ChatGPT

Azul Solver and Analysis Toolkit – Research Findings

A. Game-Theoretic Foundations

1. **Move-Space Size & Branching Factors:** In two-player base Azul, the branching factor is highest at the start of each round and shrinks toward the end. On the very first turn, there can be on the order of 100 legal moves (each move = choose all tiles of one color from a factory or center **and** pick a pattern-line placement) [1]. Dom Wilson reports "at the start of a round there can be ~100 moves to choose from, maybe 90 and 80 on turns 2 and 3" [1]. This is because initially 5 factories × up to 4 color choices each ≈ 20 tile-pick options, and each such pick can be placed on 5 different pattern lines (or floor line), yielding ~100 combinations. As tiles are taken and pattern lines fill, the branching factor drops sharply – toward round's end there are often <10 moves available [1]. Overall game-tree complexity is enormous: a naïve depth-4 minimax search in Azul can expand hundreds of millions of nodes and exhaust 32GB of RAM [2]. Therefore, **exact** game-theoretic solving (through brute-force) is intractable for full games, though smaller subproblems can be tackled with search and pruning.

2. **Symmetry Reductions and Solved Substates:** No full **solved** strategy for Azul is published (the game remains unsolved). However, researchers have identified symmetries in the *pattern completion* aspect. The mosaic wall (5×5 grid with a fixed color layout) has geometric symmetries: rotations and reflections of the grid can map one tile configuration to another. Considering all 8 symmetrical transformations (dihedral group of the square) can reduce distinct board states by roughly an 8× factor in certain analyses [3] [4]. For example, there are $2^{25} = 33,554,432$ possible ways to fill or leave empty the 25 wall spots; modulo rotations/reflections this reduces to about 4.19 million unique "pictures" [5]. In practice, not all positions get full 8-fold symmetry (some are self-symmetric), but symmetry-breaking can still cut the state space notably [6]. These symmetries could be used in a solver's transposition table to merge equivalent states. Aside from geometric symmetry, Azul's colors can be considered interchangeable *early* in the game – any permutation of color names combined with corresponding permutation of wall columns yields an equivalent scenario. Exploiting such color-symmetry could, in theory, reduce branching by treating moves that differ only by color swap as one (though the fixed wall pattern breaks color symmetry once specific tiles are placed). **Sub-state solving:** Certain endgame scenarios with few tiles can be brute-forced. For instance, with one round left and a small number of tiles, a retrograde search could find optimal play. We found a thesis that computed the *maximum score* possible in Azul by exhaustive search of tilings [7] [8], but that was a single-player optimization (ignoring an adversary). There is no known publicly solved subset of Azul akin to solved endgames in Chess; at best, researchers have used heuristics to evaluate partial states.

3. **Heuristics for Tile Value & Drafting:** Strong Azul AI relies on heuristics to prune and guide search. One common idea is **"move filtering"** to skip obviously poor or redundant moves. For example, a competition-winning minimax agent filtered out "totally meaningless actions" – moves that overfill a

pattern line causing maximum penalties or picking a color that can't legally be placed anywhere [9]. By eliminating such moves from consideration, the effective branching factor was greatly reduced, allowing deeper lookahead. Another proven heuristic is to prioritize moves that complete a row or column on the wall (due to their high immediate and bonus scoring) and de-prioritize moves that dump many tiles to the floor (incurring penalties). Dom Wilson's AI used a simple **value function** evaluating how many points each player would score *if the round ended immediately*, effectively valuing filled pattern lines and penalizing floor tiles [10] [11]. This guides search toward moves that improve one's score potential or reduce the opponent's. For Monte Carlo simulations, **heavy playouts** (using heuristics during rollouts) significantly improve quality [12]. In one academic implementation, *heavy playout* policy preferred moves using the same heuristic as move ordering (e.g. prefer placing tiles in nearly complete rows) instead of random moves [12]. This led to stronger evaluations than light (random) playouts. To shorten search, heuristics also value *flexibility* – e.g. leaving empty spaces that can accommodate any color vs. blocking yourself with duplicated colors in a row. In sum, a combination of **rule-based filters** (avoid illegal or clearly bad moves) and **evaluative heuristics** (score gain, pattern completion potential) is used to prune game trees dramatically without missing good moves [13] [14].

B. Algorithmic Approaches

1. **Algorithm Trade-offs (Minimax vs MCTS vs PNS vs Retrograde):** Traditional minimax with alpha–beta pruning has been applied successfully to Azul, but the large branching factor means it can only search a few ply except in trivial positions. Alpha–beta yields the same optimal result as naive minimax while exploring far fewer nodes – Dom Wilson observed a **400× node reduction** at depth 3 (396k nodes vs only ~9400 with alpha–beta) [15]. With move ordering, alpha–beta pruned even more (~7,373 nodes at depth 3) [15]. In 100 simulated games with 100ms per move, an alpha–beta AI beat a plain minimax AI 69–27 (with 4 draws) [16], highlighting the practical strength gained by pruning. **Monte Carlo Tree Search (MCTS)** is appealing for Azul because of its success in other complex games. MCTS can effectively handle Azul's high branching by sampling likely lines instead of exhaustive search. One team implemented MCTS and reported that with enough iterations it approaches minimax performance [17]. The Xebia AI project used pure MCTS (without explicit opponent modeling) to maximize its own score; after fixing a bug with random tile restocks, their MCTS agent achieved strong play. They found that beyond ~225 simulations per move, the agent's average score exceeded 50 points (a solid human-level score) [18]. At ~300 simulations per move, the MCTS agent's win-rate and scoring stabilized high, and it could finish games in ~0.5 seconds (CPU) [19]. MCTS has the advantage of anytime performance – more iterations yield better play – whereas alpha–beta requires depth increments. **Proof-Number Search (PNS)** and similar algorithms for *solving* game outcomes are less applicable to Azul's scoring (non-binary win/loss outcome) and high randomness (due to tile draws). PNS shines in solving endgames or puzzles (like checkmate problems) by focusing on proving a win or loss, but Azul's goal is maximizing points over several rounds, so PNS would need adaptation (perhaps treating it as a combinatorial game with terminal utilities). We found no literature of PNS on Azul – likely because the state space is too large for full proof search. **Retrograde analysis** (solving from terminal states backward) could be used for final-round scenarios or to build endgame databases. However, Azul's state (including bag contents) is complex, and the branching backward (multiple predecessors for a given state) is large. So far, retrograde analysis has only been used for simplified subproblems (like computing the maximum possible score by tiling the wall optimally [20]). In practice, a mix of *minimax (for a few ply)* and *MCTS (for deeper simulation)* may yield the best results: e.g. use minimax in late-game tactical positions

(where exact calculation of points is feasible) and MCTS or policy-guided search in early game for strategy. Real-world Azul AIs from competitions favored minimax for tactical accuracy – one team tried MCTS and deep Q-learning but ultimately their **minimax with pruning outperformed** the others in tournament play [21] [22].

2. **Transposition Table Collisions (64-bit vs 128-bit):** Using a transposition table (TT) is crucial to avoid re-exploring duplicate states reached via different move orders. Azul's state can be encoded into a hash (e.g. Zobrist hashing for tile distributions and board occupancy). A 64-bit hash is standard and generally **"more than sufficient"** for game engines [23]. The probability of a collision (two different states yielding the same 64-bit key) is extremely low. For example, even storing 10^7–10^8 positions, the birthday paradox gives a collision probability on the order of 0.01% or less with 64 bits [24]. Researchers Hyatt and Cozzie found that in practice *64-bit signatures cause effectively no significant errors* [23]. Using 128-bit hashes would reduce collision probability to near-zero, but at double the memory cost – typically not worth it for board games. Many engines compromise by storing a 64-bit primary key and a secondary 16–32 bit "lock" to verify the entry, achieving collision rates comparable to 128-bit without full 128-bit storage. In Azul's context (state-space likely <10^12 for practical searches), 64-bit TT keys are safe. A discussion on TalkChess notes that with 2^32 states (~4 billion), one might expect only ~1 collision with 64-bit keys in practice [25]. So if our solver explores millions of states, collisions will be vanishingly rare. The bottom line: **64-bit TT hashing is sufficient** [23]; use 128-bit only if memory is abundant or absolute certainty is required (e.g. if solving, where even a rare collision could spoil a proof). If using SQLite to persist states, an indexed 64-bit integer key (or 16-byte UUID for 128-bit) can serve as state ID. Given the minimal gain from 128-bit, we lean toward 64-bit Zobrist keys with collision checks (store full state or partial state info alongside to verify on hit).

