Hua Wu. Autoregressive pre-training on pixels and texts. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3106–3125, Miami, Florida, USA, November 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.182. URL <https://aclanthology.org/2024.emnlp-main.182/>.
- Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. Canine: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 01 2022. ISSN 2307-387X. doi: 10.1162/tacl\_a\_00448. URL [https://doi.org/10.1162/tacl\\_a\\_00448](https://doi.org/10.1162/tacl_a_00448).
- Marco Cagnetta and Naoaki Okazaki. Tokenization as finite-state transduction, 2024. URL <https://arxiv.org/abs/2410.15696>.
- Marco Cagnetta, Vilém Zouhar, Sangwhan Moon, and Naoaki Okazaki. Two counterexamples to tokenization and the noiseless channel. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (eds.), *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 16897–16906, Torino, Italia, May 2024. ELRA and ICCL. URL <https://aclanthology.org/2024.lrec-main.1469/>.
- Cohere. The command r model (details and application), 2024.
- Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. Getting the most out of your tokenizer for pre-training and domain adaptation, 2024. URL <https://arxiv.org/abs/2402.01035>.
- Mark Davis and Michel Suignard. Unicode security considerations. Unicode Technical Report 36, Unicode Consortium, 2014. URL <https://www.unicode.org/reports/tr36/>.
- Mark Davis, Martin Dürst, and Ken Whistler. Unicode normalization forms. Unicode Standard Annex 15, Unicode Consortium, 2024. URL <https://www.unicode.org/reports/tr15/>.
- Matthias Gallé. Investigating the effectiveness of BPE: The power of shorter sequences. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 1375–1381, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1141. URL <https://aclanthology.org/D19-1141/>.

- Juan Luis Gastaldi, John Terilla, Luca Malagutti, Brian DuSell, Tim Vieira, and Ryan Cotterell. The foundations of tokenization: Statistical and computational concerns, 2024. URL <https://arxiv.org/abs/2407.11606>.
- Renato Lui Geh, Zilei Shao, and Guy Van den Broeck. Adversarial tokenization, 2025. URL <https://arxiv.org/abs/2503.02174>.
- Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. Coercing llms to do and reveal (almost) anything, 2024. URL <https://arxiv.org/abs/2402.14020>.
- Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size, 2024. URL <https://arxiv.org/abs/2408.00118>.
- Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report, 2025. URL <https://arxiv.org/abs/2503.19786>.
- Saibo Geng, Sankalp Gambhir, Chris Wendler, and Robert West. Byte bpe tokenization as an inverse string homomorphism, 2024. URL <https://arxiv.org/abs/2412.03160>.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models, 2024. URL <https://arxiv.org/abs/2407.21783>.
- Gregory Grefenstette and Pasi Tapanainen. What is a word, what is a sentence?: problems of tokenisation. 1994.
- Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. Olmo: Accelerating the science of language models, 2024. URL <https://arxiv.org/abs/2402.00838>.
- Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need, 2023. URL <https://arxiv.org/abs/2306.11644>.
- hnyls2002. Decode incrementally. GitHub Pull Request, 2024. URL <https://github.com/sgl-project/sglang/pull/517>.
- Valentin Hofmann, Hinrich Schuetze, and Janet Pierrehumbert. An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 385–393, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-short.43. URL <https://aclanthology.org/2022.acl-short.43/>.
- Hugging Face. Text generation inference, 2025. URL <https://github.com/huggingface/text-generation-inference>.
- Huggingface. tokenizers. GitHub Repository, 2025. URL <https://github.com/huggingface/tokenizers>.
- Eugene Jang, Kimin Lee, Jin-Woo Chung, Keuntae Park, and Seungwon Shin. Improbable bigrams expose vulnerabilities of incomplete tokens in byte-level tokenizers, 2024. URL <https://arxiv.org/abs/2410.23684>.

