

Chess Bot

Aayush Bajaj

July 10, 2026

Contents

1	the from-scratch engine	1
1.1	board representation: the 120-square mailbox	1
1.2	move generation and perft	2
1.3	evaluation	3
1.4	search	3
2	voice chess	5
2.1	the engine wrapper and uci	5
2.2	the llm commentary pipeline	6
2.3	training and spaced repetition	6
3	what each project taught	7
4	see also	7

two chess projects live in this codebase, and honesty requires separating them up front. the first is a **from-scratch javascript engine** (in `arcade/references/chess/js/`, built following the classic bluefever software series): 120-square mailbox board, hand-rolled move generator validated by perft, material + piece-square evaluation, alpha-beta with quiescence, a principal-variation hash table, and iterative deepening. the second is **voice chess** (`code/private/chess-bot/`, published as github.com/abaj8494/ollama-voice-chess): there i did *not* write the engine — stockfish plays the moves over uci — and the engineering is in the wrapper: skill throttling, an ollama llm that provides spoken commentary without being allowed to hallucinate moves, neural tts, and spaced-repetition opening training. one project teaches you how engines work; the other teaches you what to build *around* an engine.

everything measured on this page — perft counts, engine analysis, commentary — is real output from running the code on this machine.

1 the from-scratch engine

1.1 board representation: the 120-square mailbox

the board is not an 8×8 array. it is a flat array of 120 cells — a 10×12 frame with the real board embedded in the middle (`defs.js`: `BRD_SQ_NUM = 120`, `SQUARES.A1 = 21`, `SQUARES.H8 = 98`, `OFFBOARD = 100`). the mapping is

$$\text{sq}(f, r) = 21 + f + 10r, \quad f, r \in \{0, \dots, 7\}, \quad (1)$$

exactly `FR2SQ()` in the code. why the padding? **sentinels**. a knight on a1 reaching for offsets $\{-8, -19, -21, -12, +8, +19, +21, +12\}$ (`KnDir`) lands either on a legal square or on a cell holding `OFFBOARD` — one array read replaces eight boundary

checks. sliding pieces run `+=dir` until they hit a piece or the sentinel frame. two extra ranks top and bottom (but only one extra file each side) suffice because the knight, the longest jumper, never reaches more than two ranks or one-plus-wrapped file outside.

the wrapped file is the subtle bit: a 1-o'clock knight move off the h-file wraps around to the a-file of another rank in a naive 64-cell array — the 10-wide row catches it as offboard instead.

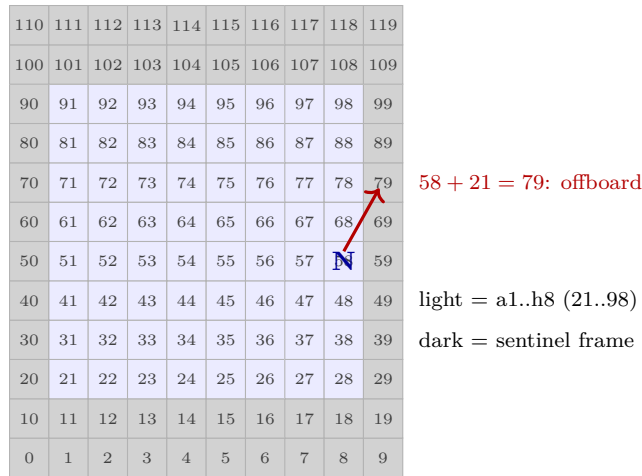


Figure 1: the 120-square mailbox. the playable 8×8 board (light) is embedded in a 10×12 frame of offboard sentinels (dark). a knight on h4 (sq 58) probing +21 lands on sq 79 — offboard, rejected by one array lookup.

alongside the square array the engine keeps **piece lists** (`pList[PCEINDEX(pce, n)]` — where each piece of each type stands), incremental **material counts** per side, and a **zobrist position key**: xor of a random number per (piece, square), plus side-to-move, en-passant and castling keys. moves are packed into a single integer bitfield (`movegen.js`):

```
1 // from arcade/references/chess/js/movegen.js
2 MoveGenController.MOVE = function (from, to, captured, promoted, flag) {
3     return (from | (to << 7) | (captured << 14) | (promoted << 20) | flag);
4 };
```

7 bits of from-square, 7 of to-square, piece codes for capture and promotion, flag bits for en passant, pawn start, and castling — the whole move history is an array of ints.