3. **Move Ordering Techniques:** Effective move ordering dramatically improves alpha–beta pruning efficiency. For Azul, domain-specific ordering rules help explore promising moves first. One heuristic is **placing higher-value moves first**: moves that complete a wall row or column (scoring bonuses) or secure the last of a color (color completion bonus) should be tried before lower-impact moves. Another rule is to prefer moves that *avoid penalties* – e.g. moves that do not overflow tiles onto the floor. In Dom Wilson's AI, an iterative deepening framework was used to sort moves: at depth=1, each move's evaluated value was recorded, then at depth=2 those moves were expanded in order of the previously assessed value [26]. This "presort" method improved pruning: alpha–beta with move sorting expanded ~7,373 leaves at depth 3 vs 9,401 without sorting [15] (~20% fewer nodes), and yielded a higher win rate in head-to-head play [27]. Generic chess heuristics like the **killer move** heuristic can be applied: if a move causes a beta-cutoff in one branch (e.g. a strong counter-move by the opponent), that move is likely strong in similar contexts and can be tried early in other branches. Similarly, a **history heuristic** could tally which moves often lead to cutoffs and sort by that score. In Azul terms, a "killer move" might be drafting a color that denies the opponent a big score or forces them into a penalty; if such a tactic was effective in one branch, the engine should try similar denial moves first elsewhere. **Action filtering**, as mentioned, is a pre-ordering step: by removing moves that violate certain strategic criteria, we effectively put them last (or not at all) in the move list. For example, one competition agent disallowed placing tiles in a pattern line that already matches the wall color (since those tiles can never be placed on the wall) [28] – this is more filtering (pruning illegal/futile moves) than just ordering, but it greatly streamlines search. In MCTS, move ordering is less explicit, but the **prior policy** from a neural net or heuristic can bias the search to focus on promising moves. In summary, Azul engines benefit from move ordering rules such as: *complete a*

row first, *avoid floor if possible*, *prefer moves setting up future scoring combos*, and standard alpha–beta enhancements like iterative-deepening ordering [29] and killer moves. Empirically, alpha–beta + ordering + pruning was the strongest approach in a 2020 student competition [21] [22].

4. **Monte Carlo Rollout Quality:** The performance of MCTS (or any Monte Carlo method) in Azul depends heavily on the quality and quantity of rollouts. **Plain random rollouts** are fast but often poor in a tactical game like Azul – random play may fill floors needlessly or miss obvious points, providing a noisy value estimate. Even so, random simulations can be run in large numbers. In a .NET implementation, ~300 random MCTS iterations per move (i.e. 300 full game simulations) yielded a strong agent that finishes a game in ~0.5 seconds [19]. That equates to roughly *600 rollouts/sec on a single CPU core*. On a modern laptop, a pure Python implementation will be slower (due to interpreter overhead), but we can accelerate critical parts (Cython/Numba or vectorized numpy) to approach similar magnitudes. With optimizations, a few hundred random rollouts in <200ms seems feasible. However, replacing random policy with a **heuristic policy** improves rollout quality so fewer simulations are needed. The concept of **heavy playouts** uses domain knowledge during simulation: e.g. always place tiles in the row that maximizes immediate points (or minimizes future pain) instead of random placement. A thesis on Azul AI found that heavy (smart) playouts dramatically improved an MCTS player's strength [12]. By using the same heuristic as the move ordering (favor filling lines with matching colors, etc.), the AI got stronger results with the *same number of iterations* compared to light (random) playouts [12] [30]. The trade-off is that heavy playouts take slightly more computation per simulation. With a 200ms budget, one might run perhaps 100 heavy rollouts versus 300 light rollouts – but if each heavy rollout is much more informative, it's worth it. Another approach is incorporating a **neural network** to guide rollouts or evaluate states (as in AlphaZero). A small neural network (e.g. a few fully connected layers or a simple CNN encoding the board) can provide a value estimate or policy prior. Using PyTorch or JAX, we could batch-evaluate many states on the GPU to amortize cost. For instance, KataGo and LeelaZero batch 16 or 32 positions per inference, achieving thousands of evaluations per second on a GPU. For Azul, a "small" network (maybe ~100k parameters) could likely evaluate a state in <1ms on GPU when batched – meaning we could get hundreds of rollouts in 200ms if the GPU is utilized. However, for very small networks, GPU overhead can outweigh benefits unless we batch. Developers note that a single forward pass on a tiny network might be ~5ms on CPU vs 6ms on GPU [31], so it's only when *batching* multiple evaluations that GPU shows advantage [32] [33]. A middle-ground is to use a neural policy to *sample moves* (reducing branching) and do a few random simulations for outcome. The question specifically asks how many rollouts fit in <200ms: Based on the Xebia result, roughly 150–300 random rollouts were done per move in ~0.2–0.5s on a strong CPU [19]. In Python, without optimization, that might be more like 50–100 rollouts in 0.2s. With vectorized logic or C++ backends, we can push this closer to a few hundred. If we introduce a neural network, the number of rollouts might drop (since each simulation calls the network many times). But the idea would be that we need far fewer rollouts for the same accuracy – e.g. 50 smart rollouts might equal 500 random ones. As a concrete benchmark: a thesis measured MCTS performance against a depth-3 minimax – with 500 iterations (heavy playouts), MCTS won ~75% [34] [35]; at 1000 iterations it was >90% winrate. So to reliably outperform a medium-depth search, on the order of 10^3 simulations may be needed if they're random. With better rollouts or neural guidance, a few hundred might suffice. Our toolkit will likely allow toggling rollout mode: **random (fast)** vs **heuristic (medium)** vs **neural (slower)**. For live hints under 0.2s, a combination of heuristic + moderate rollouts should yield good advice.

C. Data & Storage

1. **State Encoding Schemes:** Efficient state representation is key both for speed and for storing a database of positions. Possible approaches include:
2. **Bitboard/Bitset Encodings:** Represent certain components of the state as bit masks. For example, each player's 5×5 wall can be a 25-bit mask (1 for a filled tile, ignoring color since the fixed pattern determines color by position). Pattern lines could be encoded in a few bits each (each line has a color and a count of tiles). The factories and center could be encoded as counts of each color. A researcher on AI StackExchange suggested encoding each factory as either 5 integers (counts per color) or a binary vector [36]. A full Azul state (factories + center + both players' boards + bag) can be packed into roughly a few dozen integers. One Reddit user reported using a ~700-bit binary vector to encode a 4-player Azul state (which includes more factories) [37]. For 2 players, this could be pared down to ~400 bits (~50 bytes). Bit-level encodings allow very fast copying (just copy a 64-bit word, etc.) and hashing, and enable bitwise operations for certain calculations (e.g. checking adjacencies on the wall).
3. **Flat Arrays / Tensor:** A simpler approach is to use fixed-size arrays for each area. For instance, use an array of length 5 (for 5 factories), each containing 5 integers (tile counts per color) – that's 25 numbers. The center can be an array of 5 integers (count of each color in center). Each player's pattern lines can be a 5×2 array: for each line, store [color, count] (using an encoding for "no color" when empty). The wall pattern can be a 5×5 boolean array or a 5×5 color-id array (with 0 for empty, 1–5 for a tile color placed). All together, this might be on the order of 50–100 integers to describe state. This is very manageable. Such arrays can be wrapped in a NumPy structure for fast vectorized operations (e.g. updating counts when simulating a move) – or even flattened into a single NumPy vector to feed a neural net.
4. **Protobuf / Structs:** If we need to send or store states externally (e.g. caching in a database or sending over a network), a binary serialization via **Protocol Buffers** could be defined. Protobuf would produce a compact byte string representing the state (only a few dozen bytes). Its advantage is language interoperability and forward compatibility (we can add fields as game variants are added, etc.). The overhead is minor – serializing perhaps <100 bytes – which is fine for offline storage or occasional web API calls. However, within the Python solver, we wouldn't use Protobuf for every node (too slow); we'd use native representations and only serialize when saving to DB or sending to the UI.

Serialization Speed: Using Python pickle is easy but not optimal (and not secure if untrusted). Instead, we can serialize states as tuples of integers (which SQLite can store efficiently as BLOB or multiple columns). For example, one could compose a 64-bit key: 5 bits for each factory's tile composition, etc. But more transparently, storing a JSON or Proto-encoded state in SQLite is fine for small DBs. The overhead of (de)serializing ~50–100 numbers is negligible (sub-millisecond). If using SQLite, it may be simpler to store each element in a column (normalized schema), but that can bloat the table and indices. We might opt for a single BLOB column of compressed bytes (see below).