- Mojan Javaheripi and Sébastien Bubeck. Phi-2: The surprising power of small language models.
- jeffwang0516. bug: Output text from completionchunk is different with tokenizer.decode. Github Issue, 2023. URL <https://github.com/bentoml/OpenLLM/issues/809>.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Léo Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL <https://arxiv.org/abs/2310.06825>.
- Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Léo Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL <https://arxiv.org/abs/2401.04088>.
- G. Kiczales. Towards a new model of abstraction in software engineering . In *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*, pp. 127,128, Los Alamitos, CA, USA, October 1991. IEEE Computer Society. doi: 10.1109/IWOOOS.1991.183036. URL <https://doi.ieeecomputersociety.org/10.1109/IWOOOS.1991.183036>.
- G. Kiczales, M. Theimer, and B. Welch. A new model of abstraction for operating system design. In [1992] *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pp. 346–349, 1992. doi: 10.1109/IWOOOS.1992.252962.
- Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding, 2024. URL <https://arxiv.org/abs/2407.08103>.
- Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In Iryna Gurevych and Yusuke Miyao (eds.), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1007. URL <https://aclanthology.org/P18-1007/>.
- Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Eduardo Blanco and Wei Lu (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL <https://aclanthology.org/D18-2012/>.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles, 2023*.
- J. Lambek. From word to sentence: a pregroup analysis of the object pronoun who(m). *Journal of Logic, Language and Information*, 16(3):303–323, February 2007. ISSN 1572-9583. doi: 10.1007/s10849-006-9035-9. URL <http://dx.doi.org/10.1007/s10849-006-9035-9>.
- Joachim Lambek. Type grammar revisited. In *Selected Papers from the Second International Conference on Logical Aspects of Computational Linguistics, LACL '97*, pp. 1–27, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540657517.
- Sander Land and Max Bartolo. Fishing for magikarp: Automatically detecting under-trained tokens in large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 11631–11646, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.649. URL <https://aclanthology.org/2024.emnlp-main.649/>.

- Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report, 2023. URL <https://arxiv.org/abs/2309.05463>.
- Jindřich Libovický, Helmut Schmid, and Alexander Fraser. Why don't people use character-level machine translation? In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 2470–2485, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.194. URL <https://aclanthology.org/2022.findings-acl.194/>.
- Tomasz Limisiewicz, Terra Blevins, Hila Gonen, Orevaoghene Ahia, and Luke Zettlemoyer. MYTE: Morphology-driven byte encoding for better and fairer multilingual language modeling. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15059–15076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.804. URL <https://aclanthology.org/2024.acl-long.804/>.
- M. Lothaire (ed.). *Combinatorics on Words*. Cambridge University Press, 1997. ISBN 9780511566097. doi: 10.1017/cbo9780511566097. URL <http://dx.doi.org/10.1017/CBO9780511566097>.
- João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. Syntactic and semantic control of large language models via sequential monte carlo. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=xoXn62FzD0>.
- Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, et al. Gemma: Open models based on gemini research and technology, 2024. URL <https://arxiv.org/abs/2403.08295>.
- Microsoft, :, Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras, 2025. URL <https://arxiv.org/abs/2503.01743>.
- Sabrina J. Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp, 2021. URL <https://arxiv.org/abs/2112.10508>.
- Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pp. 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5\_37. URL [https://doi.org/10.1007/978-3-030-79876-5\\_37](https://doi.org/10.1007/978-3-030-79876-5_37).
- Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023a. URL <https://arxiv.org/abs/2305.02309>.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023b. URL <https://arxiv.org/abs/2203.13474>.
- OpenAI. tiktoken. GitHub Repository, 2025.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report, 2024. URL <https://arxiv.org/abs/2303.08774>.