1.2 move generation and perf

generation is **pseudo-legal**: pawns/castling handled specially, non-sliders (`KnDir`, `KiDir`) probe fixed offsets, sliders (`BiDir`, `RkDir`) walk rays to the sentinel. legality is settled the lazy way — `makeMove()` plays the move, then asks whether the mover's king is attacked; if so it is taken back and reported illegal. no pinned-piece bookkeeping, at the cost of some wasted make/unmakes.

how do you know a move generator is *correct*? **perft**: count all leaf nodes of the legal move tree to fixed depth and compare against published values. one missed en-passant discovery or castling-through-check bug and the totals diverge. running the repo's `perft.js` under node (dom-free harness, real output):

```
$ node runperft.js 4          # start position
Test Complete : 197281 leaf nodes visited
elapsed ms: 2690
```

```
$ node runperft.js 5
Test Complete : 4865609 leaf nodes visited
elapsed ms: 61489
```

```
$ node kiwipete.js          # r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBPPPP/R3K2R w KQkq
Test Complete : 97862 leaf nodes visited
```

197,281 / 4,865,609 / 97,862 are exactly the published perft(4), perft(5) and kiwipete-perft(3) values — the generator handles castling rights, en passant and promotions correctly. (~80k leaves/sec is also an honest statement of what interpreted javascript with make/unmake legality checking costs.)

kiwipete is the standard torture position for movegen bugs: lots of castling, pins, en passant and promotion interplay.

1.3 evaluation

`evaluate.js` is deliberately simple — material plus piece-square tables (psts), from the side-to-move's perspective for negamax:

- **material** (`PieceVal`): pawn 100, knight 325, bishop 325, rook 550, queen 1000, king 50000 — centipawn units.
- **psts** for pawn/knight/bishop/rook: 64-entry tables encouraging central pawns, developed knights, long-diagonal bishops, and 7th-rank rooks (+25 across the entire rank). black uses `MIRROR64` to flip the table.
- **bishop pair**: +40 if a side retains both bishops.
- a quirk worth confessing: queens are scored with the *rook's* pst — `RookTable[SQ64(sq)]` — rather than a queen table. cheap, roughly right (centralise, love the 7th rank), and exactly the kind of shortcut you find when you read your own old code carefully.

no pawn-structure terms, no king safety, no mobility. the philosophy: at low depth, material + development is most of the signal, and a fast dumb evaluator searched deeper usually beats a slow clever one searched shallower.

1.4 search

`search.js` is negamax alpha-beta (see [ultimate tic tac toe](#) for the theory and the pruning-soundness argument) with the full complement of chess-specific machinery:

- **iterative deepening**: `searchPosition()` loops depth 1, 2, ... until the clock expires (`checkUp()` polls every 2048 nodes). each iteration's result seeds the next one's move ordering — this is what makes deepening *cheaper* than searching the final depth cold.
- **pv table**: a 10,000-entry hash table keyed by `posKey % PVENTRIES` storing the best move found for each position. on re-entry the stored move is scored 2,000,000 so it is searched first; after the search, `getPvLine()` walks the table to recover the whole principal variation for display.
- **move ordering** (`pickNextMove()` selection-sorts the highest score to the front on demand):

move class	score
pv move	2,000,000
captures	1,000,000 + mvv-lva
killer move, slot 1	900,000
killer move, slot 2	800,000
quiet moves	history score ($+= d^2$ on improving α)

mvv-lva = most valuable victim, least valuable attacker: `MvvLvaValue[victim] + 6 - MvvLvaValue[attacker]/100`, so p x q sorts above q x p. **killers** are quiet moves that caused a beta cutoff at the same ply in a sibling subtree — two slots, newest first. the **history heuristic** accumulates d^2 per (piece, to-square) that raised alpha, a long-run average of “this quiet move tends to be good around here”.

