Short answer: don't use a general text embedding model on raw FEN. You'll get token-level artifacts that don't reflect chess similarity. Use a purpose-built board encoder and (optionally) learn a small embedding head.

Here's a practical, RAG-friendly stack that works well for chess positions:

1) Canonicalization & dedup (pre-embedding)

- **Normalize FEN**: drop halfmove/fullmove counters; keep side-to-move, castling, en-passant.
- **Symmetry canonicalization**: pick a canonical transform over {identity, flip ranks/files, rotate 180°} that makes equivalent patterns map to the same representative (e.g., always put the side-to-move as "White", flip board if needed).
- **Hash**: store a Zobrist (or any 64-bit) hash of the canonical FEN for exact dedup and a unique key.

2) Baseline embedding (deterministic, zero-training)

Bitboard one-hot → float vector

- 12 piece planes × 64 squares = **768 dims** (binary 0/1).
- - side-to-move (1), castling rights (4 one-hots), en-passant file (9 one-hots incl. "none") → **~782 dims total**.
- Store as float32 and use **cosine** distance.
  This already gives sensible "same/similar piece layouts ≈ close" behavior.

3) Motif-aware head (small, optionally trained)

If you care about *strategic* motifs (e.g., queenside majority), add a compact feature block and/or learn a projection:

- **Pawn-structure features (~40–60 dims)**
  - File counts: white pawns fA…fH (8), black pawns fA…fH (8).

    - Binary flags: isolated, doubled, backward, passed (per file, compressed counts).
    - Pawn-island count (white, black), open/semi-open files per side, majority indicators (queenside/kingside).
- **Material & king-safety lite (~10–20 dims)**
    - Material vector (piece counts per type), king files/ranks bucketed, castling availability.
- **Concatenate** with the 782-d baseline and (optionally) learn a **64–128-d projection** with a tiny MLP trained contrastively:
    - **Positives**: positions equal under symmetries; same pawn structure; small Stockfish eval delta; same ECO bucket in early middlegame.
    - **Negatives**: random positions; different pawn structures.

> (Unverified) In practice, this small learned head often outperforms raw one-hots for motif queries while staying cheap to compute.

4) Engine-informed enrichment (optional)

If you can batch-evaluate positions:

- Store **eval score**, **phase** (opening/middlegame/endgame), and a few **engine features** (e.g., mobility, passed pawns).
- Use them as **filters** or as extra dimensions (scaled); don't force them into the core embedding if you worry about engine bias.

5) Vector-DB setup

- Use **cosine** or **dot-product** for dense floats.
- Keep two indexes:
    1. **layout index** = the 782-d (or projected 64–128-d) vector.
    2. **pawn-structure index** = a 32–64-d pawn-only vector (fast shortlist for structure-first searches).

- Store the **canonical FEN** + metadata (game id, move number, ECO, eval) alongside vectors.

6) Query patterns that work

- **"Queenside majority attack"**: search the **pawn-structure index** first (queenside majority flag + file counts), then re-rank with the layout index or engine eval proximity.
- **"Same piece placement / tactical pattern"**: hit the **layout index** directly.
- **Filters**: side-to-move, castling rights, eval window, phase, ECO.

Why not a text embedding (OpenAI, bge, etc.) on FEN?

- FEN is a compact code, not natural language; tokenizers will slice it arbitrarily.
- Distance in token space doesn't track **board** similarity.

> (Unverified) You may see occasional wins, but it's brittle and hard to reason about.

Minimal implementation sketch (what to build first)

1. **Canonicalizer** (FEN ↔ board; symmetry normalize; strip clocks).
2. **Encoder**:
   - `float[782]` baseline vector (12×64 pieces + STM + castling + EP).
   - `float[?]` pawn-features vector (counts/flags).
3. **(Optional)** Tiny MLP to 128-d with contrastive training (you can start untrained and add this later).
4. **Two indexes** in your vector DB, plus scalar metadata columns for fast filtering.

If you want, I can generate a reference encoder (Python or OCaml) that outputs both the 782-d baseline and a 48-d pawn-structure vector, plus a Qdrant schema to store them.

