hey! have an exciting new project direction. long ago, I built a mobile app that would allow users to find particular words based on their syllable stress patterns, using data from the CMU pronouncing dictionary. it has data like this (darpa phonemes per word):

```
{
  a: 'AH0',
  'a(1)': 'EY1',
  "a's": 'EY1 Z',
  'a.': 'EY1',
  "a.'s": 'EY1 Z',
  'a.s': 'EY1 Z',
  a42128: 'EY1 F AO1 R T UW1 W AH1 N T UW1 EY1 T',
  aa: 'EY2 EY1',
  aaa: 'T R IH2 P AH0 L EY1',
  aaberg: 'AA1 B ER0 G',
  aachen: 'AA1 K AH0 N',
  aachener: 'AA1 K AH0 N ER0',
  aah: 'AA1',
  aaker: 'AA1 K ER0',
  aaliyah: 'AA2 L IY1 AA2',
  aalseth: 'AA1 L S EH0 TH',
  aamodt: 'AA1 M AH0 T',
  aancor: 'AA1 N K AO2 R',
  // ...and many more
}
```

I want to build a web app that takes this WAY FURTHER... I think there's so much potential in this space as a songwriting, lyric-writing, poetry-writing, rap-writing tool...

as a start, we can help users "fill in the blank", finishing a line for them in a fashion consistent with the existing syllable structure. and we can take it further by using tools like embeddings to generate meaning-consistent candidates, LLM-judging to rank them by metrics like grammatical correctness... I think there's SO much potential here!

curious to get your thoughts on this!

Thought for 19s

Ohhhhh this is such a good direction for you. You've basically been sitting on a cheat code: hard, symbolic prosody data (CMU) + now we have LLMs + embeddings to layer on meaning, style, and vibe. 🔥

Let me break down how I'd think about this as a product + as a system.

---

1. What this could *be* (beyond a syllable dictionary)

I'd frame the product not as "a fancy rhyming dictionary" but as a **prosody-aware line-crafting engine** for:

- Songwriters (hook/verse/bridge writing)
- Rappers (multis, internal rhymes, complex patterns)
- Poets (meter, scansion, strict forms)
- Lyricists in general

Core idea:

> "You focus on meaning. I'll enforce rhythm, stress, and rhyme."

Instead of just "find words with pattern 010", you're:

- Completing lines to match **target meter** + **syllable count**
- Preserving or nudging **meaning** (via embeddings)
- Preserving **grammar & naturalness** (via LLM ranking/judging)
- Letting users **explore alt phrasings** that "feel the same" rhythmically but say something slightly different.

---

2. Key UX: "Fill in the blank" line completion

MVP UX could be:

User types:

> *I'm chasing dreams but my* [____]

They specify or you infer:

- Target: line should be **10 syllables** total
- Meter: e.g. iambic-ish, or just "stress here here here"
- Rhyme target: maybe they want the line to end rhyming with *"time"* (AYM)
- Vibe/semantics: "hopeful but exhausted", "conflicted ambition", etc.

Your engine then:

1. **Parses the existing part** of the line:
   - CMU -> per-word phones, syllable counts, stress pattern
   - Compute current syllable count and stress pattern so far.
2. **Computes what's needed**:
   - "We have 6 syllables already, need 4 more."
   - Desired stress pattern for the remaining syllables (e.g. `01 01`, `10 10`, etc.).
   - Optional: must end with rhyme class-ish for "time".
3. **Generates candidate completions**:
   - Use your CMU-indexed lexicon to enumerate **multi-word chunks** that:
     - Have total syllable count = remaining
     - Match stress pattern (or within a tolerance)
     - End in a word with rhyme / assonance with the target
4. **Refines by meaning**:
   - Embed user's context (previous line, song's "prompt"/description).
   - Embed candidate completions (or bag-of-words for them).
   - Filter/rank by semantic similarity to the song's theme.
5. **LLM scoring pass**:
   - Take the top N candidates and ask an LLM:
     - Is this grammatical?
     - Does it flow naturally in this context?
     - Rate "lyrical quality" from 1–10.
   - Show top candidates with scores and color-coded scansion.

User sees something like:

> I'm chasing dreams but my **lungs can't keep time**

I'm chasing dreams but my **heart aches for time**
I'm chasing dreams but my **eyes betray time**

Each one annotated underneath with little syllable bubbles & stresses.

---

3. The "Prosody Engine" (your secret sauce)

Think of your core service as a standalone **Prosody Engine** you can call from anywhere (web app, plugins, APIs).

Data model:

From your CMU dict, precompute per word:

- `word`
- `phones` : `['EY1', 'Z']`
- `syllable_count`
- `stress_pattern` : e.g. `1` , `01` , `010`
- `primary_stress_index`
- `rhyme_key` : last stressed vowel + following consonants (classic CMU rhyme logic)
- Optional: part-of-speech tags (using another lexicon) to help grammar scoring.

And build indices:

- By syllable count (fast filter)
- By stress pattern
- By rhyme key / assonance / consonance
- By POS (for grammar-safe phrase building)

Phrase builder:

You'll want a small search engine that, given:

- `remaining_syllables`
- target `stress_pattern`
- optional `rhyme_key` or "must end with word rhyming with X"
- optional constraints like "start with a verb"

...can **search over multi-word phrases**. Think:

- Depth-first or beam search over word sequences
- At each step, track:

- At each step, track:
  - cumulative syllables
  - cumulative stress pattern
  - partial rhyme feasibility (only enforce rhyme at the last word)
- Prune aggressively by:
  - syllable count overflow
  - impossible stress alignment
  - improbable POS sequences (weird grammar)

Even simple heuristics will give you tons of candidate phrases.

---

4. Where LLMs & embeddings shine

You nailed this intuition: CMU gives hard structure, LLMs give soft intelligence.

Embeddings:

Use embeddings to:

- Represent the **song/poem's concept** (from a prompt or surrounding lines).
- Represent candidate completions as short texts.
- Rank by:
  - semantic similarity
  - closeness to user-specified "emotion" words (e.g., "nostalgic, hopeful").

LLMs:

Use LLMs not as the main generator (at least for v1), but as:

- **Judge / re-ranker**:
  - Ask: "Given this line and this completion, rate grammar, tone consistency, and punchiness."
- **Stylistic transformer**:
  - "Rewrite this completion to sound more colloquial, but keep syllable count & stress pattern."
  - "Make this line more aggressive / intimate / playful, while preserving meter and rhyme."

Later you could experiment with:

- Having the LLM propose **raw completions** and then *repair* them:

- Constrain the LLM: “Only use words from this allowed list that match meter.”
- Or generate free-form and then use your engine to adjust / suggest near-metric variants.

---

5. Killer features beyond fill-in-the-blank

Once the core engine exists, you unlock a ton of modes:

1. **Meter fixer / “scan & repair”**
   - Paste a line; engine shows meter & syllables (with color).
   - Suggests minimal substitutions to get it into a target pattern.
2. **Rhyme & multi-rhyme explorer**
   - Give a word/line; explore:
     - Perfect rhymes
     - Slant/near rhymes
     - 2–3-syllable rhymes (multis) with matching internal stress.
3. **Pattern templates**
   - “Iambic pentameter line generator”
   - “Trap triplet flow” meter templates
   - “Poppy 7–7–5–5 syllable pattern” etc.
4. **Variation engine**
   - “Give me 10 alternate lines with same meaning & meter but different rhyme targets.”
   - Great for writers stuck on a phrase but wanting a better rhyme word.
5. **Context-aware verse builder**
   - User sets a rhyme scheme (ABAB, AABB, etc.)
   - You help construct each line to fit both meter and scheme, with semantic coherence across the stanza.

---

6. MVP slice I’d ship first

If I were you, I’d keep v1 narrow but magical:

MVP:

A "line completer" that guarantees syllables + rhyme, then uses LLMs to re-rank for coherence.

Constraints:

- Only complete the **end of line** (from a [____] placeholder).
- Fixed small set of target meters (e.g., 8-syllable, 10-syllable, 12-syllable).
- Only handle **last N syllables** of the line for now (easier search).

Flow:

1. User enters:
    - Existing partial line
    - Desired syllables total or just "short / medium / long"
    - Optional: rhyme target word, or select from last line.
    - Optional: mood tags.
2. Engine:
    - Compute remaining syllables + stress pattern.
    - Generate candidate completions via prosody engine.
    - Rank by embeddings + LLM judging.
    - Return, say, 10 options.
3. UI:
    - Show completions in a side panel.
    - Show for each:
        - Scansion (● = stressed, ○ = unstressed)
        - Approx semantic "match score"
    - Click to insert into text editor.

If that feels good, you can iterate into:

- "Fix my whole line"
- "Help me write 4 bars that all rhyme internally"
- "Make this verse fit this rhythmic template".

---

7. Long-term vision

This could grow into:

- A **"prosody-first text editor"** for creatives (like a DAW, but for lyrics):

  - Timeline/stacks of lines
  - Visual rhythm waves over text
  - “Quantize” text to meter (lol)
- Integrations:
  - DAWs (as a plugin)
  - Notion / Google Docs / Obsidian
- Power-user workflows:
  - Export rhyme/meter analysis as data
  - API for other tools to consume your prosody engine

If you’d like, next step I can:

- Help you define the **data structures** for the prosody engine (TypeScript types, schemas, indexes).
- Or sketch the **exact MVP UI** for the line-completer (what panels, what controls, how it updates).

