Engineering a Pluggable Tetris® AI Workbench: Bin-Packing Heuristics, Double DQN with ARC Episodic Replay, and an Honest CPU/GPU Acceleration Study

Shyamal Suhana Chandra
Chief Engineer (Manager)
Sapana Micro Software
Pittsburg, KS 66762, USA

***Abstract*—We present an industry-track experience report on the design, debugging, and empirical evaluation of a pluggable Tetris® artificial-intelligence (AI) workbench implemented as a portable Swift 6 package. The system unifies classical evaluation heuristics (Dellacherie, El-Tetris), multi-ply search (beam search, expectimax, Monte Carlo), an exact offline dynamic-programming solver, a family of agents derived directly from one-dimensional *bin-packing* strategies (First/Best/Worst/Next Fit), and a deep-reinforcement-learning agent: a Double Deep Q-Network (Double DQN) over board afterstates whose experience replay is governed by the Adaptive Replacement Cache (ARC). We report three results of practical interest. First, a single coordinate-enumeration defect in the placement-search routine silently restricted every agent to two of four rotations and roughly the left two-thirds of the playfield, capping all agents at zero cleared lines; isolating and fixing it raised a representative heuristic from 0 to $198+$ cleared lines under an identical budget. Second, framing placement selection as online bin packing yields a Best-Fit agent that is competitive with state-of-the-art hand-tuned heuristics (within 0.5% of El-Tetris over 1000-piece games), while First/Worst/Next Fit reproduce the known ordering from packing theory. Third, a controlled CPU-versus-GPU study shows that for the small, branchy workloads in this domain a vectorized CPU backend (Accelerate/BLAS) dominates a Metal/MPS GPU backend at every batch size we tested (up to $9\times$ over scalar CPU, and $\sim 2\times$ over the GPU even at large batches), so the system defaults to CPU and exposes GPU as an opt-in throttled path. We distill the debugging and acceleration episodes into reusable lessons for shipping game-playing AI.**

***Index Terms*—Tetris®, heuristic search, reinforcement learning, Double DQN, experience replay, ARC cache, bin packing, GPU acceleration, Metal, Accelerate, software engineering.**

I. Introduction

Tetris® is a canonical testbed for sequential decision making: the state space is enormous, the piece stream is stochastic, and optimal play is NP-complete even in the offline setting with full knowledge of the sequence [1]. These properties make Tetris® attractive both for research and as an engineering exercise, because a working Tetris® agent exercises state representation, search, learning, real-time rendering, and performance engineering simultaneously.

This paper is an *experience report*: rather than introducing a single new algorithm, we describe the end-to-end engineering of a workbench in which many agent families share one game core, one placement-search routine, one feature extractor, and one benchmark harness. The shared substrate makes agents directly comparable, but it also means a defect in shared code affects *all* agents at once—a double-edged property that motivates one of our central lessons.

a) Contributions:

- A modular architecture (§III) that hosts heuristic, search, exact-DP, bin-packing, and deep-RL agents behind a single protocol, with a headless benchmark mode and an ncurses terminal UI.
- A family of *bin-packing* agents (§IV-C) that recast placement selection as one-dimensional online packing, and an evaluation showing Best-Fit rivals state-of-the-art heuristics (§VI).
- A Double DQN value agent over afterstates with *ARC-governed episodic replay* (§IV-D), trainable both by self-play and by observing a human player.
- An honest CPU/GPU acceleration study (§VII) with a reproducible micro-benchmark that quantifies when (and whether) a GPU helps.
- A documented root-cause analysis of a shared-code defect that masked all agent performance (§V), and the lessons we draw from it (§IX).

II. Background and Related Work

a) Heuristic controllers: The dominant approach to real-time Tetris® is a linear evaluation of board features applied to the one-step afterstate. Dellacherie's features (landing height, eroded piece cells, row/column transitions, holes, well sums) remain a strong baseline [2]; El-Tetris [3] refines the weight vector. Learning the weights via cross-entropy [4] or evolutionary methods [5] yields agents that clear millions of lines. Approximate dynamic programming over the same features is classical in the neuro-dynamic-programming literature [6].