Given Python's slowness, we may implement critical state updates in C/C++ or Cython. Dom Wilson's implementation in TypeScript defined a `GameState` class with simple arrays for tile bag, factories, etc., and provided a `smart_clone()` method to copy state for search [38] [39]. We'll likely do similar: maintain our state as a Python class or namedtuple of arrays, and implement a fast clone (maybe using `numpy.copy` or struct copy of the bitboard integer). The RL environment `AzulRL` uses an object-oriented state but under the hood likely tracks counts and placements in arrays (we see files like `environment_state.py` in that project, suggesting a structured approach) [40] [41]. In summary, we aim for **memory-compact, integer-based state encodings**, which enable both quick hashing and fast cloning. A 64-bit Zobrist hash will be maintained for each state for TT lookups.

1. **Position Database Size & Compression:** If we enumerate many states (for opening exploration or training data), we need to store them efficiently. Let's estimate: one Azul state (2 players) can be encoded in ~40–60 bytes (binary). If we compress each state with Zstd, we might achieve around 50% compression or better (because many states share empty spots patterns and have repetitive structure). For instance, **1 million states** × 60 bytes = ~60 MB raw; with Zstd compression (let's say 4:1 compression ratio for repetitive game states), that might be ~15 MB on disk per million states. This is an approximation – actual compression depends on how data varies. If stored uncompressed in SQLite, 1M states of 60 bytes each plus indexing overhead might be on the order of 100 MB. With Zstd (either compressing each entry or compressing whole DB pages), it could drop to a few tens of MB. For perspective, solving Checkers (much larger state space) resulted in databases of many gigabytes; our needs are far smaller. We likely won't enumerate full game trees beyond a few ply due to branching explosion. But we may store, say, all states up to depth 4 from the initial position for opening analysis. If branching averages ~50, that's on the order of 50^4 ≈ 6.25 million states. Storing those might be ~6M * 60 bytes = 360 MB raw; maybe ~100 MB compressed. This is quite manageable on modern disks. Using SQLite with an efficient BLOB column and Zstd compression functions (SQLite has an extension for Zstd, or we compress in Python before insert) could yield ~0.1 GB for a few million states, which is fine. Each additional ply multiplies state count by ~branching factor, so full game (perhaps ~10 rounds * 10 moves each = ~100 ply) is astronomically large – we won't store anywhere near that. Our position DB will more likely be a cache of evaluated states or interesting scenarios, not the entire game tree.

Disk usage per million states: Rough estimate: **~20–30 MB per million states** (compressed). If states are more sparse or we store partial info, that could be even less. If we included some metadata (score, best move, etc.), that adds a few bytes per state. For example, a record might be: [state_hash (8 bytes), encoded_state (~40 bytes), optimal_move (1 byte), value (2 bytes), visits (4 bytes)] – total ~55 bytes. Compressed, maybe ~15–20 bytes average. So ~15–20 MB per million. These are ballpark figures consistent with known results (for example, a Tic-tac-toe DB with 26k states compresses to ~60 KB, ~2.3 bytes/state).

The bottom line: storage is not a limiting factor for reasonable analysis. A mid-game position database with a few million entries can live in a <500MB SQLite file. Zstd compression offers a good balance of speed and ratio – at level 1 it can do 300+ MB/s compression [42], so compressing on the fly is feasible. We will also consider *delta encoding* or storing differences between subsequent states in sequences (though in a random-access DB that's harder).

1. **Schema and DB Performance (Flask + SQLAlchemy + SQLite vs Postgres):** For our prototype, SQLite is convenient – it's embedded, requires no separate service, and can handle moderate read loads well. However, we must consider multi-user and concurrent writes. SQLite locks the entire database per write transaction, so if multiple users (or threads) try to insert or update simultaneously, they'll serialize. For a low-traffic web app this is usually fine, but heavy concurrent self-play simulations could contend. **Schema design:** Using SQLAlchemy (with Flask-Migrate/Alembic for migrations) allows us to develop against SQLite and later switch to PostgreSQL for production. Migrations are typically portable across engines (Alembic generates neutral schema changes). A StackOverflow Q&A confirms that Flask-Migrate definitions will work on Postgres if developed on

SQLite, as long as only standard types are used [43]. We should be mindful of SQLite quirks: e.g. SQLite is lax with types and doesn't enforce foreign keys unless configured. Postgres, by contrast, will enforce schemas strictly (which is beneficial for data integrity) [44].

Best practices for migrations: Keep SQLAlchemy models as the single source of truth, run `flask db migrate` to generate Alembic scripts when models change, and **don't manually edit the SQLite file**. One pitfall: SQLite lacks some ALTER TABLE operations (like dropping a column easily), so Alembic might recreate tables behind the scenes. This is usually fine for small tables but can be slow for huge ones. Thus, plan schema carefully to minimize disruptive changes (e.g. use nullable columns or separate tables for new data to avoid expensive migrations). In development, it's easy to wipe and rebuild the DB, but in production (even if SQLite) we'd apply migrations carefully.

Performance tuning: With SQLite, ensure we create indices on any keys we use to lookup states (likely the hash). SQLite can handle millions of rows if indexed, but queries will be slower than Postgres as data grows. Read-mostly workloads are okay – SQLite can achieve thousands of queries per second on an index. If we find that inadequate, migrating to Postgres on a cloud host (Fly.io offers free Postgres, Render as well) is wise. Postgres handles concurrency far better (multiple connections, row-level locking) and can scale to large datasets with proper indexing. The FMularczyk article notes that while Postgres has more overhead, it *"provides superior performance for multi-user applications and enforces data integrity"* [44]. Another option is to use an in-memory cache (like Redis) for hot data (recent positions) to reduce hitting the DB frequently.

For our use case, likely DB writes happen when we store analysis results or game logs, which are not extremely frequent. We can queue writes (so the web response isn't blocked). If using SQLite, we might run into the "Queued write" issue: if many writes come, they queue behind the lock – but since our writes are small (inserting one state or result at a time), it should be fast. We can also open the DB in WAL (write-ahead log) mode, which allows concurrent reads during writes and generally improves write throughput.

Upgrading to Postgres: We should ensure our code doesn't use SQLite-specific SQL. If we stick to SQLAlchemy Core/ORM, that's handled. One caveat is data types: e.g. SQLite doesn't have a native BOOLEAN or DATETIME type (it stores them as INTEGER or TEXT), but SQLAlchemy abstracts this. When moving to Postgres, run the migrations and test on a copy of the data. Another strategy is to use Postgres from the start in development via Docker, but given simplicity, we can start with SQLite and switch when needed.

Schema design: It might be simplest to have a table `Position` with columns: `hash PRIMARY KEY, state_blob, eval_score, best_move, visits`. The `hash` can be 64-bit int (which SQLite will store as INTEGER) and Postgres as BIGINT. Alternatively, use the state itself as primary key (composite of all state fields), but that's cumbersome; a hash key is fine with minimal collision risk (we can make it UNIQUE and assume no collisions, or use (hash, encoded_state) combined key to absolutely avoid collision ambiguity). With the proper index on hash, lookups and inserts by hash will be O(log N). We might also have a separate table for game records, user info, etc., which is straightforward.

In summary: Use SQLite for now with caution (WAL mode, single-threaded writes), plan to switch to Postgres for multi-user deployment. Flask-SQLAlchemy makes that easy. Use Alembic for migrations; it works across engines as long as you stick to common features. Test migrations on both SQLite and Postgres if possible (there are known edge cases, e.g., SQLite doesn't enforce foreign keys unless PRAGMA enabled).

By following these practices, we mitigate migration issues and ensure good query performance in production.