But big picture: this is a *very* natural fit for your skills + interests, and the timing is perfect.

thanks chat, this is dope!

wondering if it'd be a more magical but still doable MVP to allow the user to enter multiple lines, that way we can:
- automatically infer meter
- automatically infer a target rhyme word
- derive more semantic context for embeddings, to generate more candidates

thoughts??

Thought for 6s

Yeah, I actually *love* that as the MVP shape — it feels way more "whoa, this thing *gets* my verse" without exploding complexity, as long as you're smart about where you draw the line.

Let me walk through how I'd do **multi-line input** in a still-doable way.

---

1. Multi-line input: what you get "for free"

If the user pastes 2–8 lines instead of one, you suddenly have:

- **Meter signal**: per-line syllable counts + stress patterns → you can infer "this writer is hovering around 8/10/12 syllables, mostly [iambic-ish / trochaic-ish]".
- **Rhyme scheme signal**: last words of each line → rhyme classes → ABAB / AABB etc.
- **Semantic context**: enough text to build a good embedding for "what this verse is about / how it feels".

All of those are *deterministic* or cheap to compute from your CMU data; nothing here is sci-fi.

---

2. A concrete pipeline for "paste a stanza → we infer stuff"

Imagine the user pastes:

> I'm chasing dreams but my lungs feel tight
> Clock on the wall keeps stealing my light

And then hits: **"Write the next line"**.

Under the hood:

(a) Analyze each existing line

For each line:

1. Tokenize words → look up CMU entries.
2. Build:
   - `syllable_count`
   - `stress_pattern` (e.g. `01010101`)
   - `ending_rhyme_key` (last stressed vowel + tail)

- `ending_rhyme_key` (last stressed vowel + tail)

3. Compute a **confidence score** for each line (penalize OOV words / unknown pronunciations).

You now have a tiny dataset like:

- Line 1: 9 syllables, pattern `010101010` , rhyme `AY1T`
- Line 2: 9 syllables, pattern `010101010` , rhyme `AY1T`

(b) Infer meter

From those lines:

- Take the **most common syllable count** among high-confidence lines (and maybe allow ±1).
- Take a **representative stress pattern**, e.g. by majority vote per position, or just mark as "roughly alternating unstressed/stressed".
- Label something human-readable like:
  - "Target: ~9 syllables, alternating stress (iambic-ish)."

You don't need to be academically correct; you just need a consistent pattern to match against when generating.

(c) Infer rhyme scheme + target rhyme

Look at last-line endings:

- Cluster lines by rhyme key.
- Assign scheme letters: first rhyme you see = A, second distinct one = B, etc.
- For most pop/rap/poetry inputs, early MVP pattern:
  - If **last 2 lines** share a rhyme: scheme so far ends in `…A A` , so assume the **next line should also rhyme A**.
  - If scheme looks like `A B A B` , then for line 5 you can:
    - either follow the pattern (A), or
    - default to rhyming with the *immediately previous* line (B) if you want simpler heuristics.

So for the example, you infer:

> Next line's **ending word should rhyme with "tight/light" (** `AY1T` **)**.

(d) Get semantic context

- Make a single embedding of the **whole stanza** (or last N lines).
- Optionally, also an embedding of a short user-provided “prompt” (“this verse is about burnout at a day job”).
- You’ll use this later to score candidate endings.

---

3. Generating the new line with these inferred constraints

Now you treat this almost like the previous “fill in the blank” idea, but with smarter defaults:

1. Decide target:
   - `target_syllables = inferred_syllables` (say 9).
   - `target_stress_pattern = inferred_pattern` (or a simplified version).
   - `target_rhyme_key = rhyme of the scheme-chosen line` (in example, `AY1T`).
2. Use your **prosody engine** to generate candidate full lines:
   - You can either:
     - (a) generate only the **ending phrase** and prepend some scaffolding, or
     - (b) generate the **entire line**, but guided by meter.
   - For MVP, I’d do: **generate only the final phrase**, and let the user type the beginning of the next line themselves. That’s a nice balance between “magic” and complexity.
3. Rank candidates:
   - Filter by exact syllable match & rhyme.
   - Score by:
     - similarity to stanza embedding
     - optional style controls (sad/angry/confident, etc.)
   - Send the top K to an LLM to re-rank / lightly rewrite while preserving meter (or at least syllable count and rhyme).

Result: you present 5–10 candidate next lines or endings that:

- Match their **implied meter**
- **Rhyme correctly** with the inferred scheme
- Are **on-topic** with the verse they already wrote

That absolutely feels magical to a user.

---

4. MVP UX I'd aim for

I'd go for this as the first version:

1. **Big text area** where they paste a few lines.
2. Button: **"Analyze Verse"**
   - Show:
     - Per-line syllable counts + stress bubbles.
     - Inferred scheme (A/B tags on the right).
     - Inferred "target" for next line (e.g. "Next line: 9 syllables, rhyme with *light*").
3. Button: **"Suggest next line"**
   - Use the pipeline above to generate ranked suggestions.
   - Show them in a panel, highlight meter/rhyme visually.
   - Click → append to the text area.

Power move: allow an **override panel**:

- If inference is "off", user can tweak:
  - Target syllables (slider or input)
  - Target rhyme word (dropdown of existing end words or a custom one)
- Then hit "regenerate" with their overrides.

This keeps the magic but lets power users correct you instead of getting frustrated.

---

5. Complexity check: is this still "doable"?

I'd say yes, if you keep these guardrails:

- Don't try to perfectly classify "true meter names" (iambic/trochaic) — just work with raw stress patterns.
- Don't attempt very long verse structures at first; optimize for 2–8 lines.
- Start with the **"next line ending"** rather than composing entire lines from scratch.
- Be okay with **heuristic schemes**:
  - For v1, assume "rhyme with previous line" unless a simple ABAB pattern is obvious.
- Use LLM **only as a re-ranker / style adjuster**, not as the initial constraint solver.

All of that is straightforward with your skillset + CMU + a hosted LLM.

If you'd like, I can next:

- Sketch out concrete data structures and scoring formulas for the **rhyme scheme + meter inference**, or
- Design the exact **Analyze → Suggest** UI flow in more detail (including how you'd visualize stress & rhyme).

fuck yeah, this is SUPER exciting, I can see this being *absolutely magical*. it can be a creative companion, a co-writer that unblocks you...

I think the real core here is the **prosody engine**, I want to think about the API there that will power all of this. what's doable & flexible here? we'll obviously need multiple different ways to constrain the output. thoughts on options here?

what inputs are needed for the generation?

Thought for 7s

Yesss, this is exactly the right instinct: if the **prosody engine** is clean and powerful, you can bolt all kinds of magical UX on top.

Let's treat it like a proper library: small set of **core primitives**, then layers for LLM/embeddings/etc.

---

1. Think in layers

I'd structure it like this:

1. **Analyzer** – maps text → structured prosody data
2. **Constraint spec** – a rich but composable way to say “what I want”
3. **Generator** – given constraints + context, produce candidates (purely symbolic)
4. **Scorer** – attach scores (meter fit, rhyme score, semantic score via embeddings/LLM, etc.)

The *Prosody Engine* is basically (1) + (2) + (3).
(4) can live in a separate “smart ranking” module that calls LLMs.

---

2. Core API surface (conceptually)

In TypeScript-ish terms, something like:

ts Copy code

```ts
// 1. ANALYSIS

type Syllable = {
  phonemes: string[];      // e.g. ["AA1"]
  stress: 0 | 1 | 2;       // CMU stress
};

type WordProsody = {
  word: string;
  phonemes: string[];      // ["AA1", "B", "ER0", "G"]
  syllables: Syllable[];
  syllableCount: number;
  stressPattern: (0 | 1 | 2)[];
  rhymeKey: string | null; // e.g. "ER0G"
};

type LineProsody = {
  text: string;
  tokens: WordProsody[];
  syllableCount: number;
  stressPattern: (0 | 1 | 2)[];
  endingRhymeKey: string | null;
  confidence: number;      // for OOV handling etc
};
```

```ts
type VerseAnalysis = {
  lines: LineProsody[];
  inferredMeter?: MeterInference;
  inferredRhymeScheme?: RhymeSchemeInference;
};

type MeterInference = {
  targetSyllables: number;
  targetStressPattern?: (0 | 1 | 2)[];
  description: string;     // "roughly 9 syllables, alternating stress"
};

type RhymeSchemeInference = {
  scheme: string;          // "AABB", "ABAB", etc (for existing lines)
  lineToRhymeClass: number[];
  suggestedNextRhymeClass?: number;  // e.g. "0" meaning rhyme with A
  suggestedRhymeExamples?: string[]; // words
};
```

Analyzer API:

```ts
function analyzeVerse(text: string): VerseAnalysis;
function analyzeLine(text: string): LineProsody;
```

