Build a Chess Engine

From Legal Moves to Neural Networks

Selected reader sample

A complete project promise, required board path, Bug Laboratory, worked alpha-beta trace, Janus case study, NNUE roadmap, and finished-engine close.

Lennart A. Conrad

First Edition · Reader Sample

SCOPE OF THIS SAMPLE

This sample contains selected, connected excerpts from the same authoritative source used for the complete and print editions. It is not the complete build path.

The classical engine is built end to end in the full book. Neural chapters are advanced implementation roadmaps; no trained network or turnkey training release is included. Janus appears as a production case study; its source is not included with this edition.

Self-published by the author. Contact: la.paloma.gordita@gmail.com

Inside This Sample

Inside This Sample 1
1 The Promise 2
2 How the Engine Fits Together 4
3 The Implementation Contract 7
4 Build the First Board 9
5 Bug Laboratory: Follow the First Wrong Branch 13
6 Worked Trace: Alpha-Beta 19
7 Evidence in Practice 24
8 Neural Roadmap: The NNUE Accumulator 26
9 The Engine You Built 28
Continue in Dependency Order 33

Inside This Sample

> This sample follows one coherent arc from promise to implementation, diagnosis, search, neural extension, and a finished-engine stopping point. It is not a stack of unrelated pages.

Excerpt	What it demonstrates
The promise	What the reader builds and how correctness shapes the journey
The architecture	How rules, search, evaluation, protocol, and tests exchange state
The contract	The shared types and ownership boundaries that keep chapters compatible
The first representation	The complete required flat-board path, its tests, and observable gate
A Bug Laboratory	Perft divide as a diagnosis funnel, followed by a worked answer
A search trace	Alpha-beta windows and one complete cutoff
Evidence in practice	A Janus aspiration-window case study
A neural roadmap	NNUE structure and the accumulator update
The stopping point	What the finished engine proves and what the reader practiced

The complete classical engine is built end to end. Neural material is an advanced implementation roadmap; no trained network or turnkey training release is included.

CHAPTER 1

The Promise

> *The project begins with an outcome and an evidence standard, not a catalogue of algorithms.*

There is a particular kind of magic in watching a program you wrote play a move you didn't expect, and play it *well*. You typed every line. You know there's no hidden intelligence in there, only loops, arrays, and a few clever heuristics. And yet the thing reaches across the board, forks your queen and rook, and you sit back and think: I didn't see that coming, and I built it.

Everything in this book builds toward that moment. By the end you will have written a real classical UCI engine: a program that reads a position, generates every legal move, searches under explicit depth, node, and time limits, evaluates what it finds, and always returns a legal move. Its speed and playing strength will depend on your language, hardware, and which optional extensions you earn through testing. It is not a toy that merely shuffles wood; it has the same architectural bones as the programs that beat grandmasters, without pretending that a first implementation inherits their Elo.

A strong player will ask the obvious question first: will it beat me? Nobody can honestly promise you a number. Strength is measured, not asserted, and the chapter on engine testing shows how to answer that question yourself: a paired match against a known opponent, reversed colors, and a confidence interval declared before the first game. Public rating lists such as CCRL and CEGT report comparable engines under their own pools, which is the closest calibration available before you have results of your own.

What You Will Build

You will build a complete classical engine through two kinds of checkpoint. A **chapter deliverable** is a tested component or runnable diagnostic tool; it need not play chess yet. A **named engine milestone** is a complete executable that can be loaded or played. The first such milestone arrives after legal move generation and perft: a deliberately terrible but fully legal random-move UCI engine. From there, each engine milestone remains runnable while its strength grows.

The required path is one vertical slice: position and attacks; reversible moves and complete legality; perft; evaluation; negamax and alpha-beta; iterative deepening, ordering, quiescence, and transposition reuse; clock control; UCI ownership; and reproducible matches. Optional lanes explain bitboards, selective search, books, tablebases, parallelism, NNUE, and policy-value systems without making any of them a hidden condition of finishing the classical engine.

The Joy and the Rigor

Engine programming is unusually satisfying because the feedback is brutally honest. Chess does not care about your intentions. Either your move generator produces exactly the legal moves or it does not. There is a famous test for this called *perft*: you count the leaf nodes of the game tree to a fixed depth and compare against known totals. From the starting position there are exactly 20 moves, then 400 replies, then 8902, then 197281, then 4865609 move paths

at depth 5. If your count is off by a single node, you have a bug, and you will find it, because the number does not lie.

That ruthlessness is the gift. Most software has fuzzy correctness; a chess engine has a stack of golden numbers waiting to catch your mistakes. You get the rare pleasure of a domain where "is it right?" has a real answer, and the even rarer pleasure of a program whose strength you can *measure* in rating points by having versions play each other.

Whatever brought you here is a good enough reason. Teaching engines, compact engines, reinforcement-learning projects, and competitive engines began with different definitions of success. Your motive is part of the specification; write it down before feature accumulation quietly replaces it.

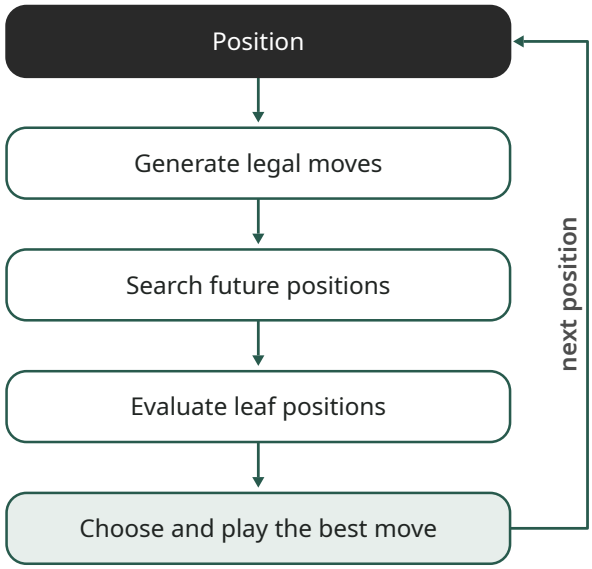
CHAPTER 2

How the Engine Fits Together

> *Before individual algorithms, the reader sees the one-way architecture they will assemble and the invariants owned by each layer.*

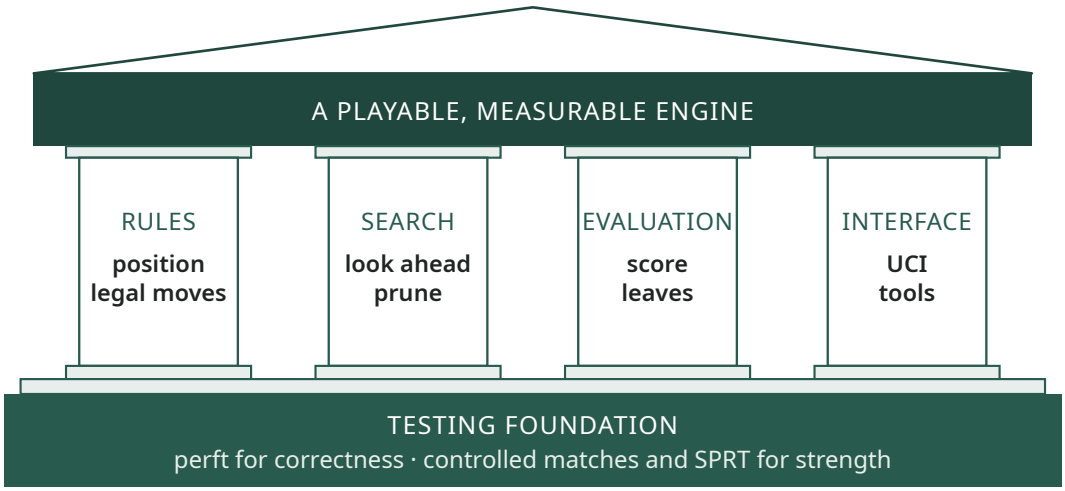
The Pipeline

At the highest level, every classical engine answers one question, repeatedly: *given this position, what is the best move?* It answers by looping a small set of stages.