Below is a self-contained OCaml reference encoder for chess positions in FEN.
It produces:

- a **782-d baseline vector**: 12×64 one-hot piece planes (P,N,B,R,Q,K × white/black = 768) + side-to-move (1) + castling rights K,Q,k,q (4) + en-passant file one-hot (a–h, none = 9)
- a **48-d pawn/metadata vector** with file counts, islands, open/semi-open files, isolated/doubled/passed counts, side majorities, material counts, castling/STM/EP flags, and a crude "phase" scalar

No external deps. Keep or adapt the feature ordering as you need.

```
(* File: chess_fen_encoder.ml *)

module ChessFenEncoder = struct
  (*
    --- Public API ----------------------------------------------
---------------

    val encode_layout   : string -> float array   (* 782-d
baseline *)
    val encode_features : string -> float array   (* 48-d
pawn+meta *)
    val encode_all      : string -> float array * float array

    ------------------------------------------------------------
---------------

    Baseline (782 dims), in order:
      [0..767]   12 piece planes × 64 squares (a1..h1, a2..h2,
... a8..h8)
                 Plane order: WP, WN, WB, WR, WQ, WK, BP, BN,
BB, BR, BQ, BK
```

```
      [768]      side-to-move: 1.0 = White, 0.0 = Black
      [769..772] castling rights one-hot flags: K, Q, k, q (1.0
or 0.0)
      [773..781] en-passant file one-hot: a..h plus index 8 =
"none"

    Features (48 dims), in order:
      [ 0.. 7]  white pawn file counts a..h
      [ 8..15]  black pawn file counts a..h
      [16]      white pawn islands
      [17]      black pawn islands
      [18]      open files (no pawns either side)
      [19]      semi-open for White (no white pawns, ≥1 black
pawn)
      [20]      semi-open for Black (no black pawns, ≥1 white
pawn)
      [21]      isolated white pawns
      [22]      isolated black pawns
      [23]      doubled white pawns (sum over files max(0,
count-1))
      [24]      doubled black pawns
      [25]      passed white pawns (simplified, see code)
      [26]      passed black pawns (simplified)
      [27]      queenside majority White  (a-d > e-h)
{0.0,1.0}
      [28]      queenside majority Black  (a-d > e-h)
{0.0,1.0}
      [29]      kingside majority White   (e-h > a-d)
{0.0,1.0}
      [30]      kingside majority Black   (e-h > a-d)
{0.0,1.0}
      [31..36]  white material counts: P,N,B,R,Q,K
      [37..42]  black material counts: P,N,B,R,Q,K
      [43]      castling right exists (White): {0.0,1.0}
      [44]      castling right exists (Black): {0.0,1.0}
      [45]      side-to-move (1.0 white, 0.0 black)
      [46]      en-passant available (not "-"): {0.0,1.0}
      [47]      phase (crude 0.0 opening .. 1.0 endgame; see
code)

    Notes:
```

```
    - Coordinates: squares are indexed a1..h1, a2..h2, ... ,
a8..h8 (file-major).
    - FEN parsing ignores halfmove/fullmove counters in the
baseline planes.
  *)

  (* --- Types & basics ---------------------------------------
-------------- *)

  type color = White | Black
  type piece = Pawn | Knight | Bishop | Rook | Queen | King

  let is_between x a b = (x >= a) && (x <= b)

  let piece_of_char = function
    | 'P' -> Some (White, Pawn)
    | 'N' -> Some (White, Knight)
    | 'B' -> Some (White, Bishop)
    | 'R' -> Some (White, Rook)
    | 'Q' -> Some (White, Queen)
    | 'K' -> Some (White, King)
    | 'p' -> Some (Black, Pawn)
    | 'n' -> Some (Black, Knight)
    | 'b' -> Some (Black, Bishop)
    | 'r' -> Some (Black, Rook)
    | 'q' -> Some (Black, Queen)
    | 'k' -> Some (Black, King)
    | _ -> None

  let plane_index (c, p) =
    match c, p with
    | White, Pawn   -> 0
    | White, Knight -> 1
    | White, Bishop -> 2
    | White, Rook   -> 3
    | White, Queen  -> 4
    | White, King   -> 5
    | Black, Pawn   -> 6
    | Black, Knight -> 7
    | Black, Bishop -> 8
    | Black, Rook   -> 9
    | Black, Queen  -> 10
```

```
    | Black, King   -> 11