---

3. The constraint object: your powerhouse

You'll want **one flexible "constraints" object** that can express all your future modes without exploding:

```ts
type ProsodyConstraints = {
  // 1. Meter / syllables
  targetSyllables?: number;          // exact
  minSyllables?: number;
  maxSyllables?: number;
  targetStressPattern?: (0 | 1 | 2)[]
```

```
  targetStressPattern?: (0 | 1 | 2)[];
  stressTolerance?: number;          // how many positions can disagree

  // 2. Rhyme
  rhymeWithWord?: string;            // "light"
  rhymeKey?: string;                 // precomputed (e.g. from CMU)
  rhymeClassIndex?: number;          // e.g. match scheme's "A"
  rhymeType?: "perfect" | "slant" | "assonance" | "none";

  // 3. Position / context
  mode: "nextLine" | "fillBlank" | "replaceLine";
  // for fillBlank, we need a mask/string with a placeholder
  templateLine?: string;             // "I'm chasing dreams but my ____"
  blankToken?: string;               // default "____"
  lineIndex?: number;                // which line in the verse we're targe

  // 4. Lexical / grammatical constraints
  mustIncludeWords?: string[];       // ["heart", "clock"]
  mustEndWithPOS?: string[];         // e.g. ["NOUN", "VERB"]
  bannedWords?: string[];            // profanity or user's banlist
  allowedPOSSequences?: string[][]; // advanced grammar templates
  language?: "en";                   // future-proofing

  // 5. Search knobs
  maxCandidates?: number;            // raw before scoring
  beamWidth?: number;                // search control
  maxWords?: number;                 // hard limit

  // 6. Soft scoring weights (symbolic side, not embeddings/LLM)
  weights?: {
    meterFit?: number;
    rhymeFit?: number;
    lengthCompactness?: number;
    wordFrequency?: number;          // prefer more common vocab, or the op
  };
};
```

So the **generation request** could look like:

ts Copy code

```ts
type GenerationRequest = {
  verseContext: VerseAnalysis;  // from analyzeVerse()
  semanticContextText?: string; // for later embedding/LLM scoring
  constraints: ProsodyConstraints;
};
```

And the output:

```ts
type CandidateLine = {
  text: string;
  prosody: LineProsody;
  scores: {
    meterFit: number;       // 0-1
    rhymeFit: number;       // 0-1
    lexicalScore: number;   // 0-1
    totalSymbolic: number; // weighted sum
    // these can be filled later by other modules:
    semanticFit?: number;   // embedding-based
    llmQuality?: number;    // LLM judge
  };
  debug?: any;              // path, chosen words, etc
};
```

Core engine method:

```ts
function generateCandidates(req: GenerationRequest): CandidateLine[];
```

---

4. Different "modes" as just different constraint presets

With that structure, all the cool UX modes are just **different ways of filling** `ProsodyConstraints`.

A. "Write my next line" (multi-line mode)

Inputs:

- `verseContext` : result of analyzing all prior lines
- `constraints` :
  - `mode: "nextLine"`
  - `targetSyllables` : from `verseContext.inferredMeter`
  - `rhymeClassIndex` : from `verseContext.inferredRhymeScheme.suggestedNextRhymeClass`
  - `rhymeType: "perfect"`
  - maybe `maxCandidates: 200` , `beamWidth: 8`

B. "Fill in the blank in this line"

User gives:

`I'm chasing dreams but my ____`

- First, analyze everything *before* and optionally *after* the blank to figure out:
  - how many syllables are already present
  - how many you have left to hit target meter (either user-specified or inferred from surrounding lines)
- Then constraints:
  - `mode: "fillBlank"`
  - `templateLine` : `"I'm chasing dreams but my ____"`
  - `targetSyllables` : `remainingSyllables`
  - optional `rhymeWithWord` if they want the whole line to rhyme with something.

C. "Fix my meter but keep my meaning"

User: `Line X feels off, fix it`

You can:

1. Analyze original line.
2. Keep a set of **mustIncludeWords** (key nouns/verbs).
3. Set `targetSyllables` & `targetStressPattern` based on inferred verse meter.

Constraints:

ts Copy code

```ts
{
  mode: "replaceLine",
```

```
    lineIndex,
    targetSyllables,
    targetStressPattern,
    mustIncludeWords: ["dreams", "clock"], // from heuristic
    stressTolerance: 1,
  }
```

Then, after you get candidates, a later module uses embeddings or an LLM to pick the ones closest in meaning to the original line.

---

5. What are the *minimal* required inputs?

For a basic generation to work, you really only need:

1. **Some context** about meter & rhyme
   - Either:
     - `verseContext` from previous lines, or
     - explicit: `targetSyllables` and maybe `rhymeWithWord` .
2. **A mode**
   - Are we generating an entire line from scratch (`nextLine`)?
   - Or completing just a segment (`fillBlank`)?
   - Or replacing a whole line (`replaceLine`)?
3. **Rhyme constraint (optional but common)**
   - `rhymeWithWord` or `rhymeKey` .

Everything else (POS, mustInclude, bannedWords, weights) defaults to reasonable values.

So a "minimal viable generator call" could be:

ts Copy code

```ts
generateCandidates({
  verseContext: analyzeVerse(previousLines),
  constraints: {
    mode: "nextLine",
    targetSyllables: 10,
    rhymeWithWord: "time",
    maxCandidates: 100
```

```
    maxCandidates: 100
  }
});
```

And a super-basic blank-fill:

ts Copy code

```ts
generateCandidates({
  verseContext: analyzeVerse("I'm chasing dreams but my ____"),
  constraints: {
    mode: "fillBlank",
    templateLine: "I'm chasing dreams but my ____",
    targetSyllables: 4,
  }
});
```

---

6. Where embeddings + LLMs plug in (without polluting the core API)

I'd keep LLM/embedding concerns in *separate* functions that operate on `CandidateLine[]`:

ts Copy code

```ts
type SemanticContext = {
  text: string;        // whole verse, or user prompt about the song
  extraTags?: string[]; // "sad", "confident", "club"
};

async function scoreWithEmbeddings(
  candidates: CandidateLine[],
  context: SemanticContext
): Promise<CandidateLine[]> { /* return with semanticFit filled */ }

async function scoreWithLLM(
  candidates: CandidateLine[],
  context: SemanticContext
): Promise<CandidateLine[]> { /* fill llmQuality + maybe rewrites */ }
```

Then your high-level "magical" API:

Then your high-level "magical" API:

ts Copy code

```ts
async function suggestNextLine(
  verseText: string,
  userOverrides?: Partial<ProsodyConstraints>,
  semanticContext?: SemanticContext
): Promise<CandidateLine[]> {
  const verse = analyzeVerse(verseText);
  const baseConstraints = inferDefaultConstraintsFromVerse(verse);

  const candidates = generateCandidates({
    verseContext: verse,
    constraints: { ...baseConstraints, ...userOverrides }
  });

  const withSemantic = semanticContext
    ? await scoreWithEmbeddings(candidates, semanticContext)
    : candidates;

  const withLLM = semanticContext
    ? await scoreWithLLM(withSemantic, semanticContext)
    : withSemantic;

  // final ranking logic...
  return rankFinal(withLLM);
}
```

The important bit: **the prosody engine stays deterministic and testable**.
LLMs are "just" a scoring/ranking layer on top.

If you want to go one level deeper next, we can:

- Design the **beam search / phrase generation** strategy in detail (how you actually walk the word graph with these constraints), or
- Flesh out concrete **TS types + function signatures** you could drop into a real repo as `prosody-engine` and start unit-testing.

love this model! here's some initial thoughts:
- the constraints object is complex, but it seems like it has to be? unless you can think of a way to simplify this while still providing power?
- is such an engine feasible to build with what we have here (symbolically)
- can we improve performance by using structs like enums over string patterns?
- can this handle fill-in-the-blank *inside* lines, not just the ending?
- will this engine be able generate candidates that are *not* perfect matches, but rather the best it can? (best to always give the user something rather than be strict and fail)
- LOVE the deterministic + testable aspect, this will be key for development!

Thought for 10s

Yessss, I'm so glad this is clicking for you. This is *exactly* the kind of project where your "I like clean systems" brain is going to have a field day. Let me go through your bullets one by one.

---

1. Is the constraints object too complex? Can we simplify?

You're right: the full power version is inherently kinda chunky… **but** that doesn't mean the *public* surface has to feel that way.

Two key ideas:

a) Simple public API → rich internal constraints

Externally you expose things like:

ts Copy code

```
suggestNextLine({
  verseText,
  targetSyllables,   // optional
  rhymeWith,         // optional
  mood,              // optional semantic stuff
});
```

or

ts Copy code

```ts
fillBlank({
  templateLine: "I'm chasing dreams but my ____",
  verseContext,
  rhymeWith,
});
```

Internally, these functions build a full `ProsodyConstraints` object and call the engine. So:

- **Users of the engine (your UI / plugins)** see *simple function signatures*.
- **The engine itself** gets a normalized, more detailed constraints struct.

b) Split "hard" vs "soft" constraints

You can also conceptually split:

ts Copy code

```ts
type HardConstraints = {
  maxWords?: number;
  bannedWords?: string[];
  language?: "en";
  // things we truly won't violate
};

type SoftTargets = {
  targetSyllables?: number;
  targetStressPattern?: (0|1|2)[];
  rhymeWithWord?: string;
  weights?: { meterFit?: number; rhymeFit?: number; ... };
};
```

```
type ProsodyConstraints = HardConstraints & SoftTargets;
```

Hard constraints = safety rails.
Soft targets = "do your best and we'll score/penalize deviations."

That mental split makes it feel conceptually simpler and maps nicely to your "always return something" desire.

---

2. Is this actually feasible to build symbolically?

Yes. Very much yes.