- **quiescence search**: at depth 0 the engine does not evaluate immediately — it first plays out *captures only* (with the same alpha-beta window and a stand-pat lower bound) until the position is quiet. this kills the **horizon effect**: a depth-4 search that ends mid-queen-trade would otherwise score the position a queen up.
- **check extension**: in-check nodes get **depth++** — forced sequences are searched deeper.
- **terminals**: no legal moves is mate (scored $-29000 + \text{ply}$, preferring faster mates) or stalemate (0); repetition and the fifty-move rule return 0.
- **instrumentation**: the engine reports **fhf/fh** — the fraction of beta cutoffs delivered by the *first* move tried. with the ordering above this typically sits above 90%, which is the practical proxy for “how close am i to the $O(b^{d/2})$ best case”.

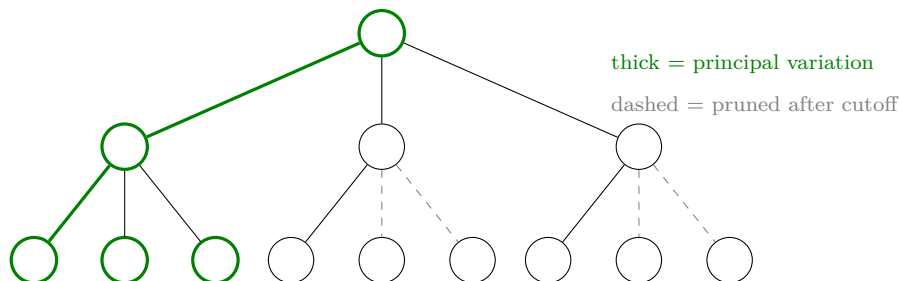


Figure 2: iterative deepening with a pv table: the previous iteration’s best line (thick) is searched first at every node it passes through, so most siblings die to immediate cutoffs (dashed).

one honest caveat on the hashing: `RAND_32()` builds zobrist keys by or-ing four `Math.random()` bytes — 32-bit keys, no seeding discipline. real engines use 64-bit keys precisely because a 10,000-entry table indexed by `posKey % PVENTRIES` will suffer collisions in long games; the code half-guards this by verifying the stored `posKey` matches before trusting the move, which reduces a collision to a wasted probe rather than a corrupted search.¹

¹zobrist hashing is from a. l. zobrist (1970), *a new hashing method with application for game playing*, tech. report 88, univ. of wisconsin. the trick: xor is self-inverting, so making/unmaking a move updates the key incrementally in $O(1)$.

2 voice chess

the second project inverts the effort. stockfish is a superhuman oracle you can apt-install; the interesting problem is the *interface*: playing against it at a human level, hearing it explain itself in a voice, and mining your games for training material. the stack: a **fastapi** server (python) wrapping stockfish via **python-chess**, an **ollama** llm for conversational commentary, **edge-tts** for speech, and a **svelte** frontend talking rest + websocket.

