YAKUBU GOWON UNIVERSITY, ABUJA · CAIIT

MSc in Embedded Artificial Intelligence

EAI 820 — Speech and Language Technologies

Week 1 · Summative Model Examination Questions with Answers

Turning Input into Tokens: Tokenization, Byte-Pair Encoding & Discrete Audio Tokens

Question 1 — Foundations of Tokenization

A sequence model operates over a fixed, finite vocabulary of discrete symbols, each mapped to a learnable embedding vector.

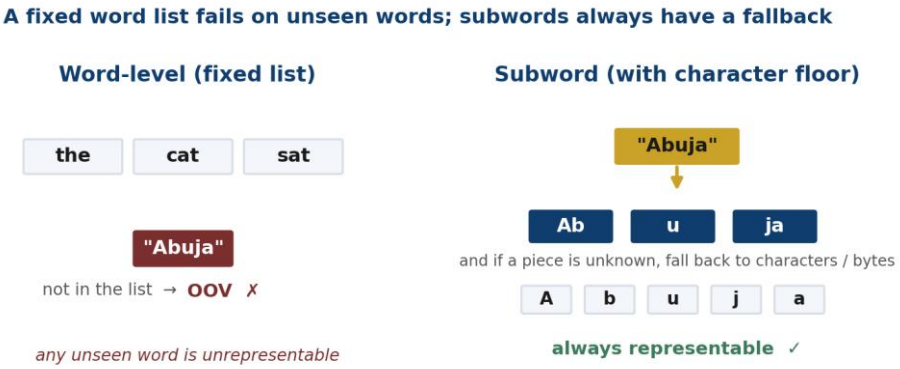


Figure Q1. *A fixed word list fails on an unseen word; subwords fall back to characters, so any string is representable.*

(a) Explain why neither a raw text string nor a raw audio waveform can be fed directly into such a model, and state precisely what tokenization provides that makes the input usable. **[6 marks]**

(b) Contrast representing each word by a one-hot vector over the entire word vocabulary with representing words by subword tokens. Focus your answer on the unknown-word (out-of-vocabulary) and long-tail problem. **[8 marks]**

(c) Explain why including the individual characters (or bytes) in the base vocabulary guarantees that any input string can always be represented — the "fallback" property. **[6 marks]**

MODEL ANSWER

(a) A sequence model consumes a sequence of **discrete symbols drawn from a fixed, finite vocabulary**, each looked up in an embedding table. Raw text arrives as characters or bytes and raw audio as a stream of continuous amplitude samples (about sixteen thousand per second) — neither is a sequence of linguistic units the model can embed. Audio is worse than merely inconvenient: it is *continuous*, so there is no finite symbol set at all. Whole words are also unsuitable because the set of possible words is effectively unbounded (an open vocabulary). Tokenization solves the interface problem: it converts messy, real input into a sequence of tokens drawn from a bounded, finite vocabulary, each of which can be embedded and processed.

(b) A **one-hot word vector** has one dimension per vocabulary word, so its dimension equals $|V|$ — tens or hundreds of thousands — and it is almost entirely zeros. More seriously, it can only represent words that were placed in the vocabulary in advance; every other word is **out-of-vocabulary (OOV)** and has no representation at all. Because the tail of rare and novel words — proper names, borrowings, misspellings, freshly coined terms, and the many inflected forms of morphologically rich languages — is unbounded, a fixed word list always misses some, and the miss is unrecoverable. **Subword**

tokenization instead keeps a modest vocabulary of subword units and represents any word as a sequence of them: frequent words become a single token, while rare or unseen words are decomposed into known pieces. Nothing is un-representable, so the OOV / long-tail problem is dissolved rather than merely reduced.

(c) Byte-pair encoding builds its vocabulary by starting from the individual characters (or, in byte-level variants, the 256 possible bytes) and only ever **merging pieces that already exist**. Consequently there is always at least one valid tokenization of any string: in the worst case the string is left as its raw characters or bytes. Because every Unicode character decomposes into bytes and all 256 byte values are in the base vocabulary, a byte-level tokenizer can represent *any* string whatsoever — there is no true OOV at the character/byte level. This is the fallback property: merges make common strings compact, but the character/byte floor guarantees universal coverage.