Break it down:

- **Analysis** (word → CMU → syllables, stress, rhyme)
  - 100% straightforward. CMU is small (~130k entries).
  - Simple tokenization + lookup + some heuristics for OOV words.
- **Generation**
  - You're not trying to generate *arbitrary* English, just **short phrases** that:
    - have N syllables
    - match (roughly) a stress pattern
    - optionally end in a rhyme class
  - This is classic search / constraint solving:
    - Pre-index words by syllable count, stress pattern, rhyme key.
    - Use **beam search** over word sequences:
      - State = (words so far, syllables so far, stress so far).
      - Expand by adding words whose syllables don't overshoot.
      - Prune aggressively.

With beam widths like 16–64 and phrase lengths 1–6 words, this is totally tractable. Especially if you:

- Bias toward **frequent words** (fewer rare weird ones).
- Precompute some small "legal POS sequences" to avoid ungrammatical stuff.
- Cap maxCandidates per request.

The engine is not "AI-hard"; it's basically a clever combinatorics + scoring problem.

---

3. Enums vs strings for performance?

Yep, there are some easy wins here:

- **Stress**: already ints (0, 1, 2).
- **Rhyme key**:
  - Initially: string like `"AY1 T"` .
  - Later: you can hash or intern them into small integers (e.g. `rhymeId: number`), so rhyme comparisons are O(1) integer equality.
- **POS tags** (if you add them): map to small enums/ints too.

You can treat the "human-readable" bits as metadata and run the core search on **integer IDs** (word index, rhyme index, stress sequence encoded as bytes, etc.).

That said, I'd optimize for clarity first:

- v1: strings are fine.
- Once it feels right & you see hot spots in profiling, you tighten into enums/ints.

But your instinct is right: numeric encodings make the engine *both* faster and more deterministic/testable.

---

4. Can this handle blanks *inside* lines, not just endings?

Yes, totally.

Approach:

1. **Split the template** around the blank:

   txt

   Copy code

   ```
   "I'm chasing dreams but my ____ keeps racing"
   => prefix: "I'm chasing dreams but my"
      suffix: "keeps racing"
   ```

2. **Analyze prefix & suffix** separately:
   - Get syllable counts + stress patterns:
     - `prefixProsody`

- `prefixProsody`
- `suffixProsody`

3. **Decide target meter for the whole line**:
   - From verse context: maybe we want ~10 syllables & a certain stress pattern.
   - Or user-specified.
4. **Compute the "middle" requirements**:
   - `middleTargetSyllables = targetTotal - prefix - suffix`.
   - `middleTargetStressPattern` = the slice of the desired pattern that falls in the blank region.
   - If that's negative or tiny, you relax and just aim for "reasonable" rather than perfect.
5. **Generate phrases ONLY for the blank**:
   - Run the prosody engine to generate candidate phrases of `middleTargetSyllables`.
   - Optionally require that the **overall line end rhyme** is preserved (so suffix may be doing that job already).
6. **Verify whole-line meter post-hoc**:
   - After you plug a candidate middle in, re-analyze the full line and score:
     - How close is the overall stress pattern to desired?
     - How far off is the final syllable count?

So yes: the same engine works — you just carve out the middle and solve a "mini-problem" for the blank that's informed by global meter.

---

5. Can it give best-effort, non-perfect matches instead of failing?

Absolutely — and I agree that's critical for UX.

This is where the **soft vs hard** distinction matters:

- **Hard constraints**: things you really don't want to break:
  - max words
  - banned words
  - language
- **Soft targets**: everything else:
  - target syllables

- target stress pattern
- rhyme class

You turn soft targets into a **score function**:

```ts
meterPenalty = abs(actualSyllables - targetSyllables);       // or square
stressPenalty = HammingDistance(actualStress, targetStress);  // # mismatc
rhymePenalty = rhymeMatches ? 0 : 1;

totalSymbolicScore =
  w_meter * meterPenalty +
  w_stress * stressPenalty +
  w_rhyme * rhymePenalty + ...
```

Then:

- Search doesn't throw away all paths that deviate; it just **prefers** good ones.
- At the end, you sort candidates by lowest penalty (or highest transformed "fit" score).
- You *always* return the top K, even if they're a little off.

You can also do staged relaxation if you like:

1. Try strict (exact syllables & rhyme). If no candidates →
2. Allow ±1 syllable. If still nothing →
3. Drop rhyme requirement but keep syllables. → etc.

But even without staged retries, a soft scoring approach means you never return "no results."

---

6. Deterministic + testable: yes, you can have this cake 😄

Because the core is symbolic and pure:

- Same inputs → same outputs. No randomness unless you explicitly add it.
- You can unit-test things like:
  - "Given target 10 syllables and this stress pattern, all candidates obey it within tolerance ≤ 1."
  - "When I ask to rhyme with 'time', all top 10 candidates share the correct rhyme

class."
  - "FillBlank with this template never exceeds maxWords 4."

You can even write **golden tests**:

- Fix a tiny fake lexicon.
- Run generation with fixed constraints.
- Assert the exact list of candidates & scores.

Then keep LLM & embeddings strictly in a **post-processing layer** so your core engine stays rock solid and easy to reason about.

---

7. Can we simplify *how you think about the constraints*?

One way to keep your brain happy:

- Design the **internal constraints struct** like we sketched (rich & explicit).
- Define a small set of **mode-specific builders**:

```ts
buildNextLineConstraints(verseAnalysis, userOverrides?)
buildFillBlankConstraints(verseAnalysis, templateLine, userOverrides?)
buildReplaceLineConstraints(verseAnalysis, lineIndex, userOverrides?)
```

Those builders:

- Infer defaults (syllables, rhyme, etc.).
- Apply sensible weights.
- Hide 80% of the complexity from your day-to-day coding.

So in actual code, you mostly see:

```ts
const verse = analyzeVerse(verseText);
const constraints = buildNextLineConstraints(verse, { rhymeWithWord: "time
const candidates = generateCandidates({ verseContext: verse, constraints }
```

And the hairy stuff lives in one well-documented place.

If you want, next I can:

- Sketch a **concrete** `generateCandidates` **algorithm** (step-by-step beam search with example state structure), or
- Propose an **actual TS interface** for `prosody-engine` v0.1.0 (files, function names, and a first-pass minimal constraints type that you can literally paste into a repo).

love this, thanks for addressing everything, let's sketch a concrete generateCandidates algorithm!

Thought for 6s

Yesss, let's turn this into something you could actually implement. I'll focus on a **beam search**-style `generateCandidates` that's:

- Deterministic
- Meter- and rhyme-aware
- Best-effort (soft scoring, not brittle)

I'll assume v1 is mostly for **"generate a phrase / line end"** (next-line or fill-in-the-blank), which you can reuse everywhere.

---

0. Ingredients / precomputed stuff

Before `generateCandidates` ever runs, you have a lexicon:

ts Copy code

```ts
type LexWord = {
  id: number;                  // internal int ID
  text: string;                // "light"
  phonemes: string[];          // ["L", "AY1", "T"]
  syllables: (0 | 1 | 2)[];    // [0,1] or [1] etc
  syllableCount: number;
  rhymeKey: number | null;     // integer ID for rhyme class
```

```
  pos?: string[];              // optional POS tags
  freqScore: number;           // e.g. log frequency (higher = more common)
};
```

And some indexes for fast lookup:

- `wordsBySyllables[count: number] -> LexWord[]`
- `wordsByRhyme[rhymeKey: number] -> LexWord[]`
- Maybe `wordsByPOS["NOUN"] -> LexWord[]`, etc.

You also have a `analyzeLine()` that can take a sequence of words and return `LineProsody`.

---

1. Inputs to `generateCandidates`

We'll keep this shape:

ts

Copy code

```ts
type GenerationRequest = {
  verseContext: VerseAnalysis;      // from analyzeVerse()
  constraints: ProsodyConstraints; // from builders
};

type CandidateLine = {
  text: string;
  prosody: LineProsody;
  scores: {
    meterFit: number;
    rhymeFit: number;
    lexicalScore: number;
    totalSymbolic: number;
  };
};
```

Assume `constraints` has at least:

- `targetSyllables?`
- `minSyllables?` / `maxSyllables?`
- `targetStressPattern?`

- `stressTolerance?`
- `rhymeKey?` or `rhymeWithWord?`
- `maxWords?`
- `maxCandidates?`
- `beamWidth?`
- `bannedWords?`
- `mustIncludeWords?` (optional, v2)
- `weights` for scoring

---

2. Search state

Inside the search, we operate on **partial phrases**:

ts Copy code

```ts
type SearchState = {
  wordIds: number[];          // sequence of LexWord.id
  syllablesSoFar: number;
  stressSoFar: (0 | 1 | 2)[];
  usedWords: Set<number>;     // optional; for mustInclude logic
  lastWordRhymeKey: number | null;
  complete: boolean;          // meets soft target criteria
  partialPenalty: number;     // for beam pruning (lower is better)
};
```

You can derive everything else from `wordIds`.