That move produces the next position, and the loop starts over.

Step back from the loop and the whole book resolves into **four pillars on one foundation** — a model worth memorizing, because every chapter ahead lives in exactly one of them:

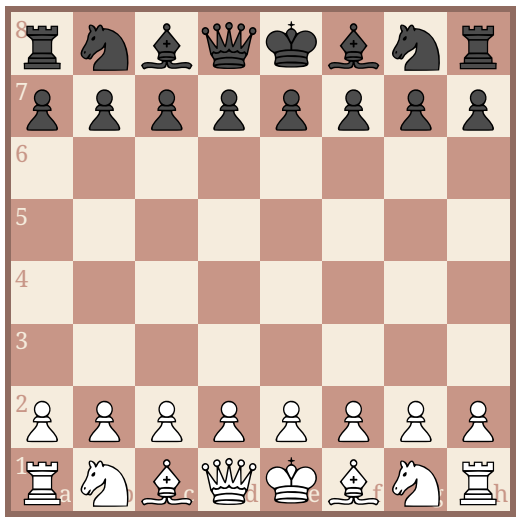


Four pillars, one foundation: every pillar is only as strong as the tests beneath it.

Rules covers the board, legal move generation, and make/unmake. This pillar has to be *perfect*, because every other one silently trusts it. **Search** is minimax, alpha-beta, and the refinements layered on them: the machinery of looking ahead. **Evaluation** decides what the search's leaves are worth, whether the terms are hand-written or learned by a network. **Interface** is UCI, the protocol through which the engine meets GUIs, tools, and opponents. Beneath all four sits **Testing** — perft for the rules, SPRT for strength. It touches every pillar, which is why it keeps its own chapters.

Six concepts flesh out those pillars. Learn their names now; each gets its own chapters later.

Board representation. The data structure that holds the position: where every piece sits, whose turn it is, castling rights, the en passant square, and the move clocks. Everything else reads from and writes to this. The classic high-performance choice is the *bitboard* — a 64-bit integer per piece type, one bit per square — so that "all white pawns" is a single machine word and "squares a knight attacks" is a table lookup. A simpler starting point is an 8 × 8 (or 10 × 12) array of pieces.



That starting position, in a bitboard engine, is just a handful of 64-bit constants. We cover representation first because every other part depends on it.

Move generation. Given a position, produce the legal moves. This is pure rule-keeping: how each piece slides or jumps, plus the awkward special cases — castling, en passant, promotion, and the rule that you may never leave your own king in check. Move generation must be *correct* (no illegal moves, no missed moves) and *fast*, because search will call it millions of times per second. We verify correctness with a technique called *perft*, which counts leaf nodes to a fixed depth and compares against known answers.

Search. The engine cannot see the best move directly; it discovers it by looking ahead. Search explores a tree of possibilities: my moves, your replies, my replies to those, and so on. The core algorithm is *minimax* — I maximize my score, you minimize it — made practical by *alpha-beta pruning*, which skips branches that cannot affect the result. Real engines layer many refinements on top: iterative deepening, transposition tables, quiescence search, move ordering, and selective pruning. Search is where most of an engine's strength lives.

Evaluation. Search eventually reaches positions it will not look past (the *leaves*). Something must put a number on those positions. The public evaluator in this book is side-to-move-relative: positive favors the player about to move, negative favors the opponent, and zero is equal. A component may accumulate White minus Black internally, but flips exactly once at the boundary. A simple evaluation counts material (a queen is worth about nine pawns) and

adds positional terms — piece activity, king safety, pawn structure. Modern engines replace hand-written terms with a small neural network (*NNUE*) evaluated quickly on the CPU. We start with a hand-crafted evaluation because you must understand *what* to measure before you let a network learn it.

Move choice. After search scores every root move, the engine picks the highest-scoring one (with care for draws, repetitions, and time). This stage is small but real: it is where the search result becomes a decision, and where time management decides how long to think.

UCI. An engine needs to talk to the outside world. The *Universal Chess Interface* is the de facto text protocol that graphical front-ends (Arena, Cute Chess, Banksia) and tournament managers use to drive engines. You read commands on standard input and write replies on standard output: `uci`, `isready`, `position`, `go`, `bestmove`. Implementing the UCI **transport shell** early makes the process discoverable and testable. It becomes playable only after the legal generator is connected at the named random-engine milestone; timed matches wait for search and clock ownership.

CHAPTER 3

The Implementation Contract

> *These shared names and boundaries are what let a language-neutral project remain mechanically coherent from chapter to chapter.*

The Implementation Contract

Before the modules grow, freeze their seams. This one-page contract governs every final implementation snippet; the Canonical Engine Contract supplies the complete layouts, score conversions, failure policy, and integration audit.

Boundary	Canonical contract
Squares	LERF: `a1 = 0`, `h8 = 63`; bit operations use unsigned 64-bit values
Types	`Position`, `Move`, `Undo`, `SearchState`, `SearchLimits`, `SearchResult`, `TTEntry`
Notation	`parse_fen(text, mode)` and `write_fen(pos, policy)`
Rules	`generate_legal(pos, mode)`, `make_move(pos, move) -> Undo`, `unmake_move(pos, move, undo)`
Engine	`evaluate(pos) -> Score`, `search_root(state, limits) -> SearchResult`
Scores	Node scores favor the side to move; a root result favors the root side; normal eval, tablebase, mate, and infinity occupy separate ranges
State	Move identity excludes clocks but includes effective legal EP; search state adds rule-50 and repetition history; notation adds fullmove/policy data
Ownership	Rules own state and transitions; evaluation reads rules; search owns path/TT/time; UCI owns one active worker and one `bestmove`

`Move` flags are combinable: a capture-promotion is both `CAPTURE` and `PROMOTION`. `Undo` stores the full captured piece, prior castling rights, en passant candidate, both clocks, prior key, and any incremental evaluator state that cannot be safely derived. In-place code always uses the same transaction:

```
undo = make_move(pos, move)
...
unmake_move(pos, move, undo)
```

En passant has three deliberately separate meanings. `ep_candidate` preserves the notation/last-double-push target; geometric availability ignores king safety; `effective_ep_file(pos)` exists only when at least one capture is fully legal. The default FEN writer prints an en passant target only in that last case. Move identity and repetition use the effective file. Polyglot uses its own specified geometric adapter. The effective test is a direct occupancy-and-attack check, never a recursive call back into ordinary legal generation.

The classical engine is built end to end. The neural chapters are advanced implementation roadmaps. This edition does not include the training-data format, training command, licensed compatible model, and synchronized inference package required to call a neural path runnable. The neural material therefore defines implementation and verification contracts without pretending that the reader has received those artifacts.

CHAPTER 4

Build the First Board

> *The required path comes first. Alternative representations remain valuable reference material, but they do not interrupt this milestone.*

Square Indexing

Fix an indexing convention before choosing a data structure. The most common is **a1** = **0**, counting left to right along a rank, then bottom to top:

	a	b	c	d	e	f	g	h
8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7

Given a square index `sq`:

```
file(sq) = sq % 8        # column, 0..7  (a..h)
rank(sq) = sq / 8        # row,    0..7  (1..8)
square(f, r) = r * 8 + f
```

This is the **little-endian rank-file** mapping (LERF) used by Stockfish, Leela, and most open-source engines. Some engines put **a8** = **0** to match FEN reading order. Either works; pick one, write it at the top of your source, and never waver. The board diagrams in this book always show rank 8 at the top, but index 0 is always a1.

PREDICT · P-02-01 · The far corner

Before you run it. With LERF indexing (`a1 = 0`, files increasing first), predict the numeric index of `h8` before evaluating the formula.

The Flat 64-Square Array — Required Implementation

This is the representation the rest of the book builds on. Everything after it in this chapter is optional: read it for understanding, but do not implement it before you have a working perft.

DESIGN DECISION · One authoritative placement

Problem. Redundant board views can disagree. **Teaching default.** Keep the 64-square placement array authoritative and derive queries from it. **Production alternative.** Maintain synchronized piece lists and bitboards. **Cost.** Derived scans are slower. **Switch when.** Fixed-work profiling shows representation work, rather than search design, limits useful nodes.