Marking guidance — *(a) finite discrete vocabulary + embeddings and both modalities named (3), why raw forms fail incl. audio being continuous (3). (b) OOV/long-tail explained as unbounded and unrecoverable (5); size/sparsity of one-hot (3). (c) merges only combine existing pieces → character/byte floor guarantees coverage (6).*

Question 2 — Byte-Pair Encoding by Hand

A training corpus consists of the following words with their frequencies of occurrence: **hug (10), pug (5), pun (12), bun (4), hugs (5).** Byte-pair encoding is to be learned from this corpus. (For simplicity, ignore end-of-word markers.)

(a) Write the initial (character-level) split of each word and state the base vocabulary. **[4 marks]**

(b) Perform the first three (3) merges. At each step, identify the most frequent adjacent pair and its count, state the merge, and give the corpus after the merge. Show your working. **[12 marks]**

(c) Using the three merges you learned, in order, tokenize the unseen words *"bug"* and *"mug"*. Explain what happens in each case. **[6 marks]**

(d) State how the size of the final vocabulary is controlled, and what determines when merging stops. **[3 marks]**

MODEL ANSWER

(a) Each word is split into characters. Initial splits: **h u g (10), p u g (5), p u n (12), b u n (4), h u g s (5)**. The base vocabulary is the set of characters that appear: **{ b, g, h, n, p, s, u }** — seven symbols.

Working through the three merges (pair counts are weighted by each word's frequency):

Step	Most frequent pair	Count	Merge → new token	Corpus after merge
1	(u, g)	20	u + g → ug	h ug (10) · p ug (5) · p u n (12) · b u n (4) · h ug s (5)
2	(u, n)	16	u + n → un	h ug (10) · p ug (5) · p un (12) · b un (4) · h ug s (5)
3	(h, ug)	15	h + ug → hug	hug (10) · p ug (5) · p un (12) · b un (4) · hug s (5)

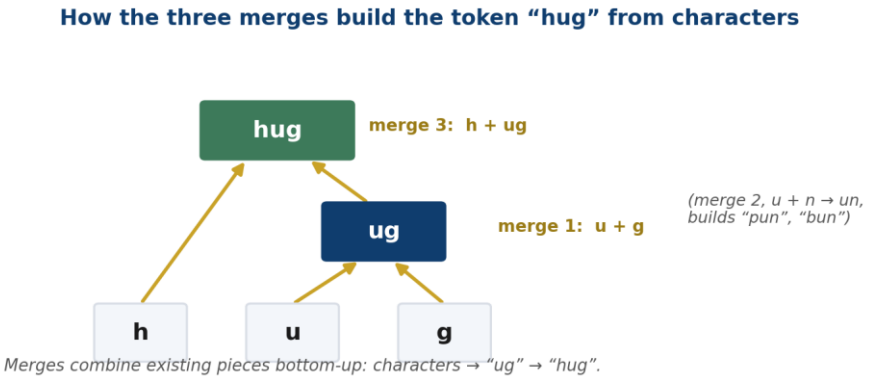


Figure Q2. *The learned merges combine existing pieces bottom-up: characters → "ug" → "hug".*

(b) Step 1: the candidate pairs and counts are (h,u)=15, (u,g)=20, (p,u)=17, (u,n)=16, (b,u)=4, (g,s)=5; the maximum is (u,g)=20, giving the token **ug**. Step 2: now (u,n)=16 is the maximum (over (h,ug)=15, (p,u)=12, (p,ug)=5, (ug,s)=5, (b,u)=4), giving **un**. Step 3: (h,ug)=15 is the maximum (over (p,un)=12, (p,ug)=5, (ug,s)=5, (b,un)=4), giving **hug**. The ordered merge list is therefore: 1) u+g→ug, 2) u+n→un, 3) h+ug→hug.

(c) Apply the merges in the order learned. *"bug"* → b u g; the **ug** merge applies (un and hug do not), giving **[b, ug]**. Every piece is in the vocabulary, so "bug" is represented even though it never appeared in training — this is the OOV-robustness of subwords. *"mug"* → m u g; the letter **m** never occurred in the corpus, so it is not in the base vocabulary. Under this character-level scheme "mug" cannot be fully represented (the m maps to an unknown symbol). This is exactly the case that **byte-level** byte-pair

encoding fixes: because every character decomposes into bytes and all byte values are in the base vocabulary, even a never-seen character is representable.