D. User-Facing Features

1. **AI UI/UX Patterns (PV, Heatmaps, EV Deltas):** Borrowing ideas from other game AI UIs:
2. **Principal Variation (PV) Display:** Show the engine's predicted best sequence of moves (the PV) and outcome. For Azul, this could be a sequence of tile drafts and placements for, say, the next round or two. Chess engines show PVs as move algebra; for Azul, we might list something like "Factory 3: Blue → Row 2; then Opponent takes Red from Center → Row 5; then …". This helps users follow the AI's reasoning. Including the expected score difference or final score at the end of the PV is useful (e.g., "AI expects to lead by +5 after these moves"). We should update the PV in real-time as the user or AI plays moves.
3. **Move Heatmaps / Highlights:** Engines like Leela Zero (Go) and KataGo present a **heatmap overlay** on the board to indicate move quality. In Go, you see colored dots on each intersection – greener/brighter for higher win-rate moves. We can do similar: highlight each factory or center option on the board with a color or intensity reflecting the AI's evaluation. For example, if a particular tile choice is very good, put a bright green glow or a star icon on that factory; if bad, maybe a red tint. KataGo in Sabaki will display a "blob" on each recommended move, size or color scaled by win rate [45]. We can also overlay numbers: e.g., +3 or -5 indicating how many points that move gains or loses relative to the best move (this is the **EV delta**). Poker solvers and Backgammon bots do this: they list every possible action with an EV in terms of final outcome, and highlight the **mistake size** if not choosing the top action. For Azul, we can compute the expected score difference or final margin. So if a move is 2 points worse in expectation, we might label it "-2" in red.
4. **Probability or Score Graphs:** Though Azul isn't strictly win/lose until game end, we could graph the AI's evaluation over the course of the game (similar to how chess websites show evaluation graphs). This gives users a sense of momentum and where mistakes happened. If using simulation, maybe we track win probability (assuming both play optimally thereafter). If using minimax, a score differential can be shown (like "AI thinks it's +8 points ahead").
5. **Interactive Analysis:** Leela-like interfaces allow users to *try out hypothetical moves* and see the AI's response (the engine plays out a variation). We can implement a "Analysis Mode" where the user can drag tiles to a pattern line (a hypothetical move) and the AI will then show the expected counter-move and continuation. Essentially, allow the user to explore "what if I do this?" by consulting the engine. This is akin to hovering a move in Sabaki (Go software) to see the continuation [46].
6. **Comparative Analysis:** For training purposes, we might integrate a feature like KataGo's score analysis: after a game or move, highlight which move was the "turning point" by EV. E.g., mark a move with a big red X if it swung the score by more than some threshold (like a blunder). This helps users learn from mistakes.

In summary, the UI should present the AI's thinking in a user-friendly manner: *highlight best moves on the board* (heatmap), *list top few moves with expected outcomes*, and *explain differences*. We should be careful not to overwhelm with numbers – a common pattern is to show only a few top choices. For example, LeelaChessZero shows the best 3 moves with their evals and percent chances, and the rest are hidden. We can do similarly: maybe display the top 3 moves: "Move A (expected +5 score), Move B (+3), Move C (-1)" and indicate that anything else is suboptimal by >5 points. Also, providing an **option for full analysis** vs **simple**

hint is good: casual users might just want a nudge ("consider taking blue from center"), while advanced users might want the full PV and heatmap.

1. **Visualizing Factories and Boards (Canvas vs SVG vs React):** Azul's components – factories with tiles, a central pool, and player boards – can be rendered in various ways on the web:
2. **SVG:** Using Scalable Vector Graphics is a convenient way to render the board and tiles as shapes. Each tile could be an SVG circle or image, and factories can be circles that contain those. SVG has the benefit of being easily styleable and clickable (each element is part of the DOM, so you can attach events to, say, a tile or a factory circle). For example, an SVG representation might have groups for each factory, with sub-elements for each tile of a color. This would allow highlighting (change fill color or opacity) to indicate recommended moves. SVG is also resolution-independent, so it will look crisp on high-DPI displays. The downside is performance if there are *very* many elements or frequent updates – but Azul has at most on the order of 100 tiles visible, which is trivial for SVG. A Stack Overflow discussion noted SVG is easier for cases like dragging and dropping objects because each element is native and can listen for events [47]. For Azul, we likely don't need hundreds of moving parts, so SVG is a good choice for clarity and ease of development.
3. **Canvas:** The HTML5 canvas is a pixel-based drawing surface. It can redraw the board very fast (great for animations or thousands of objects) but doesn't retain object structure after drawing. Using canvas might yield better raw performance if we were doing complex animations or updating many times per second (like an arcade game). But in our solver UI, updates are infrequent (a user makes a move, AI responds – not 60fps animation). Also, implementing interactive features (hover, click on a specific tile) is more manual with canvas: you'd have to translate mouse coordinates and track what region was clicked. Canvas is ideal if we later incorporate WebGL or custom visual effects. For now, it likely adds unnecessary complexity. As one source put it, canvas trades a retained mode (DOM/SVG) for an immediate mode (just pixels), giving performance at the cost of having to manage interactions yourself [48]. Given Azul's relatively static display, canvas is not needed for performance – but it could be used for a polished look (for example, rendering tile textures or shadows).
4. **React Components (DOM):** We could also build the board with plain HTML elements (e.g. `<div>` or `<img>` for each tile, absolutely positioned). With React, each tile could be a component in state. This might be simplest (no need to learn SVG), but managing many DOM elements and their styling can get messy. React's reconciliation can handle a few hundred elements easily, but if we aren't careful, re-rendering the whole board for every engine update could cause slight lag. Still, with only tens of elements, React is fine. The pitfall is that updating via React (setState => re-render) might be overkill for small changes like "highlight this one factory": one might accidentally re-render every tile. We can mitigate via shouldComponentUpdate or React's diffing (which will likely diff on keys and update only those elements).

Pitfalls and considerations: Using a pure React/DOM approach with dozens of tiles might lead to **layout thrashing** if not done carefully. E.g., if we animate tile movement by changing CSS top/left, too many style recalculations could occur. But frameworks or direct CSS transitions can handle it. **SVG vs Canvas**: One JointJS article summarizes: SVG is easier for rich interactivity, canvas is better for raw performance with many objects [49] [50]. In our context, clarity of implementation is more important. Another pitfall is touch support – ensuring the UI works on mobile (dragging tiles on touch events) might be easier with DOM events than with canvas (where we'd handle touch coordinates).

Recommendation: Use **SVG or a lightweight component library** for the board. We can embed SVG in our React app (React supports SVG elements). Each tile can be an `<circle>` or `<image>` inside an SVG. We can then manipulate attributes (like add a glow filter or change opacity for heatmap effect). This approach avoids heavy canvas math and is scalable. If we encounter performance issues, we can optimize by only updating the parts that change (e.g., use D3.js data joins or React keys to update only relevant elements). Given the small scale, we don't anticipate major issues. We just need to avoid an anti-pattern of "redraw everything all the time" in React if not needed.

Additional pitfalls: Browsers may resize an SVG if the viewBox is not set correctly – ensure the board scales properly. Also, z-index (render order) in SVG matters if we want to highlight a tile on top of others. We must manage layering (maybe draw highlights last or use separate SVG layers). If using canvas, a known pitfall is that it is *resolution-dependent* – on high DPI screens one should scale the canvas or use `window.devicePixelRatio` to avoid blur [51]. We'd have to handle that if we go canvas. With SVG/DOM, the browser handles DPI scaling automatically.

In summary, we lean toward **SVG for rendering factories and wall** because it offers fine-grained control, interactivity and is sufficiently performant for Azul's needs. React can still manage the high-level UI, injecting an SVG element for the board. We will be cautious about too frequent re-rendering; most likely, updates happen on discrete events (user move or AI move), which is fine.

1. **Security & Multi-User (Hosting on Fly.io/Render):** When deploying the solver web app, we must consider both application security and the challenges of concurrent users:
2. **Preventing Abuse of Compute:** An AI solver can be resource-intensive (CPU/GPU). On a multi-user host, we should ensure one user cannot hog the engine. Rate limiting or requiring login for heavy analysis can mitigate denial of service. For example, allow at most X analyses per minute per IP (could be enforced via a simple middleware or using Render's rate limit add-ons). Fly.io allows deploying multiple lightweight VMs – we might dedicate one VM for the engine. We should also consider using a task queue for engine requests so they are serialized or distributed.
3. **Isolation:** Ensure that one user's analysis or data isn't exposed to another. This is mostly standard web app practice: use per-session or per-user IDs for any stored data, check authentication on requests, etc. If we allow users to save games or analysis, apply proper access controls (don't fetch another user's data just by incrementing an ID).
4. **Input Validation:** The API endpoints (especially if we provide a REST endpoint for analysis or a move recommendation) should validate inputs. Malicious inputs could be JSON with unexpected structure or extremely large/fuzzed states. We should implement checks: e.g., ensure a submitted game state has a valid number of tiles (100 tiles total, correct distribution), no illegal configurations (e.g., two tiles in wall same color in a row). This not only prevents crashes but also avoids pathological cases that could hang the solver.
5. **No Code Injection:** If we ever integrate a feature where users can input formulas or scripts (unlikely here), sandbox it. More relevant is making sure to sanitize any data that is rendered in the UI. For instance, if usernames or comments are shown, escape HTML to prevent XSS. In our solver context, perhaps not an immediate concern unless we implement a shared database of user-generated puzzles or notes.
6. **HTTPS and CORS:** Both Fly and Render provide HTTPS by default. We should enforce secure cookies for sessions. If the frontend is served separately from an API, configure CORS properly to only allow our domain. But likely we serve everything from one domain.
7. **Secrets Management:** Use environment variables for any API keys or admin passwords. For example, if we integrate Rollbar or a database URL, Fly/Render allow setting those via dashboard, and they'll be injected securely.