Industry-track experience report. Artifact: a Swift 6 package (`tetris-ai-tui`) implementing all agents, benchmarks, and the compute-backend harness described herein.

b) Bin packing: One-dimensional bin packing and its simple online heuristics—First Fit, Best Fit, Worst Fit, Next Fit—have well-understood worst-case bounds [7], [8]. Tetris® rows are fixed-width bins and a completed row is a packed bin, making the analogy natural; we make it concrete and measure it.

c) Deep RL and replay: DQN [9] and Double DQN [10] stabilize value learning; prioritized experience replay [11] focuses updates on high-error transitions. We additionally borrow ARC [12] from the storage-systems literature as the *retention* policy for replay memory.

d) Acceleration: Apple's Accelerate framework exposes BLAS/vDSP CPU kernels [13]; Metal Performance Shaders [14] and MLX [15] target the GPU. We evaluate these as interchangeable backends for the only dense kernel in the system.

III. System Architecture

The artifact is a Swift 6 package split into four targets: `TetrisCore` (rules, board, placement search), `TetrisAI` (agents, features, learning, compute backends), `TUI` (ncurses rendering), and `TetrisApp` (the command-line front end). Every agent conforms to a single protocol:

```
protocol TetrisAIAlgorithm: Sendable {
  var name: String { get }
  func chooseMove(for snapshot: BoardSnapshot,
    legalPlacements: [PiecePlacement]) -> AIMove
}
```

A *placement* is a (rotation, column) pair; the shared routine `PlacementSearch.allLegalPlacements` enumerates the full set of legal landings, and `BoardSimulator.apply` produces the resulting afterstate (with cleared lines and a game-over flag) without mutating live state. All agents therefore reason over the same legal-move set and the same afterstate features, which is what makes the benchmark a fair comparison.

A. Board features

The feature extractor computes aggregate height, maximum height, holes, bumpiness, row and column transitions, well sums, and several dispersion metrics. Care with the coordinate system is essential: row 0 is the top of the board and the floor is the largest row index. Column height is therefore the distance from the floor up to the topmost filled cell, and a hole is an empty cell with a filled cell *above* it. As §V recounts, getting these conventions wrong is a recurrent source of silent failure.

IV. Agent Families

a) Shared primitives: Let $W{=}10$ be the board width, $H{=}22$ the visible height, P the number of legal placements per move ($P{=}\Theta(W)$ after the fix of §V, with $P{\approx}34$ empirically), and F the feature-vector dimension ($F{=}\Theta(1)$, $F{\approx}20$ in our extractor). Enumerating placements costs $O(P)$; simulating one placement and extracting features costs $O(WH{+}F)$, which we write $O(WH)$ since F is constant.

A. Knuthian complexity per agent

Table I summarizes per-move asymptotics in Knuth's Θ-notation [16]. Space excludes the board itself ($O(WH)$).

a) Heuristic controllers: Dellacherie and El-Tetris evaluate a fixed linear form over F features for each of P afterstates, yielding $\Theta(P{\cdot}WH)$ per lock. This matches the industry-standard one-ply controller and sits strictly below any multi-ply method in D.

b) Search planners: Beam search expands D plies while retaining at most B hypotheses, so work is $\Theta(D{\cdot}B{\cdot}P{\cdot}WH)$ rather than the naive $O(B^D)$. Expectimax with a 7-outcome piece model incurs $\Theta(7^D{\cdot}P{\cdot}WH)$; in practice $D{\leq}2$ keeps it usable. Monte Carlo trades branching for sampling: $\Theta(R{\cdot}L{\cdot}P)$ with variance $\Theta(1/R)$ on the rollout mean.