---

3. Pre-step: normalize constraints

Inside `generateCandidates(req)`:

ts Copy code

```ts
function generateCandidates(req: GenerationRequest): CandidateLine[] {
  const { verseContext, constraints } = req;
  const {
    targetSyllables,
    minSyllables
```

```
    minSyllables,
    maxSyllables,
    targetStressPattern,
    stressTolerance = 2,
    rhymeKey: explicitRhymeKey,
    rhymeWithWord,
    maxWords = 6,
    maxCandidates = 100,
    beamWidth = 32,
    bannedWords = [],
    weights = defaultWeights,
  } = constraints;

  // 1. Resolve rhymeKey if only rhymeWithWord is given
  const targetRhymeKey = explicitRhymeKey ??
    (rhymeWithWord ? getRhymeKeyForWord(rhymeWithWord) : null);

  // 2. Resolve syllable range
  const [minSyl, maxSyl] = resolveSyllableRange(targetSyllables, minSyllab

  // 3. Precompute banned word IDs
  const bannedIds = new Set(bannedWords.map(w => getLexWord(w)?.id).filter

  // 4. Initial beam: empty phrase
  let beam: SearchState[] = [{
    wordIds: [],
    syllablesSoFar: 0,
    stressSoFar: [],
    usedWords: new Set(),
    lastWordRhymeKey: null,
    complete: false,
    partialPenalty: 0,
  }];

  const completed: SearchState[] = [];
  ...
}
```

`resolveSyllableRange` might return e.g. `[target-1, target+1]` if only `targetSyllables` is given, for built-in leniency.

4. Scoring: penalties for meter / rhyme / lexical

Define helpers:

ts Copy code

```ts
function meterPenalty(
  syllables: number,
  minSyl: number,
  maxSyl: number,
  targetSyl?: number
): number {
  if (syllables < minSyl) return minSyl - syllables;       // underfull
  if (syllables > maxSyl) return syllables - maxSyl;       // overflow
  if (targetSyl != null) return Math.abs(syllables - targetSyl) * 0.5;
  return 0;
}

function stressPenalty(
  stressSoFar: (0|1|2)[],
  targetStressPattern?: (0|1|2)[],
  tolerance = 2
): number {
  if (!targetStressPattern) return 0;
  const len = Math.min(stressSoFar.length, targetStressPattern.length);
  let mismatches = 0;
  for (let i = 0; i < len; i++) {
    if (targetStressPattern[i] === 0 || targetStressPattern[i] === 2) {
      // treat 0/2 as equivalent-ish if you want
    }
    if (stressSoFar[i] !== targetStressPattern[i]) {
      mismatches++;
    }
  }
  // Allow some mismatches with a soft penalty
  return Math.max(0, mismatches - tolerance);
}

function rhymePenalty(
```

```ts
function rhymePenalty(
  lastRhyme: number | null,
  targetRhymeKey: number | null
): number {
  if (!targetRhymeKey) return 0;
  // No last rhyme yet = small penalty, not infinite
  if (lastRhyme == null) return 0.5;
  return lastRhyme === targetRhymeKey ? 0 : 1;
}

function lexicalScore(words: LexWord[]): number {
  // Higher is better; we convert to penalty later
  const sumFreq = words.reduce((acc, w) => acc + w.freqScore, 0);
  return sumFreq / (words.length || 1);
}
```

Combine into a single **partial penalty** for beam pruning:

ts Copy code

```ts
function computePartialPenalty(
  state: SearchState,
  params: {
    minSyl: number;
    maxSyl: number;
    targetSyl?: number;
    targetStressPattern?: (0|1|2)[];
    targetRhymeKey?: number | null;
    weights: { meterFit: number; stressFit: number; rhymeFit: number; lexi
  }
): number {
  const { minSyl, maxSyl, targetSyl, targetStressPattern, targetRhymeKey,

  const words = state.wordIds.map(id => lexicon[id]);
  const meter = meterPenalty(state.syllablesSoFar, minSyl, maxSyl, targetS
  const stress = stressPenalty(state.stressSoFar, targetStressPattern);
  const rhyme = rhymePenalty(state.lastWordRhymeKey, targetRhymeKey ?? nul
  const lex = lexicalScore(words);

  // Lexical is a "good" score, so we subtract it
  const penalty =
```

```
    weights.meterFit * meter +
    weights.stressFit * stress +
    weights.rhymeFit * rhyme -
    weights.lexical * lex;

  return penalty;
}
```

This is soft: even if meter/rhyme aren't perfect, they're just "worse", not discarded.

---

5. Main beam search loop

ts Copy code

```ts
const paramsForPenalty = {
  minSyl: minSyl,
  maxSyl: maxSyl,
  targetSyl: targetSyllables,
  targetStressPattern,
  targetRhymeKey,
  weights,
};

for (let step = 0; step < maxWords; step++) {
  const newBeam: SearchState[] = [];

  for (const state of beam) {
    // If this state is already complete, keep it as a candidate and don't
    if (state.complete) {
      newBeam.push(state);
      continue;
    }

    // Expand with candidate next words
    const candidates = getCandidateNextWords(state, bannedIds);

    for (const word of candidates) {
      const newSyl = state.syllablesSoFar + word.syllableCount;
      // Hard constraint: if even maxSyl exceeded, skip
```

```
        if (newSyl > maxSyl) continue;

        const newStress = state.stressSoFar.concat(word.syllables);
        const newUsedWords = new Set(state.usedWords);
        newUsedWords.add(word.id);

        const newState: SearchState = {
          wordIds: [...state.wordIds, word.id],
          syllablesSoFar: newSyl,
          stressSoFar: newStress,
          usedWords: newUsedWords,
          lastWordRhymeKey: word.rhymeKey,
          complete: false,
          partialPenalty: 0, // fill below
        };

        newState.partialPenalty = computePartialPenalty(newState, paramsForP

        // Decide if this should be marked "complete" yet
        if (newSyl >= minSyl && newSyl <= maxSyl) {
          newState.complete = true;
        }

        newBeam.push(newState);
      }
    }

    // Sort by partialPenalty, keep top beamWidth
    newBeam.sort((a, b) => a.partialPenalty - b.partialPenalty);
    beam = newBeam.slice(0, beamWidth);

    // Collect completed states into `completed` pool
    for (const s of beam) {
      if (s.complete) completed.push(s);
    }

    // Optional early stop: if we've collected enough completed candidates
    if (completed.length >= maxCandidates) break;
  }
```

`getCandidateNextWords` can be simple v1:

ts

Copy code

```ts
function getCandidateNextWords(state: SearchState, bannedIds: Set<number>)
  // v1: just all words, filtered by bannedIds
  // later: restrict by POS, rhyme hints, etc.
  return lexicon.filter(w => !bannedIds.has(w.id));
}
```

Later you can:

- Prefer words whose syllable count doesn't obviously make it impossible to ever reach `minSyl` within `maxWords` .
- Use rhyme hints: if you're near the end (syllables close to max), bias toward words with `targetRhymeKey` .

---

6. Finalization: turn states into CandidateLines

After the loop:

ts

Copy code

```ts
// Deduplicate completed by text (optional)
const uniqueCompleted = dedupeByText(completed);

// Re-score with full line prosody (more accurate than incremental)
const scored: CandidateLine[] = uniqueCompleted.slice(0, maxCandidates).ma
  const words = state.wordIds.map(id => lexicon[id].text);
  const text = words.join(" ");
  const prosody = analyzeLine(text);

  const meter = meterPenalty(prosody.syllableCount, minSyl, maxSyl, target
  const stress = stressPenalty(prosody.stressPattern, targetStressPattern)
  const rhyme = rhymePenalty(prosody.endingRhymeKey, targetRhymeKey ?? nul
  const lex = lexicalScore(state.wordIds.map(id => lexicon[id]));

  const totalSymbolic =
    weights.meterFit * meter +
    weights.stressFit * stress +
```

```
      weights.stressFit * stress +
      weights.rhymeFit * rhyme -
      weights.lexical * lex;

    return {
      text,
      prosody,
      scores: {
        meterFit: meter,
        rhymeFit: rhyme,
        lexicalScore: lex,
        totalSymbolic,
      },
    };
  });

  // Sort by totalSymbolic ascending (lower penalty -> better)
  scored.sort((a, b) => a.scores.totalSymbolic - b.scores.totalSymbolic);

  return scored.slice(0, maxCandidates);
```

Because everything is:

- deterministic (no RNG)
- based on pure data structures

…it's super unit-testable.

---

7. How this extends to blanks & "inside line" use-cases

For **inside-line blank**:

1. Analyze `prefix` and `suffix` using `analyzeLine` -ish logic but with care to treat them as partials.
2. Decide a total target syllable count for the whole line (from meter inference or user).
3. Compute:

ts Copy code

```ts
middleMin = max(1, targetSyl - suffixSyl - 1); // or reasonable lower
```

```
middleMax = targetSyl - prefixSyl + 1;        // allow a bit of slack
```

4. Create constraints:

ts Copy code

```ts
const constraints: ProsodyConstraints = {
  minSyllables: middleMin,
  maxSyllables: middleMax,
  // Optionally derive a target stress slice for the middle
  targetStressPattern: middleStressSlice,
  ...
};
```

5. Run `generateCandidates` to get middle phrases.
6. For each candidate, build full line: `prefix + " " + middle + " " + suffix`, re-analyze full line, and re-score if you want.

Same core engine, just used on the **middle segment**.

---

8. Why this is "doable"

This is all:

- Straightforward data structures
- O(N * beamWidth * vocabSubset) style search, with plenty of pruning knobs
- Completely testable with small toy lexicons

You can start with:

- Tiny lexicon of like 100 handpicked words.
- `beamWidth = 8`, `maxWords = 4`.
- Simple constraints: only `targetSyllables` and `rhymeKey`.