8. **Multi-User Data Consistency:** If multiple people use the solver concurrently, our backend should handle it. The engine itself might be single-threaded (especially if Python GIL). We might use an async queue to handle requests sequentially or spin up multiple processes. A potential risk is one user's request timing out could affect another if not handled. Using an async worker model (like Celery or just spawning a thread) can isolate tasks. On Render, we can horizontally scale if needed (multiple dynos).
9. **Database concurrency:** As mentioned, SQLite with multiple users could become a bottleneck if many write at once. We might mitigate by using Postgres in production, which is designed for concurrency. Fly has LiteFS which can replicate SQLite, but Postgres is simpler for multi-region writes.
10. **Session management:** Use Flask's built-in session or an extension, and ensure session cookies are secure and httpOnly. If deploying globally, consider sticky sessions or a shared session store if scaled out.
11. **Hosting-specific:** Fly.io runs apps in Firecracker VMs – ensure memory and CPU limits are set so the engine doesn't OOM the instance. For example, if a search could use a lot of RAM, we might want to impose an artificial limit (e.g., don't expand beyond X nodes). Render and Fly both have free tiers with limited resources; we may integrate monitoring to avoid exceeding those (like check CPU usage and possibly scale down search depth if too high).
12. **Privacy:** If we allow multiple users, maybe implement a simple user system. Don't store sensitive PII (not needed here, just maybe an email or username). But if we do, hash passwords (Flask's security libs or passlib).

In summary, **the common pitfalls are uncontrolled compute use and data leaks**. We mitigate by rate-limiting engine calls, validating inputs (so someone can't, e.g., feed a 10,000-tile "state" that breaks our logic), and isolating each user's data. Both Fly.io and Render containerize the app, which gives some isolation at the OS level. We should also keep dependencies updated to patch any known vulnerabilities (e.g., Flask releases, etc.). Since our app deals with potentially heavy computation, one approach is to **separate the web frontend from the solver engine** – e.g., the frontend accepts requests and enqueues a job to a worker process that does the search. This way, even if many requests come, they queue rather than crashing the server. Tools like RQ (Redis Queue) or Celery could be used, or simply Python threads with a job queue.

Fly.io allows deploying multiple VM instances easily, but we must ensure state (like our DB) is consistent – using a remote Postgres or their volume for SQLite. Render similarly can use a managed Postgres.

Summarily, our deployment will start simple (one server, SQLite or Postgres, engine integrated) but with careful limits. As load grows, we can scale horizontally and move the engine to a background service if needed. Security-wise, by sticking to Flask's robust defaults and following standard web best practices, we will provide a safe multi-user experience.

E. Performance & Deployment

1. **Profiling & Bottlenecks in Azul Engines:** Profiling early will help us pinpoint slow spots. Likely bottlenecks include:
2. **Move Generation & State Cloning:** Generating the list of all legal moves each turn and applying a move to get a new state is heavy, especially in Python. In Dom's implementation, cloning the game state for each node was essential [39] . This is typically O(state_size) ~ O(100) operations, but when

done millions of times it adds up. We can profile how much time is spent in `generate_moves()` and `apply_move()`. Often this is the #1 hotspot in game solvers. We might mitigate by using object pools or by applying moves and undoing them (to avoid full copies). A command pattern was used in one C++ Azul project for move apply/undo [52] [53]. Python doesn't have cheap undo, so copying or using immutable state with structural sharing could be alternatives.

3. **State Evaluation:** In minimax, evaluating leaf nodes (i.e., computing a heuristic score for a given state) can also consume time if done repeatedly. Azul's scoring involves checking adjacency on the wall and counting bonuses – which is not too bad (bounded board size). Still, doing it millions of times could matter. We can cache evaluations in the transposition table: store not just score bounds but also exact score for terminal states, etc. If we find evaluation is hot, we might implement it in C (since it's arithmetic over 25 wall cells and some pattern counts).
4. **Transposition Table Lookups:** Hash table operations in Python (for TT) are fairly fast, but if we use Python dict with state->value mappings, the hashing of a complex state (especially if it's a tuple of many elements) can be costly. A 64-bit int hash as key would be ideal (then it's just a single integer hash). We should measure collision checking overhead too (comparing states if hash matches). If TT lookups become significant (should be O(1) average, but many millions could stress memory caches), we could consider a custom C extension or use an array as a big TT (like an array of size N for hash->value mapping, which is typical in C/C++ engines). In Python, likely a dict is fine for reasonable node counts (tens of millions might be too slow in pure Python).
5. **Python-Specific Overhead:** Function call overhead, dynamic typing, and garbage collection can all cause slowness. If our profiler (e.g. cProfile) shows a lot of time in Python internals or object allocations, that's a sign to optimize. For example, constructing many short-lived objects (like move objects, state tuples) can pressure the garbage collector. We might mitigate by reusing objects (e.g., have a single `Move` object we mutate with new values instead of creating millions).
6. **Memory Bottleneck:** Storing millions of states in memory (transposition table) can lead to high memory usage. Dom Wilson noted running out of 32GB RAM at depth 4 search [2]. We must decide on a memory limit – e.g., if TT grows beyond, say, a few hundred thousand entries, consider pruning or clearing it per search iteration (common practice in iterative deepening – clear TT for each new root to avoid memory blow-up).
7. **Parallelism:** Python's GIL prevents multi-threaded CPU-bound speedup. If we need to utilize multiple cores, we might use multiprocessing. That introduces overhead of IPC. We could parallelize Monte Carlo simulations (e.g., 4 processes doing rollouts) and combine results. But syncing them might diminish returns for short searches. Still, using all available cores is a consideration if performance is insufficient on one core.

Case study metrics: In Dom's blog, a depth-4 minimax without pruning had to evaluate 396 million leaf nodes and "took minutes and ran out of memory (32GB) before completing" [2] – clearly an extreme scenario. With alpha–beta and sorting, depth 4 was achievable in under the time limit (presumably seconds, since he ran matches with 100ms move limit and reached up to depth 5 in mid-game in some cases) [54] [55]. This tells us our engine must carefully prune and possibly cut off search at some depth/time.

Another empirical source: the Reddit RL project reported an interesting performance challenge – out of 300 possible actions encoded, only ~40 are valid on average [56]. Initially, they brute-forced by picking the highest network output and checking validity, falling back if invalid – which is inefficient. They moved to masking invalid moves before selecting [57]. This highlights that *invalid move checking* can be a bottleneck if done repeatedly without optimization. We will incorporate move validity mask to avoid wasted computation on illegal options, especially in RL or MCTS playouts.

Optimization steps: We will profile using representative scenarios (early round with ~100 moves, mid-game with ~30 moves, end-game with <10). We expect *move generation* to dominate early, *evaluation* to dominate late (since fewer moves but deeper branches). We'll consider using Numba or Cython on the move generator and simulator – e.g., a Cython function to apply a move (remove tiles from factory, update pattern line, update center) could cut Python overhead significantly. Likewise, using numpy for operations like "find all factories that have color X" might batch some work.

In summary, likely **hot spots** are: move gen/apply, state copy, and hashing. We will mitigate by: - Writing critical loops in Cython (e.g., iterate factories and tiles in C instead of Python loop). - Using simpler data structures (e.g., bitmasks for quick legal move checks, as mentioned). - Possibly disabling Python's garbage collector during deep search (since we manage our memory carefully), to avoid GC pauses.

We will document these optimizations in the design doc's risk mitigation (e.g., risk: "combinatorial explosion and slow Python loops" – mitigation: pruning + native code + limiting search depth).