c) Offline DP and Demaine's hardness: Demaine et al. [1] show that even *offline* clearing is NP-hard with a known piece sequence; our memoized DP therefore targets only horizon $K{=}\Theta(1)$, giving $O(K{\cdot}P{\cdot}M)$ time and $O(M)$ space rather than polynomial-time optimality in the game length.

d) Bin packing: Each candidate requires $\Theta(W{+}H)$ contact/hole bookkeeping, so all four online fit heuristics are $\Theta(P{\cdot}WH)$ per move—asymptotically identical to Dellacherie, but with different constants and tie-breaking that align with Johnson's First/Best/Worst Fit bounds [7].

e) Double DQN and ARC: Inference batches P afterstates through an MLP of width N, costing $\Theta(P{\cdot}N)$ per move. Training adds $\Theta(\text{batch}{\cdot}N)$ per SGD step. ARC replay [12] maintains signature lists in $O(c)$ space with $O(1)$ amortized insertion and $O(\text{batch})$ prioritized sampling—decoupling *retention* ($O(1)$ touch) from *replay* (prioritized by TD error [11]).

B. Heuristic and search agents

We implement linear-weight controllers (Dellacherie, El-Tetris, and learned weight vectors) and multi-ply planners (beam search, expectimax over the piece distribution, and Monte Carlo rollouts) that share the feature evaluator. A separate exact offline dynamic-programming solver memoizes board fingerprints to compute optimal play over short, known piece horizons, providing a small-scale optimality reference.

C. Bin-packing agents

We treat each row as a width-W bin and select placements with classical packing rules. For each candidate placement we measure a *fit*: the number of exposed piece faces that press against an existing block, a wall, or the floor (more contact = tighter pack), the number of newly created holes (irrecoverable wasted bin space), and the number of completed rows (closed bins). Strategies differ only in how they break ties among waste-free placements:

- **Best Fit:** maximize contact (tightest pack).
- **First Fit:** leftmost waste-free landing (column-major scan).
- **Next Fit:** First Fit resuming from the last column used.
- **Worst Fit:** minimize contact (loosest landing).

TABLE I
PER-MOVE TIME AND AUXILIARY SPACE (KNUTHIAN ANALYSIS). B=BEAM WIDTH, D=SEARCH DEPTH, R=ROLLOUTS, L=ROLLOUT DEPTH, K=DP HORIZON, M=MEMO CAP, N=MLP WIDTH.

Agent	Time	Space	Notes
Random	$\Theta(P)$	$\Theta(1)$	Uniform over P candidates
Greedy height	$\Theta(P)$	$\Theta(1)$	Min landing height
Dellacherie / El-Tetris	$\Theta(P{\cdot}WH)$	$\Theta(1)$	One-ply linear score
Genetic / linear policy	$\Theta(P{\cdot}WH)$	$\Theta(F)$	Fixed weight vector
Bin First/Best/Worst/Next Fit	$\Theta(P{\cdot}WH)$	$\Theta(1)$	Contact scan per candidate
Tactical Strategy (1-ply)	$\Theta(P{\cdot}WH)$	$\Theta(1)$	Split tactical/strategic weights
Beam search	$\Theta(D{\cdot}B{\cdot}P{\cdot}WH)$	$\Theta(B{\cdot}WH)$	Keeps top-B partial plans
Expectimax	$\Theta(7^D{\cdot}P{\cdot}WH)$	$\Theta(7^D)$	7-bag branching; depth capped
Monte Carlo	$\Theta(R{\cdot}L{\cdot}P)$	$\Theta(L)$	Rollout to depth L or game over
Offline DP	$O(K{\cdot}P{\cdot}M)$	$O(M)$	Memo on board fingerprint; K fixed
Double DQN (inference)	$\Theta(P{\cdot}N)$	$\Theta(N)$	Batch forward over P afterstates
Double DQN (train step)	$\Theta(\text{batch}{\cdot}N)$	$O(\text{replay})$	ARC eviction $O(1)$ amortized
ARC replay insert/sample	$O(1)$ / $O(\text{batch})$	$O(c)$	c=signature capacity

All variants prefer closing a full bin (clearing a line) when possible, then prefer waste-free placements, falling back to minimum-waste placements otherwise.