Concept

The simplest board is a flat array of 64 cells. Each cell holds a piece code, an integer such as `WHITE_PAWN = 1` or `BLACK_KNIGHT = 8`, with `EMPTY = 0`.

```
0  = empty
1  = white pawn    7  = black pawn
2  = white knight  8  = black knight
3  = white bishop  9  = black bishop
4  = white rook    10 = black rook
5  = white queen   11 = black queen
6  = white king    12 = black king
```

Lookups and updates are both O(1):

ALGORITHM

```
function piece_at(board, sq):
    return board[sq]

function move_piece_unchecked(board, from_sq, to_sq):
    board[to_sq]   = board[from_sq]
    board[from_sq] = EMPTY
```

`move_piece_unchecked` demonstrates only the array writes. It is not the engine's move transaction: the canonical `make_move` introduced in the reversible-state chapter also returns `Undo` and updates captures, rights, clocks, en passant, keys, and incremental evaluation state.

The Off-Board Problem

The sting appears in move generation. A knight on h1 has a candidate at `sq + 17`. But `7 + 17 = 24`, which is a4 — the a-file, three ranks up, not a knight's move from h1 at all. Nothing in a flat array stops you wandering off an edge, so every candidate needs an explicit bounds and file-wrap check:

ALGORITHM

```
function is_valid(sq):
    return 0 <= sq < 64

# Rook sliding east from sq:
target = sq + 1
while is_valid(target) and file(target) != 0:
    visit(target)
    target = target + 1
```

A rook on h4 must not slide to a5, yet `sq + 1` steps from index 31 (h4) to 32 (a5), both inside 0..63. The extra `file(target) != 0` test catches the wrap. These checks are cheap individually but clutter the code and are easy to forget.

For a learning engine, a puzzle solver, or a perft validator to depth 4 or 5, an 8 × 8 array is adequate. Its simplicity is a feature: bugs have fewer places to hide.

Representation-Neutral Queries and Derived Occupancy

Later chapters should ask the position *questions*, not reach into its storage. Give the required array a small public surface:

ALGORITHM

```
function piece_at(pos, sq):
    return pos.placement[sq]

function count_piece(pos, color, piece_type):
    return count(sq where piece_at(pos, sq) == piece(color, piece_type))

function count_color(pos, color):
    return count(sq where color_of(piece_at(pos, sq)) == color)

function occupancy_count(pos):
    return count(sq where piece_at(pos, sq) != EMPTY)
```

An occupancy view is a derived answer to "which squares are occupied?" It may be a Boolean array or a 64-bit mask. Build it by scanning `placement`; do not keep a second authoritative board:

ALGORITHM

```
function derive_occupancy(pos, wanted_color = ANY):
    occupied = empty_square_set()
    for sq in A1 .. H8:
        piece = piece_at(pos, sq)
        if piece != EMPTY and
           (wanted_color == ANY or color_of(piece) == wanted_color):
            occupied.add(sq)
    return occupied
```

If profiling later justifies cached occupancies, the scan becomes their from-scratch oracle. That is the recurring production pattern in this book: start with one truth, then make every redundant speedup prove agreement.

Text Rendering

Your first visualization should be selectable text, not a GUI. Map empty squares to `.`, White pieces to uppercase letters, and Black pieces to lowercase. Print rank 8 first even though LERF index zero is a1:

ALGORITHM

```
function print_board(pos):
    for rank in 7 down to 0:
        line = ""
        for file in 0 .. 7:
            line += piece_symbol(piece_at(pos, square(file, rank)))
        print(line)
```

This tiny renderer will localize FEN, move, and unmake bugs long before a board widget would. Keep coordinates or rank/file labels available in debug mode.

Required Tests

Load the starting placement and require representation-neutral facts:

```
count_piece(pos, WHITE, PAWN) == 8
count_color(pos, BLACK)       == 16
piece_at(pos, E1)             == WHITE_KING
occupancy_count(pos)          == 32
```

Then round-trip several FEN *placement fields*: print the board, serialize the placement, and require the same eight rank widths and piece locations. No legal generator is needed. If any derived view is cached, rebuild it after every test mutation and compare it with the cache.

RUN IT NOW · Chapter gate

Command or action. Construct the initial position, print its eight ranks, and query E1, the White pawn count, the Black piece count, and total occupancy. **Expected output shape.** The text board has rank 8 at the top; the four queries return the king, 8, 16, and 32. **What failure means.** A rotated display suggests presentation trouble; wrong counts or E1 indicate indexing or encoding trouble. **Safe next step.** Keep one placement array authoritative and make every occupancy view derived.

01 BUILD

What to implement

- ☐ Fix a square convention (`a1 = 0`) and implement `file(sq)`, `rank(sq)`, and `square(file, rank)`.
- ☐ Build one 64-entry `placement` array with `piece_at`, `set_piece`, and `clear_square`. These are storage operations, not the later rule-aware `make_move` transaction.
- ☐ Implement `count_piece`, `count_color`, and a derived occupancy view.
- ☐ Implement `print_board` from rank 8 to rank 1.
- ☐ Pass the boundary, count, placement-round-trip, and derived-view tests.

That is the chapter gate. You may continue directly to position state. The gallery below is reference material; it becomes implementation work only when a measured bottleneck or later project needs it.

CHAPTER 5

Bug Laboratory: Follow the First Wrong Branch

> *This excerpt shows the book's diagnostic style: reduce an aggregate failure until one invariant and one regression fixture remain.*

The Divide Output

Counting one big number tells you *that* you are wrong. **Perft divide** tells you *where*. Divide runs perft to depth `d`, but instead of one total it prints, for each root move, the perft count of the resulting position at depth `d-1`:

ALGORITHM

```
function divide(pos, depth):
    if depth == 0:
        print("total", 1)
        return 1
    total = 0
    for move in generate_legal(pos, ALL):
        undo = make_move(pos, move)
        n = perft(pos, depth - 1)
        unmake_move(pos, move, undo)
        print(move, n)
        total += n
    print("total", total)
    return total
```

Thus `divide(pos, 0)` has no move rows and total 1; at depth 1 every legal root move has count 1. Reject negative depths. The depth-1 bulk shortcut is valid only because `generate_legal` returns a truly legal list.

The output looks like this (start position, depth 5):

```
a2a3: 181046
a2a4: 217832
b2b3: 215255
...
e2e4: 405385
...
total: 4865609
```

Stockfish and many perft tools print one root move and count per line, but spelling, separators, ordering, and the final total vary. Normalize moves to UCI notation and sort the lines before diffing two implementations.

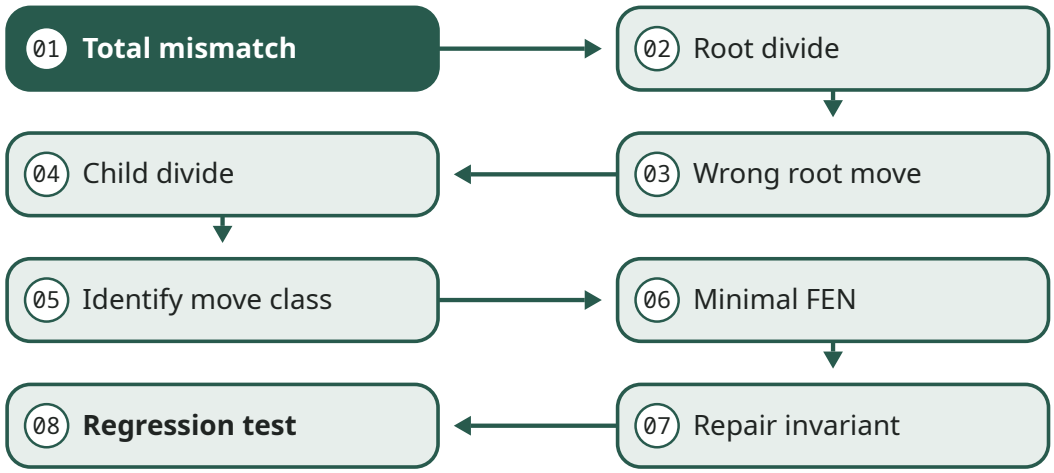
EXTEND · X-07-01 · Move-class divide counters

Contained extension. Extend divide with capture, en-passant, castle, promotion, check, and mate counters without changing the canonical node total.