Once the behavior feels right, scale up to full CMU + actual word frequencies.

If you want, next we can:

- Sketch a **minimal in-memory lexicon** and a couple of **unit tests** (e.g. "generate 4-syllable phrase rhyming with 'TIME'"), or

- Design a `prosody-engine` folder structure (files, modules) so you can drop this into a Bun/Node project and start iterating.

amazing! thank you for sketching this out! some thoughts:
- do we really need to constrain on number of words? is that meaningful for song writers? (maybe it is, I don't know)
- what's the difference between incremental and full-line prosody scoring-wise?

yes, please sketch out a minimal in-memory lexicon and some unit tests!

then we'll talk tech stack...

Thought for 6s

Yesss, this is coming together so nicely. Let me hit your two questions first, then I'll drop a concrete mini-lexicon + tests.

---

1. Do we really need a max-words constraint?

Short answer: it's more about **controlling search complexity & style** than about "songwriters care how many words".

- From the *engine*'s perspective:
  - Without *some* bound, the search can keep adding 1-syllable words forever while staying under a syllable limit (if you allow slack).
  - Max words keeps the tree finite and biased towards reasonably tight phrases.
- From the *creative* side:
  - Many lyricists *do* have feel preferences like "short punchy endings" vs "long rambly endings".
  - But you don't need to expose it to the user early.

- You can:
  - Keep a **reasonable default** (e.g. 6 words).
  - Make it an **advanced knob** later (“concise vs verbose”).

So: I’d keep `maxWords` in the engine, but:

- hide it from basic UI
- treat it as a tuning parameter, not a “songwriting constraint”.

---

2. Incremental vs full-line prosody scoring

Why both?

Incremental scoring (inside beam search)

While the search is building phrases one word at a time, you need a quick way to decide:

> “Is this partial path worth keeping in the beam?”

You only know **so far**:

- syllablesSoFar
- partial stressSoFar
- lastWordRhymeKey

So you approximate:

- meter penalty based on current syllables vs expected range
- stress penalty on prefix vs target pattern
- maybe a soft rhyme direction (e.g. “no rhyme yet is okay”)

This is **fast & heuristic**. The goal is to prune obviously bad branches early.

Full-line scoring (after you have complete candidates)

Once you have full phrases, you can:

- Re-run `analyzeLine(text)` to get **exact** syllable count, full stress pattern, and final rhyme.
- Compare against full target:
  - Entire stress sequence (not just prefix)
  - Ending rhyme vs target

- Ending rhyme to target
- Exact syllable bounds

This is **final, authoritative scoring**, and you might even use a slightly different metric than the incremental one. It also makes it easier to evolve the scoring logic later without touching the search internals.

So:

- Incremental = **guidance** for search.
- Full-line = **ground truth** for ranking.

---

3. Minimal in-memory lexicon + unit tests

Let's do a tiny, concrete example in TypeScript that you could literally drop into a `prosody-engine` repo.

I'll define:

- A small `LexWord` type and `lexicon` array.
- Helpers to get words, rhyme keys, etc.
- A tiny `analyzeLine`.
- A **super simplified** `generateCandidates` that only cares about:
  - `targetSyllables`
  - optional `rhymeWithWord`
- A few Vitest-style tests.

`lexicon.ts`

ts Copy code

```ts
// lexicon.ts

export type Stress = 0 | 1 | 2;

export type LexWord = {
  id: number;
  text: string;
  phonemes: string[];          // CMUish, but we'll keep it simple
  stressPattern: Stress[];     // one entry per syllable
```

```
  stressPattern: Stress[];   // one entry per syllable
  syllableCount: number;
  rhymeKey: string | null;   // simple string rhyme class
  freqScore: number;         // arbitrary; higher = more common
};

// Helper to create entries
function w(
  id: number,
  text: string,
  phonemes: string[],
  stressPattern: Stress[],
  rhymeKey: string | null,
  freqScore: number
): LexWord {
  return {
    id,
    text,
    phonemes,
    stressPattern,
    syllableCount: stressPattern.length,
    rhymeKey,
    freqScore,
  };
}

/**
 * Tiny toy lexicon:
 * - "time", "rhyme", "light", "tight" share rhyme-ish classes
 * - Some function words: "my", "the", "in", "on"
 * - A few content words: "heart", "mind", "dreams", "clock", "wall"
 */
export const LEXICON: LexWord[] = [
  // id, text,    phonemes,         stress, rhymeKey, freq
  w(0, "time",   ["T", "AY1", "M"],      [1],    "AYM",    5),
  w(1, "rhyme",  ["R", "AY1", "M"],      [1],    "AYM",    3),
  w(2, "light",  ["L", "AY1", "T"],      [1],    "AYT",    5),
  w(3, "tight",  ["T", "AY1", "T"],      [1],    "AYT",    4),

  w(4, "my",     ["M", "AY1"],           [1],    "AY",     6),
  w(5, "the",    ["DH", "AH0"],          [0],    "AH",     8),
```

```
  w(6, "in",     ["IH1", "N"],          [1],   "IHN",   7),
  w(7, "on",     ["AA1", "N"],          [1],   "AAN",   7),

  w(8, "heart",  ["HH", "AA1", "RT"],   [1],   "AART",  4),
  w(9, "mind",   ["M", "AY1", "ND"],    [1],   "AYND",  4),
  w(10,"dreams", ["D", "R", "IY1", "M", "Z"], [1], "IYMZ", 4),
  w(11,"clock",  ["K", "L", "AA1", "K"],[1],   "AAK",   3),
  w(12,"wall",   ["W", "AO1", "L"],     [1],   "AOL",   3),
];

const byText = new Map<string, LexWord>(
  LEXICON.map((w) => [w.text.toLowerCase(), w])
);

export function getLexWord(text: string): LexWord | undefined {
  return byText.get(text.toLowerCase());
}

// Simple rhymeKey accessor for constraints
export function getRhymeKeyForWord(text: string): string | null {
  return getLexWord(text)?.rhymeKey ?? null;
}
```

analyzeLine.ts

Very minimal version: just split on spaces, look up words, aggregate.

ts Copy code

```ts
// analyzeLine.ts

import { LEXICON, getLexWord, LexWord, Stress } from "./lexicon";

export type LineProsody = {
  text: string;
  words: LexWord[];
  syllableCount: number;
  stressPattern: Stress[];
  endingRhymeKey: string | null;
  confidence: number; // 0–1 (how many words we recognized)
```

```
};

export function analyzeLine(text: string): LineProsody {
  const tokens = text.trim().split(/\s+/).filter(Boolean);
  const words: LexWord[] = [];
  let known = 0;

  for (const t of tokens) {
    const w = getLexWord(t);
    if (w) {
      words.push(w);
      known++;
    } else {
      // OOV: we could skip or give a dummy entry; for this toy version,
      // just ignore syllables but keep token count.
    }
  }

  const syllableCount = words.reduce((acc, w) => acc + w.syllableCount, 0)
  const stressPattern: Stress[] = words.flatMap((w) => w.stressPattern);
  const last = words[words.length - 1];
  const endingRhymeKey = last?.rhymeKey ?? null;

  const confidence = tokens.length === 0 ? 0 : known / tokens.length;

  return {
    text,
    words,
    syllableCount,
    stressPattern,
    endingRhymeKey,
    confidence,
  };
}
```

Minimal `generateCandidates` (toy version)

Let's do a **super simple** version that:

- Only cares about:
  - `targetSyllables`

- targetSyllables
- optional `rhymeWithWord`
- `maxCandidates`
- Uses a tiny DFS-ish search instead of full beam, just to test the concept.

ts Copy code

```ts
// generateCandidates.ts

import { LEXICON, getRhymeKeyForWord, LexWord } from "./lexicon";
import { analyzeLine, LineProsody } from "./analyzeLine";

export type ProsodyConstraints = {
  targetSyllables: number;
  rhymeWithWord?: string;
  maxCandidates?: number;
};

export type CandidateLine = {
  text: string;
  prosody: LineProsody;
};

export function generateCandidates(
  constraints: ProsodyConstraints
): CandidateLine[] {
  const { targetSyllables, rhymeWithWord, maxCandidates = 20 } = constrain

  const targetRhymeKey = rhymeWithWord
    ? getRhymeKeyForWord(rhymeWithWord)
    : null;

  const results: CandidateLine[] = [];

  // Simple DFS: build phrases up to 3 words
  const maxWords = 3;

  function dfs(wordsSoFar: LexWord[], syllablesSoFar: number) {
    if (results.length >= maxCandidates) return;

    // If we've hit exactly target syllables, consider this a candidate
```

```
      // If we've hit exactly target syllables, consider this a candidate
      if (syllablesSoFar === targetSyllables && wordsSoFar.length > 0) {
        const text = wordsSoFar.map((w) => w.text).join(" ");
        const prosody = analyzeLine(text);

        // If rhyme is specified, require last word to match that rhymeKey
        if (targetRhymeKey) {
          if (prosody.endingRhymeKey !== targetRhymeKey) {
            // not a rhyme match; skip in this toy version
            // (in real engine you'd keep as lower-scored)
            // but here it's clear for tests
            // so just return.
            return;
          }
        }

        results.push({ text, prosody });
        // Note: we don't return immediately; we might still explore other v
        // at the same depth or further (but keep it small for toy).
      }

      if (wordsSoFar.length >= maxWords) return;
      if (syllablesSoFar >= targetSyllables) return; // no more room

      for (const w of LEXICON) {
        const newSyl = syllablesSoFar + w.syllableCount;
        if (newSyl > targetSyllables) continue;
        dfs([...wordsSoFar, w], newSyl);
      }
    }

    dfs([], 0);

    return results;
  }
```

This is intentionally **simplified** (no stress patterns, no soft penalties), but it's perfect for getting your first tests green and proving the plumbing.