- Artidoro Pagnoni, Ram Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srinivasan Iyer. Byte latent transformer: Patches scale better than tokens, 2024. URL <https://arxiv.org/abs/2412.09871>.
- David D. Palmer. *Tokenization and Sentence Segmentation*, chapter 2. Marcel Drekker, 2000.
- Aleksandar Petrov, Emanuele La Malfa, Philip H.S. Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA, 2023. Curran Associates Inc.
- Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models, 2022. URL <https://arxiv.org/abs/2201.11227>.
- Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019. URL [https://cdn.openai.com/better-language-models/language\\_models\\_are\\_unsupervised\\_multitask\\_learners.pdf](https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf).
- AbRAR Rahman, Garry Bowlin, Binit Mohanty, and Sean McGunigal. Towards linguistically-aware and language-independent tokenization for large language models (llms), 2024. URL <https://arxiv.org/abs/2410.03568>.
- Phillip Rust, Jonas F. Lotz, Emanuele Bugliarello, Elizabeth Salesky, Miryam de Lhoneux, and Desmond Elliott. Language modelling with pixels. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=FkSp8VW8RjH>.
- Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, October 2009. ISBN 9781139195218. doi: 10.1017/cbo9781139195218. URL <http://dx.doi.org/10.1017/CBO9781139195218>.
- Elizabeth Salesky, David Etter, and Matt Post. Robust open-vocabulary translation from visual text representations. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7235–7252, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.576. URL <https://aclanthology.org/2021.emnlp-main.576/>.
- Craig W Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. Tokenization is more than compression. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 678–702, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.40. URL <https://aclanthology.org/2024.emnlp-main.40/>.
- Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models, 2021. URL <https://arxiv.org/abs/2109.05093>.
- Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith (eds.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL <https://aclanthology.org/P16-1162/>.
- Joel Spolsky. The law of leaky abstractions. *Joel on Software*, 2002. URL <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/>.
- Stability-AI. Stable lm. GitHub Repository. URL <https://github.com/stability-AI/stableLM/>.

- Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Dingo: Constrained inference for diffusion llms, 2025. URL <https://arxiv.org/abs/2505.23061>.
- Yi Tay, Vinh Q. Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. Charformer: Fast character transformers via gradient-based subword tokenization, 2022. URL <https://arxiv.org/abs/2106.12672>.
- Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 olmo 2 furious, 2025. URL <https://arxiv.org/abs/2501.00656>.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023a. URL <https://arxiv.org/abs/2302.13971>.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shrutu Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esioibu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023b. URL <https://arxiv.org/abs/2307.09288>.
- Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. Itegen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL <https://openreview.net/forum?id=ac93gRzxxV>.
- Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syn-code: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*, 2025b. ISSN 2835-8856. URL <https://openreview.net/forum?id=HiUZtgAPoH>.
- Unicode Technical Committee. Unicode standard. Standard, Unicode, Inc., 2025. URL <https://www.unicode.org/versions/latest/>.
- Hendrik van Antwerpen and Alexander Neubeck. bpe readme. *GitHub*, 2024. URL <https://github.com/github/rust-gems/blob/main/crates/bpe/README.md>.
- Anne van Kesteren. Encoding. Standard, WHATWG, 2024. URL <https://encoding.spec.whatwg.org>.
- W3Techs. Usage statistics of character encodings for websites. Q-Success, 02 2025. URL [https://w3techs.com/technologies/overview/character\\_encoding](https://w3techs.com/technologies/overview/character_encoding).
- Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models, 2023. URL <https://arxiv.org/abs/2307.09702>.

- Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 03 2022. ISSN 2307-387X. doi: 10.1162/tacl\_a\_00461. URL [https://doi.org/10.1162/tacl\\_a\\_00461](https://doi.org/10.1162/tacl_a_00461).
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL <https://arxiv.org/abs/2505.09388>.
- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025b. URL <https://arxiv.org/abs/2412.15115>.
- Zhixuan Yang and Nicolas Wu. Modular models of monoids with operations. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023. doi: 10.1145/3607850. URL <https://doi.org/10.1145/3607850>.
- Yard1. Use tgi-like incremental detokenization. GitHub Pull Request, 2023. URL <https://github.com/vllm-project/vllm/pull/984>.
- Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL <https://arxiv.org/abs/2312.07104>.
- Vilém Zouhar, Clara Meister, Juan Gastaldi, Li Du, Mrinmaya Sachan, and Ryan Cotterell. Tokenization and the noiseless channel. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5184–5207, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.284. URL <https://aclanthology.org/2023.acl-long.284/>.
- Vilém Zouhar, Clara Meister, Juan Gastaldi, Li Du, Tim Vieira, Mrinmaya Sachan, and Ryan Cotterell. A formal perspective on byte-pair encoding. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 598–614, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.38. URL <https://aclanthology.org/2023.findings-acl.38/>.