Done when: Divide prints every requested counter, preserves the ordinary node total, and matches each available manifest breakdown.
Regression: Check the counters only at depths where independently sourced manifest values exist.

The Debugging Loop

This is the technique that makes perft so powerful. Diff normalized divide output from your engine and a trusted reference, then follow only the first subtree whose count differs.



The diagnosis funnel converts one aggregate mismatch into one named rule and one permanent fixture.

For example, if only the `e1c1` subtotal differs in a castling-rich position, follow that child rather than rereading the whole generator. The move already classifies the likely subsystem. Reduce pieces until the disagreement still reproduces, repair the violated castling occupancy or attack invariant, and keep that minimal FEN beside the aggregate perft vector. At depth 1, the diff is a concrete set of wrong moves in one fully known position.

DIAGNOSE · D-07-01 · Only one Kiwipete root move is wrong

Name the violated invariant before editing. The total is wrong, and divide shows exactly one differing root move. What should you do next instead of rereading the whole generator?

BUG LABORATORY · BL-07-01 · From mismatch to one rule

Work the laboratory in order. Start-position perft passes, Kiwipete fails, and the first divide mismatch is a queenside castle. Use the diagnosis funnel to produce a minimal regression rather than a speculative patch.

The Standard Test Positions

The community has converged on a small set of positions that, between them, stress every tricky rule. Memorize the first; keep the rest in a test file. All are given in FEN.

Position 1 — the start position.

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Depth	Nodes	Captures	E.p.	Castles	Promos	Checks	Mates
1	20	0	0	0	0	0	0
2	400	0	0	0	0	0	0
3	8902	34	0	0	0	12	0
4	197281	1576	0	0	0	469	8
5	4865609	82719	258	0	0	27351	347
6	119060324	2812008	5248	0	0	809099	10828

The extra columns classify the *final-ply* move that reached each leaf: captures (en passant included, and also counted in its own column), castles, promotions, checks, and checkmates. They localize a bug the node total only detects — a wrong E.p. column with correct captures points straight at the en-passant rule. The Perft Test Positions reference defines each column and extends every table deeper.

Position 2 — "Kiwipete". Dense and tactical; catches castling, pins, and many capture cases.

```
r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1
```

Depth	Nodes	Captures	E.p.	Castles	Promos	Checks	Mates
1	48	8	0	2	0	0	0
2	2039	351	1	91	0	3	0
3	97862	17102	45	3162	0	993	1
4	4085603	757163	1929	128013	15172	25523	43

Position 3 — en-passant and discovered checks.

```
8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - 0 1
```

Depth	Nodes	Captures	E.p.	Castles	Promos	Checks	Mates
1	14	1	0	0	0	2	0
2	191	14	0	0	0	10	0
3	2812	209	2	0	0	267	0
4	43238	3348	123	0	0	1680	17
5	674624	52051	1165	0	0	52950	0
6	11030083	940350	33325	0	7552	452473	2733

Position 4 — promotions, including underpromotion, with the side to move in a sharp spot.

```
r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1
```

Depth	Nodes	Captures	E.p.	Castles	Promos	Checks	Mates
1	6	0	0	0	0	0	0
2	264	87	0	6	48	10	0
3	9467	1021	4	0	120	38	22
4	422333	131393	0	7795	60032	15492	5

Depth	Nodes	Captures	E.p.	Castles	Promos	Checks	Mates
5	15833292	2046173	6512	0	329464	200568	50562

Position 5 and 6 round out the set with more promotion and castling traps:

```
rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8
```

Depth 3 = 62379, depth 4 = 2103487.

```
r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10
```

Depth 3 = 89890, depth 4 = 3894594.

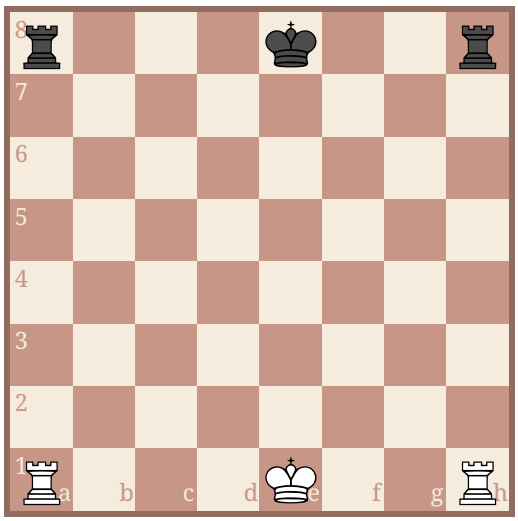
If your engine matches all six to the depths shown, your move generation is almost certainly correct.

Classic Bugs Perft Exposes

Move generation has a handful of recurring bugs. Perft finds every one of them. Here are the usual suspects, with the rule that is easy to get wrong.

Castling

Castling is the single most bug-ridden move. The rules: neither king nor the chosen rook has moved; the squares between them are empty; the king is not currently in check; and the king does not pass *through* or *land on* an attacked square. The rook may pass through an attacked square; only the king's path matters.

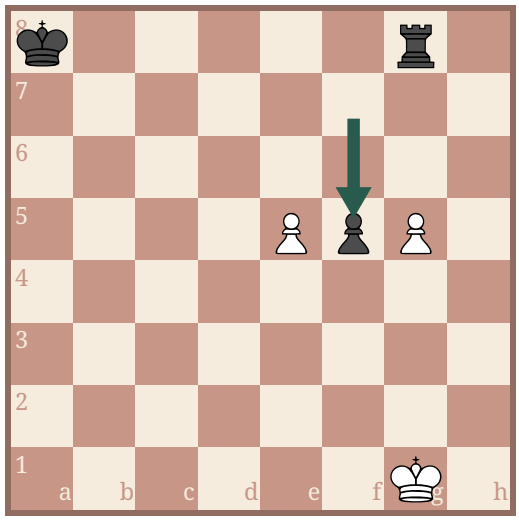


Common mistakes: checking the rook's path for attacks (wrong, only b1/b8 must be *empty* for queenside, not safe); forgetting that the king cannot castle out of check; updating castling rights only when the king moves but not when a rook moves or is **captured** on its home square. That last one is a favorite: if a rook is captured on h1, White instantly loses kingside castling rights, and many engines forget to clear the flag on the capturing side's make_move.

En passant

En passant is legal only on the move immediately after the opposing pawn's two-square advance, and only by a pawn on the correct adjacent file. Bugs: leaving the en-passant target square set for too long (clear it every move, then retain a new candidate only when a legal

capture exists); allowing en passant by the wrong pawn; and, the nastiest, the **en-passant discovered check**.



Black has just played f7-f5. The target f6 is canonical here because White's e5-pawn has the legal capture exf6 en passant. White's g5-pawn also appears able to play gxf6, but moving it off the g-file opens the g8-rook's line to the white king on g1, so *that* capture is illegal. A naive legality check that only removes the captured pawn's square (not the moving pawn's origin) admits both moves. This two-pawn construction tests the edge case without relying on a noncanonical en passant field. Position 3 above reaches related discovered-check cases deeper in its tree.

Promotion

A pawn reaching the last rank must promote, and it can promote to **four** different pieces. Bugs: generating only a queen promotion (missing N, R, B), which undercounts; generating promotion to king or pawn, which overcounts; and mishandling **capture promotions** (a pawn capturing onto the last rank also yields four moves). Under-promotion to a knight matters in real play, so this is not academic.

Watch the node arithmetic. If your `perft 1` is right but `perft 2` is too low by a multiple of three on a promoting position, you are generating one promotion piece instead of four.

Pins and Check Evasion

When the king is in check, only three responses are legal: move the king to safety, capture the checker, or block the line (against sliders). A pinned piece may not move off the king–pinner line. A compatible slider or pawn may still move along that line, including capturing the pinner; a pinned knight cannot. Generators that filter pseudo-legal moves with a full make/check-king pass handle these automatically; generators that try to be clever with pin masks often get the pawn-capture and en-passant-along-the-pin cases wrong.

