People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research



University 20 August 1955 – Skikda



Faculty of Science

Department of Computer Science

End-of-Studies Dissertation

Submitted in partial fulfilment of the requirements for the degree of

Academic Master

Field: Computer Science **Specialty: Artificial Intelligence**

THEME

Artificial Intelligence Techniques for Chess Puzzle Generation and Automated Evaluation

Submitted by:

Louahem M'sabah Mohamed Akram

Mokhtari Rezkallah

Supervised by:

Touil Ghassen

Board of Examiners

President: Cheribet Mohamed

Supervisor: Touil Ghassen

Examiner: Layadi Said

Academic year: 2025 / 2026

Declaration of Authenticity

We, the undersigned, Louahem M'sabah Mohamed Akram and Mokhtari Rezkallah, students in the Department of Computer Science at the University 20 August 1955 – Skikda, hereby declare that this dissertation, entitled "Artificial Intelligence Techniques for Chess Puzzle Generation and Automated Evaluation", submitted in partial fulfilment of the requirements for the Academic Master's degree in Computer Science (specialty: Artificial Intelligence), is the product of our own work, carried out under the supervision of Touil Ghassen.

We affirm that the research reported here was conducted by us, and that the ideas, designs, experiments, and results it contains are our own, except where explicit reference is made to the work of others. All sources, whether published or unpublished, that have been used or quoted have been duly acknowledged in the text and listed in the bibliography.

We further declare that this work has not been submitted, in whole or in part, for any other degree, diploma, or qualification at this university or at any other institution. To the best of our knowledge, it contains no material previously written or published by another person, except where due credit has been given within the text.

We are fully aware that plagiarism and any other form of academic dishonesty constitute serious violations of academic integrity, and we accept full responsibility for the content of this dissertation.

Skikda, June 2026

Louahem M'sabah Mohamed Akram Signature: ……

Mokhtari Rezkallah Signature: ……

Acknowledgements

Before all else, we would like to express our sincere gratitude to everyone who contributed, in one way or another, to the completion of this work.

We are especially grateful to our supervisor, Touil Ghassen, for his guidance, his availability, and his valuable advice throughout this project. His insight and his encouragement were essential in bringing this dissertation to completion, and we thank him for the trust he placed in us.

We also extend our sincere thanks to the members of the jury for the honour they do us in agreeing to read and evaluate this work. We are grateful for the time and the attention they devote to it, and we look forward to their remarks and their constructive criticism.

Finally, we owe our deepest gratitude to our families, for their unfailing support, their patience, and their constant encouragement throughout our years of study. Their presence has been a source of strength at every stage of this journey.

To all those who, near or far, helped us to see this work through, we offer our heartfelt thanks.

ملخص

تعد ألغاز الشطرنج أداة أساسية لتنمية المهارة التكتيكية، غير أنّ تجميع مجموعات كبيرة وعالية الجودة منها يدويا لا يتوسّع بالقدر الكافي، كما أن صعوبة لغز جديد لا يمكن معرفتها قبل أن يحاول حلها عدد كبير من اللاعبين. تعالج هذه المذكرة هاتين المشكلتين عبر سلسلة آلية قائمة على محرك شطرنج لتوليد الألغاز التكتيكية وتقييمها. يفصل النظام بين التوليد والتقييم؛ تُستخلَص الوضعيات المرشحة من مباريات حقيقية بصيغة PGN أو تولّد بإجراء تعديلات قانونية صغيرة على وضعيات بذرية، ثم يمرّ كلّ مرشح عبر المراحل نفسها من التحقق والتنقيط. يجمع التحقق بين فحص القانونية، وغربلة سطحية، واختبار تعاودي للحركة الوحيدة يستخدم محرّك Stockfish 16 ببحثٍ ذي مسارين (MultiPV) للتأكد من أنّ الحل إجباري وأفضل بصورة قاطعة عند كلّ نقلة. تُقيّم الألغاز المقبولة من حيث هامش التفرد والطابع غير البديهي والجدّة والتنوع، وتسند إليها صعوبة دون أي سجل حلّ سابق باستخدام مقياس استدلالي شفاف ونموذج تعلم اختياري مبني على ثلاث عشرة سمة. نُفذت السلسلة في تطبيق Tactica، وهو تطبيق وِب مستقل يعمل بالكامل داخل المتصفّح باستخدام React وchess.js ونسخة WebAssembly أحادية الخيط من Stockfish. يدعم التنفيذَ ثمانية وثلاثون اختبارًا آليًا نجحت جميعها، وتشغيل مقاس على قاعدة Lichess Elite. يبلغ متوسط الخطأ المطلق لنموذج الصعوبة المتعلّم نحو 309 نقاط تصنيف ومعامل ارتباط بيرسون نحو 0.72 مع الصعوبة البشرية، متفوقًا بوضوح على المرجعين. الإسهام الرئيس هو إطار عمل قابل لإعادة الإنتاج وقائم بالكامل على المحرّك، يوحّد توليد الألغاز والتحقّق من سلامتها وتفرّدها وتقدير صعوبتها.

