WINMORPH: DOES TOKENIZING CODE IDENTIFIERS AT MORPHEME BOUNDARIES IMPROVE LANGUAGE MODEL PERFORMANCE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Code identifiers such as `AddAccessAllowedAce` combine meaningful subparts (morphemes), such as `Add`, `Access`, `Allowed`, and `Ace`. However, a language model observes only tokenizer outputs, and a tokenizer may place boundaries that do not match these morphemes. Although better boundary placement is often assumed to help, it is difficult to isolate because standard tokenizers confound alignment with token count: making more correct splits typically also changes the number of tokens. We therefore ask: *when the token count is held fixed, does aligning token boundaries with reference morpheme boundaries improve language modeling*? We address this question with WINMORPH, a corpus of $83{,}900$ Windows identifiers with gold cut points derived from published naming rules, and a benchmark that varies the fraction of correct cuts from none to all while keeping tokens-per-identifier constant. We study two controls: (i) segmentation matching under a shared vocabulary cap, and (ii) matching the average number of model-input symbols on the training set by adjusting vocabulary coverage. On held-out identifiers, full alignment reduces bits per character by 39.8% under segmentation matching and by 18.9% under training-symbol matching, compared to no reference-aligned cuts. These results show that, when we control for token count, splitting identifiers at reference morpheme boundaries improves language model performance.

1 INTRODUCTION

Every language model first divides text into pieces called tokens. A tokenizer determines where these divisions occur, but it is often chosen by convention. Byte-pair encoding (Sennrich et al., 2016) and the Unigram model (Kudo, 2018) remain the default choices. Because training a model for every candidate tokenizer is expensive, researchers have developed inexpensive measures that use only tokenized text to predict a model's eventual performance. Three measures dominate performance of a tokenizer. *Compression* measures how many tokens a word is divided into (Gallé, 2019; Gowda and May, 2020). *Rényi efficiency* measures how evenly tokens are used (Zouhar et al., 2023). *Morphological alignment* measures how often token boundaries coincide with morpheme boundaries (Bostrom and Durrett, 2020). A morpheme is the smallest part of a word that carries meaning; for example, the morphemes in `AllocateMemory` are `Allocate` and `Memory`.

Rényi efficiency was introduced after reports of a strong correlation with downstream translation quality, but later work presented clear counterexamples (Cognetta et al., 2024). Results for compression are also mixed: one study found no association with accuracy across 64 models (Schmidt et al., 2024), whereas another found a consistent negative association (Goldman et al., 2024). Morphological alignment appears least informative. Across 70 languages, it explains almost none of the variation in downstream performance, and the small association it shows goes in the opposite direction (Arnett et al., 2025).

The main obstacle is not a lack of models or metrics, but the absence of an appropriate dataset. **To test whether morphological alignment helps, we must compare tokenizers that are identical except for their alignment, which ordinary text does not allow.** In natural language, enforcing more morpheme boundaries typically increases the number of tokens, so alignment and token count

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

vary together. In addition, gold morpheme boundaries are limited and often contested, and most resources provide a single fixed segmentation. They indicate where boundaries are assumed to lie, but they do not allow us to move boundaries and measure the effect. As a result, we cannot evaluate how different boundary placements would change outcomes.

In some domains, "morpheme" boundaries follow written conventions rather than human judgment. Windows system identifiers are a clear example. Under Microsoft's published naming rules, `MmAllocateContiguousMemory` segments into `Mm`, `Allocate`, `Contiguous`, and `Memory`. In practice, three independent conventions operate simultaneously: underscore-delimited constants (e.g., `FILE_ATTRIBUTE_READONLY`), PascalCase function names (e.g., `RegSetValueExW`), and Hungarian type prefixes in declarations (e.g., `lpszFileName`). Using open-source header files, export tables, and open-source code bases, we derive a reference segmentation for $83{,}900$ identifiers. With this reference in hand, alignment becomes a controllable variable: we can construct tokenizers that match all boundaries, match none, or achieve any intermediate level while keeping the token count fixed (Figure 2).

Our contributions:

- **A dataset whose cut points come from rules, not from opinion.** WINMORPH contains $83{,}900$ Windows system identifiers divided into $21{,}950$ distinct morphemes by published naming rules. Three rules coexist and check one another, so the segmentation needs no human judgment. We release the data, the rules, and the code.
- **A benchmark in which boundary precision is a dial.** Because the correct cuts are known, we build tokenizers instead of picking existing ones. Our *arms* (We call each tokenizer we construct an *arm*) cut at all correct boundaries, at none of them, or at any fraction in between, always with the same number of tokens per identifier (Table 1). We evaluate them on three tasks: a training-free test on pairs of near-identical identifiers (**Pair-task**), language modeling on identifiers seen in training (**Seen-task**), and language modeling on held-out identifiers (**Unseen-task**).
- **A direct answer to the question in the title.** For identifiers the model has not seen, using correct boundaries reduces bits per character fall by 39.8% overall. So the answer is yes. Cutting at correct boundaries has huge impact on model performance.