1. **GPU Batch Rollouts (PyTorch, JAX, RLlib):** If we incorporate a neural network for evaluation or policy, leveraging the GPU can massively accelerate computations *provided we batch them*. The typical approach (as seen in AlphaZero implementations) is to collect states that need evaluation from different search threads and evaluate them together. In our context, we could maintain a batch of states during MCTS that need value network evaluation, and run them through a PyTorch model in one go. PyTorch makes this easy – if our input encoding is, say, a tensor of shape (N, features), we can stack 32 states and evaluate in one forward pass. The latency for one state vs 32 states can be similar on a GPU due to parallelism. This yields big throughput gains. For example, an implementation of batched MCTS for robotics reported a **30× speedup** using a GPU simulator with 500 parallel simulations [58]. Even for board games, others have noted ~10× speedups by batching neural net calls vs sequential (the Julia GPU MCTS medium article mentioned ~13× for Gumbel-MCTS with batch processing) [59].

If we use **Ray RLlib**, it can distribute simulations across CPU workers and also utilize a policy model on GPU. RLlib has a concept of *vectorized environments* and *policy server* that can batch inference. But RLlib might be heavyweight for our needs unless we do learning. For interactive play, a simpler custom batching might suffice (e.g., our search code itself aggregates calls).

We should also consider using JAX if we want differentiable or extremely optimized computations. JAX can JIT-compile the rollout policy or evaluation function, possibly giving near-C speed for parts of the simulation on CPU/GPU. However, integrating JAX might be overkill if we only have a small network.

Another angle: If we treat rollout simulation as a vectorized problem (like simulate 16 games in lockstep), we could use numpy or PyTorch to do it. For instance, represent 16 game states as matrices and update them simultaneously for one move – this is complex for Azul due to branching (players may take different moves), but for random rollouts one could randomize moves in bulk. Not straightforward, but possible for playouts if they don't diverge. More practical is running many rollouts in parallel threads or processes. Python threads won't speed up due to GIL, so we'd use `multiprocessing` or `ray` to run, say, 4 rollouts in separate processes. This scales roughly linearly with cores for independent simulations.

The question specifically mentions a 3k series NVIDIA card available – that's powerful for neural nets. If we incorporate a small CNN or MLP to evaluate Azul positions, we can easily batch dozens of positions per

0.1ms on such a GPU (the net likely being tiny relative to say a ResNet for Go). So the key is to keep the GPU busy by sending batches. If the solver is running in a cloud without GPU, we'd stick to CPU.

Using the GPU for pure simulation (without neural nets) is less straightforward. One could theoretically encode game logic in a GPU kernel (some researchers have done this for other games). But development cost is high, and given Azul's branching (if-else logic for picking tiles and placing), a GPU kernel might not gain much unless doing thousands of simulations in parallel (which in real-time play isn't necessary).

Ray RLlib can help if we do reinforcement training: it can run many environments in parallel processes, and utilize vectorized ops for the model. For deployment, we likely won't involve RLlib, but for training an Azul bot via self-play, RLlib is a great choice. We could spawn 8 workers each playing games and training a neural net on a GPU – RLlib handles the communication and gradient aggregation. This might be a longer-term research utility (one of our objectives is training quizzes and exploring strategies, so training an agent could be part of that).

In conclusion, **GPU acceleration** is mainly beneficial if we use a neural network. We anticipate being able to evaluate **hundreds of states in a single 10ms GPU batch**, meaning the engine can explore much broader/deeper if guided by a net (like AlphaZero style). Empirically, AlphaZero Chess did ~80k NN evaluations per second on a TPU; a single 3080 GPU can do tens of thousands for a small net. For us, even 1k evals/sec is plenty. The marginal speed-up from GPU for non-NN tasks is limited, but for NN inference it can be huge (10-50× faster than CPU per eval for larger nets, and ability to batch).

We will design our engine to be **neural-aware**: perhaps implement a hook where it asks a value function for a batch of states. If no GPU, it can default to a simple heuristic evaluation. This way, an advanced user with an RTX 3000 series can plug in a trained model and see stronger, faster analysis.

1. **Docker Best Practices (Alpine vs Slim, Multi-stage, Native Dependencies):** Containerizing the toolkit is important for reproducible deployment. We have options for the base image:

2. Python **Slim (Debian slim)** vs **Alpine:** Alpine Linux images are small (e.g. `python:3.11-alpine` ~56 MB) but can cause **huge pain for scientific Python libraries**. This is because Alpine uses musl libc and many PyPI wheels (NumPy, SciPy, etc.) are built for glibc (manylinux standard). Using Alpine often forces these packages to compile from source, which is slow and may require installing build tools and libraries. In our case, if we use Numba, SciPy, etc., Alpine could drastically increase build times. A developer noted that **using Alpine can make Python Docker builds 50× slower** in some cases [60] [61], and sometimes even result in larger images due to needing compilers and dev libraries. By contrast, `python:3.x-slim` is based on Debian, slightly larger base, but can use precompiled manylinux wheels for Numpy/Scipy, etc. For example, installing `numpy` on Alpine will compile it (unless using experimental musl wheels), whereas on slim it just downloads a wheel (no compile) [62] [63]. The consensus in Python Docker community is *"it's usually not worth the hassle to use Alpine for Python"* [62]. Our project likely will include Numpy, possibly Cython or PyTorch – all of which have wheels for Debian-based images. So we will use a slim image to save time and avoid obscure issues (floating point differences on musl, etc.).

3. **Multi-stage Builds:** We can minimize the final image size by using multi-stage builds. For example, we might use a full `python:3.11-slim` in the build stage to install dependencies (and possibly compile any C extensions), then copy only the needed files to a final lightweight image (which could

even be Alpine or a slim base without dev tools). One approach: use `manylinux2014` Docker image to build any binary wheels for our project, then install those wheels in a clean environment. However, since we're not packaging a library but an app, a simpler method: use one stage to `pip install` with build deps, then in final stage use the same Python base and copy the site-packages. But often, using the same base for build and run (both slim) is fine; multi-stage is more crucial if we had heavy build deps (like SciPy's Fortran libs) that we don't want in final. In our case, PyTorch (if used) will be installed via pip wheel (no build needed), Numba has wheels for slim (no build), Flask pure Python, etc. Cython might compile some code – for that we need `gcc` and Python headers in build stage, but then can omit them in final.

4. **Reproducible Wheels (Numba/Cython):** If we do compile any extensions (say we write some Cython for performance), we should pin the versions and perhaps build wheels that can be reused. One strategy is to use `cibuildwheel` to build wheels for multiple platforms. But internally, for our Docker, we can simply ensure that when we run `pip install`, a wheel is either downloaded or built deterministically. We should lock dependency versions via a `requirements.txt` or better, a `poetry.lock/Pipfile.lock` to ensure reproducibility. For instance, ensure that the version of Numba we use has prebuilt wheels for our Python version (most do). If we had to compile Numba (which includes LLVM), it would be painful on Alpine. Fortunately, PEP 656 musllinux wheels now exist for many packages [64], reducing Alpine issues, but not all packages provide them. We will likely avoid that by sticking to slim.

5. **Alpine vs Slim trade summary:** Slim ~100MB larger image but saves potentially *minutes* in build and avoids pitfalls [61] [62]. Given we are not extremely constrained on image size (a few hundred MB is fine for an AI toolkit), we choose reliability and speed of build. PythonSpeed's guide explicitly warns that Alpine can lead to obscure runtime bugs and slower builds for Python apps [65] [66]. We heed that warning.

6. **Multi-stage example:**

    1. `FROM python:3.11-slim AS build` – install gcc, musl-dev (if needed), etc., then `pip install -r requirements.txt`. This will compile any C extensions.
    2. `FROM python:3.11-slim AS runtime` – copy only needed files from build (e.g., `/usr/local/lib/python3.11/site-packages` and our app code). This way, none of the build tools (gcc, headers) are in the final image. We can also use `--no-cache-dir` with pip to avoid cache files. If some dependencies are heavy, we could even strip symbol tables or remove test data installed by pip to slim down (some use `pip install --no-dev` etc., but typically not needed for production).

7. **Numba/Cython deployment:** If we use Numba, note that it compiles functions at runtime on first use (unless using ahead-of-time compile). We might ensure the first compilation happens during container build or startup (to avoid a runtime slowdown for the first user request). We could call any Numba-decorated function once on start (Numba will cache the machine code). Similarly, if using PyTorch with CUDA, the first CUDA context initialization can take a couple seconds; we might "warm up" the model on startup.