الكلمات المفتاحية: توليد ألغاز الشطرنج؛ التقييم الآلي؛ تفرّد الحلّ؛ تقدير الصعوبة؛ محرّك الشطرنج (Stockfish)؛ التوليد الإجرائي للمحتوى.

Résumé

Les problèmes d'échecs constituent un outil essentiel pour l'entraînement tactique, mais la constitution manuelle de grandes collections de qualité ne passe pas à l'échelle, et la difficulté d'un problème nouveau ne peut être connue avant de nombreuses tentatives. Ce mémoire traite ces deux questions par une chaîne automatisée, fondée sur un moteur d'échecs, pour générer et évaluer des problèmes tactiques. Le système sépare la génération de l'évaluation : les positions candidates sont obtenues par extraction de parties réelles (PGN) ou par de petites perturbations légales de positions de départ, puis chaque candidat traverse les mêmes étapes de vérification et de notation. La vérification associe un contrôle de légalité, un filtrage superficiel et un test récursif de coup unique qui pilote le moteur Stockfish 16 avec une recherche à deux variantes (MultiPV), afin de confirmer que la solution est forcée et strictement meilleure à chaque coup. Les problèmes retenus sont notés selon la marge d'unicité, le caractère contre-intuitif, la nouveauté et la diversité, et reçoivent une estimation de difficulté sans historique, par une heuristique transparente et un modèle appris optionnel à treize caractéristiques. La chaîne est implémentée dans Tactica, une application web autonome côté client (React, chess.js et Stockfish en WebAssembly mono-thread), où les problèmes sont générés, résolus et stockés dans le navigateur. L'implémentation est validée par trente-huit tests automatisés, tous réussis, et par une exécution mesurée sur la base Lichess Elite. Le modèle de difficulté appris, entraîné et testé sur de vraies évaluations Lichess, atteint une erreur absolue moyenne d'environ 309 points et une corrélation de Pearson d'environ 0,72 avec la difficulté humaine, surpassant nettement les deux références. La contribution principale est un cadre reproductible, entièrement fondé sur un moteur, unifiant la génération de problèmes, la vérification de validité et d'unicité, et l'estimation de la difficulté sans historique.

Mots-clés : *génération de problèmes d'échecs ; évaluation automatique ; unicité de la solution ; estimation de la difficulté ; moteur d'échecs (Stockfish) ; génération procédurale de contenu.*

Abstract

Chess puzzles are a central tool for developing tactical skill, yet assembling large, high-quality collections by hand does not scale, and the difficulty of a newly created puzzle cannot be known until many players have attempted it. This dissertation addresses both problems through an automated, engine-based pipeline for generating and evaluating tactical chess puzzles. The system keeps generation separate from evaluation: candidate positions are obtained either by extracting them from real games in PGN form or by applying small, legal perturbations to seed positions, and every candidate then passes through the same verification and scoring stages. Verification combines a legality check, a shallow plausibility screen, and a recursive only-move test that drives the Stockfish 16 engine with a two-line (MultiPV) search to confirm that the solution is forced and uniquely best at each move. Accepted puzzles are scored for uniqueness margin, counter-intuitiveness, novelty, and diversity, and are assigned a difficulty rating without any solving history, using a transparent heuristic together with an optional learned model built on thirteen history-free features. The pipeline is implemented as Tactica, a self-contained client-side web application built with React, chess.js, and a single-threaded WebAssembly build of Stockfish, in which puzzles are generated, solved, and stored entirely in the browser. The implementation is backed by thirty-eight automated tests, all passing, and by a measured generation run on the Lichess Elite database. The learned difficulty model, trained and tested on real Lichess puzzle ratings, reaches a mean absolute error of about 309 rating points and a Pearson correlation of about 0.72 with human difficulty, clearly outperforming both baselines. The main contribution is a reproducible, fully engine-based framework that unifies puzzle generation, soundness and uniqueness verification, and history-free difficulty estimation.