Worked answer · BL-07-01: From mismatch to one rule

Minimal fixture. Remove unrelated pieces while preserving rights, rook, blockers, and the suspect attack geometry. **Plausible causes.** emptiness mask; safety mask; rook motion; rights update; unmake restoration **Discriminator.** Compare legal lists and full snapshots before/after the castle in the minimized position. **Violated invariant.** Castling preconditions and the reversible two-piece transition must both be exact. **Repair.** Fix only the failed mask

or state transition identified by the discriminator. **Expected observation.** The minimal fixture isolates one mask or restored field instead of many unrelated move classes. **Regression.** Keep the minimal FEN, legal-list assertion, transaction snapshot, and original Kiwipete vector. **Common wrong answer.** Editing en-passant or pin logic merely because Kiwipete contains those features elsewhere.

Return to BL-07-01.

CHAPTER 6

Worked Trace: Alpha-Beta

The trace makes every incoming window, negation, update, return, and cutoff visible.

The Core Idea: Two Bounds

Alpha-beta carries two numbers down the tree:

- **alpha** is the lower edge of the score window: values at or below it cannot improve the choice already available to the relevant ancestor.
- **beta** is the upper edge: reaching or exceeding it proves enough for the relevant ancestor to reject the remaining alternatives at this node.

Together `[alpha, beta]` is the **search window**. A node's true value below alpha is irrelevant (we already have something better elsewhere), and a value at or above beta is "irrelevant" in a more useful way — the opponent would never have allowed this position, so the parent can stop searching.

When that second case fires, we get a **beta cutoff**: the search abandons the remaining sibling moves and returns immediately. This is where all the savings come from.

A Complete Worked Cutoff

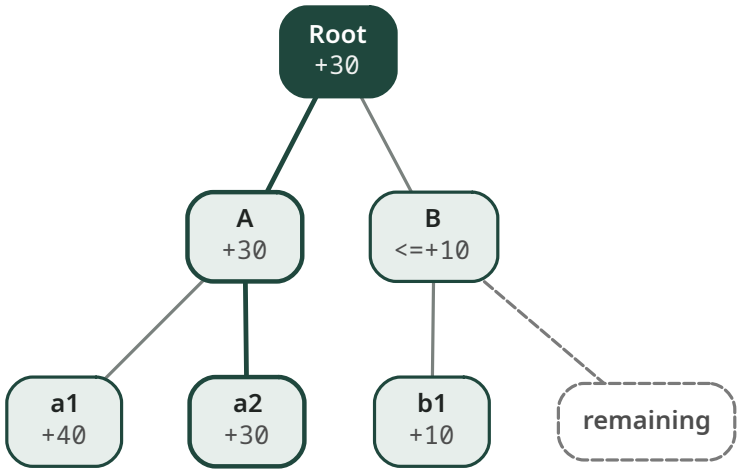
Use a two-ply tree with root moves A and B. Under A, the opponent can hold the root side to 30; under B, its first reply already holds the root side to 10. Every score in the trace belongs to the *side to move at that node*. The child returns first; the parent negates it before comparing it with its own window.

Step	Node	STM	d	α_{in}	β_{in}	Child	Negate	α_{out}
1	R→A	Us	2	$-\infty$	$+\infty$	—	—	$-\infty$
2	A→a1	Them	1	$-\infty$	$+\infty$	+40	−40	−40
3	A→a2	Them	1	−40	$+\infty$	+30	−30	−30
4	R after A	Us	2	$-\infty$	$+\infty$	−30	+30	+30
5	R→B	Us	2	+30	$+\infty$	—	—	+30
6	B→b1	Them	1	$-\infty$	−30	+10	−10	−10
7	R after B	Us	2	+30	$+\infty$	−10	+10	+30

Step	Return or bound	Cutoff reason
3	A returns −30 exact	Both replies searched; value lies inside the window
6	B returns −10 lower bound	At B, −10 is at least beta (−30); remaining replies are pruned
7	R returns +30 via A	B backs up to +10 at the root and cannot raise alpha

The apparent sign flip in step 6 is the whole mechanism: B's `−10` is good enough for the opponent to refute B from the root's perspective. The unvisited B replies cannot make B exceed the already available +30, so their exact values are irrelevant.

Root order	Leaf probes	First cutoff	Returned score
A then B (good)	3	B after `b1`	+30
B then A (poor)	4	None in this tree	+30



The completed A branch raises alpha to +30. The first reply under B proves that B cannot improve that result, so the dashed siblings are never visited.

Ordering changes the work, not the minimax answer. That equivalence is the correctness oracle you will use throughout the next chapters.

Fail-High and Fail-Low

Two pieces of vocabulary you will see everywhere:

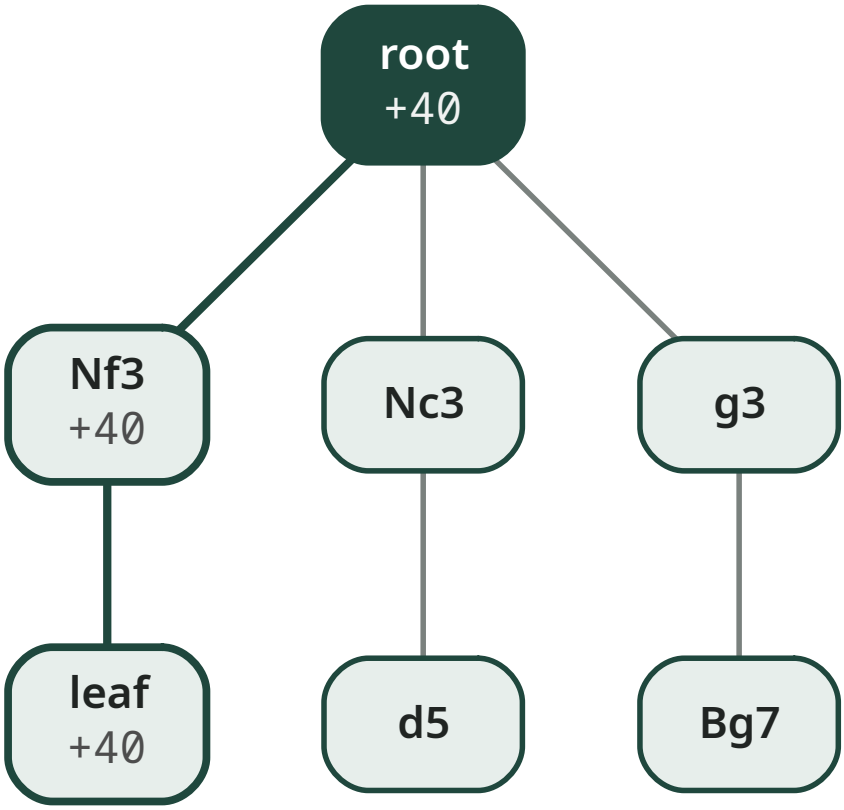
- **Fail-high**: a node returns a score `>= beta`. The move was *too good* — the opponent will avoid this line, so we stop searching siblings. The returned value is a *lower bound* on the true score, not the exact score.
- **Fail-low**: a node returns a score `<= alpha`. No move here was good enough to raise alpha. The returned value is an *upper bound*; we learned only that nothing here improves on what we already had.

A node that returns a score strictly *inside* the window (`alpha < score < beta`) is an **exact** result. The path through these exact nodes is the principal variation.

This bound/exact distinction is exactly the information the transposition table will store. A node searched to completion after raising alpha can return an exact value; principal-variation nodes are the most visible examples, but "PV" and "exact" are not interchangeable labels in every implementation.

The Principal Variation

The **principal variation** (PV) is the sequence of best moves for both sides — the line the engine actually expects to be played. The score at the root is the score of the PV's final leaf.



The bold path is the principal variation. Every node on it carries the same score: the root's +40 is the PV leaf's +40, passed back up the line.