## A Extended related work

**Tokenization notation and terminology.** The process we call “cutting” is a standard step in tokenization but is represented variously in the literature. Sennrich et al. (2016) use  $()$  or a space; Gastaldi et al. (2024) use  $()$ ; Schmidt et al. (2024) and Bostrom & Durrett (2020) use a space; Koo et al. (2024) use  $()$ ; Kudo (2018) uses  $()$ ; Kudo & Richardson (2018) wrap tokens with square brackets; Cогnetta & Okazaki (2024) use  $\_$ ; and Geng et al. (2024) use color to distinguish tokens. We follow Berglund & van der Merwe (2023) in using  $()$ , though we extend their notation by requiring that cut sequences begin and end with  $()$ .

**Morphisms in tokenization.** We use the term homomorphism the way Geng et al. (2024) use it. For Sakarovitch (2009), a homomorphism is a bijective morphism. Gastaldi et al. (2024) call *multiplicative* what we refer to as a homomorphism; with the following additional constraint that it map non-empty sequences to non-empty sequences, it also has a *trivial kernel*:  $\delta \neq \epsilon_\Delta \rightarrow \kappa(\delta) \neq \epsilon_\Sigma$ . Lothaire (1997) and Sakarovitch (2009) call a *morphism* what we call a homomorphism with the additional requirement that it map the identity member to the identity member:  $\kappa(\epsilon_\Delta) = \epsilon_\Sigma$ . We include neither of these additional restrictions because they are not necessary for our work here.

**Subword tokenization methods.** A cutting tokenizer like ours is a generalization of a subword tokenizer (Sennrich et al., 2016; Kudo, 2018; Kudo & Richardson, 2018). The performance of subword tokenizers is contested. The reader is directed *inter alia* to Gallé (2019); Zouhar et al. (2023a) for a favorable impression and to Bostrom & Durrett (2020); Schmidt et al. (2024); Chai et al. (2024a) for a negative one. Practically representing the negative camp, tokenizer-free models work directly on input characters (e.g. Tay et al., 2022; Clark et al., 2022), bytes (e.g. Xue et al., 2022; Pagnoni et al., 2024), or on images of input text (e.g. Salesky et al., 2021; Rust et al., 2023; Chai et al., 2024b).

**Byte-level and character-level approaches.** Byte-pair encoding has been studied extensively: Bostrom & Durrett (2020) argue that Unigram is superior to BPE. Gallé (2019) argue that BPE performs highly because it compresses the input, but Schmidt et al. (2024) perform experiments that suggest that compression is not necessary or sufficient for performance in a tokenizer. Zouhar et al. (2023a) propose an information theoretic standard, Rényi entropy, for why certain tokenizers perform better than others; Cогnetta et al. (2024) supply counter examples to the argument of Zouhar et al. (2023a). Libovický et al. (2022) examine the efficacy of tokenizationless character-level machine translation compared to character-level BPE and find that the former performs at best only as well as the latter.

**Inter-language performance comparisons.** Limisiewicz et al. (2024); Hofmann et al. (2022) examine adapting subword tokenization to respect morphological boundaries in language in order to improve performance in non-European languages. Petrov et al. (2023); Ahia et al. (2023) show that byte-level and character-level tokenization introduce severe discrepancies among languages, due in part to the varying lengths of byte sequences used to represent text in various languages.

## B Details about UTF-8

Table 4: UTF-8 bit distribution, showing how to convert code points, represented as binary numbers, into the bytes of UTF-8. (Unicode Technical Committee, 2025, Table 3-6).

Code Point	First Byte	Second Byte	Third Byte	Fourth Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

In the main body of the paper we ignore the concept of code points entirely and exclusively treat UTF-8 code units. However, in Appendix D we discuss a hack to store byte-level tokens

in well-formed UTF-8 files; we will need to discuss code points as such there. Each character in Unicode (Unicode Technical Committee, 2025) is mapped to a number in the range 0 to  $10FFF_{16}$ , called a code point. To turn a code point into its UTF-8 code unit, convert the code point to bytes as shown in Table 4. A process that interprets UTF-8 bytes as characters, a process we waved away in the body of the paper, must disassemble the bytes it receives on its input to recover the code point of the character it must display.

Several properties of UTF-8 referenced in the main text are immediately obvious by visual inspection of Tables 4 and 1. For example, any sequence made up exclusively of bytes starting 10, that is, in the range 80..BF, can never be well-formed UTF-8. Similarly, no sequence made up exclusively of bytes beginning with 110, 1110, or 11110 can ever contain any well-formed encoded forms. Also, the bytes C0-C1 and F5-FF never appear in UTF-8 at all, so they can also be used to disrupt well-formed encoding forms.