8. **Testing in Docker:** We'll include tests in the build stage, possibly. But for production image, exclude tests to keep slim.

9. **Continuous Integration:** We should ensure that the Docker build is part of CI so any missing dependencies are caught.

10. **Other Docker tips:** Use pinned versions in `pip install` to avoid nondeterminism. Clean up any apt caches (`apt-get clean && rm -rf /var/lib/apt/lists/*`) if we installed system libs, to reduce image size. If using Alpine (we likely won't), one has to install `py3-numpy` etc., but as StackOverflow suggests, it's often better to just use slim to avoid that [67].

Summary: We will favor `python:3.x-slim` as the base. Use multi-stage to remove build deps. Ensure all necessary binary deps (like libopenblas for Numpy) are present – in slim, installing numpy via pip will bring manylinux wheels that include OpenBLAS. If using Flask with mod WSGI, we might need to install that, but likely we'll use Flask's built-in or gunicorn, which pip provides wheels for. Gunicorn can be used to serve on Fly/Render, and we'll ensure to bind to the correct port (Fly uses 8080 by convention).

Using these practices, our Docker image might be ~300MB (mostly due to PyTorch if included). If that's too big, an alternative is **conda-pack** or a smaller base like `debian:bullseye-slim` with specific packages – but the official Python slim is convenient and optimized. Also consider that Render's free tier has limited disk, but 300MB is fine.

We'll document these choices in the design doc and create a Dockerfile that follows these principles (with comments about why not Alpine, referencing known issues). This ensures a reproducible and efficient deployment pipeline.

F. Similar Open-Source Projects & Prior Art

1. **Existing Azul Solvers and Repos:** There have been a few notable implementations of Azul AIs:
2. **Dom Wilson's Azul AI (2023)** – A strong minimax-based AI written in TypeScript [68] [69]. It uses depth-limited minimax with alpha–beta and move ordering. Dom published a blog detailing the architecture and performance [70] [16]. The source code (on GitHub under `domw95/azul-tiles` and `domw95/minimaxer`) shows a clear separation of game logic and AI. Notably, he implemented the game state as a class with clone method [71] and integrated an interactive web UI. **License:** It appears to be an open personal project; the GitHub doesn't specify a license in the snippet, so we must assume **"All rights reserved"** unless stated. We can reference his ideas and results, but direct code reuse might not be allowed if no license.
3. **mgsweet's Monte Carlo AI (2020)** – A project by a student team ("Diamond_Three") that placed 2nd in a competition [72] [73]. It's a Python framework for Azul with a GUI and supports multiple AI agents (naive, MCTS, etc.) [74]. Their best agent used MCTS with some enhancements and placed highly in the tournament [72]. The repository is licensed under GPL-3.0 [75]. **Architectural choices:** They forked a game framework by Michelle Blom [74], so the game model was likely given. They mention using heuristics in future work (referring to a blog on "cmath-school" for pruning heuristics) [76]. Because of GPL, if we want to reuse any code, we'd have to open-source our project under GPL as well – likely undesirable for a toolkit we want more flexibility with. So we'll avoid copying GPL code; instead, we'll implement our own logic, possibly informed by their approach (e.g., how they structured the search). We can, however, run their AI to generate data or compare performance.
4. **Kaiyoo's AI-Agent Azul (2019)** – Another student project (COMP90054 at Univ. of Melbourne) that tried **Minimax, MCTS, and Deep Q-learning** [21]. They ultimately picked minimax with alpha–beta

for the final agent as it performed best [77] [78]. They did interesting work with **action filtering** to reduce branching [79] and had a custom evaluation that "maximizes the gap in points" [80]. The repo is GPL-3.0 as well [81]. **Data model:** likely similar to mgsweet's as they might have had the same base framework (the file structure looks similar). They also provided a PDF report in their repo which could have insights (though behind GitHub login, but snippet suggests they measured performance vs a baseline). We'll glean ideas like the "unnecessary action filtering" from their documentation [82].

5. **Michal Počátko's Master Thesis (2020)** – *"AI for the Board Game Azul"* [83]. This academic thesis (Charles University, Prague) developed an Azul AI in C++ with both minimax and MCTS. It implemented a full MVC game engine and compared the AIs. The thesis includes heavy playout heuristics and results of MCTS iterations vs winrate [35] [34]. It likely used Qt for GUI (given mention of GUI view) and perhaps could be open-sourced. If the thesis code is available (not sure, but maybe via university dspace), it could be a valuable reference. It might not have a declared license, but being academic, it's probably okay to use for inspiration and even code if credit is given (still, we must be cautious without explicit license).
6. **AzulRL by wert23239 (2020)** – An open-source project focusing on reinforcement learning for Azul [84]. It's written in Python and provides an **environment and some RL agents (DQN, policy gradient)**. It's marked as a draft, but importantly it's under MIT License [85]. This is great for us: we can reuse the environment (game logic) rather than reinventing. Skimming its README and code, it defines the game rules clearly [86] [87] and likely has functions for step (apply move) and reward calculation (score difference). If well-written, we could adopt this as the core game model – saving development time and ensuring correctness. We'd need to verify its completeness (since it was draft, maybe not all features implemented). But having an MIT-licensed reference is extremely useful. We should still test it thoroughly against official rules.
7. **BoardGameArena (BGA) Azul implementation** – BGA has an online Azul with AI (for solo play) but that AI is closed source (BGA's AIs are typically simple scripted bots, not public). No direct reuse possible.
8. **Mobile/Official App AIs** – There is an official Azul mobile app with AI opponents (Pro and Master levels). Those are proprietary and likely use a strong minimax or MCTS under the hood, but no info on them is public. We can only glean that strong Azul AI is feasible (since the app AI is quite good according to player reports).
9. **Other student projects and forks:** Searching GitHub reveals a few personal projects (like `tylin30/AI-Minimax-Azul` which might be another fork of the above competition code, since its description is identical [88]). Also an **Amber Gao's project** (found via a Wiki) that tried Greedy vs BFS vs Minimax [89]. These are likely academic and open-source (likely GPL or MIT). We should double-check any license if we peek at their code.

Architectural Decisions & Data Models from these: A common theme is separating the *game state representation* from the *AI algorithm*. Most implemented a class for game state and then different AI classes that use it (minimax, MCTS, etc.). We will do the same. The game rules code in these projects can guide us on tricky parts like scoring logic and legal move generation. For example, verifying that the rule "cannot place a color in a row if that color is already on the wall in that row" is handled – these projects have already encoded that.

From **AzulRL (MIT)**, we see they clearly define the game step by step (the README excerpt essentially outlines the rules) [86] [90]. We can use their approach to ensure our simulation is correct. Since it's MIT, we could even directly use parts of it (with attribution). That saves time and ensures consistency if we want to integrate learning algorithms later (we could plug our solver into their environment for training).

Regarding **license compatibility:** If we incorporate MIT-licensed code (AzulRL), it's permissive and fine to include in our (we'll likely also choose MIT or BSD for our toolkit to encourage wide use). GPL code (like the competition AIs) we will *not* include directly to avoid copyleft obligations. We might, however, mimic some of their heuristics (since algorithms themselves aren't copyrightable, only the code expression is). For example, we can implement our own "action filter" based on their description [82] without copying their code.

We will give credit in documentation to these projects for inspiration: e.g., "Our engine's move generation logic was cross-verified with the AzulRL environment (MIT Licensed) for correctness" or "Our MCTS implementation is informed by techniques used in Diamond_Three's AI (GPL) [76], though implemented from scratch."