(d) The number of merges is a hyperparameter; each merge adds exactly one token, so the **final vocabulary size ≈ (number of base characters) + (number of merges)**. Merging therefore stops when the chosen target vocabulary size is reached (equivalently, after a fixed number of merges). Larger targets give longer merge lists and larger vocabularies.

Marking guidance — *(a) correct splits (2) + base vocab of 7 characters (2). (b) 4 marks per merge for the correct top pair, its count, and the resulting corpus — full marks require the counts shown, not just the answer. (c) [b, ug] with reasoning (3); "m" OOV at character level and the byte-level fix (3). (d) vocab size = base + merges, stop at target size (3).*

Question 3 — Subwords, Morphology and Audio Tokens

(a) Explain, with a worked example, how subword tokenization helps a model handle morphologically productive words (many forms built from a stem) and rare long-tail words. **[6 marks]**

(b) A neural audio codec turns a waveform into a sequence of discrete audio tokens. Explain the parallel between this and subword tokenization of text, and why this idea is described as "the move that merged" natural language processing and speech processing. **[6 marks]**

(c) State the central trade-off in choosing a small versus a large subword vocabulary. **[3 marks]**

MODEL ANSWER

(a) Morphologically productive languages build many surface forms from a stem plus affixes. A word-level model treats *walk, walks, walking, walked* as four unrelated symbols and cannot represent a form it never saw. Subword tokenization splits them into shared pieces — **walk + s, walk + ing, walk + ed** — so the model reuses the single unit "walk" across all of them and generalises to an unseen combination such as "rewalking" → re + walk + ing. For the **long tail**, a word that occurs zero times in training is still assigned a sensible representation as a sequence of common subwords, so rare and novel words degrade gracefully instead of failing.

(b) A neural audio codec learns, by vector-quantising short frames of sound, a **finite vocabulary of discrete audio tokens** — exactly as byte-pair encoding learns a finite vocabulary of subword text tokens. Once a waveform is a sequence of discrete tokens, the very same sequence-model machinery used for text (embed the tokens, predict the next token) applies to speech. This is why it is called the move that merged the two fields: natural language processing (language as text) and speech processing (language as sound) had used different toolkits for half a century, but discretising audio recast speech as **"predict the next token over a finite vocabulary"** — the same problem as text, solvable with the same architecture.

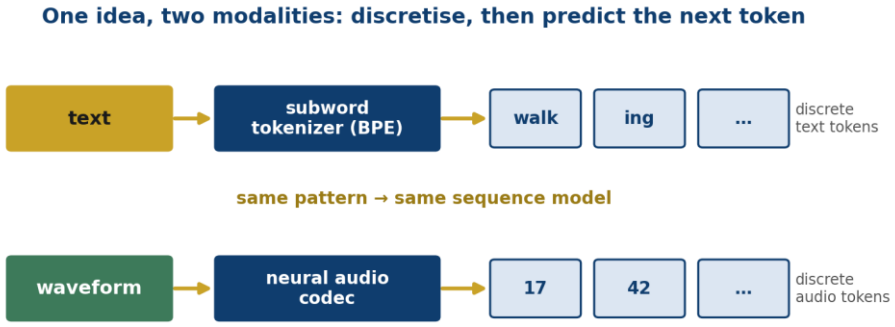


Figure Q3. *Text and audio follow the same pattern — discretise into tokens, then run one sequence model.*

(c) A **small** vocabulary means shorter tokens and therefore longer token sequences (more time steps and compute, longer context needed) but more sharing and robustness to rare words; a **large** vocabulary means each token carries more and sequences are shorter, but the embedding table is bigger (memory) and rarer tokens make the data sparser. The choice balances sequence length against table size and sparsity.

Marking guidance — *(a) shared-stem reuse with an example (4) + long-tail graceful degradation (2). (b) codec learns discrete audio vocabulary paralleling BPE (3) + both become next-token prediction, unifying the fields (3). (c) sequence length vs table size / sparsity (3).*

Question 4 — Application — A Multilingual Voice Assistant for Nigeria

Real-life scenario. A start-up in Abuja is building a voice assistant that must understand Hausa, Yoruba and Igbo as well as English. Only modest text corpora are available for the three Nigerian languages, and users frequently code-switch — mixing English words into a Hausa or Yoruba sentence within a single utterance.