2 RELATED WORK

Benchmarks for tokenizer quality and Alignment measures. Most evaluations of tokenizers sample existing algorithms, measure their properties, and regress performance on those properties (Schmidt et al., 2024; Goldman et al., 2024; Arnett et al., 2025). This protocol cannot separate properties that move together. Uzan et al. (2024) shows that the segmentation strategy used at inference matters independently, and Petrov et al. (2023) documents how tokenizer choice distributes cost unevenly across languages. Our benchmark differs by supplying the tokenizers rather than sampling them. Bostrom and Durrett (2020) shows that Unigram aligns better with morphology than byte-pair encoding and transfers better. The predictive picture is discouraging (Arnett et al., 2025). Comparing tokenizers requires a shared unit; bits per character is standard (Mielke et al., 2019), so we adopt it.

Tokenizing source code. Whole identifiers would make a code model's vocabulary unmanageably large, so open-vocabulary code models typically use subword tokenization (Karampatsis et al., 2020). Subsequent work compared subtokenization choices for code pretraining (Chirkova and Troshin, 2023) and found that code-specialized tokenizers trade off compression and downstream accuracy (Dagan et al., 2024). Identifier splitting into constituent words has also been studied extensively in software engineering (Enslen et al., 2009; Hill et al., 2014). Most closely related to our setting, Shi et al. (2022) introduced rule-based splitting for a BPE code model and reported that splitting alone reduced performance, whereas a hybrid approach helped; however, splitting also changed the token count, introducing the same confound that our design removes.

Making tokenizers more morphological. A line of work modifies a tokenizer so its output aligns better with morphology, then measures what changes downstream. Hofmann et al. (2022) keep a pretrained BPE vocabulary unchanged but alter how it is applied at inference, producing more plausible

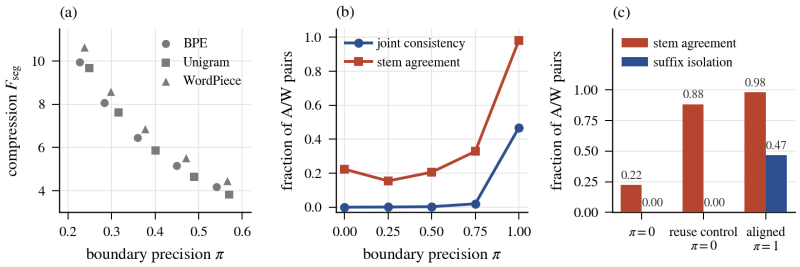

Figure 1: **(a)** Ordinary tokenizers cannot separate the two properties: fifteen BPE, Unigram and WordPiece tokenizers on our corpus. Higher precision always arrives with fewer tokens, so no member of this family lets one vary while the other is held. **(b)** The training-free pair probe: joint consistency and stem agreement against precision. Stem agreement is substantial even at low precision, which is why joint consistency requires two conditions rather than one. **(c)** The reuse control nearly matches the aligned arm on stem agreement while isolating the suffix in no pair at all.

segmentations and improved classification accuracy. García et al. (2025) study morphology-aware segmentation for a single language, and Fusco et al. (2025) evaluate linguistic units as tokens both intrinsically and extrinsically under a fixed data budget. These studies are the closest in motivation to ours, and they share the difficulty we address: in each case the intervention that improves alignment also changes how many tokens a text becomes, so an observed gain is attributable to either property. That is not a flaw in their execution but a limit of what natural text allows. Our design differs in fixing the token count by construction, which lets the same comparison be made with one quantity varying rather than two.

3 Problem Statement and Proposed Solution

The standard approach trains several conventional tokenizers and tests which tokenizer property predicts performance. This approach fails for two reasons. First, no standard algorithm allows precision to be controlled directly. Byte-pair encoding and Unigram optimize merge frequency or likelihood, while alignment emerges only as a side effect. Second, *precision* and *compression* move together almost perfectly among conventional tokenizers (Figure 1 **a**). A larger identifier produces both more accurate boundaries and fewer tokens, so the study cannot determine which property causes the improvement. Evaluating more models does not resolve this problem.

A tokenizer divides an identifier into pieces by placing cuts between its characters. Some cuts coincide with true morpheme boundaries, while others do not. **Does placing more cuts on true boundaries improve performance of a language model?** Let W denote the set of identifiers. For a tokenizer T and an identifier w, let $B_T(w) \subseteq 1, \ldots, |w|-1$ denote the cut positions, which produce $|B_T(w)|+1$ tokens. Let $B^*(w)$ denote the true morpheme boundaries (figure 2 (A)). We call each tokenizer we construct an *arm* and evaluate it using standard boundary precision (π) and recall (ρ) (Arnett and Bergen, 2025). figure 2 (B) shows arms with different boundary cuts.

$$\pi(T) = \frac{\sum_w |B_T(w) \cap B^*(w)|}{\sum_w |B_T(w)|}, \qquad \rho(T) = \frac{\sum_w |B_T(w) \cap B^*(w)|}{\sum_w |B^*(w)|} \tag{1}$$

where precision (π) measures the fraction of an arm's cuts that are correct, and recall (ρ) measures the fraction of true boundaries that the arm finds. Compression is the mean number of tokens per identifier, $F(T) = 1 + |W|^{-1} \sum_w |B_T(w)|$ (Gallé, 2019), and F^* denotes the corresponding value for the reference compression (segmentation).

3.1 Proposed Solution

In this section, we provide two propositions, and we propose our solution based on these propositions.