D. Double DQN with ARC episodic replay

The learning agent is a value function V over afterstates, represented by a small multilayer perceptron ($11 \to 48 \to 24 \to 1$, ReLU hidden layers, linear output) implemented from scratch with Adam, so the artifact has no external ML dependency. Action selection is greedy in afterstate value plus the immediate reward; learning is Double DQN [10]: the next afterstate is *selected* with the online network and *evaluated* with a periodically synchronized target network.

a) ARC-governed episodic replay: Transitions are grouped into *episodes* keyed by a quantized afterstate signature and retained by ARC [12], which self-tunes the balance between recency (list T_1) and frequency (list T_2) using ghost lists B_1, B_2 and an adaptive parameter p. A signature is promoted toward the frequent list both when it recurs in play and when it is *replayed* during training, so durable, frequently-useful memories survive eviction. Sampling for stochastic gradient descent remains prioritized by temporal-difference error in the spirit of prioritized replay [11]; ARC decides *what to keep*, priority decides *what to replay*. The agent can be trained by headless self-play or by an active-learning mode that observes a human playing in the terminal UI and learns online from each locked piece.

V. THE SHARED-CODE DEFECT THAT HID ALL PERFORMANCE

During benchmarking, *every* agent—including strong heuristics—cleared zero lines and topped out after roughly 30 pieces, with random play sometimes competitive. Because the symptom was global, the fault was almost certainly in shared code rather than in any agent. A single instrumented game revealed two coupled defects in the legal-placement enumeration:

1) The column range was derived from *absolute* block coordinates, which already included the spawn offset, shifting every placement left by the spawn column and rendering the rightmost columns unreachable. Pieces piled on the left and rows could never be completed.
2) Rotation states 2 and 3 produced negative vertical offsets that failed the spawn-time legality check, so only two of four rotations were ever generated (observed as $\sim$11 candidates instead of $\sim$34).

TABLE II
EFFECT OF THE PLACEMENT-ENUMERATION FIX (EL-TETRIS, SEED 7).

	Before fix	After fix
Candidate placements / move	~ 11	~ 34
Rotations generated	2	4
Reachable columns	left $\sim 2/3$	full width
Lines cleared (500-piece cap)	0	198+
Pieces survived	~ 31	500 (cap)

Both the enumerator and the landing routine shared the faulty logic, so the agents' internal simulations were *self-consistent* with the buggy gameplay—which is precisely why the bug was invisible to the agents and only manifested as uniformly poor outcomes. Rewriting the enumeration to range over shape-relative offsets (and to seed pieces with their topmost block at row 0) restored the full action set. Table II quantifies the effect on a representative agent under an identical budget.

VI. EXPERIMENTAL EVALUATION

A. Gameplay benchmark

Table III reports a headless benchmark over 3 games per agent with a 1000-piece cap (level 1, seed 7). One-ply and learned-weight controllers (Genetic, Linear Policy, El-Tetris, Tactical Strategy) cluster at $\approx$398 average lines with 100% survival. Dellacherie and Bin Best-Fit trail by $\approx$4 lines ($\approx$1%)—confirming that Best Fit matches hand-tuned heuristics. Bin Next/First/Worst Fit, Greedy Height, and Random fail early, reproducing the First/Next/Worst ordering from one-dimensional packing theory.

TABLE III
GAMEPLAY BENCHMARK (3 GAMES, 1000-PIECE CAP, LEVEL 1, SEED 7).