`prosody-engine.test.ts` (Vitest style)

ts Copy code

```ts
// prosody-engine.test.ts
import { describe, it, expect } from "vitest";
import { analyzeLine } from "./analyzeLine";
import { generateCandidates } from "./generateCandidates";
import { getRhymeKeyForWord } from "./lexicon";

describe("analyzeLine", () => {
  it("computes syllables and rhymeKey for a single word", () => {
    const line = analyzeLine("time");

    expect(line.syllableCount).toBe(1);
    expect(line.stressPattern).toEqual([1]);
    expect(line.endingRhymeKey).toBe(getRhymeKeyForWord("time"));
    expect(line.confidence).toBe(1);
  });

  it("aggregates multiple words", () => {
    const line = analyzeLine("my time");

    expect(line.syllableCount).toBe(2); // "my" (1) + "time" (1)
    expect(line.stressPattern).toEqual([1, 1]);
    expect(line.endingRhymeKey).toBe(getRhymeKeyForWord("time"));
  });

  it("handles unknown words with reduced confidence", () => {
    const line = analyzeLine("mystery time");

    expect(line.syllableCount).toBe(1); // only "time" is known in toy lex
    expect(line.confidence).toBeCloseTo(0.5); // 1/2 tokens known
  });
});

describe("generateCandidates (toy)", () => {
  it("generates 2-syllable phrases", () => {
    const candidates = generateCandidates({
      targetSyllables: 2,
    });
```

```
    // Expect at least some phrases like "my time", "the wall", etc.
    const texts = candidates.map((c) => c.text);
    expect(texts).toContain("my time");
    expect(texts).toContain("the wall");

    for (const c of candidates) {
      expect(c.prosody.syllableCount).toBe(2);
    }
  });

  it("can constrain to rhyme with 'time'", () => {
    const candidates = generateCandidates({
      targetSyllables: 2,
      rhymeWithWord: "time",
    });

    const texts = candidates.map((c) => c.text);
    // "my time" is 2 syllables and rhymes with time
    expect(texts).toContain("my time");
    // Something like "my light" should *not* appear in this toy version
    expect(texts).not.toContain("my light");

    for (const c of candidates) {
      expect(c.prosody.endingRhymeKey).toBe(
        getRhymeKeyForWord("time")
      );
    }
  });

  it("respects maxCandidates", () => {
    const candidates = generateCandidates({
      targetSyllables: 2,
      maxCandidates: 3,
    });

    expect(candidates.length).toBeLessThanOrEqual(3);
  });
});
```

You can run these with:

bash Copy code

```bash
npm install -D vitest
npx vitest
```

(Or swap in Jest if you prefer.)

From here, the path is:

1. Get this toy setup working end-to-end.
2. Gradually evolve `generateCandidates` :
   - Add stress patterns & penalties.
   - Add beam search instead of naive DFS.
   - Add soft scoring & best-effort candidates.
3. Then plug it into a tiny Bun/Node backend and start playing with a simple web UI.

Whenever you're ready, we can talk tech stack next:

- backend runtime (Bun + Hono vs Next API routes, etc.)
- where to host the prosody engine
- how to wire in LLMs + embeddings cleanly on top.

yes, let's talk tech stack!

I'm wondering if js/ts is the right move for building this out. i can see how it'd let us iterate fast, which may be the best thing?

im just thinking that this is super CPU and memory-intensive, and we're going to be constrained on that in a very real way. the more efficient it performs, the more broader search we can do, the higher quality we can get (or the faster UX we can have). so we might end up rewriting the engine down the line?

cool thing is, with great tests we can rewrite against a spec and know it's production-quality before migrating (and be able to evaluate performance, of course)

wdyt is the best more for now? is nextjs on vercel the way? (been doing other apps like that, which has been uber fast dev). or do you see something better suited for this?

Thought for 16s

Love where your head's at on this. You're asking exactly the right "foundations" questions.

I'll split this into:

1. JS/TS vs "fast" languages (Rust/Go/etc)
2. Where the real bottlenecks actually are
3. A concrete stack recommendation *now* + an easy path to a later rewrite
4. How to structure the code so you don't paint yourself into a corner

---

1. JS/TS vs Rust/Go/etc for the engine

Short version:
Start in TypeScript, *design like you'll rewrite later*, and only drop to Rust/Go if profiling proves you're CPU bound.

Why TS is a good starting point:

- You get **stupid-fast iteration** on:
  - Data structures (lexicon, indexes, scoring)
  - Search heuristics (beam width, penalties, etc.)
  - API shape
- Same language on **frontend + backend**, shared types for:
  - `ProsodyConstraints`
  - `LineProsody`

- `CandidateLine`
- The first real bottleneck in practice is very likely to be:
  - **LLM + embeddings network latency**
  - Not the prosody search itself (at least at MVP scales).

Rewriting later is totally viable:

- Your prosody engine is **pure compute**, no IO.
- If you keep a **clean interface** (`generateCandidates` , `analyzeLine` etc.), you can:
  - Reimplement that in Rust/Go/Zig
  - Expose the *same* HTTP/FFI/WebAssembly interface
  - Keep the rest of the app untouched.
- Good tests = your migration contract.

So: I'd treat "Rust rewrite" as *an optimization you earn* after you have users and real traffic patterns.

---

2. How intensive is this actually?

Ballpark:

- CMU lexicon: ~130k words. Even with derived indexes, you're talking **a few–tens of MB** in memory.
- A beam search with:
  - beamWidth ~ 16–64
  - maxWords ~ 4–6
  - heavy pruning on syllable counts
- …is *not* insane for Node/TS, especially if:
  - You load the lexicon once per process (cache in memory).
  - You keep the search reasonably constrained.

What's likely to cost more:

- Network round trips to OpenAI/Anthropic for:
  - embeddings
  - LLM judging
- Serialization/deserialization of bigger payloads for those calls.

So, in the early days, JS/TS performance will be "good enough" and LLM calls will dominate latency.

If you later discover that:

- People are slamming "generate 100 candidates" every keystroke, or
- You want super-deep search for wild patterns,

then it's time to consider a lower-level rewrite.

---

3. Concrete stack recommendation (for now)

Given:

- You're comfy and fast with Next.js on Vercel.
- You want SEO-friendly marketing + a nice frontend.
- You want low-friction dev right now.

I'd do this:

a) Monorepo with a shared `prosody-engine` package

Structure like:

txt Copy code

```
/ apps
  /web          # Next.js app (UI + simple API routes)
/ packages
  /prosody-engine   # pure TS library (no Node/Vercel specifics)
```

- `prosody-engine` exposes:
  - `analyzeLine`, `analyzeVerse`
  - `generateCandidates`
  - Types: `LineProsody`, `CandidateLine`, `ProsodyConstraints`
- `web`:
  - Uses Next.js App Router.
  - Imports `prosody-engine` in server components / API routes.
  - Handles auth, persistence, billing, etc.

b) Deployment

UI + simple engine calls:

- Host the Next.js app on **Vercel**.
- Put the engine calls in **Node runtime** API routes or Server Actions (not Edge):
  - You want:
    - More CPU
    - Stable warm processes with the lexicon in memory.
  - Edge functions are better for small, cheap per-request logic; your engine wants some heft and caching.

Later, if/when needed:

- Factor the `prosody-engine` into its own microservice:
  - Could be:
    - a Node/Bun service (using the exact same TS library), on Fly.io/Railway.
    - or a Rust/Go service once you rewrite.
  - Next.js app just hits it over HTTP.

This gives you a **clean evolution path**:

1. v0: everything in Vercel monolith, super fast to iterate.
2. v1: move engine to its own Node/Bun service if you hit runtime/memory limits.
3. v2: reimplement engine internals in Rust/Go/WASM behind the same API if you truly need the perf.

---

4. How to future-proof the TS implementation

To make a later rewrite trivial, do these things from day 1:

a) Keep the engine *pure*

- `prosody-engine` should have:
  - **No HTTP clients**
  - **No DB access**
  - **No environment-variable reads**
- Just: “given this input, compute this output.”

This makes:

- Testing trivial.
- Reimplementation in other languages a 1:1 mapping of functions + types.

b) Define a small, stable core interface

For example, this is your “spec”:

ts Copy code

```ts
// prosody-engine core interface
export type ProsodyRequest = {
  verseText: string;
  constraints: ProsodyConstraints;
};

export type ProsodyResponse = {
  verseAnalysis: VerseAnalysis;
  candidates: CandidateLine[];
};

export function analyzeVerse(text: string): VerseAnalysis;
export function generateCandidatesForVerse(
  req: ProsodyRequest
): ProsodyResponse;
```

If you later move to a Rust HTTP service, it just exposes:

- `POST /prosody/generate` with JSON body `ProsodyRequest`, returns `ProsodyResponse`.