Proposition 1 (Recall follows from precision and compression)**.** *For any tokenizer with* $\sum_w |B_T(w)| > 0$*,*

$$\rho(T) = \pi(T)(F(T)-1)/(F^*-1)$$

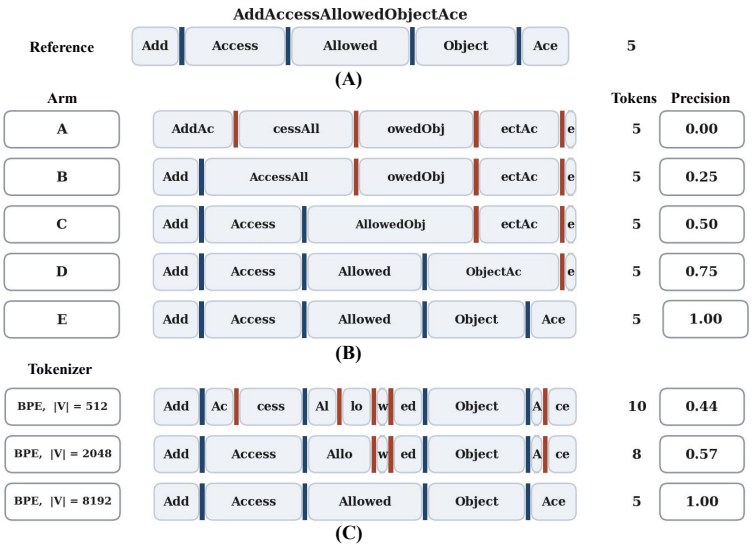

Figure 2: Why ordinary tokenizers cannot vary precision independently of compression. Navy bars indicate cuts on true morpheme boundaries, while red bars indicate cuts inside morphemes. **(A)** The reference contains five morphemes and four true boundaries. **(B)** Five constructed arms each make four cuts and therefore produce five tokens. Compression remains fixed while precision ranges from 0.00 to 1.00. **(C)** The same identifier is tokenized using our arm (tokenizer) at three vocabulary sizes. All three tokenizers find every true boundary. Precision increases only because the additional incorrect cuts disappear, and the token count decreases at the same time.

Proof. Let $K = \sum_w |B_T(w)|$ and $G = \sum_w |B^*(w)|$. Equation 1 gives $\rho = \pi K/G$. Substituting $K = |W|(F-1)$ and $G = |W|(F^*-1)$ yields the stated identity. □

Although the algebra is simple, the implication is important: **recall is not independent.** This identity matters for two reasons. First, it makes the confound exact; any alignment score that uses recall goes up and down with the number of cuts. Therefore, when past studies correlate such scores with model performance, part of what they measure is compression, not alignment. Second, it simplifies our design; once F is fixed, recall follows from precision, so precision is the only alignment quantity we need to vary ($\pi = 0$ to $\pi = 1$). The ρ column in Table 1 only checks that the construction worked.

An identifier cap limits the number of *embedding rows*. The model stores one learned vector for each token type that receives a row, and the cap determines how many types it can store. However, the cap does not determine where the tokenizer places its cuts. A token type without an embedding row cannot be looked up, so it is represented by its individual characters, each of which has a row. Therefore, **two arms (tokenizers) can produce the same number of *tokens* while giving the model very different numbers of *symbols***. We distinguish F_{seg}, the mean number of tokens per identifier, which the construction fixes exactly, from F_{sym}, the mean number of symbols read by the model, which depends on which token types fit within the embedding budget. Let V_T^a denote the token types produced by arm a, and let $E_a \subseteq V_T^a$ denote the types that receive an embedding row. We rank types by their training frequency, so E_a contains the most frequent types that fit within the budget. Let $n_a(v)$ denote the number of times type v occurs in the corpus. Each occurrence of a type outside E_a is replaced by its $|v|$ characters, adding $|v|-1$ symbols. Summing these additional symbols over all uncovered types gives the following relationship:

$$F^a_{\text{sym}} - F^a_{\text{seg}} = \frac{1}{|W|} \sum_{v \notin E_a} n_a(v)\big(|v|-1\big) \tag{2}$$

Proposition 2 (Exact symbol matching requires embedding all multi-character types)**.** *Let M_a be the set of token types with length > 1 that arm a produces on the identifier used to compute symbol counts. Then $F^a_{\text{sym}} = F^a_{\text{seg}}$ holds if and only if every such type has an embedding row, i.e., $M_a \subseteq E_a$.*

```
{"flag": "", "id": "ACTRL_KERNEL_VM_READ", "n": 4, "seg": ["ACTRL", "KERNEL", "VM", "READ"], "src": "accctrl.h", "strat": "underscore"}
{"flag": "no Hungarian prefix (plain field name)", "id": "ProtectedModeDataBaseAddress", "n": 5, "seg": ["Protected", "Mode", "Data", "Base", "Address"], "src": "cmtypes.h", "strat": "declaration"}
{"flag": "", "id": "PssNtValidateDescriptor", "n": 4, "seg": ["Pss", "Nt", "Validate", "Descriptor"], "src": "ntdll.dll", "strat": "case"}
```


Figure 3: Three records from WINMORPH as stored in the released JSON file, one identifier per line. Each record holds the identifier as it appears in the source (*id*), its reference segmentation (*seg*) and morpheme count (*n*), the header or export table it was collected from (*src*), the naming rule that segmented it (*strat*), and an optional note (*flag*). Long lines are wrapped for display.