Note also that not all characters are encoded by the same number of bytes. In tokenizers based on merging pairs of tokens, the number of bytes that make up a character could influence the likelihood that that character is represented by more than one token. We leave verifying or disproving this conjecture to future work.

## C Python implementation of Algorithm 1

**Listing 1** Implementation of Algorithm 1 from the serving engines discussed in Section 5.

In the real listing, the variable `replacement_char` is assigned the value `?`. Note that the variables we call  $i$  and  $j$  are named `prefix_offset` and `read_offset` respectively. The fast version of the Hugging Face tokenizers library uses a Rust translation of this logic (Huggingface, 2025).

---

```

1 def decode_token(
2     self,
3     all_input_ids: List[int],
4     prefix_offset: int = 0,
5     read_offset: int = 0,
6     skip_special_tokens: bool = False,
7 ) -> Tuple[str, int, int]:
8     prefix_text = self.tokenizer.decode(
9         all_input_ids[prefix_offset:read_offset],
10        skip_special_tokens=skip_special_tokens,
11    )
12    new_text = self.tokenizer.decode(
13        all_input_ids[prefix_offset:],
14        skip_special_tokens=skip_special_tokens
15    )
16
17    if len(new_text) > len(prefix_text) and not
18        ↪ new_text.endswith(replacement_char):
19        new_text = new_text[len(prefix_text) :]
20        return new_text, read_offset, len(all_input_ids)
21    else:
22        return "", prefix_offset, read_offset

```

---

Table 5 gives an example of incrementally decoding a sequence of byte tokens. The first column of the execution trace contains the byte tokens to be detokenized, read in order from top to bottom. The second column shows the characters emitted as they are completed. The third and fourth columns show the values of  $i$  and  $j$  as they are advanced through the sequence of byte tokens.

Table 5: Trace of executing Algorithm 1 on the tokens from Example 1a.

Byte token	Text emitted	i	j
<code>\xE0 A4\</code>		0	0
<code>\85\</code>	अ	0	2
<code>\xE0 A4\</code>		0	2
<code>\97\</code>	ग	2	4
<code>\xE0 A5 8D E0 A4\</code>		2	4
<code>\A8\</code>	न	4	6
<code>\xE0 A4 BF E0 A4\</code>		4	6
<code>\AE\</code>	मि	6	8
<code>\xE0 A5 80\</code>	ी	8	9
<code>\xE0 A4\</code>		8	9
<code>\B3\</code>	ळ	9	11
<code>\xE0 A5 87\</code>	ं	11	12

## D Mapping bytes to Unicode and back

Г	р	а	д	г	р	а	д	и	л	а	
<code>\D093\</code>	<code>\D180</code>	<code>D0B0</code>	<code>D0B4\</code>	<code>\20</code>	<code>D0B3\</code>	<code>\D180</code>	<code>D0B0</code>	<code>D0B4\</code>	<code>\D0B8</code>	<code>D0BB</code>	<code>D0B0\</code>
<code>\Ej\</code>	<code>\NG</code>	<code>Đ°</code>	<code>Đ´\</code>	<code>\G</code>	<code>Đ³\</code>	<code>\NG</code>	<code>Đ°</code>	<code>Đ´\</code>	<code>\D,</code>	<code>Đ»</code>	<code>Đ°\</code>

Table 6: The first two words of *The Building of Skadar* (Serbian: Зидане Скадра), “Град градила”, transliterated “*Grad gradila*” (“The city was built”). The top row is the sequence of characters making up these words, the middle row is the sequence of bytes that are the UTF-8 encoding of this sequence of code points, and the bottom row is the bytes of the middle row mapped to characters according to the mapping defined in Listing 2. The squiggles `\` in the bottom two rows indicate the tokenization according to Qwen3’s vocabulary (Yang et al., 2025a).