  (* board: 64 squares, each is (color * piece) option
     index: file 0..7 (a..h), rank 0..7 (1..8); idx = file +
8*rank *)
  type board = (color * piece) option array

  type castling = {
    wk : bool;  (* K *)
    wq : bool;  (* Q *)
    bk : bool;  (* k *)
    bq : bool;  (* q *)
  }

  type fen = {
    board : board;
    stm   : color;             (* side to move *)
    cast  : castling;
    ep    : int option;        (* en-passant file 0..7, or None
*)
    (* clocks ignored for encoding purposes *)
  }

  (* --- FEN parsing ----------------------------------------
-------------- *)

  let empty_board () : board = Array.make 64 None

  let idx_of_file_rank file rank = (file land 7) + (rank land
7) * 8

  let parse_ep (s : string) : int option =
    (* s is like "e3" or "-" *)
    if s = "-" then None
    else
      let file_ch = s.[0] in
      if is_between (Char.code file_ch) (Char.code 'a')
(Char.code 'h')
      then Some (Char.code file_ch - Char.code 'a')
      else None

  let parse_castling (s : string) : castling =
```

```
    if s = "-" then {wk=false; wq=false; bk=false; bq=false}
    else
      let has c = String.exists (fun x -> x = c) s in
      {
        wk = has 'K';
        wq = has 'Q';
        bk = has 'k';
        bq = has 'q';
      }

  let parse_fen (fen_str : string) : fen =
    (* FEN: pieces / stm / castling / ep / halfmove / fullmove
*)
    let parts = String.split_on_char ' ' fen_str in
    match parts with
    | pieces :: stm :: cast :: ep :: _half :: _full :: _ ->
        let b = empty_board () in
        (* pieces rows: rank 8 to 1, slash-separated *)
        let ranks = String.split_on_char '/' pieces in
        if List.length ranks <> 8 then failwith "Invalid FEN:
ranks";
        List.iteri
          (fun i rstr ->
             (* i=0 -> rank 8; want to place on rank=7 down to
0 *)
             let rank = 7 - i in
             let file = ref 0 in
             String.iter (fun ch ->
               if is_between (Char.code ch) (Char.code '1')
(Char.code '8')
               then file := !file + (Char.code ch - Char.code
'0')
               else match piece_of_char ch with
                 | Some cp ->
                     if !file >= 8 then failwith "Invalid FEN:
file overflow";
                     let idx = idx_of_file_rank !file rank in
                     b.(idx) <- Some cp;
                     incr file
                 | None -> failwith "Invalid FEN: piece char"
             ) rstr;
```

```
            if !file <> 8 then failwith "Invalid FEN: file
count"
          ) ranks;
        let side =
          match stm with
          | "w" -> White
          | "b" -> Black
          | _ -> failwith "Invalid FEN: stm"
        in
        let cast = parse_castling cast in
        let epf  = parse_ep ep in
        { board = b; stm = side; cast; ep = epf }
    | _ -> failwith "Invalid FEN: fields"

  (* --- Helpers for features ------------------------------
------------ *)

  let is_pawn = function (_, Pawn) -> true | _ -> false
  let is_white = function White, _ -> true | _ -> false
  let is_black = function Black, _ -> true | _ -> false

  let piece_value = function
    | Pawn -> 1 | Knight -> 3 | Bishop -> 3 | Rook -> 5 | Queen
-> 9 | King -> 0

  let material_counts (b:board) =
    (* counts: White P,N,B,R,Q,K ; Black P,N,B,R,Q,K *)
    let w = Array.make 6 0 and bl = Array.make 6 0 in
    Array.iter (function
      | None -> ()
      | Some (c,p) ->
          let i = match p with
            | Pawn -> 0 | Knight -> 1 | Bishop -> 2
            | Rook -> 3 | Queen -> 4 | King -> 5
          in
          (match c with
           | White -> w.(i) <- w.(i) + 1
           | Black -> bl.(i) <- bl.(i) + 1)
    ) b;
    (w, bl)

  let file_of_idx idx = idx mod 8
```

```
  let rank_of_idx idx = idx / 8

  let pawn_file_counts (b:board) =
    let wf = Array.make 8 0 and bf = Array.make 8 0 in