Consequently, if all arms share the same embedding budget and we want $F^a_{\text{sym}} = F^a_{\text{seg}}$ for every arm, the budget must include at least $\max_a |M_a|$ embedding rows.

Proof. In Equation 2, each summand $n_a(v)(|v| - 1)$ is nonnegative and is strictly positive exactly when v occurs ($n_a(v) > 0$) and $|v| > 1$. Therefore the sum is zero (equivalently, $F^a_{\text{sym}} = F^a_{\text{seg}}$) precisely when no occurring multi-character type is left uncovered, i.e., when $M_a \subseteq E_a$. □

An arm reads no extra symbols only if every multi-character token it outputs has its own embedding row. A misaligned arm creates many more token types than an aligned one, so closing the symbol gap by increasing the embedding budget would mostly add very rare tokens, each seen only a few times in training. Instead, we do the opposite: in the symbol-matched design, we reduce the aligned arm's coverage until the average number of symbols is the same (§4).

In order to measure the performances of language models on morpheme cuts, we used two separate designs. The **Segmentation-matched** design keeps the number of segmentation tokens the same in every arm and uses the same identifier cap. Coverage and the number of symbols the model reads can still differ. The **Symbol-matched** design instead changes coverage so training identifiers have the same mean number of input symbols; with a fixed symbol budget, this also equalizes the number of training passes.

3.2 ARM (TOKENIZER) CONSTRUCTION

The aligned arm ($\pi = 1.00$). We begin with the reference segmentation of the training split and repeatedly merge the most frequent pair of adjacent tokens until the mean number of tokens per identifier reaches the target F. Each merge removes a boundary without creating a new one, so $B_T(w) \subseteq B^*(w)$ throughout the process and precision remains exactly 1.

The misaligned arms ($\pi = 0.00, 0.25, 0.50, 0.75$). A misaligned arm with target precision π takes its number of cuts for every identifier from the aligned arm: if the aligned arm makes k_w cuts in identifier w, so does every misaligned arm, which fixes the number of tokens per identifier across arms by construction. The arm places c_w of its k_w cuts on true boundaries and the remaining $k_w - c_w$ cuts on non-boundary positions. We choose c_w by stochastic rounding of πk_w: $c_w = \lfloor \pi k_w \rfloor + 1$ with probability $\pi k_w - \lfloor \pi k_w \rfloor$, and $c_w = \lfloor \pi k_w \rfloor$ otherwise, so that corpus precision equals π in expectation.

Reuse-matched. This arm is derived from the aligned arm by shifting every cut left or right by the same small offset (1–4 characters). Since all cuts are moved, none falls on a true boundary, so $\pi = 0$. Because the offset is shared, the sequence of piece lengths is unchanged, which preserves cross-identifier reuse more than a fully random placement would. We also vary a *coordination* parameter that controls what fraction of identifiers use the same shift. We sweep both the offset and coordination, and select the pair whose reuse index R (Section 4.2) is closest to the target. This arm tests how much benefit remains when pieces are frequently reused across identifiers but all cuts are incorrect.

Table 1: Properties of the constructed arms. $|V_T|$ is the number of token types an arm produces over the corpus; ρ is boundary recall (Equation 1) at the precision π given in the row label, and Reuse-matched arms have $\pi = 0$; R is the reuse index (§4.2). All arms produce $F = 2.50$ tokens per identifier; the reference segmentations have $F^* = 3.43$ in both corpora, so the aligned arm's recall of 0.62–0.63 follows from Proposition 1.

	WINMORPH			Hungarian		
Arm	$\|V_T\|$	ρ	R	$\|V_T\|$	ρ	R
$\pi = 0.00$	131,730	0.00	1.61	65,574	0.00	3.20
$\pi = 0.25$	124,714	0.16	1.68	65,150	0.16	3.22
$\pi = 0.50$	108,834	0.31	1.93	54,639	0.31	3.84
$\pi = 0.75$	88,872	0.47	2.36	42,612	0.46	4.93
$\pi = 1.00$	38,219	0.63	5.55	8,995	0.62	23.34
Reuse-matched	93,903	0.00	2.26	51,295	0.00	4.09

4 BENCHMARK DESIGN

The benchmark includes three tasks that use the same morpheme identifiers, so the arm (tokenizer) is the only difference between the systems being compared. **Pair-task:** Windows provides most string-handling functions in two versions: one for ANSI text and one for wide characters. For example, `RegSetValueExA` and `RegSetValueExW` represent the same function and differ only in their final character, which the reference treats as a separate morpheme. Pair-task tests whether a tokenizer splits both identifiers in the same way while keeping the final character separate. **Seen-task:** A model is trained on the training split and evaluated on identifiers from that same split. This task deliberately does not measure generalization. Instead, it measures how efficiently a tokenizer allows a model to compress text that it has already seen, which is a quantity disputed in the compression literature.