The first move of the PV is the move the engine plays. The rest is its prediction of the opponent's and its own follow-ups. Printing the PV is the single most useful debugging aid you will build: if the PV is nonsense, your evaluation or search has a bug. Most engines collect the PV during search and emit it in the UCI `info` line.

You can recover the PV two ways: a **triangular PV table** (collect moves as the search returns) or a walk of the transposition table after the search. The PV table is simpler and bug-resistant, so prefer it first.

Negamax Alpha-Beta

Negamax exploits the fact that `max(a, b) = -min(-a, -b)`. Instead of separate maximize and minimize code, every node maximizes its own score, and we negate the child's value and swap/negate the window when we recurse. One code path, half the bugs.

Here is a clean recursive alpha-beta in negamax form:

ALGORITHM

```
function alphabeta(state, depth, alpha, beta, ply):
    pos = state.pos
    moves = generate_legal(pos, ALL)
    if moves is empty:
        if in_check(pos): return -(MATE - ply)
        return 0                          # stalemate

    # Apply your rule-accurate/engine draw policy only after
    # mate and stalemate have been distinguished.
```

```
    if draw_by_documented_search_policy(state): return 0

    if depth <= 0:
        return evaluate(pos)   # +ve = side to move

    best = -INF
    for move in moves:
        undo = make_move(pos, move)
        state.repetition_keys.append(pos.key)
        score = -alphabeta(state, depth - 1,
                           -beta, -alpha, ply + 1)
        state.repetition_keys.pop()
        unmake_move(pos, move, undo)

        if score > best:
            best = score
            if score > alpha:
                alpha = score  # new PV / raise floor

        if score >= beta:
            return best        # fail-high (cutoff)

    return best
```

A few details worth dwelling on:

- The window flips on recursion: the child is called with `(-beta, -alpha)`. From the child's viewpoint, our beta is its alpha and vice versa, both negated.
- The evaluation returns a score *relative to the side to move*. Positive means "good for whoever is on move." If your `evaluate()` is absolute (positive = White), wrap it: `return side == WHITE ? eval : -eval`.
- Returning `best` even when it lies outside the window is the "fail-soft" style: a result above `beta` or below `alpha` is still a valid *bound*, and those tighter bounds pay off once you add a transposition table. The "fail-hard" variant clamps instead — `return beta` on a cutoff, `return alpha` when no move reaches the window. Either is correct, but pick one and use it *everywhere* (main search and quiescence); this book uses fail-soft throughout.

PREDICT · P-11-01 · The child's window

Before you run it. At a negamax node with window `[alpha, beta)`, predict the window passed to the child before reading the recursive call.

Wiring Up the Root

The root is a special case: you must remember *which* move produced the best score, not just the score itself.

ALGORITHM

```
function search_root(state, limits) -> SearchResult:
    assert limits.depth is set
    pos = state.pos
    root_moves = generate_legal(pos, ALL)
    if root_moves is empty:
        score = -MATE if in_check(pos) else 0
        return SearchResult(best_move=NONE, score=score,
                            depth=0, completed=true)
```

```
    best_move = root_moves[0]       # legal fallback before any search
    if draw_by_documented_root_policy(state):
        return SearchResult(best_move=best_move, score=0,
                            depth=0, completed=true)

    alpha = -INF
    beta  = +INF
    for move in root_moves:
        undo = make_move(pos, move)
        state.repetition_keys.append(pos.key)
        score = -alphabeta(state, limits.depth - 1,
                           -beta, -alpha, 1)
        state.repetition_keys.pop()
        unmake_move(pos, move, undo)
        if score > alpha:
            alpha = score
            best_move = move
    return SearchResult(best_move=best_move, score=alpha,
                        depth=limits.depth, completed=true)
```

Call the root with the full open window `(-INF, +INF)`. Never let a beta cutoff happen at the root in this simple form — there is no parent to benefit, so we keep beta at infinity here.

BUG LABORATORY · BL-11-01 · The engine chooses the slower mate

Work the laboratory in order. Two moves both force mate, but the engine prefers the line that mates later. The plain score signs look correct. Diagnose the distance handling.

CHAPTER 7

Evidence in Practice

> *A production technique earns its default through correctness gates, telemetry, and confirmation—not familiarity.*

Aspiration Windows

A full-width root search uses the window `(-INFINITY, +INFINITY)`. But after iterative deepening completes depth d you have a score estimate `s`, and the depth d+1 score is usually close to it. So open a *narrow* window around `s`:

ALGORITHM

```
function aspiration_search(state, depth, previous_score) -> SearchResult:
    if depth <= 1:
        return search_depth(state, depth, -INF, +INF)

    delta = 25                  # illustrative quarter-pawn starting band
    alpha = previous_score - delta
    beta  = previous_score + delta

    loop:
        result = search_depth(state, depth, alpha, beta)
        if result.aborted: return result
        if result.score <= alpha:       # fail-low
            beta  = (alpha + beta) / 2
            alpha = alpha - delta
            delta = delta * 2
        else if result.score >= beta:   # fail-high
            beta  = beta + delta
            delta = delta * 2
        else:
            return result                # inside window: exact
```

A narrower window produces more cutoffs, so when the guess is right the search is cheaper. When the true score lands outside the window, the search *fails* (low or high) and returns a bound, not an exact score, so you must widen and re-search. Doubling `delta` (or jumping straight to infinity after one or two failures) keeps the number of re-search rounds small — though their node cost can still be anything but small, as the measurements below show.

The trade-off: aspiration windows are a net win at higher depths where the score is stable, and a small loss if your evaluation is noisy or the position is volatile. Start `delta` around 1/4 to 1/2 a pawn. On a fail-low at the root, also consider granting extra time — the engine just learned the position is worse than it thought.

A worked picture. Suppose depth 9 returns +0.40. For depth 10 you search the window (+0.15, +0.65). Three outcomes:

- True score is +0.45: inside the window, search is fast. Done.
- True score is +0.90 (you found something good): fail-high at +0.65, widen the upper bound, re-search.

- True score is -0.30 (something went wrong): fail-low at +0.15, widen the lower bound, re-search, and buy more time.

JANUS CASE STUDY · Set aspiration width by evidence

Question. Would a wider initial band reduce costly retries without giving away the benefit of a narrow search?
Hypothesis. Fewer re-searches would more than repay the wider first window.
Correctness gate. Fixed-depth scores and completed root results had to remain identical.
Measurement. A controlled Janus campaign instrumented retries, searched work, and confirmation games.
Result — Confirmed. The wider band was retained only after confirmation, not from a single benchmark.
Decision. Keep aspiration search and its measured width behind a switch.
Reader lesson. Retry rate is diagnostic telemetry; games decide whether its remedy belongs in the engine.

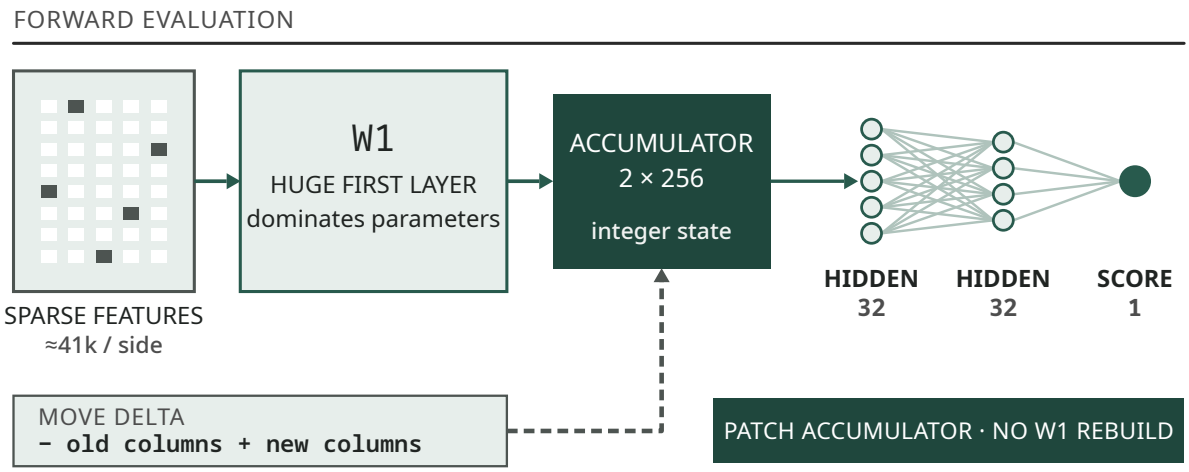
CHAPTER 8

Neural Roadmap: The NNUE Accumulator

> *The classical engine is the end-to-end build. This neural excerpt shows the level of implementation detail in the advanced roadmap without implying that a trained model is included.*

An Illustrative Network Shape

A classic HalfKP topology looks like this; dimensions and activations differ among formats and engine versions:



The first layer dominates the parameter count: tens of millions of weights. The dense layers are tiny by comparison. This imbalance is the lever NNUE pulls. The big layer is the one we make incremental; the small layers we just recompute every node, because they are cheap.