    Array.iteri (fun idx cell ->
      match cell with
      | Some (White, Pawn) -> wf.(file_of_idx idx) <- wf.
(file_of_idx idx) + 1
      | Some (Black, Pawn) -> bf.(file_of_idx idx) <- bf.
(file_of_idx idx) + 1
      | _ -> ()
    ) b;
    (wf, bf)

  let islands_of_files (fcounts:int array) =
    (* number of contiguous groups of files with count>0 *)
    let islands = ref 0 in
    let in_group = ref false in
    for f=0 to 7 do
      if fcounts.(f) > 0 then (
        if not !in_group then (in_group := true; incr islands)
      ) else in_group := false
    done;
    !islands

  let open_and_semiopen (wf:int array) (bf:int array) =
    let open_files = ref 0
    and semi_w = ref 0
    and semi_b = ref 0 in
    for f=0 to 7 do
      let w = wf.(f) and b = bf.(f) in
      if w = 0 && b = 0 then incr open_files;
      if w = 0 && b > 0 then incr semi_w;
      if b = 0 && w > 0 then incr semi_b;
    done;
    (!open_files, !semi_w, !semi_b)

  let isolated_pawns (fcounts:int array) =
    let iso = ref 0 in
    for f=0 to 7 do
      if fcounts.(f) > 0 then
        let left = if f>0 then fcounts.(f-1) else 0 in
```

```
        let right = if f<7 then fcounts.(f+1) else 0 in
        if left = 0 && right = 0 then iso := !iso + fcounts.(f)
    done;
    !iso

  let doubled_pawns (fcounts:int array) =
    let d = ref 0 in
    for f=0 to 7 do
      if fcounts.(f) > 1 then d := !d + (fcounts.(f) - 1)
    done;
    !d

  (* Passed pawns (simplified):
     - White pawn on (f,r) is passed if there is NO black pawn
on files f-1,f,f+1
       on ranks strictly greater than r (ahead of it).
     - Symmetric for Black (look for white pawns behind).
  *)
  let passed_pawns (b:board) =
    let w_passed = ref 0 and b_passed = ref 0 in
    let black_pawns_on f r =
      (* is there a black pawn on files f-1..f+1 with rank>r ?
*)
      let found = ref false in
      for ff = max 0 (f-1) to min 7 (f+1) do
        for rr = r+1 to 7 do
          let idx = idx_of_file_rank ff rr in
          match b.(idx) with
          | Some (Black, Pawn) -> found := true
          | _ -> ()
        done
      done;
      !found
    in
    let white_pawns_on f r =
      (* is there a white pawn on files f-1..f+1 with rank<r ?
*)
      let found = ref false in
      for ff = max 0 (f-1) to min 7 (f+1) do
        for rr = r-1 downto 0 do
          let idx = idx_of_file_rank ff rr in
          match b.(idx) with
```

```
          | Some (White, Pawn) -> found := true
          | _ -> ()
        done
      done;
      !found
    in
    Array.iteri (fun idx cell ->
      match cell with
      | Some (White, Pawn) ->
          let f = file_of_idx idx and r = rank_of_idx idx in
          if not (black_pawns_on f r) then incr w_passed
      | Some (Black, Pawn) ->
          let f = file_of_idx idx and r = rank_of_idx idx in
          if not (white_pawns_on f r) then incr b_passed
      | _ -> ()
    ) b;
    (!w_passed, !b_passed)

  let majority_flags (wf:int array) (bf:int array) =
    let sum range arr =
      let s = ref 0 in
      List.iter (fun f -> s := !s + arr.(f)) range; !s
    in
    let w_q = sum [0;1;2;3] wf
    and w_k = sum [4;5;6;7] wf
    and b_q = sum [0;1;2;3] bf
    and b_k = sum [4;5;6;7] bf in
    let w_q_maj = if w_q > w_k then 1.0 else 0.0
    and b_q_maj = if b_q > b_k then 1.0 else 0.0
    and w_k_maj = if w_k > w_q then 1.0 else 0.0
    and b_k_maj = if b_k > b_q then 1.0 else 0.0 in
    (w_q_maj, b_q_maj, w_k_maj, b_k_maj)

  let phase_scalar (w:int array) (bl:int array) =
    (* crude phase: normalize non-pawn material to [0..1]
endgame-ish *)
    let nonpawn_sum counts =
      let n = counts.(1) and b = counts.(2) and r = counts.(3)
and q = counts.(4) in
      (n * 3) + (b * 3) + (r * 5) + (q * 9)
    in
    let tot = nonpawn_sum w + nonpawn_sum bl in
```

```
    let max_tot = 62. (* start of game: N4,B4,R4,Q2 ->