Unseen-task: The same model is scored on the held-out identifiers, which it has never seen in any form. This task carries the central hypothesis. If aligned tokens are reusable pieces of meaning, their value should be greatest where an identifier cannot be memorized and must instead be assembled from parts the model already knows. Suppose the model has seen `AddAccessAllowedAce` and `RegSetValueExW` during training, and now meets `AddAccessDeniedObjectAce` for the first time. An aligned tokenizer cuts it into `Add`, `Access`, `Denied`, `Object`, `Ace`, and four of those five pieces are ones the model already has representations for; only `Denied` is new, and it sits in a familiar frame. A misaligned tokenizer cutting the same identifier into pieces such as `AddAcc`, `essDenie` , and `dObjectAce` produces fragments that appeared in no earlier identifier, so the model has nothing to build from, even though the underlying parts are all familiar. The identifier is equally unfamiliar to both; what differs is whether its familiar parts survive the cut. This is why Unseen-task, rather than Seen-task, is the benchmark's primary endpoint. On identifiers seen in training, a model can memorize the whole string, and the segmentation matters less. On held-out identifiers, memorization is unavailable by construction, so what remains is whatever the tokenizer makes reusable.

4.1 DATASET

WINMORPH includes 83,900 Windows system identifiers gathered over two months from open-source header files, export tables, and open-source code bases. Following published naming guidelines from Microsoft, we segment these identifiers into 21,950 unique morphemes. Figure 3 shows some samples of WINMORPH dataset. To test whether our method depends on the Windows domain, we also run the same experiments on the SIGMORPHON 2022 morpheme-segmentation dataset for agglutinative Hungarian (Batsuren et al., 2022).[1] We sample 84,000 Hungarian words and stratify them by morpheme count to match the WINMORPH distribution. Code for reproducibility is available in the WinMorph repository.[2] For both corpora, we create training and held-out test splits of

[1] `https://github.com/sigmorphon/2022SegmentationST`

[2] `https://anonymous.4open.science/r/WinMorph-D07D`

Table 2: What each design holds fixed. *Params* is the number of trainable parameters. Each cell is the mean over the three model seeds ± the half-width of a 95% confidence interval.

	Segmentation-matched				Symbol-matched			
Arm	Covered	Symbols (train / held)	Passes	Params	Covered	Symbols (train / held)	Passes	Params
WinMorph								
$\pi = 0.00$	32,768	14.23±0.13 / 17.10±0.09	3.95±0.9	$13.8M$	32,266	14.36 / 17.11±0.11	3.92	$13.6M$
$\pi = 0.25$	32,768	13.81±0.07 / 16.56±0.10	4.07±0.07	$13.8M$	28,397	14.36 / 16.62±0.13	3.92	$12.2M$
$\pi = 0.50$	32,768	12.50±0.09 / 15.28±0.06	4.50±0.10	$13.8M$	20,614	14.36 / 15.51±0.09	3.92	$9.2M$
$\pi = 0.75$	32,768	10.58±0.08 / 13.41±0.12	5.32±0.12	$13.8M$	10,148	14.36 / 14.41±0.06	3.92	$5.1M$
$\pi = 1.00$	32,768	4.79±0.10 / 6.50±0.08	11.76±0.14	$13.8M$	2,825	14.36 / 13.52±0.11	3.92	$2.3M$
Reuse-matched	32,768	11.49±0.12 / 14.09±0.12	4.90±0.09	$13.8M$	15,067	14.36 / 15.09±0.09	3.92	$7.0M$
Hungarian								
$\pi = 0.00$	32,768	6.76±0.09 / 8.44±0.06	8.28±0.11	$13.8M$	32,268	6.81 / 8.44±0.11	8.22	$13.6M$
$\pi = 0.25$	32,768	6.70±0.10 / 8.48±0.11	8.35±0.09	$13.8M$	31,582	6.81 / 8.51±0.09	8.22	$13.4M$
$\pi = 0.50$	32,768	5.98±0.06 / 7.70±0.13	9.37±0.20	$13.8M$	23,606	6.81 / 8.04±0.13	8.22	$10.3M$
$\pi = 0.75$	32,768	5.01±0.14 / 6.64±0.10	11.18±0.07	$13.8M$	13,956	6.81 / 7.58±0.06	8.22	$6.6M$
$\pi = 1.00$	8,893	4.50±0.011 / 4.57±0.08	12.45±0.14	$4.7M$	2,732	6.81 / 6.95±0.14	8.22	$2.3M$
Reuse-matched	32,768	5.64±0.08 / 7.47±0.13	9.92±0.08	$13.8M$	19,843	6.81 / 7.82±0.08	8.22	$8.9M$

85%/15%: WINMORPH has 71,065 training and 12,835 test identifiers, and Hungarian has 71,400 training and 12,600 test words.

4.2 METRICS

Pair-task reports three rates over the pairs. Let $B_T(w)$ and $B_T(w')$ denote the cut positions produced by an arm for the two members of a pair. *Stem agreement* is the fraction of pairs for which $B_T(w) = B_T(w')$, meaning that both members are cut at the same positions. *Suffix isolation* is the fraction of pairs for which both members have a cut immediately before the final character, leaving that character as a separate token. *Joint consistency*, our primary measure, is the fraction of pairs that satisfy both conditions. We require both conditions because either one alone is weak. An arm can cut both members identically while still placing the final character inside a longer token, which often occurs at low precision. Conversely, an arm can isolate the final character while cutting the shared stem differently, leaving the two members with almost nothing in common.