(a) Explain why subword tokenization is preferable to word-level tokenization for this product, referring specifically to out-of-vocabulary words, morphology, and code-switching. **[8 marks]**

(b) The team trains a single shared subword vocabulary across all four languages rather than one vocabulary per language. Give two advantages and one significant risk of this decision. **[7 marks]**

(c) Yoruba text uses letters with diacritics and tone marks (for example ọ, ẹ, ṣ). Explain, in terms of Unicode and character handling, why inconsistent diacritics can fragment tokens and harm the model, and give one mitigation. **[5 marks]**

MODEL ANSWER

(a) Word-level tokenization fails on every count here. With only modest corpora, the great majority of word forms are unseen, so a fixed word list produces constant **out-of-vocabulary** failures that cannot be recovered. Hausa, Yoruba and Igbo are **morphologically rich** — extensive affixation, and reduplication in Yoruba and Igbo — so a single stem generates many surface forms a word list will never fully cover, whereas subwords share the stem across forms. And because users **code-switch**, a single subword inventory spanning all four languages tokenizes an English word dropped into a Hausa sentence with the same machinery, with no separate path or language detector required mid-utterance. Subwords also absorb spelling variation gracefully. In short, subword tokenization gives universal coverage from small data, which word-level cannot.

(b) Advantages of one shared vocabulary: (1) **parameter efficiency** — a single embedding table serves all languages, which matters for an eventual on-device product; and (2) **cross-lingual transfer** — shared pieces (the common Latin script, shared affixes, loanwords and cognates) let the three low-resource languages borrow statistical strength from English and from one another, and code-switching is handled natively. The significant **risk** is domination: because English (and whichever language has the most data) contributes the most frequent pairs, the merges are learned largely in its favour, so the low-resource languages are **over-fragmented** — their words split into many short tokens, lengthening sequences and degrading quality. The usual remedy is to balance the corpora when learning the vocabulary (for example by temperature up-sampling the smaller languages).

(c) A character such as Yoruba ọ or ṣ can be encoded in Unicode either as a single **precomposed codepoint** (normalization form NFC) or as a base letter followed by a separate **combining diacritic** (form NFD). If the training data mixes the two forms, "the same" letter appears to the tokenizer as two different byte sequences, so it is treated as two different symbols. The statistics for that letter are split across both forms, the affected words fragment into different tokens, and a piece that should have been common instead looks rare — which hurts learning and inference. The standard mitigation is to apply **consistent Unicode normalization** (for example NFC) to all text during preprocessing, so every occurrence of a diacritic letter has one canonical encoding; byte-level tokenization combined with normalization is also robust.

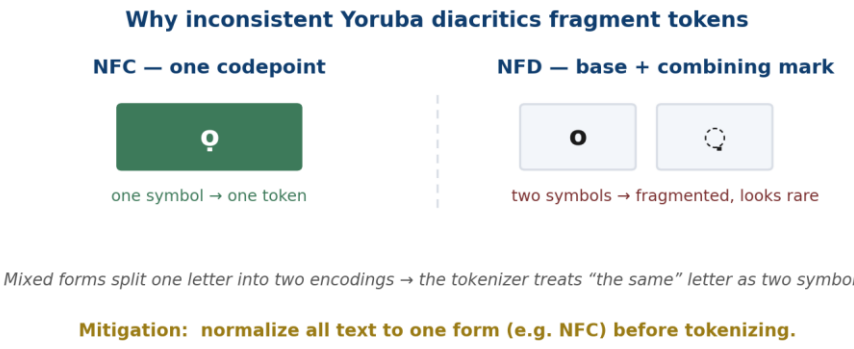


Figure Q4. *NFC encodes a diacritic letter as one codepoint; NFD splits it into base + combining mark, fragmenting tokens. Normalize to one form.*

Marking guidance — *(a) OOV from small data (3), morphology/reduplication sharing (3), code-switching handled by one inventory (2). (b) two valid advantages (2+2), the domination/over-fragmentation risk with a balancing remedy (3). (c) NFC vs NFD splitting a letter into two symbols and fragmenting statistics (3) + normalization mitigation (2).*

Question 5 — Application — On-Device Speech on Constrained Hardware

Real-life scenario. An embedded team is building an on-device Hausa command recogniser for a low-power microcontroller, and later a small speech interface. They wish to use one architecture for both text and audio, and the device has very little memory and no reliable network.