The Accumulator

The accumulator is the output of the first layer *before* the nonlinearity. Each king/color perspective owns a vector of, say, 256 integers. Crucially, the first layer is a plain matrix multiply of a *sparse* binary input vector: the input is a list of "active features", each 0 or 1.

Because the input is binary, the matrix product reduces to a sum of weight columns:

```
accumulator = bias
for each active feature f:
    accumulator += W1[:, f]
```

That is just vector addition. And vector addition has an inverse: subtraction. So when a move changes the set of active features, we do not redo the whole sum. We add the columns for features that turned on and subtract the columns for features that turned off.

ALGORITHM

```
function patch_accumulator(acc, removed_features, added_features):
    for feature in removed_features:
        acc -= W1[:, feature]
    for feature in added_features:
        acc += W1[:, feature]
```

Here is the whole update on a four-lane toy accumulator. The numbers are illustrative; the add/subtract contract is the production one.

Stage	Vector	Operation
Old accumulator	`[12, 7, 3, 9]`	Sum of bias and active feature columns
Removed column	`[2, -1, 4, 0]`	Subtract the feature that switched off
Added column	`[-1, 3, 2, 5]`	Add the feature that switched on
New accumulator	`[9, 11, 1, 14]`	`old - removed + added`

A normal quiet move changes only a handful of features, so the engine performs a few hundred integer additions instead of a sparse refresh across every active column. Refresh from scratch when the perspective's king anchor or bucket changes, when the cached version is incompatible, or when a lazy-update chain cannot be reconstructed exactly. Incremental work—not a different network answer—is the speedup.

ML RESEARCH NOTE · Incremental inference needs a scalar truth

Every patched accumulator must equal a fresh scalar rebuild within the declared integer contract. King-bucket changes force refresh; ordinary feature deltas subtract removed columns and add new ones. Bound products and fan-in before choosing integer widths, then compare scalar and every SIMD backend on the same versioned model.

PREDICT · P-26-01 · Increment or refresh?

Before you run it. In a king-bucketed NNUE, the king crosses into a different bucket. Predict whether ordinary remove/add feature deltas are sufficient for that perspective's accumulator.

To support unmaking moves cheaply, keep one accumulator per search ply: copy, patch, and restore the old copy on undo. The integration section below adds the lazy-update policy most engines layer on top.

CHAPTER 9

The Engine You Built

> *The course closes with objective capabilities, retained evidence, portfolio language, and explicit permission to stop.*

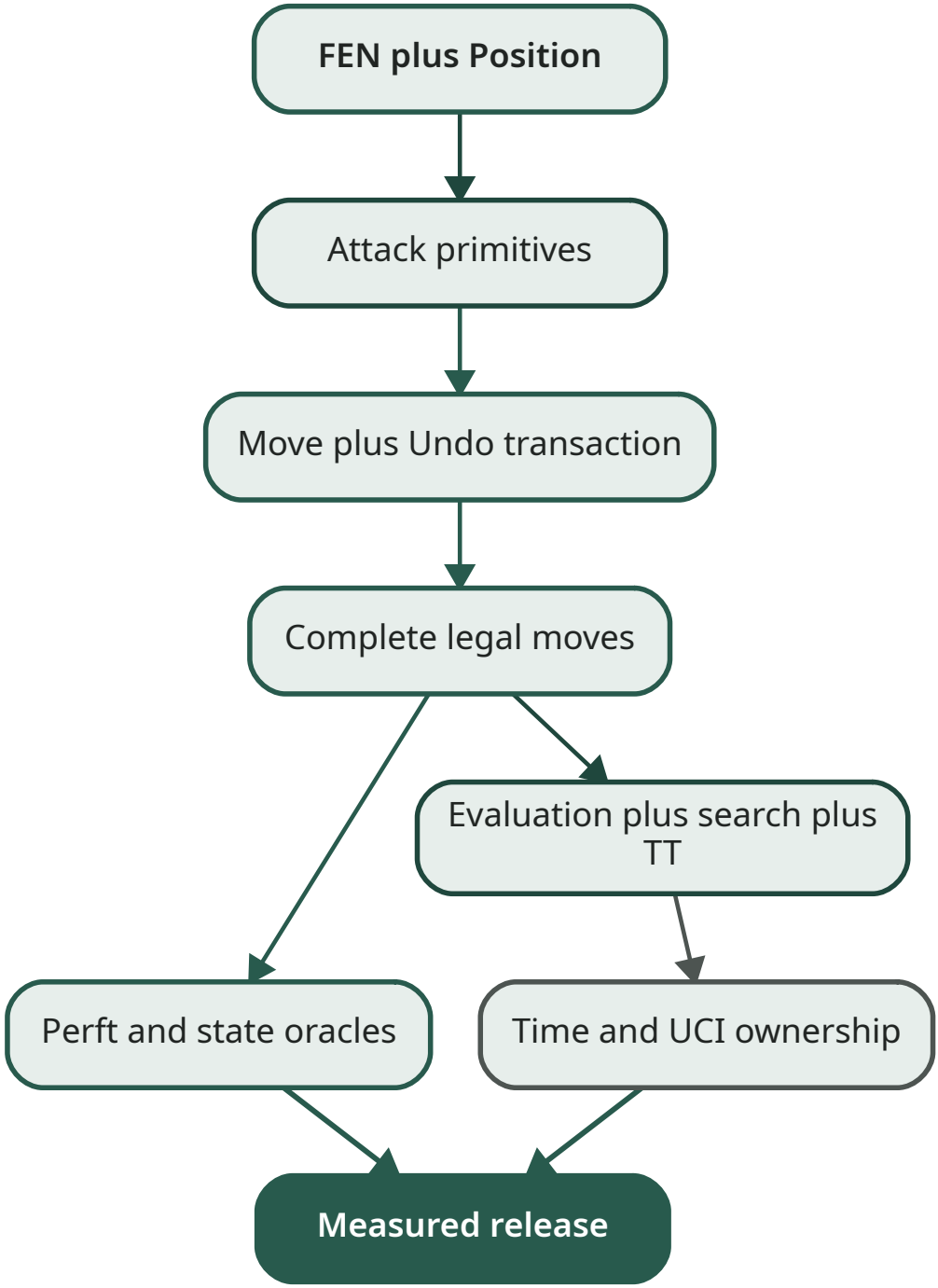
You began with sixty-four squares and a promise: every move would be legal. Now there is a complete program on the other side of that promise. It can read a position, preserve the state that chess rules actually depend on, generate and reverse every move, prove the rules core with perft, search under depth, node, and time limits, evaluate leaves, speak UCI, stop cleanly, and return one legal move. It can benchmark itself and play controlled matches without pretending that one noisy result is a law.

That is a chess engine. It does not need every technique in the literature to earn the name.

What the Engine Can Now Do

Capability	Evidence you retained
Represent and exchange positions	Six-field FEN modes, invariants, and semantic round trips
Play every legal move	Special-move fixtures, differential sets, perft, and full-state restoration
Choose moves deterministically	Bounded evaluation, terminal semantics, negamax, and alpha-beta equivalence
Search usefully	Completed iterations, a PV, staged ordering, check-aware qsearch, and safe TT reuse
Play under real controls	Fake-clock properties, a legal fallback, responsive stop, and one `bestmove` owner
Measure a change	Fixed-work benchmarks, paired openings, intervals, declared hypotheses, and saved manifests
Ship a baseline	Known feature profile, version and binary metadata, license inventory, package, and GUI smoke test

The architecture is now a chain of proved boundaries. No later layer is allowed to reinterpret the state owned below it.