Seen-task and Unseen-task are evaluated in bits per character. This unit is necessary because each arm predicts a different vocabulary, making token-level losses incomparable across arms. Our primary statistic is corpus bits per character, defined as the total loss over the evaluation set divided by the total number of characters. It represents the expected number of bits needed to encode one character, and lower values are better. For Unseen-task in particular, we report five additional metrics. $|V_T|$ is the number of distinct token types produced by an arm across the corpus. The *reuse index* **R** is the number of token occurrences divided by the number of types, $R = \frac{|W|\,F}{|V_T|}$. That is, it represents the mean number of occurrences per type; a higher R indicates that an arm's pieces recur more often across identifier. *Covered* is the number of tokens that receive an embedding row. The difference between $|V_T|$ and *Covered* is, therefore, the number of types that the model cannot look up. *Symbols* is the mean number of inputs that the model actually reads per identifier after uncovered tokens are spelled out as characters. *Passes* is the number of times the model sees the entire training set under the fixed symbol budget described in §4.3. An arm with shorter sequences can make more passes through the data using the same budget.

4.3 REPRODUCIBILITY AND IMPLEMENTATION DETAILS

We use a small pre-norm decoder Transformer to run experiments. The model operates causally for the Seen-task and Unseen-task. We evaluate the model with 4 layers, dimension 192, 3 heads (64 dimensions per head), and a Feed-forward width of 384. Training uses AdamW with a weight decay of 0.01, dropout of 0.1, gradient clipping at 1.0, a batch size of 512. Arithmetic uses `bfloat16`. We run every cell with model seeds 2026, 2027, and 2028, and build every non-deterministic arm with construction seeds 2026, 2027, and 2028.

5 EXPERIMENTAL RESULTS

Table 1 describes the arms, Table 2 shows what each design holds fixed, and Table 3 gives bits per character for every arm.

Pair task. We start with a zero-training task. Figure 1(b,c) reports the three rates from §4.2. Joint consistency (our main metric) rises from 0.00 at $\pi = 0$ to 0.468 at $\pi = 1$. Stem agreement behaves differently: it is 0.223 at $\pi = 0$ and reaches 0.981 at $\pi = 1$. This shows that applying the *same* cut pattern to two related identifiers is easy. The aligned arm is stricter: it both cuts the shared stem consistently *and* separates the final letter as a suffix. The reuse-matched control highlights the difference (Figure 1c). It attains high stem agreement (0.884), close to the aligned arm, but it never isolates the suffix because each cut is shifted one character into the morpheme. Therefore, consistent cutting is not equivalent to correct cutting.

Seen task. For Seen task in WINMORPH, raising precision from $\pi = 0$ to $\pi = 1$ reduces bits per character from 2.80 to 1.39 under segmentation matching and from 2.81 to 2.32 under symbol matching. This corresponds to reductions of 50.5% and 17.44% (Table 3). Under segmentation matching, the aligned arm also makes more passes over the training data (11.76) than the zero-precision arm (3.95). Under symbol matching, all arms make 3.92 passes (Table 2). Even when we control for the mean number of training symbols and the number of passes, the aligned arm performs best. The effect of higher precision is not strictly monotonic: under symbol matching, bits per character are 2.43, 2.56, and 2.32 at $\pi = 0.50$, 0.75, and 1.00, respectively. The aligned arm also outperforms the reuse-matched control (bits per character 3.26). Overall, these results suggest that accurate segmentation benefits identifiers the model has already seen.

Unseen task. This experiment is our primary endpoint. On held-out identifiers, increasing precision from 0 to 1 reduces bits per character from 3.86 to 2.32 (-39.8%) under the segmentation-matched design and from 3.85 to 3.12 (-18.9%) under the symbol-matched design (Table 3). Bits per character decreases at every step in both precision sweeps. **Correct cuts therefore improve performance on identifiers the model has never seen, although the estimated gain differs by about a factor of two across the two designs.** Table 2 explains this gap. When cuts align with true boundaries, the tokenizer creates many fewer distinct token types ($38{,}219$ for the aligned arm vs. $131{,}730$ for the misaligned arm). As a result, almost all aligned types fit within the $32{,}768$ embedding rows, whereas only about one-quarter of the misaligned types fit. Token types that lack an embedding row are emitted as character sequences. Consequently, the aligned arm uses fewer symbols per identifier (4.79 vs. 14.23 for the misaligned arm). With a fixed symbol budget, this lets the aligned arm traverse the training set more often (11.76 passes vs. 3.95), so the 39.8% gain partly reflects additional training exposure. The symbol-matched design removes this exposure advantage by reducing the aligned arm's coverage to $2{,}825$ embedding rows. Under this design, all arms consume 14.36 symbols per training identifier and complete 3.92 passes. The remaining 18.9% therefore reflects performance under matched exposure, although coverage, parameter count, and held-out sequence length still vary across arms (Table 2).

What the control show. The **Reuse-matched** arm never cuts at true morpheme boundaries, but its token fragments repeat across identifiers more often than fragments produced by random cuts. Its reuse index (R) is 2.26, versus 1.61 for random cuts and 5.55 for the aligned arm (Table 1). On held-out identifiers, it captures 54% of the aligned arm's improvement under the symbol-matched design and 30% under the segmentation-matched design. This shows that reuse by itself explains a substantial portion of the gain; since this control still has less reuse than the aligned arm, reuse could account for an even larger share.