Keywords: *chess puzzle generation; automated evaluation; only-move uniqueness; difficulty estimation; chess engine (Stockfish); procedural content generation.*

Table of Contents

General Introduction 1
Chapter 1: State of the Art 4
1. Introduction and Problem Framing 4
1.1 Motivation 5
1.2 Objectives and Structure 5
2. Historical Background and Conceptual Foundations 6
2.1 From Hand Composition to Computer-Assisted Composition 6
2.2 Making Aesthetics Computable: The Chesthetica Programme 6
2.3 Computational Creativity and the DSNS Approach 7
2.4 The Wider Context: Procedural Content Generation 7
3. A Taxonomy of AI Techniques for Puzzle Generation 8
3.0 Position Representation as a Design Decision 8
3.1 Search-Based Generation and Extraction from Games 9
3.2 Evolutionary and Metaheuristic Generation 10
3.3 Computational-Creativity and Knowledge-Based Composition 11
3.4 Deep Generative Models and Reinforcement Learning 11
4. Automatic Evaluation of Generated Puzzles 13
4.1 Verification: Legality, Solubility, Uniqueness 13
4.2 Difficulty Estimation 13
4.3 Aesthetic and Creativity Assessment 14
4.4 System-Level Metrics and Human Studies 14
5. Datasets, Engines, and Shared Infrastructure 15
6. The Generation-Evaluation Loop in Practice 16
7. Comparative Synthesis 17
8. Open Challenges and Positioning of the Thesis 18
9. Conclusion 19
Chapter 2: Problem Analysis and Methodology for an Engine-Based Chess Puzzle Generator 20
2.1 Introduction 20
2.2 Problem Definition 20
2.3 Research Questions 21
2.4 Objectives of the Proposed System 22
2.5 Requirements 22
2.5.1 Functional Requirements 22

2.5.2 Non-Functional Requirements 23
2.6 General Architecture of the Proposed Framework 23
2.7 Scope: Implemented Core, Experimental Extensions, and Future Work 25
2.8 Puzzle Representation 26
2.9 Candidate Generation Strategy 26
2.10 Engine-Based Verification 27
2.11 Puzzle Evaluation Criteria 27
2.12 Difficulty Estimation 28
2.13 Dataset and Tools 29
2.14 Experimental Protocol 29
2.15 Evaluation Metrics 30
2.16 Methodological Limitations 31
2.17 Conclusion 31
Chapter 3: Implementation and Tests 33
3.1 Introduction 33
3.2 Implemented Scope of the Application 33
3.3 Technologies Used 34
3.3.1 React and Vite 34
3.3.2 chess.js 35
3.3.3 Stockfish WASM 35
3.3.4 Local Storage 35
3.4 Application Architecture 35
3.5 Input Data and PGN Processing 36
3.6 Candidate Generation 37
3.6.1 Extraction from PGN 37
3.6.2 Perturbation of Positions 37
3.7 Engine-Based Verification 37
3.7.1 Stockfish Configuration 37
3.7.2 Shallow Filtering 38
3.7.3 MultiPV Uniqueness Test 38
3.8 Puzzle Scoring 39
3.8.1 Uniqueness Margin 39
3.8.2 Counter-Intuitiveness 39
3.8.3 Novelty 39
3.8.4 Diversity 40
3.9 Difficulty Estimation 40

3.10 User Interface 42
3.10.1 Puzzle Library 42
3.10.2 Solving Interface 43
3.10.3 Statistics 43
3.10.4 Generation Report 44
3.10.5 Settings 45
3.11 Storage and Export 46
3.12 Tests 47
3.12.1 Build Test 47
3.12.2 Module Tests 47
3.12.3 Known Puzzle Tests 47
3.12.4 Reproducibility Tests 47
3.12.5 Performance Tests 48
3.13 Results 48
3.14 Limitations 51
3.15 Conclusion 52
General Conclusion 53
Bibliography 57
Appendix A: Test Evidence 63
A.1 How to run the tests 63
A.2 Run environment 63
A.3 Automated test results 64
A.4 Summary 65