Agent	Tier	Avg. lines	Survival	Best
Genetic Weights	sota	398.7	100%	399
Linear Policy	sota	398.7	100%	399
El-Tetris	sota	398.3	100%	399
Tactical Strategy (1)	sota	398.3	100%	399
Dellacherie	classic	394.7	100%	399
Bin Best-Fit	classic	394.3	100%	398
Bin Next-Fit	classic	15.0	0%	25
Greedy Height	baseline	14.0	0%	21
Bin First-Fit	classic	10.3	0%	15
Random	baseline	0.3	0%	1
Bin Worst-Fit	classic	0.0	0%	0

Multi-ply agents (beam search, expectimax, Monte Carlo, offline DP) are omitted from Table III because their per-move cost (Table I) makes a 1000-piece sweep impractical on a laptop without GPU batching; we report them qualitatively via the Knuthian bounds in §IV-A.

B. Learning curve

Trained by self-play with ϵ-greedy exploration decayed across 12 short games (150-piece cap), the Double DQN agent with ARC replay improves from 0 to a best of 26 cleared lines (average 5.2 lines/game) as exploration decays, confirming the value-learning loop, ARC retention, and reward shaping operate end-to-end. Longer schedules and larger networks are expected to close the gap to the tuned heuristics; the contribution here is the engineering of the pipeline, not a new state of the art in line count.

VII. CPU/GPU ACCELERATION STUDY

The only dense, parallelizable kernel in the system is the value network's batched forward pass; heuristic and search agents are branchy integer logic over a 10×22 board and are not GPU-amenable. We therefore implemented three interchangeable backends for the kernel $Y = \mathrm{ReLU?}(XW^\top + b)$: a scalar CPU loop, an Accelerate/BLAS path (`cblas_sgemm`) [13], and a Metal Performance Shaders path (`MPSMatrixMultiplication`) [14]. A throttle keeps batches below a threshold on the CPU, and an `auto` mode micro-benchmarks the available backends once per workload shape and memoizes the winner.

Table IV shows measured time per dense forward (in $=$ out $=$ 48). Accelerate wins at *every* batch size, by up to $9\times$ over scalar CPU and roughly $2\times$ over the GPU even at the largest batch, because GPU dispatch overhead dominates the small kernels and Apple's matrix coprocessor is highly effective for this shape. Crucially, the typical Tetris® decision evaluates only ~ 34 candidate afterstates, deep in the regime where the GPU is counterproductive. The system therefore defaults to CPU and treats the GPU as an opt-in, throttled backend—an evidence-based instance of "accelerate only if it actually accelerates."

TABLE IV
DENSE FORWARD, MS/ITER (in $=$ out $=$ 48). LOWER IS BETTER.

Batch	CPU (scalar)	Accelerate	GPU (MPS)	Fastest
1	0.0010	0.0003	0.5262	Accelerate
34	0.0469	0.0026	0.5543	Accelerate
256	0.2414	0.0162	0.4087	Accelerate
1024	1.0753	0.0602	0.7910	Accelerate
4096	5.3013	0.2153	1.2156	Accelerate
16384	17.2061	1.8431	3.6855	Accelerate

VIII. TERMINAL RENDERING ROBUSTNESS

The ncurses UI initially discarded entire frames—appearing as "pieces not showing"—because the renderer drew at fixed coordinates and the underlying wide-character output throws on any out-of-bounds write. The footer fell on row 24, off-screen on a standard 80×24 terminal, so the exception fired *before* the screen refresh and the whole buffered frame (board and pieces) was dropped. We made the renderer bounds-aware: every cell and string write is clipped to the live window size and never throws, the footer is clamped to fit, and the run loop always refreshes. The fix was verified across 50×120, 80×24, and 20×60 pseudo-terminals.