5.1 ABLATION: DOES THE RESULT DEPEND ON THE DOMAIN?

Windows identifiers include conventions that are largely absent from Hungarian, such as type prefixes, `A`/`W` suffixes, and a small vocabulary of subsystem terms. To test whether our results rely on these domain-specific cues, we hold the entire experimental setup fixed arm construction, target $F = 2.50$, budgets, model, and training procedure and change only the corpus. For the Hungarian dataset, we run each experimental cell with model seeds 2026, 2027, and 2028.

Table 3: Bits per character for every arm, both tasks, both designs, both corpora; lower is better. **Seg** and **Sym** denote the segmentation-matched and symbol-matched designs. Each cell is the mean over the three model seeds ± the half-width of a 95% confidence interval.

	WinMorph				Hungarian			
	Seen-task		Unseen-task		Seen-task		Unseen-task	
Arm	Seg	Sym	Seg	Sym	Seg	Sym	Seg	Sym
$\pi = 0.00$	2.80±0.03	2.81±0.09	3.86±0.07	3.85±0.08	3.03±0.03	3.05±0.05	3.69±0.07	3.68±0.06
$\pi = 0.25$	2.79±0.04	2.71±0.04	3.83±0.08	3.82±0.08	3.06±0.06	3.09±0.06	3.77±0.09	3.75±0.04
$\pi = 0.50$	2.67±0.08	2.43±0.08	3.67±0.05	3.66±0.05	2.75±0.03	2.92±0.04	3.55±0.07	3.49±0.05
$\pi = 0.75$	2.38±0.06	2.56±0.06	3.35±0.06	3.39±0.09	2.40±0.08	2.94±0.07	3.29±0.08	3.18±0.07
$\pi = 1.00$	1.39±0.07	2.32±0.10	2.32±0.04	3.12±0.04	1.96±0.07	2.81±0.03	2.17±0.10	2.56±0.11
Reuse-matched	2.73±0.05	3.26±0.07	3.39±0.07	3.46±0.06	2.39±0.06	2.61±0.05	3.25±0.05	3.08±0.04
Change, $\pi = 0 \rightarrow 1$ (%)	−50.5±1.1	−17.44±0.3	−39.8±0.7	−18.9±1.1	−35.4±0.8	−7.9±0.6	−41.2±0.6	−30.5±1.0
Reuse share of change (%)			30±0.6	54±0.8			29±1.0	54±1.1

Hungarian morphemes are reused much more frequently than Windows morphemes (reuse index 23.4 for the aligned arm vs. 5.55; Table 1). As a result, aligned cutting reduces the type inventory from 65,574 to 8,995. This also shortens the average input from 6.76 to 4.50 symbols per word and increases training exposure from 8.28 to 12.45 passes (Table 2). In the symbol-matched setting, all Hungarian arms are constrained to 6.81 symbols and 8.22 passes. Increasing precision from 0 to 1 reduces bits per character on held-out words by 41.2% under segmentation matching and by 30.5% under symbol matching (compared with 39.8% and 18.9% in WINMORPH). The **Reuse-matched** control again explains a substantial share of the improvement (54% under symbol matching; 29% under segmentation matching)

Two corpus differences align with this accounting. First, the symbol-matched gain on unseen words is larger in Hungarian than in WINMORPH (30.5% vs. 18.9%) because symbol matching removes less of the aligned arm's advantage in Hungarian: the aligned arm grows from 4.50 to 6.81 symbols per word, while in WINMORPH it grows from 4.79 to 14.36. Second, the segmentation-matched gain on seen words is smaller in Hungarian (−35.4% vs. −50.5%) because alignment increases training exposure less: from 8.28 to 12.45 passes in Hungarian (about a 1.5× increase) versus from 3.95 to 11.76 passes in WINMORPH (about a 3× increase). Consistent with this, under symbol matching the bits per character on seen Hungarian words *decrease* from 3.05 to 2.81 (−7.9%) as precision increases from 0 to 1, which we have same pattern in WINMORPH.

6 CONCLUSION

Evaluating whether splitting code identifiers at morpheme boundaries helps a language model is difficult because standard tokenizers confound two factors: more accurate boundaries usually also reduce the token count. We address this with WINMORPH, a corpus of 83,900 Windows identifiers with gold segmentations derived from published naming rules, and a benchmark that varies the fraction of correct boundaries from 0 to 1 while keeping the number of tokens per identifier fixed. The results support our claim that correct boundaries help beyond what shorter input explains: on unseen identifiers, using every correct boundary reduces bits per character by 39.8% under token-count matching and by 18.9% when the number of symbols read is also matched; corresponding reductions on 84,000 Hungarian words are 41.2% and 30.5% (Table 3). Thus, morpheme-consistent segmentation improves a model's ability to compose identifiers it has not seen before. We release WINMORPH as a controlled testbed for tokenizer design. Future work can use it to quantify how closely practical tokenizers match the reference boundaries, to test whether improving this match reproduces the gains we report, and to disentangle the effects of piece reuse from those of morphological meaning.

7 EXTRAS

AI USE STATEMENT

The authors used generative AI solely for proofreading and language editing during manuscript preparation. The authors reviewed all edits and take full responsibility for the content of this publication.

ETHICS STATEMENT

This work analyzes how tokenizers segment code identifiers. It does not involve human participants, personal data, or crowdsourced annotation. The reference segmentation for WINMORPH is derived from published naming conventions rather than subjective human judgments. The dataset includes only identifier strings and their segmentations, collected from headers, export tables, and code bases; it contains no implementation code and no user-generated content. All identifiers are public API names that Microsoft also documents. We release the dataset and code under the MIT license, consistent with the licenses of the source materials. The Hungarian data are taken from the publicly available SIGMORPHON 2022 shared task. We do not anticipate direct harmful uses of a dataset consisting of segmented identifier names. All experiments use small models (Section 4), and the computational cost is therefore low.