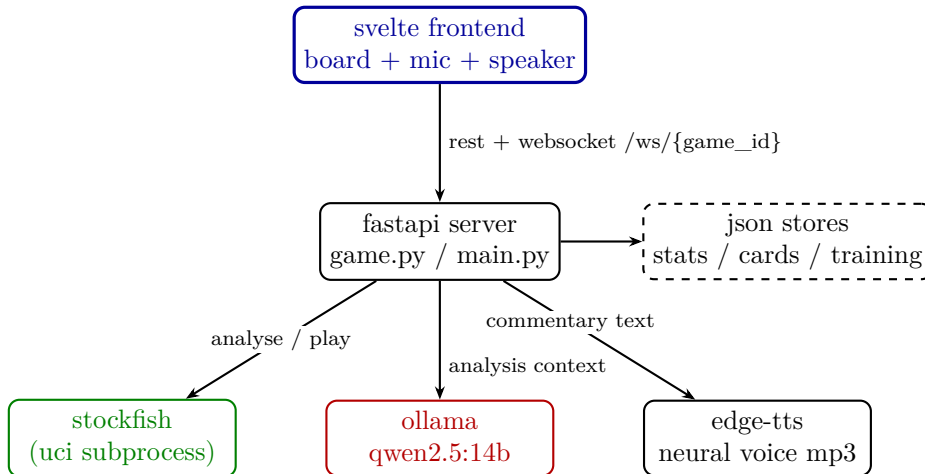


Figure 3: voice-chess architecture. stockfish decides moves; the llm is only allowed to talk about analysis it is handed; tts renders the words. json files persist stats, review cards and training progress.

2.1 the engine wrapper and uci

uci (universal chess interface) is a line protocol over stdin/stdout: the gui sends `position fen ...` and `go depth 15 movetime 1000`, the engine streams `info depth ... score cp ... pv ...` lines and finishes with `bestmove e2e4`. **python-chess** hides the plumbing behind `SimpleEngine.popen_uci()`; the repo's wrapper (`server/engine.py`) configures it and normalises the results:

```
1 # from chess-bot/server/engine.py (trimmed)
2 class ChessEngine:
3     def __init__(self, skill_level: int = 10, depth: int = 15, time_limit:
4         ↪ float = 1.0):
5         self.skill_level = skill_level
6         self.depth = depth
7         self.time_limit = time_limit
8
9     def start(self) -> bool:
10         self.engine = chess.engine.SimpleEngine.popen_uci(self.engine_path)
11         self.engine.configure({"Skill Level": self.skill_level})
12         return True
13
14     def get_best_move(self, board):
15         result = self.engine.analyse(
16             board, chess.engine.Limit(time=self.time_limit, depth=self.depth))
17         move = result.get("pv", [None])[0]
18         info = {"move": board.san(move), "score":
19             ↪ self._format_score(result.get("score")),
20                 "depth": result.get("depth", 0), "pv": self._pv_to_san(board,
21                 ↪ result.get("pv", [])),
22                 "nodes": result.get("nodes", 0)}
23         return move, info
```

```

22     def set_skill_level(self, level: int):
23         self.skill_level = max(0, min(20, level))
24         self.engine.configure({"Skill Level": self.skill_level})

```

skill throttling is stockfish’s own **Skill Level** uci option (0–20): at low levels the engine deliberately picks from a wider, weaker move distribution, which plays far more like a club player than capping search depth does (depth-capped engines still never hang a piece; skill-capped ones do). the wrapper clamps to [0, 20] and exposes it as a difficulty endpoint. real run on this machine, driving the actual class against homebrew stockfish in the italian game after 1.e4 e5 2.Nf3 Nc6 3.Bc4:

```

started: True path: /opt/homebrew/bin/stockfish
best move: Bc5 | score: 0.3 | depth: 12 | nodes: 188808 | pv: Bc5 c3 Nf6 d4 exd4
commentary: Bc5. Developing.
skill 1 eval: 0.3, top: ['Bc5', 'Nf6', 'Be7']
skill 20 eval: 0.2, top: ['Bc5', 'Nf6', 'Be7']

```

(`evaluate_position()` runs `multipv=3` to surface the top three candidate moves with scores — that is what feeds both the analysis view and the llm’s context. mate scores format as **M3**; centipawns as pawns, **0.3**.)