1. **Insights from Others (Reuse/Avoid):**
2. **Data Models:** Many projects used object-oriented models (classes for Factory, Tile, PlayerBoard, etc.). That can lead to a lot of small objects and possibly overhead in Python. We might choose a more compact representation (like arrays) as discussed to avoid Python property access overhead every time. But having a clear model can improve code clarity. We might strike a balance: use simple classes or namedtuples for segments of state but avoid deeply nested objects. For instance, Dom's game used nested arrays in a class (e.g. `gameState.factory` is an array of arrays of Tiles) [38] – we can do similar but with perhaps numeric arrays.
3. **Licenses:** We noted AzulRL is MIT (we can reuse), others GPL (avoid direct reuse). The Azul thesis presumably had code attached or described – academic code might not have a formal license, so it defaults to protected. We'll treat it as reference only.
4. **Algorithmic Choices:** Notably, multiple independent implementations concluded *minimax with pruning and good heuristics beats plain MCTS* for Azul within reasonable time [21] [78]. This suggests for exact game-theoretic value or strong play, we should definitely implement a minimax solver. MCTS can still be offered as an alternative (especially with neural guidance, it could surpass depth-limited minimax in some positions by evaluating deeper stochastically). But if we want to *prove* game-theoretic values, a deterministic search (with full lookahead, if possible) is needed. Perhaps small endgames can be solved exactly by retrograde – an idea is to use minimax to create a DB of solved states for final round scenarios. None of the open projects did that due to time, but we could.
5. **UI/UX:** Dom Wilson created a web GUI for Azul (in browser) [91]. We can see a screenshot (the blog shows an image) – it wasn't as pretty as the real game but functional [92]. We might not reuse his front-end code, but it shows it's feasible to have an interactive board. We should note if any projects attempted a web API or mobile app; most were either GUI or console. Our project using Flask will likely produce a web UI that could stand on shoulders of these (maybe borrow some CSS or idea for layout).
6. **Performance:** The competition code and thesis might contain optimizations we can replicate. E.g., the thesis mentions using Command pattern to allow undo (which can avoid state cloning) [53]. In Python, implementing undo might be possible by saving a diff of state changes and applying it backward. We could consider that if copying becomes a bottleneck.
7. **Mistakes to avoid:** We saw the Xebia team initially allowed MCTS to simulate beyond one round with random restocks, which led to non-convergent results [93]. They fixed it by limiting search to the current round [94]. This is a good insight: due to randomness of tile draws, searching beyond the current round deterministically is meaningless without expectation over bag draws. So either we integrate chance nodes or (simpler) limit lookahead to within the round, except perhaps for end-of-game check (if someone is close to finishing a row, that's a game-ending condition to consider). Many AIs effectively treat each round as a separate minimax problem (with some heuristic for future

rounds). We should do similar unless we implement a full expectimax with bag probability – which is complex and potentially huge. So a *round-by-round search* with an evaluation that accounts for future potential might be the pragmatic approach (Dom's evaluation didn't consider future combos beyond the current round's immediate scoring [95]).
8. We also learned from RL projects that handling invalid moves efficiently is crucial (masking out moves in policy) [56]. Our engine should provide a fast move validity mask for any state to facilitate this.

To conclude, we have a **rich set of references**: We will **reuse**: - AzulRL's game logic (MIT) as a foundation, verifying it against rules. - Possibly Dom Wilson's evaluation ideas (point-instant scoring) and iterative deepening approach (not code, but concept). - Competition heuristics like action filters and move ordering strategies. We will **avoid**: - Depending on GPL code; instead, implement our own or use MIT alternatives. - Overly OOP designs that hamper performance. - Known pitfalls (like not accounting for randomness, or building on Alpine images for deployment which cause issues as noted).

Each of these prior projects offers lessons, and by synthesizing the best elements (Minimax+pruning from one, heavy playout MCTS from another, a ready game model from a third), our design document and eventual implementation can be both efficient and feature-rich without reinventing every wheel. We will, of course, cite these sources in our documentation (as we have with footnotes here) to credit their contributions to our understanding.

---

[1] [2] [10] [11] [14] [15] [16] [26] [27] [29] [38] [39] [54] [55] [68] [69] [70] [71] [91] [92] [95] Azul AI · Dom Wilson
https://domwil.co.uk/posts/azul-ai/

[3] [4] [5] [6] [7] [8] [20] Achieving the highest score in Azul
https://theses.liacs.nl/pdf/2022-2023-KooistraS.pdf

[9] [13] [21] [22] [28] [77] [78] [79] [80] [81] [82] GitHub - kaiyoo/AI-agent-Azul-Game-Competition: AI agent game competition - Reinforcement learning (Monte Carlo Tree Search, Deep Q-learning, Minimax)
https://github.com/kaiyoo/AI-agent-Azul-Game-Competition

[12] [30] [34] [35] [52] [53] [83] Thesis title
https://dspace.cuni.cz/bitstream/handle/20.500.11956/127953/130308802.pdf?sequence=1&isAllowed=y

[17] An AI that can play Azul : r/boardgames - Reddit
https://www.reddit.com/r/boardgames/comments/10b1p0h/an_ai_that_can_play_azul/

[18] [19] [93] [94] Writing Board Game AI Bots - The Good, The Bad, And The Ugly | Xebia
https://xebia.com/blog/writing-board-game-ai-bots-the-good-the-bad-and-the-ugly/

[23] [24] Transposition Table - Chessprogramming wiki
https://www.chessprogramming.org/Transposition_Table

[25] Increase `TypeId`'s hash from 64 bits to 128 bits. · Issue #608 - GitHub
https://github.com/rust-lang/compiler-team/issues/608

[31] [33] NN forward passes for MCTS too slow. Advice? - Reddit
https://www.reddit.com/r/reinforcementlearning/comments/hme983/nn_forward_passes_for_mcts_too_slow_advice/

[32] GPU accelerated batch MCTS · Issue #10 · fqjin/2048NN - GitHub
https://github.com/fqjin/2048NN/issues/10

[36] neural networks - How to encode Azul game state as NN input - Artificial Intelligence Stack Exchange
https://ai.stackexchange.com/questions/9375/how-to-encode-azul-game-state-as-nn-input

[37] [56] [57] Stuck in developing successful RL model for Azul Board Game : r/reinforcementlearning
https://www.reddit.com/r/reinforcementlearning/comments/1cweh20/stuck_in_developing_successful_rl_model_for_azul/

[40] [41] [84] [85] [86] [87] [90] GitHub - wert23239/AzulRL: A Reinforcement Learning Azul Bot and Environment
https://github.com/wert23239/AzulRL

[42] Better Compression with Zstandard - Gregory Szorc's Digital Home
https://gregoryszorc.com/blog/2017/03/07/better-compression-with-zstandard/

[43] Will Flask-Migrate migrations work across different database engines?
https://stackoverflow.com/questions/59163140/will-flask-migrate-migrations-work-across-different-database-engines

[44] SQLite to PostgreSQL - what to check before migration
https://www.fmularczyk.pl/posts/2023_06_sqlite_to_postgresql/

[45] [46] Go and Sabaki - Schlink's Docs
https://sts10.github.io/docs/initial-setup/go.html

[47] HTML5 Canvas vs. SVG vs. div - javascript - Stack Overflow
https://stackoverflow.com/questions/5882716/html5-canvas-vs-svg-vs-div

[48] To canvas, or not to canvas, when building browser-based games?
https://gamedev.stackexchange.com/questions/23023/to-canvas-or-not-to-canvas-when-building-browser-based-games

[49] SVG versus Canvas: Which technology to choose and why? - JointJS
https://www.jointjs.com/blog/svg-versus-canvas

[50] When to Use SVG vs. When to Use Canvas - CSS-Tricks
https://css-tricks.com/when-to-use-svg-vs-when-to-use-canvas/

[51] Canvas vs SVG: Choosing the Right Tool for the Job - SitePoint
https://www.sitepoint.com/canvas-vs-svg/

[58] GitHub - arc-l/pmbs: Parallel Monte Carlo Tree Search with Batched Rigid-body Simulations
https://github.com/arc-l/pmbs

[59] A full-GPU Implementation of MCTS in Julia: the key to Gumbel ...
https://medium.com/@guillaume.thopas/an-almost-full-gpu-implementation-of-gumbel-muzero-in-julia-1d64b2ec04ca

[60] [64] [65] [66] Using Alpine can make Python Docker builds 50× slower
https://pythonspeed.com/articles/alpine-docker-python/

[61] [62] [67] Installing numpy on Docker Alpine - python - Stack Overflow
https://stackoverflow.com/questions/33421965/installing-numpy-on-docker-alpine

[63] Docker Best Practices for Python Developers - TestDriven.io
https://testdriven.io/blog/docker-best-practices/

[72] [73] [74] [75] [76] GitHub - mgsweet/Azul-MCTS-AI: An AI agent that using Monte Carlo Tree Search to master the tile-laying game of Azul.
https://github.com/mgsweet/Azul-MCTS-AI

[88] tylin30/AI-Minimax-Azul - GitHub
https://github.com/tylin30/AI-Minimax-Azul

89 Evolution - AmberGaoYuqing/Azul-Game-AI-Agent GitHub Wiki
https://github-wiki-see.page/m/AmberGaoYuqing/Azul-Game-AI-Agent/wiki/Evolution