(a) Explain how representing audio as discrete tokens (via a neural codec) lets the team reuse the same sequence-model machinery for speech as for text, and why that unification is valuable for an embedded pipeline. **[7 marks]**

(b) In testing, proper nouns (place names, personal names, drug names) and freshly coined slang are frequently unknown to the recogniser. Using the long-tail argument, explain why a subword or byte-level tokenizer is more robust here than a fixed word list, and state the cost. **[7 marks]**

(c) On this microcontroller the vocabulary/embedding table is a large fraction of the model's memory. Discuss how the choice of tokenization granularity — characters, subwords, or words — interacts with memory use and sequence length. **[6 marks]**

MODEL ANSWER

(a) A neural codec turns the microphone waveform into a sequence of **discrete audio tokens**, so the command recogniser (and later the speech interface) operates on a token sequence just as a text model does. The team can therefore run both modalities on **one sequence-model stack**: a single architecture, one training and optimisation toolchain, and one set of compression and quantisation steps reused for text and speech. Discrete audio tokens are also small integer indices, which are friendly to the fixed-point, quantised inference used on microcontrollers. For an embedded pipeline this unification means less code to write, verify and fit into limited flash, and a single deployment path rather than two.

(b) Place names, personal and drug names, and new slang are precisely the **unbounded long tail**: no fixed word list can enumerate them, and any word missing from the list is an unrecoverable out-of-vocabulary failure. A **subword or byte-level tokenizer** represents any such item as a sequence of known pieces — down to bytes in the worst case — so nothing is un-representable and the recogniser **degrades gracefully**, spelling out an unfamiliar name from subwords rather than failing outright. The **cost** is that rare items split into several tokens, so their sequences are longer (more compute, energy and latency) and each piece carries less meaning, making accuracy on those items lower than on common words — but non-zero, which is exactly the improvement over a word list that scores them at zero.

(c) The embedding table's size is proportional to (vocabulary size × embedding dimension), and on a tiny device it can dominate the model. The granularity choice is therefore a direct memory-versus-latency knob. **Characters**: a very small vocabulary and a tiny table, but very long sequences — more time steps, more compute and energy, and harder long-range modelling. **Words**: short sequences but an enormous table that may not fit in memory at all, plus poor out-of-vocabulary behaviour. **Subwords** are the middle ground: a modest vocabulary (a few thousand units) keeps the table small enough while keeping sequences reasonably short. On the tightest devices one deliberately shrinks toward a smaller subword or byte vocabulary to reduce the table, accepting longer sequences as the price of fitting in memory.

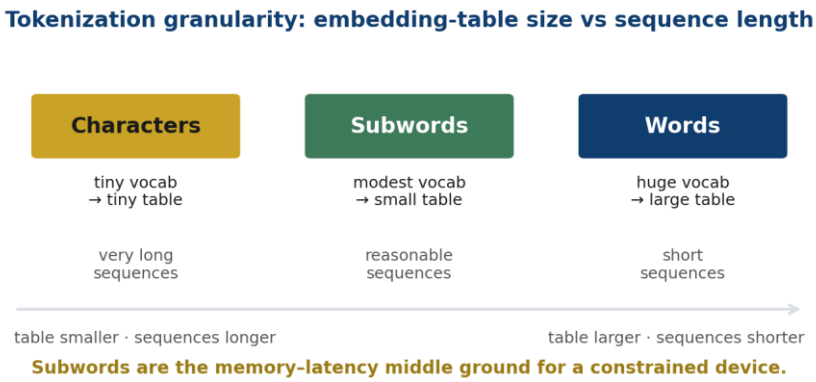


Figure Q5. *Granularity trades embedding-table size against sequence length; subwords are the memory–latency middle ground.*

Marking guidance — *(a) codec → discrete tokens → shared sequence stack (4); embedded value: one toolchain, integer tokens suit quantised inference (3). (b) long tail is unbounded and word lists fail unrecoverably (3); subword/byte gives coverage and graceful degradation (2); cost of longer sequences and lower per-item accuracy (2). (c) table size $\propto$ vocab × dim (2); characters (tiny table/long sequences), words (huge table/short), subwords as the memory-latency middle ground (4).*