REPRODUCIBILITY STATEMENT

We provide an anonymized repository with the full experimental pipeline. It includes the code for constructing the arms, training the models, and reproducing all values in Tables 1–3. The repository also contains the WINMORPH dataset, the segmentation rules, and the script used to sample the Hungarian data. Corpus sizes and the 85/15 train/test split are reported in Section 4.1. Model architecture, optimization details, and the three model seeds and three construction seeds used for each cell are described in Section 4.

REFERENCES

Catherine Arnett and Benjamin Bergen. Why do language models perform worse for morphologically complex languages? In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 6607–6623, 2025.

Catherine Arnett, Marisa Hudspeth, and Brendan O'Connor. Evaluating morphological alignment of tokenizers in 70 languages. In *Tokenization Workshop*, 2025. URL https://openreview.net/forum?id=XYRri1s6pP.

Khuyagbaatar Batsuren, Gábor Bella, Aryaman Arora, Viktor Martinovic, Kyle Gorman, Zdeněk Žabokrtskỳ, Amarsanaa Ganbold, Šárka Dohnalová, Magda Ševčíková, Kateřina Pelegrinová, et al. The sigmorphon 2022 shared task on morpheme segmentation. In *Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 103–116, 2022.

Kaj Bostrom and Greg Durrett. Byte pair encoding is suboptimal for language model pretraining. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4617–4624, 2020.

Nadezhda Chirkova and Sergey Troshin. CodeBPE: Investigating subtokenization options for large language model pretraining on source code. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=htL4UZ344nF.

Marco Cognetta, Vilém Zouhar, Sangwhan Moon, and Naoaki Okazaki. Two counterexamples to tokenization and the noiseless channel. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16897–16906, 2024.

Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. Getting the most out of your tokenizer for pre-training and domain adaptation. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 9784–9805, 2024.

Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories (MSR)*, 2009.

Achille Fusco, Maria Letizia Piccini Bianchessi, Tommaso Sgrizzi, Asya Zanollo, and Cristiano Chesi. Linguistic units as tokens: Intrinsic and extrinsic evaluation with babylm. In *Proceedings of the First BabyLM Workshop*, pages 496–507, 2025.

Matthias Gallé. Investigating the effectiveness of bpe: The power of shorter sequences. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 1375–1381, 2019.

Alba Táboas García, Piotr Przybyła, and Leo Wanner. Exploring morphology-aware tokenization: A case study on spanish language modeling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 30493–30506, 2025.

Omer Goldman, Avi Caciularu, Matan Eyal, Kris Cao, Idan Szpektor, and Reut Tsarfaty. Unpacking tokenization: Evaluating text compression and its correlation with model performance. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2274–2286, 2024.

Thamme Gowda and Jonathan May. Finding the optimal vocabulary size for neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3955–3964, 2020.

Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

Valentin Hofmann, Janet B. Pierrehumbert, and Hinrich Schütze. An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2022.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-vocabulary models for source code. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020. doi: 10.1145/3377811.3380342.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, 2018.

Sabrina J Mielke, Ryan Cotterell, Kyle Gorman, Brian Roark, and Jason Eisner. What kind of language is hard to language-model? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4975–4989, 2019.

Aleksandar Petrov, Emanuele La Malfa, Philip Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages. *Advances in neural information processing systems*, 36: 36963–36990, 2023.

Craig W Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. Tokenization is more than compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 678–702, 2024.

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, 2016.

Jieke Shi, Zhou Yang, Junda He, Bowen Xu, and David Lo. Can identifier splitting improve open-vocabulary language model of code? In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1134–1138, 2022.

Omri Uzan, Craig W Schmidt, Chris Tanner, and Yuval Pinter. Greed is all you need: An evaluation of tokenizer inference methods. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 813–822, 2024.

Vilém Zouhar, Clara Meister, Juan Luis Gastaldi, Li Du, Mrinmaya Sachan, and Ryan Cotterell. Tokenization and the noiseless channel. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5184–5207, 2023.

A Limitations and Future Work

Limitations. Our benchmark addresses a controlled setting rather than a fully practical one. Because each arm is constructed using reference boundaries, the reported improvements reflect idealized tokenizers. We do not yet know whether standard tokenizers (e.g., BPE or Unigram), which do not observe these boundaries, can be reliably steered toward them and obtain similar gains. Different results may emerge for larger models or for identifiers in context, although we do not expect the direction of the effect to reverse.

Future work. These limitations suggest two directions. First, we should bridge the gap between our constructed arms and real-world tokenizers. Because WinMorph provides gold morpheme boundaries for each identifier, it can be used either to train tokenizers to split at these boundaries or to evaluate tokenizers that attempt to do so. This would clarify whether our reported gains carry over to practical tokenization pipelines. Second, we should isolate the source of the improvement by constructing arms that match the aligned tokenizer's level of piece reuse while keeping boundary precision fixed. We should also test whether the effect persists at larger scale, for identifiers embedded in full code context, and for other rule-governed identifier families beyond Windows.