12+12+20+18 = 62 *)
    let x = float_of_int tot /. max_tot in
    let p = 1.0 -. x in
    if p < 0.0 then 0.0 else if p > 1.0 then 1.0 else p

  (* --- Baseline encoder (782-d) ----------------------------
------------ *)

  let encode_layout (fen_str : string) : float array =
    let f = parse_fen fen_str in
    let v = Array.make 782 0.0 in
    (* 12 planes × 64 squares *)
    Array.iteri (fun idx cell ->
      match cell with
      | None -> ()
      | Some cp ->
          let pidx = plane_index cp in
          let base = pidx * 64 in
          v.(base + idx) <- 1.0
    ) f.board;
    (* side to move *)
    v.(768) <- (match f.stm with White -> 1.0 | Black -> 0.0);
    (* castling K,Q,k,q *)
    v.(769) <- if f.cast.wk then 1.0 else 0.0;
    v.(770) <- if f.cast.wq then 1.0 else 0.0;
    v.(771) <- if f.cast.bk then 1.0 else 0.0;
    v.(772) <- if f.cast.bq then 1.0 else 0.0;
    (* en-passant file one-hot a..h + none *)
    let ep_idx = match f.ep with Some file -> file | None -> 8
in
    for i=0 to 8 do v.(773 + i) <- if i = ep_idx then 1.0 else
0.0 done;
    v

  (* --- Feature encoder (48-d) ------------------------------
------------ *)

  let encode_features (fen_str : string) : float array =
    let f = parse_fen fen_str in
    let feats = Array.make 48 0.0 in
    let wf, bf = pawn_file_counts f.board in
```

```
    (* file counts *)
    for i=0 to 7 do
      feats.(i)      <- float_of_int wf.(i);
      feats.(8 + i)  <- float_of_int bf.(i);
    done;
    (* islands *)
    feats.(16) <- float_of_int (islands_of_files wf);
    feats.(17) <- float_of_int (islands_of_files bf);
    (* open/semi-open *)
    let open_files, semi_w, semi_b = open_and_semiopen wf bf in
    feats.(18) <- float_of_int open_files;
    feats.(19) <- float_of_int semi_w;
    feats.(20) <- float_of_int semi_b;
    (* isolated/doubled *)
    feats.(21) <- float_of_int (isolated_pawns wf);
    feats.(22) <- float_of_int (isolated_pawns bf);
    feats.(23) <- float_of_int (doubled_pawns wf);
    feats.(24) <- float_of_int (doubled_pawns bf);
    (* passed (simplified) *)
    let w_passed, b_passed = passed_pawns f.board in
    feats.(25) <- float_of_int w_passed;
    feats.(26) <- float_of_int b_passed;
    (* majorities *)
    let wq, bq, wk, bk = majority_flags wf bf in
    feats.(27) <- wq; feats.(28) <- bq; feats.(29) <- wk;
feats.(30) <- bk;
    (* material counts *)
    let wmat, bmat = material_counts f.board in
    for i=0 to 5 do feats.(31 + i) <- float_of_int wmat.(i)
done;
    for i=0 to 5 do feats.(37 + i) <- float_of_int bmat.(i)
done;
    (* castling availability (any side) *)
    feats.(43) <- if f.cast.wk || f.cast.wq then 1.0 else 0.0;
    feats.(44) <- if f.cast.bk || f.cast.bq then 1.0 else 0.0;
    (* stm / ep flag *)
    feats.(45) <- (match f.stm with White -> 1.0 | Black ->
0.0);
    feats.(46) <- (match f.ep with Some _ -> 1.0 | None ->
0.0);
    (* phase *)
    feats.(47) <- phase_scalar wmat bmat;
```

```
    feats

  let encode_all fen_str = (encode_layout fen_str,
encode_features fen_str)

  (* --- Pretty helpers (optional) ----------------------------
------------ *)

  let pp_vec_short name (v:float array) n =
    Printf.printf "%s (%d): [" name (Array.length v);
    for i=0 to min (n-1) (Array.length v - 1) do
      if i>0 then print_string ", ";
      Printf.printf "%.0f" v.(i)
    done;
    if Array.length v > n then print_string ", ...";
    print_endline "]"

end

(* ---------------------- Quick usage example ----------------
--------------
   let fen = "r4rk1/ppp1bppp/2n2B2/8/8/2P2P2/PPP1bP1P/1K1R2R1 w
- - 0 17" in
   let layout = ChessFenEncoder.encode_layout fen in
   let feats  = ChessFenEncoder.encode_features fen in
   ChessFenEncoder.pp_vec_short "layout" layout 20;
   ChessFenEncoder.pp_vec_short "feats" feats 48;
----------------------------------------------------------------
------------ *)
```