2.2 the llm commentary pipeline

the design constraint that makes this work: **the llm never chooses or evaluates moves**. two tiers of commentary exist (`engine.py`, `ollama_client.py`):

1. **deterministic commentary** — `generate_move_commentary()` builds a sentence purely from board facts via `python-chess` predicates: is it a capture (name the victim), a check, a mate, castling, a centre pawn push, a development move off the back rank; then appends evaluation context (“mate in 3”, “gaining advantage”) computed from the stockfish score. zero hallucination risk because there is no generation — it is a template over verified facts.
2. **ollama commentary** — `OllamaClient` (`httpx` against `localhost:11434`, default model `qwen2.5:14b`) receives the move already played, the stockfish analysis (score, pv, top alternatives, formatted by `_format_analysis()`) and a system prompt that pins it down: one sentence, max 12 words, never mention the engine, never apologise. for tutoring questions (`answer_question()`) the model may also emit inline board annotations — `[ARROW e2 e4 green]`, `[HIGHLIGHT f7 red]` — which `parse_annotations()` strips from the spoken text with regexes and ships to the frontend as structured arrow/highlight json.

whatever text survives goes to `tts.py`: microsoft edge neural voices via `edge-tts` (default `en-US-BrianNeural`), returning mp3 bytes streamed to the browser. the effect at the board is an opponent that plays at your level, says “taking the knight — it was undefended”, and can draw an arrow on the square it is talking about.

2.3 training and spaced repetition

the third leg is deliberate practice infrastructure (`openings.py`, `training.py`, `spaced_repetition.py`, `tactics.py`):

- **opening trainer**: play a chosen opening line against the engine; hints degrade as your recorded mastery improves (full move -> piece-only hint -> nothing, via `calculate_hint_level()`).

- **review cards:** positions become flashcards — a fen, the expected move in san, acceptable alternatives, and an explanation. cards are minted from opening lines, from your recorded blunders, and from tactical motifs.
- **scheduling** is a classic leitner box system (Leitner, Sebastian, 1972): six boxes with review intervals of 1, 2, 4, 7, 14 and 30 days; a correct answer promotes the card one box, an error demotes it to box 1. simpler than sm-2's per-card easiness factors (Woźniak, Piotr A. and Gorzelańczyk, Edward J., 1994), but the retention curve it approximates is the same spacing-effect economics.
- **stats:** every game can be saved as pgn and mined by `analysis.py` for accuracy and blunder counts, which feed `player_stats.json` and adjust the suggested skill level.

the pedagogical stack — engine as sparring partner, llm as commentator, spaced repetition as memory — is the software version of the standard chess-improvement advice: analyse your own games and drill your openings until recall is automatic (silman and engqvist have whole books of the human protocol (Silman, Jeremy, 1999) (Engqvist, Thomas, 2016)).

3 what each project taught

the js engine teaches the invariants: if perft is off by one at depth 4 you have a move-gen bug, full stop; if fhf/fh drops, your ordering broke; if the engine blunders at the horizon, quiescence is leaking. the voice project teaches restraint: the strongest component (stockfish) needed zero of my code, and the llm becomes trustworthy exactly to the degree you *remove* its freedom — feed it verified analysis, cap its words, parse its annotations, and let deterministic templates handle anything that must be true.

4 see also

- **ultimate tic tac toe** — minimax, negamax and the alpha-beta soundness argument in full
- **adversarial searching** — parent section
- **connect 4** — a smaller adversarial game
- **arcade** — where the js engine reference lives

references

Engqvist, Thomas (2016). *300 Most Important Chess Exercises*, Everyman Chess.

Leitner, Sebastian (1972). *So lernt man lernen: Der Weg zum Erfolg*, Herder.

Silman, Jeremy (1999). *The Complete Book of Chess Strategy*, Siles Press.

Woźniak, Piotr A. and Gorzelańczyk, Edward J. (1994). *Optimization of Repetition Spacing in the Practice of Learning*, Acta Neurobiologiae Experimentalis.