List of Figures

Figure 2.1 The generation-evaluation loop, with its four stages of proposal, filtering, scoring, and feedback 24
Figure 2.2 The six modules of the framework and the data that flow between them 24
Figure 3.1 The Tactica processing pipeline: a candidate position is loaded, screened, analysed by the engine, tested for uniqueness, scored, rated, and finally stored and displayed 36
Figure 3.2 The Puzzle Library during a generation run: the filter bar (motif, difficulty band, piece, and solved status), the Upload-PGN and Regenerate controls, the live progress banner, and the grid of generated puzzles below 43
Figure 3.3 Solving an example puzzle (the Immortal Game, Anderssen versus Kieseritzky, London 1851): (a) the puzzle as presented, with the side to move, difficulty, and the hint, reveal, and next controls; (b) after the first correct move, the opponent's reply played out; (c) on completion, with the detected motifs revealed 43
Figure 3.4 The Statistics screen: the puzzles-solved count, accuracy, and current and best streaks, with solving coverage broken down by difficulty band and by motif 44
Figure 3.5 The Generation Report: the acceptance rate and creative yield, the evaluation and difficulty-distribution panels, and the JSON and CSV export buttons (top), together with the full run configuration that produced the run (bottom) 45
Figure 3.6 The Settings screen: the theme toggle, board orientation, generation mode (extraction, perturbation, or both), the engine-depth control, and the clear-puzzles and reset-progress actions 46
Figure 3.7 A Hard puzzle (Black to move): the quiet 1...Bf2+ opens a forced mating sequence, which is why the learned model rates it Hard 51
Figure 3.8 A Medium puzzle (White to move): 1.Qxd4 wins a piece by a short forcing tactic, rated Medium by the model 51

List of Tables

Table 1.1 A four-family taxonomy of AI techniques for chess puzzle generation, with the evaluator that typically drives each family. The families are complementary, not competing (Section 7) 9
Table 1.2 Dimensions of automatic puzzle evaluation, the methods most commonly used, and the principal weakness of each 15
Table 2.1 Functional requirements of the proposed system 22
Table 2.2 Non-functional requirements and their targets 23
Table 2.3 Modules of the framework, their responsibilities, and their inputs and outputs 24
Table 2.4 Scope of the work. Only the core tier is claimed as an implemented contribution 25
Table 2.5 Datasets, engine, and software, with their roles 29
Table 2.6 Experimental protocol: each research question, its experiment, and the comparison 30
Table 2.7 Evaluation metrics, what each measures, and its level 30
Table 3.1 Mapping from Chapter 2 requirements to their Chapter 3 implementation 33
Table 3.2 Technologies used by the application and their roles 34
Table 3.3 Engine and verification configuration (from the application's config module) 37
Table 3.4 Difficulty prediction on 6,000 held-out Lichess puzzles 41
Table 3.5 Band confusion matrix (row = predicted band, column = actual band; 6,000 held-out puzzles) 41
Table 3.6 Automated test results, by suite 48
Table 3.7 Number of analysed positions and outcome 48
Table 3.8 Reasons for rejection (deep-verification stage) 49
Table 3.9 Difficulty distribution of the accepted puzzles 49
Table 3.10 Timing and throughput 49
Table 3.11 Selected accepted puzzles 49
Table A.1 Unit tests — test/unit.test.js (26 of 26 passed) 64
Table A.2 Reconciliation test — test/reconciliation.test.js (1 of 1 passed) 64
Table A.3 Integration test — test/integration.test.js (1 of 1 passed) 64
Table A.4 Acceptance tests — test/acceptance.test.js (3 of 3 passed) 65
Table A.5 Negative / rejection tests — test/negative.test.js (6 of 6 passed). Each asserts the expected rejection code 65
Table A.6 Reproducibility test — test/reproducibility.test.js (1 of 1 passed) 65
Table A.7 Automated test results by suite 65

General Introduction

Chess has been a testing ground for artificial intelligence since the first search-based programs, and it remains one in the era of neural-network engines that play well above the human level. As the engines grew stronger, interest shifted from playing the game to using it, both as a domain for studying search and learning and as a teaching tool. Tactics are central to that teaching. A tactical puzzle, a position with a single forcing solution that the player must calculate, is among the best tools for training calculation and pattern recognition, and the large online platforms now serve such puzzles to millions of players every day. Unlike the slow, strategic side of the game, a tactic has a definite answer. The solver either finds the forced line or does not, the feedback is immediate, and this immediacy is what makes tactical positions so well suited to practice. The supply of good puzzles, however, has not kept up with the demand for them.