**Listing 2** Code from GPT-2 (Radford et al., 2019) that defines an injective but not surjective mapping between the set of natural numbers from 0 to 255 and the set of natural numbers. The numbers  $21_{16}, \dots, 7E_{16}, A1_{16}, \dots, AC_{16}, AE_{16}, \dots, FF_{16}$  are mapped to themselves. The remaining numbers are mapped in order to  $100_{16}$  through  $143_{16}$ . The returned dictionary maps from ints representing byte values to strs of one character representing code points.

```

1 def bytes_to_unicode():
2     bs = (
3         list(range(ord("!"), ord("~") + 1))
4         + list(range(ord("¡"), ord("¬") + 1))
5         + list(range(ord("®"), ord("ÿ") + 1))
6     )
7     cs = bs[:]
8     n = 0
9     for b in range(2**8):
10        if b not in bs:
11            bs.append(b)
12            cs.append(2**8 + n)
13            n += 1
14    cs = [chr(n) for n in cs]
15    return dict(zip(bs, cs))

```

Hugging Face distributes tokenizer vocabularies as UTF-8-encoded files.<sup>3</sup> Since programmers tend to work with UTF-8-encoded strings rather than directly with byte sequences, it

<sup>3</sup>In HTTP-speak: content-type: text/plain; charset=UTF-8

is convenient to be able to represent arbitrary byte sequences, such as those produced by arbitrarily clumping bytes, as UTF-8 sequences.<sup>4</sup> GPT-2 (Radford et al., 2019), because it had a byte-level vocabulary stored in UTF-8 files, introduced a mapping between each byte and a code point (see B) and storing the code points as UTF-8 byte sequences. The tokenizer’s implementation must convert back and forth between the code points and the bytes they represent to interact with byte-level inputs.

Listing 2 gives a Python procedure for mapping from bytes to (printable) code points. Most of the code points between  $00_{16}$  and  $FF_{16}$  are printable characters, so though the result will be an ugly mix of semantically insignificant characters, the result will be made up entirely of printable characters. The characters in the range  $00_{16}$  through  $FF_{16}$  that are control or whitespace characters are mapped to the code points beginning at  $100_{16}$ . Luckily the entire block U+0100–U+17F0 is printable characters. The transition between the middle and bottom rows of Example 6 exemplifies these transformations.

## E Unicode character handling SynCode evaluation prompt and grammar

As discussed in Section 6, to evaluate SynCode’s ability to handle non-ASCII Unicode characters, we conducted an experiment using emoji generation task. We selected a subset of the TweetEval emoji dataset Barbieri et al. (2020), filtering for three common emoji classes. The task required the model to generate exactly one emoji character in response to a given tweet, adhering to a constrained grammar specification. This evaluation directly tested SynCode’s handling of multi-byte UTF-8 sequences, which was a limitation in earlier versions.

---

### Listing 3 Grammar specification for emoji generation task using Unicode escape sequences.

```
// Lark grammar to validate single emoji output
start: emoji

// Define the 3 emojis from the TweetEval emoji dataset
emoji: "\U0001F60D" | "\U0001F602" | "\U0001F609"
emoji: "😍" | "😂" | "😞"
```

---

The prompt template instructed the model to analyze tweets and respond with exactly one emoji from the allowed set (Listing 4). We evaluated this task across 100 examples from the TweetEval test set on two versions of SynCode: v0.2.0 which used a character-level finite state machine (FSM), and the current version which implements a byte-level FSM with our recommended fix.

---

### Listing 4 Prompt template for emoji generation task (emoji symbols represented as placeholders)

```
You are evaluating tweets to assign the most appropriate emoji.
INSTRUCTIONS:
1. Read the tweet below carefully.
2. Select the SINGLE most appropriate emoji that captures the sentiment.
3. Respond with ONLY that emoji - no words or other characters.
```

The emoji must be one of the 3 valid options from this set:

😍 😂 😞

```
Tweet: "tweet_text"
Your response:
```

---

<sup>4</sup>A notable exception is OpenAI, who after GPT-2 have worked directly with byte sequences that are not guaranteed to be valid UTF-8.

The results demonstrated a significant improvement in Unicode handling capability. Syn-Code v0.2.0 with its character-level FSM failed on all 100 examples, resulting in a 100% crash rate and 0% accuracy. In comparison, the current implementation after suggested fix processed all examples without crashes, achieving 62% accuracy in emoji prediction.