IX. LESSONS LEARNED

- **Global symptoms implicate shared code.** When every agent fails identically, suspect the common substrate (enumeration, features, simulation) before tuning any single agent.
- **Self-consistent bugs are the most dangerous.** Because the placement enumerator and the landing routine shared the same defect, agents' internal models matched the buggy world and the error was invisible until measured against ground-truth outcomes (cleared lines).
- **Coordinate conventions deserve tests.** Height, hole, and landing-height definitions hinge on which axis points where; encode them once and assert them.
- **Acceleration is an empirical question.** Adopt a GPU only when a controlled benchmark on the real workload shows a win; for small, branchy, or low-batch kernels a vectorized CPU path is frequently faster and simpler.
- **Render defensively.** In a single-buffer terminal UI, one out-of-bounds write can erase an entire frame; clip all output to the live viewport.

X. THREATS TO VALIDITY

Line counts depend on seed, level, and piece budget; we report small-sample averages intended to expose *ordering* and *regressions*, not record scores. The acceleration study is specific to one Apple-silicon class of device and to the small networks used here; larger models or different hardware can shift the CPU/GPU crossover, which is exactly why the artifact ships the benchmark and an `auto` mode rather than a hard-coded choice.

XI. Conclusion

We described the engineering of a unified Tetris® AI workbench spanning heuristic, search, exact-DP, bin-packing, and deep-RL agents. Beyond the agents themselves, the most transferable outcomes are methodological: a shared-substrate architecture that makes agents comparable but concentrates risk, a disciplined root-cause process for a defect that masked all performance, a bin-packing framing that yields a competitive agent almost for free, and an evidence-driven acceleration policy that prefers CPU until a measured GPU win exists. The artifact reproduces every table in this paper.

GLOSSARY

Tetris® Registered trademark of Tetris Holdings, LLC. Used in this paper to refer to the falling-block game; module names such as `TetrisCore` are implementation identifiers only.

ARC Adaptive Replacement Cache [12]; authors Megiddo and Modha (IBM Research). Governs episodic replay *retention* in our Double DQN trainer.

Double DQN Double Deep Q-Network [10]; authors van Hasselt, Guez, and Silver (DeepMind). Decouples action selection from target evaluation.

Knuthian notation Asymptotic complexity in Θ, O, and Ω as in Knuth [16]; we express costs per locked piece unless noted.

One-ply controller Evaluates only the immediate afterstate; time $\Theta(P{\cdot}WH)$ with our feature set.

Offline DP Exact dynamic programming over a fixed horizon K; exponential in K without memoization, NP-hard in the general offline game [1].

ACKNOWLEDGMENTS

We thank the open-source Tetris®-AI community whose heuristic weight vectors and analyses informed the baselines reproduced here.

REFERENCES

[1] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," *International Journal of Computational Geometry & Applications*, vol. 14, no. 1–2, pp. 41–68, 2004, preliminary version: arXiv:cs/0210020.

[2] C. P. Fahey, "Tetris ai," Ph.D. dissertation, 2003, technical report and survey of Tetris evaluation heuristics.

[3] I. E.-A. Islam, "El-tetris: An improvement on pierre dellacherie's algorithm," https://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/, 2011.

[4] I. Szita and A. Lőrincz, "Learning tetris using the noisy cross-entropy method," in *Neural Computation*, vol. 18, no. 12, 2006, pp. 2936–2941.

[5] R. Szmit and M. Szubert, "Genetic and evolutionary optimization of tetris controllers," *Applied Soft Computing*, 2010.

[6] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[7] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[8] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, *Bin Packing Approximation Algorithms: Survey and Classification*. Springer, 2013.

[9] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," pp. 2094–2100, 2016.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *Proc. Int. Conf. on Learning Representations (ICLR)*, 2016.

[12] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. USENIX Conf. on File and Storage Technologies (FAST)*, 2003, pp. 115–130.

[13] Apple Inc., "Accelerate framework (blas, vdsp)," https://developer.apple.com/documentation/accelerate, 2024.

[14] ——, "Metal performance shaders," https://developer.apple.com/documentation/metalperformanceshaders, 2024.

[15] Apple Machine Learning Research, "MLX: An array framework for apple silicon," https://github.com/ml-explore/mlx, 2024.

[16] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1997.