Puzzles have traditionally come from two sources. Some are composed by hand; others are taken from real games and then checked, again by hand, for a clean tactical point. Both routes depend on human effort, and neither scales well. A curator working one puzzle at a time cannot match a platform that shows a different puzzle to every user on every visit, nor can hand curation easily be aimed at a particular theme, a particular difficulty, or a gap in a collection. Automation is the obvious answer, but it immediately raises a harder problem: it is not even clear what a generator should produce. Legal positions exist in astronomical numbers, yet almost all of them are useless as puzzles: dead drawn, trivially won, or dull. A good puzzle, meanwhile, must meet several partly conflicting goals at the same time. It must be legal, it must have a forced and unique solution, it must sit at a suitable difficulty, it should teach something, and, for some authors, it should show a measure of beauty or surprise. As the most recent large study of the problem notes, there is still no standard definition of a chess puzzle, which makes qualities such as creativity or beauty hard to measure objectively [1].

This dissertation confines itself to a concrete position. It does not try to settle what beauty or creativity means; it concentrates instead on the properties of a puzzle that can be defined precisely and checked by machine, and it treats the harder, subjective questions as objects of study. Two problems stand at the centre. The first is the automatic generation and verification of sound tactical puzzles: given a source of candidate positions, can a system decide, automatically and reproducibly, whether each one is a legal, soluble, and uniquely solvable tactic, and do so at a useful rate? The second is the estimation of difficulty for a puzzle that has

never been solved. This is the cold-start problem, in which the usual practice of rating a puzzle from many human attempts cannot be applied because no attempts yet exist. Rating systems capture difficulty only indirectly, by watching how many players succeed or fail over time, and that is exactly the information a brand-new puzzle lacks. Difficulty in this sense is a property of human cognition rather than of engine evaluation, which is why it is hard to predict from the position alone. Seen more broadly, the task belongs to procedural content generation and to the classic generate-and-test pattern, with one advantage peculiar to chess: for a tactical puzzle the engine supplies a strong, if engine-relative, check of correctness, since it can decide, within its own search and settings, whether a candidate has a single best move.

Both problems matter, in practice and in principle. In practice, a system that verifies and rates puzzles without human help gives a scalable supply of trustworthy material, and the supply can be aimed at a chosen theme or difficulty. In principle, such a system brings some rigour to an area where strong claims have often gone untested. Uniqueness can be checked directly with an engine, and reproducibility can be enforced by fixing the engine and its search, so the results can be repeated by anyone. The work also commits to real data and plain reporting. The figures come from real games and from real puzzle ratings, not from cases chosen to flatter the method, and where something could not be measured within the limits of the project, the text says so. Throughout, a modest reproducible result was preferred to an impressive one that could not be repeated.

The approach is an engine-based pipeline in which generation and evaluation stay separate. A generator does nothing but propose candidate positions, either extracted from real games or produced by small legal changes to existing positions, and every candidate then passes through the same checks. Verification proceeds in three steps. A legality check comes first; a shallow screen then drops unsuitable positions cheaply; and a recursive only-move test, which runs the Stockfish engine with a two-line search, confirms at each move of the solution that the best move is forced and better than the alternatives by a fixed margin. Positions that pass are scored for uniqueness, counter-intuitiveness, novelty, and diversity. Each accepted puzzle also receives a difficulty estimate that uses no solving history, first from a transparent heuristic and then from a small learned model trained on real puzzle ratings.

The pipeline is implemented as Tactica, a self-contained application that runs entirely in the browser with no server and no account. On their own machine, a user can load games, generate puzzles, solve them, and inspect the evaluation of a whole run. The client-side design, built on a WebAssembly build of the engine, keeps the system reproducible and open, because the same code and the same engine give the same results to anyone who runs them. It also

removes the practical barriers that usually stand between a method and the people who might use it.

Within this frame the work asks a connected set of questions, which Chapter 2 states formally and the later chapters put to the test: whether legality, solubility, and uniqueness can be verified automatically and reproducibly at a useful rate; whether puzzle quality can be captured on measurable axes; whether difficulty can be estimated with no solving history, and with what accuracy and stability; and how sensitive the whole process is to the engine and its settings. The emphasis falls throughout on the evaluator, the part of the system that decides whether a candidate is a puzzle at all and how good it is. Chapter 2 pairs each of these questions with an experiment and a baseline, so that every answer rests on a measurement.