Your Next app doesn’t care what language lives behind that.

c) Preprocess the lexicon once

Heavy stuff:

- Parsing CMU
- Computing stress patterns, rhyme keys, indexes

Do that **offline** into a compact JSON or even a pre-baked `.ts` module, so:

- At runtime, your engine just does `import { LEXICON } from "./lexicon-data";`
- No runtime parsing, no disk IO beyond initial bundle load.
- Makes it easier in Rust/Go later too (you can ship a preprocessed binary blob or CSV).

d) Profile *before* you rewrite

When you start to feel performance pressure:

- Add simple timing:
  - “GenerateCandidates took X ms for these constraints.”
- Check:
  - Is it the prosody engine?
  - Or is it LLM/embeddings?
  - Or Vercel cold-starts?

You might find that:

- Batching LLM calls and caching results gives more speedup than rewriting engine in Rust.

---

5. What about Bun + Hono, Fly.io, etc.?

You already have a Bun + Hono + Fly mental model from other projects. You *could*:

- Run the prosody engine as a **dedicated Bun/Hono service** on Fly.io:
  - Pros:
    - Long-lived processes → lexicon stays in memory.
    - Full control over CPU/memory sizing.
    - Bun is fast at JS/TS.
  - Cons:
    - More infra complexity than “just Vercel”.
    - Two deploy targets (Vercel for UI, Fly for engine).

I’d treat that as a **Phase 2** option:

1. Phase 1:
   - Next.js on Vercel, Node runtime.
   - Engine embedded in the Next app.
2. Phase 2:
   - If you hit Vercel runtime limits or want more control:
     - Extract `prosody-engine` into a Bun/Hono service on Fly.
     - Next.js calls it over HTTP.

Again, same TS code, just moved behind an HTTP boundary.

TL;DR recommendation

For **right now**:

- ✅ **Language:** TypeScript
- ✅ **UI:** Next.js (App Router) on Vercel
- ✅ **Engine:** Pure TS `prosody-engine` package, imported into:
  - Next.js server components or API routes
  - Node runtime (not Edge)
- ✅ **Design:** Clear core interface + offline-preprocessed lexicon, fully unit-tested
- 🚀 **Future:** If/when needed, move engine behind an HTTP API and reimplement in Rust/Go/Bun without touching UI or product logic.

If you want, next step we can sketch:

- The actual **project layout** (folders, package.json workspaces)
- Or the **first couple of Next routes / Server Actions** that call `prosody-engine` and stream candidates to the UI.

hell yeah, love this! one side question...

one quick q, for embeddings, can't we precompute those for most words? (obv not for user queries, but anything else)
and it is possible to host the embeddings model on our server?

Thought for a couple of seconds

Yeah, great instinct—this is exactly the kind of thing that can shift "LLM bottleneck" into "fast local scoring" if you do it right.

Let's split it:

1. **Can we precompute embeddings for most words/phrases?**
2. **Can we host the embedding model ourselves?**

2. **Can we host the embedding model ourselves?**
3. **How this plugs into your prosody engine.**

---

1. Precomputing embeddings: what's realistic?

Short answer: **yes, for a LOT of stuff**.

You've got a few different embedding needs:

1. **Per-word embeddings**
   - For every word in your lexicon (i.e., every CMU entry you care about).
   - Lets you do things like:
     - "Given a target concept, rank candidate *words* by semantic fit."
   - Very feasible:
     - Vocabulary size: say 50k–200k words depending on how much you include.
     - Embedding dim: 256–1024.
     - Storage:
       - 100k words × 512 dims × 4 bytes ≈ ~200MB raw float32.
       - You can compress (float16/int8) or reduce dims later if needed.
2. **Short-phrase embeddings (n-grams)**
   - E.g. "keep time", "feel tight", "my heart", etc.
   - Trickier, because the combinatorics explode if you try to precompute *all possible* phrases.
   - But you *can*:
     - Precompute embeddings for common collocations / templates you generate often.
     - Or compute phrase embeddings on the fly *from* word embeddings (e.g., weighted averages, learned projection later, etc.).
3. **User context embeddings**
   - The stanza / verse / "what this song is about" text.
   - These will always be **online** (per request) and not precomputable.

So a good plan:

- **Yes:** Precompute per-word embeddings for all lexicon entries you care about (maybe downsample rare junk words).

- **Maybe later:** Precompute some phrase embeddings for "canonical" candidates, but don't overoptimize prematurely.
- **Always online:** User context.

Then at runtime, for each candidate line:

- Approximate its embedding as:
  - either the embedding of its last content word
  - or avg/max pool of its constituent word embeddings
  - or embed it directly via a hosted model if you're not latency constrained.

That already gives you a "semantic distance" metric you can use to rank purely on your own infra.

---

2. Can you host the embedding model yourself?

Also **yes**, with tradeoffs.

Two options:

Option A: Use a hosted API (OpenAI / Anthropic / etc.)

Pros:

- No infra.
- You get very strong models out of the box.
- Easy to iterate.

Cons:

- Recurring cost if you embed a ton of words or phrases.
- Latency is bound by network.
- You're dependent on their uptime and API semantics.

For **precomputing** a lexicon, this is actually pretty great:

- You do an offline batch job:
  - Chunk your lexicon.
  - Call `embeddings.create` in batches.
  - Store vectors in a local file or vector DB.
- That's a one-time-ish cost; you only re-run if you change your lexicon or model.

Option B: Host your own embedding model

You can absolutely run a sentence/word embedding model on your own servers:

- Family of options:
  - SentenceTransformers (e.g. `all-MiniLM-L6-v2`) in PyTorch.
  - Smaller LLMs with an embedding head (e.g. `gte-small` , `e5-small`).
  - GGUF-quantized variants running via llama.cpp, Ollama, or similar.

Pros:

- No per-call cost.
- Low latency if colocated with your engine.
- Full control.

Cons:

- You're now doing **ML infra**:
  - Memory/VRAM sizing.
  - Model loading + serving.
  - Potential scaling issues if lots of users show up.
- Quality of local-only small models may be worse than state-of-the-art hosted ones, but often *good enough* for "semantic-ish similarity".

Given your skillset + desire to own the stack, a realistic trajectory:

1. **Phase 1:**
   - Use a hosted embedding API for:
     - offline lexicon precomputation
     - online user-context embeddings
   - This gives you great quality with zero infra friction.
2. **Phase 2:**
   - Once the product is real and you care about:
     - Marginal cost per user
     - Latency
   - Stand up a small embedding service (e.g., Python + FastAPI + SentenceTransformers, or Ollama running a GTE/E5-family embedding model).
   - Use *that* for:
     - fresh lexicon expansion
     - candidate-line embedding
     - maybe even user context.

maybe even user context.

3. **Phase 3 (maybe):**
   - If usage explodes, optimize further:
     - quantize model
     - use GPU
     - or buy a bigger hosted solution.

So: **yes, absolutely possible** to self-host embeddings. It's just a question of when you want to take that on.

---

3. How this plugs into your prosody engine

Here's how I'd wire it conceptually.

Precomputation step (offline)

1. Load lexicon (CMU-derived words).
2. For each word:
   - send through embedding model (hosted API or local).
   - store result:

ts Copy code

```ts
type WordEmbedding = {
  wordId: number;
  vector: Float32Array; // or Float16/int8 compression
};
```

3. Persist:
   - Either as a flat binary file (fast to mmap / load).
   - Or as JSON initially (for ease of dev), then optimize.

In your engine, you now have:

ts Copy code

```ts
// index by wordId
const WORD_EMBEDDINGS: Float32Array[] = [...];
```

Runtime scoring

Given:

- `contextEmbedding` (from user stanza or description).
- Candidate line with word IDs `[w1, w2, ..., wn]`.

You can compute `lineEmbedding` as:

- `avg(WORD_EMBEDDINGS[wi])`
- or weighted by word importance (content words vs function words).

Then:

- Semantic similarity ~ cosine(contextEmbedding, lineEmbedding)
- That gives you `semanticFit` in `[-1, 1]` / `[0, 1]`.

You plug that into your total score:

ts Copy code

```ts
totalScore = symbolicScore + alpha * semanticFit;
```

Where:

- `symbolicScore` = meter + rhyme + lexical etc.
- `semanticFit` = your embedding-based alignment.

And this entire part can be done:

- 100% **locally** on your own server.
- Without hitting any external API per request (other than for context embedding if you don't host that model yet).

Even in a hybrid:

- Precomputed word embeddings locally.
- User context embedding from a hosted API for now.
- Candidates scored entirely in your Node/Bun service.

TL;DR

- ✅ **Precomputing embeddings for words:**
  Very doable and a *great idea*. Precompute once, store with the lexicon, use to score candidates locally.
- ✅ **Hosting embedding models yourself:**

Also very possible. Start with hosted APIs for speed of dev and switch to a self-hosted embedding model (SentenceTransformers, GTE, E5, etc.) once it makes sense economically and operationally.

- 🎯 **Best near-term plan for you:**
  - Use a hosted embedding model to:
    - precompute per-word embeddings
    - get per-verse context embeddings
  - Wire semantic scoring into your prosody engine as a separate "semanticScorer" module.
  - Design the interface so swapping out "where embeddings come from" is just changing one implementation, not your whole system.

If you'd like, next we can sketch:

- The **data format + load path** for storing word embeddings alongside the lexicon.
- Or a small `semanticScorer.ts` API that sits next to `generateCandidates` and does the ranking step.