Rules own truth, search consumes it, protocol owns process lifetime, and the release gate joins deterministic correctness with measured behavior.

Three Honest Definitions of Done

Choose the definition that matches why you started.

Done as a project book. The required baseline passes; the engine plays complete timed games in a GUI; a clean release build and its configuration are archived. You can explain every module and reproduce the correctness suite.

Done as a personal engine. Add the features that make it yours: a particular evaluation style, a cautious selective-search stack, an opening repertoire, or a small NNUE experiment. Keep the last known-good release runnable while you explore.

Done as a research platform. The important product is not a rating but a fast experimental loop: one hypothesis, one switch, one baseline, one manifest, and a result that can be confirmed or rejected. Failed ideas are useful when the protocol makes the failure legible.

You may move from one definition to another later. Do not quietly replace the one you chose with an endless list of features.

Pick the Next Experiment, Not the Next Fashion

Start from a symptom you can measure. If fixed-work profiles say legality filtering dominates, compare an optimized legal generator with the trusted make-test-unmake oracle. If the search spends too much time on late quiet moves, instrument LMR rather than copying a table of reductions. If evaluation errors repeat around passed pawns, add a bounded feature and fit it against a held-out set. If one thread is stable and the shipping machine has idle cores, measure a conservative Lazy-SMP design.

Write the experiment before writing the patch:

```
hypothesis:
baseline version and feature switches:
correctness gates:
fixed-work benchmark and positions:
paired-match protocol:
retention / rejection rule:
```

Then accept the answer. A patch that loses, does nothing, or cannot be distinguished from noise is not a personal failure. It is the engine giving you information before the feature becomes permanent complexity.

Maintain the Invariants

As the program grows, keep the small set of oracles that made the first version trustworthy:

- every named FEN and perft count comes from one manifest;
- every make has its matching `Undo`, and full logical state returns exactly;
- the incremental key equals a from-scratch key;
- the scalar/reference backend remains available for differential tests;
- fixed-depth nonselective search has a known oracle;
- the root always owns a legal fallback and commits completed work only;
- one UCI owner emits one terminal response;
- every clock allocation preserves `0 <= soft <= hard <= usable`;
- feature-off configurations remain buildable;
- release binaries identify their version, options, and hardware tier.

Those checks are not scaffolding to throw away. They are what lets you change a deep, stateful program without becoming afraid of it.

Before calling a version released, confirm one final packet:

- a clean build from the recorded revision and locked dependencies;
- deterministic and production profiles with their selected switches;
- passing unit, state, perft, search, clock, and UCI transcript suites;
- a fixed-work benchmark manifest and the first controlled baseline match;

- binary name, version, platform, CPU tier, compiler, options, and digest;
- license and third-party notice inventory;
- a GUI installation smoke test and a rollback copy of the last known-good build.

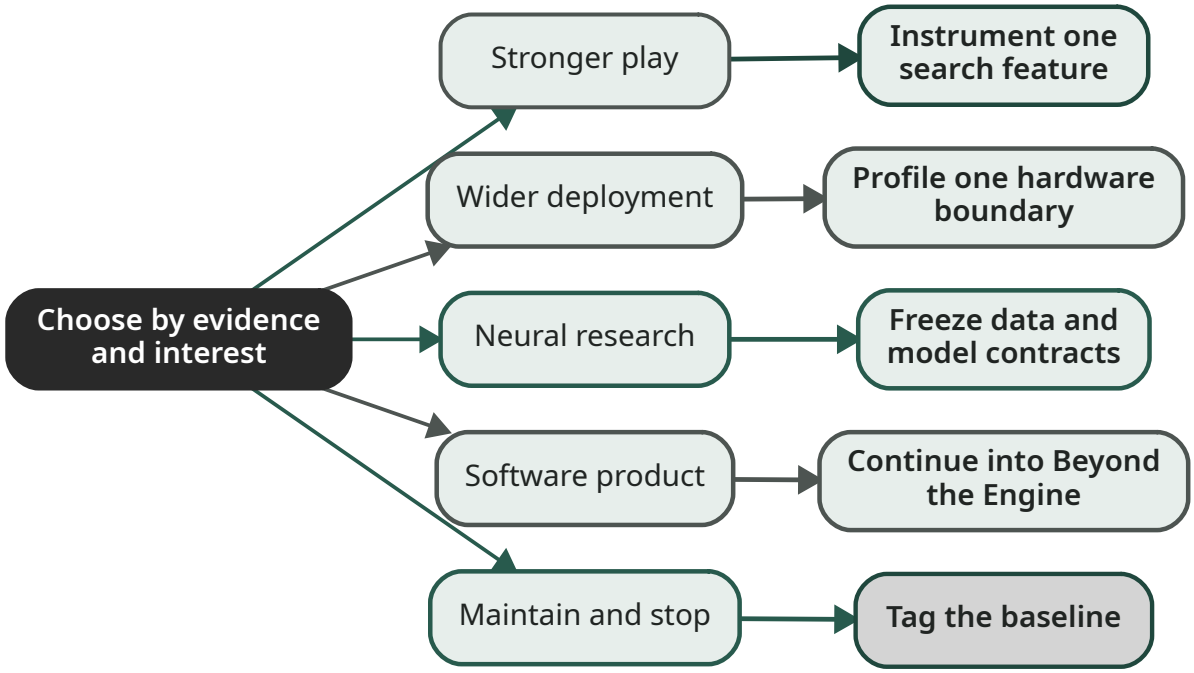
The Computer Science You Practiced

This project is broader than its chessboard. If you describe it in a portfolio, name the work you actually performed rather than claiming a rating the evidence does not support.

Domain	What you implemented or proved
Algorithms	Tree traversal, negamax, alpha-beta bounds, iterative deepening, ordering, and quiescence
Data structures	Indexed board state, attack tables, compact moves, undo records, Zobrist keys, and a TT
Systems programming	Memory layout, incremental updates, clocks, atomics, threads, SIMD dispatch, and profiling boundaries
Software architecture	One-way module dependencies, ownership, immutable snapshots, feature switches, and adapters
Testing	Oracles, property and differential tests, randomized legal walks, golden transcripts, and regression reduction
Statistics	Paired designs, confidence intervals, sequential tests, uncertainty, and predeclared stopping rules
Machine learning	Versioned features and targets, split discipline, NNUE updates, quantization, legal policy masks, and self-play contracts
Data engineering	FEN/PGN schemas, provenance, deterministic manifests, validation, and reproducible experiment records
Interfaces	UCI parsing, atomic replay, flushing, worker/output ownership, and user-facing score conversion
Product engineering	Release profiles, compatibility tiers, packaging, licensing, documentation, and honest capability claims

That is useful language for a README, interview, course reflection, or project report because every line points to a concrete artifact and test.

Choose the Next Branch—or Stop



Interest chooses the branch; evidence chooses what stays. "Maintain and stop" is a complete answer.

Beyond the Searcher

An engine can become the analysis core of many other projects: databases, position tools, opening explorers, puzzle pipelines, annotators, and visual interfaces. Those projects have their own center of gravity—data formats, provenance, accessibility, and product behavior—so they form a separate software-project path rather than delaying the engine you have just finished. The same rule still applies: define the artifact, build the smallest complete vertical slice, and verify the boundary where one layer hands work to another.

The strongest next move is rarely "add everything." It is to keep one engine you can trust, choose one question worth answering, and let measurement decide what survives.

You have built a real engine. It does not need every modern feature to count. The next step should be chosen by evidence and interest, not by guilt.

Continue in Dependency Order

> The full book continues from these same source chapters. The required path reaches a tested, timed, measurable classical UCI release before advanced selective search or neural work becomes an option. The separate *Beyond the Engine* layer then reuses that engine for databases, analysis, puzzles, visualization, and software products.