The dissertation is explicit about what is built and what is not. The full version of the wider idea, a generator trained by reinforcement learning with safeguards for novelty and diversity, together with a complete human study of quality, is more than a master's project can responsibly carry out and validate. The implemented core is the engine-based generation-and-evaluation pipeline described above, and it answers the questions of verification, difficulty, and reproducibility. A learned generator, directed generation by theme or difficulty, and a human study appear as experimental extensions, reported as partial where they are not finished. A fully learned generator is left to future work.

The rest of the dissertation is organised as follows. Chapter 1 reviews the state of the art and draws out the open challenges that motivate the work. Chapter 2 turns that review into a methodology: it states the problem, the research questions, the requirements, and the engine-based design, and it marks what is implemented and what belongs to the wider framework. Chapter 3 presents Tactica and reports its results, among them a generation run on real games and a validation of the difficulty model against real puzzle ratings. The general conclusion then gathers the findings, states the contributions and the limitations plainly, and points to the work that follows.

Chapter 1: State of the Art

1. Introduction and Problem Framing

A chess puzzle is a position, or a short sequence of positions, in which one side must find the precise continuation that forces a decisive result. Under this single heading the literature keeps two distinct families, and it frequently conflates them. Tactical puzzles come from real games, extracted at the moment a single best line exists. These are the puzzles solved in large numbers on online platforms, and their canonical pipeline is the Lichess puzzle generator, which scans hundreds of millions of analysed games for blunders and re-analyses the resulting positions with a strong engine [48, 49, 50]. Chess problems, or compositions, are built deliberately. As game positions they are often unrealistic, and they are judged against a codified body of aesthetic convention maintained by the World Federation for Chess Composition [15]. One defining property is common to both families: a unique solution, in the sense that any deviation from the intended line markedly worsens the solving side's position. Beyond that they answer to different success criteria, and a recurring weakness of the existing work is that a technique validated on one family is presented as though it generalised to the other, when the generalisation frequently does not hold.

Automatic generation of either family draws on several research areas: combinatorial search, procedural content generation (PCG) for games [24, 25], computational creativity [16, 19], and, more recently, deep generative modelling [27, 1]. The task is harder than it first appears. Legal chess positions are astronomically numerous, but almost all are useless as puzzles, whether dead drawn, trivially won, illegal, or dull. A generator must therefore navigate an enormous space under several partially conflicting objectives: legality, solubility, uniqueness, an appropriate difficulty, instructional value, and, where relevant, a measure of beauty or surprise. A further complication is that several of these objectives lack any independently agreed definition. In the words of the most recent large-scale study in the area, "a standardized definition of a chess puzzle is absent, making it challenging to objectively measure its creativity or aesthetic qualities" [1]. This missing target is arguably a deeper constraint on the field than algorithmic difficulty alone. Addressing it is a central aim of this dissertation.

Evaluation is therefore the core of the problem, not a downstream step, and it operates at two levels. For an individual puzzle, an automatic procedure must verify legality, confirm that the intended solution wins, and establish uniqueness; it must then estimate difficulty and, where

relevant, aesthetic merit. For the system as a whole, evaluation must establish whether the generator produces output that is novel, diverse, and competitive with human-authored material. Generation and evaluation are tightly coupled. In essentially every modern pipeline the evaluation function doubles as the fitness signal, reward, or filter that drives the generator [24, 1, 28]. That coupling is the organising thesis of this chapter, and it carries a consequence the descriptive literature tends to underplay: any weakness in the evaluator is inherited by the generator, and often amplified. The reinforcement-learning literature studies this mechanism under the name of reward hacking [39, 40, 41], and in chess puzzle generation it appears empirically as entropy collapse and out-of-distribution drift [1, 2].

1.1 Motivation

Three forces motivate the topic, and they are not equally strong. The practical force is the strongest. Online chess has grown rapidly; structured study and problem-solving are among the strongest predictors of chess skill [63]; and platforms need a continuous supply of correctly rated puzzles matched to each solver's strength. Manual composition cannot meet that demand, so automation is an industrial need before it is an academic curiosity [48, 49]. The scientific force is real but more rhetorical. Chess has long been a standard task environment for the study of cognition and problem-solving [62], and moving the goal from playing well to composing well turns it into a measurable testbed for computational creativity [18, 19]. The methodological force is the claim that puzzle generation is a convenient benchmark for creative problem-solving in modern generative models. It is the most recent of the three and the least settled: it rests largely on a single 2025 study and its companion expert review [1, 2], and it is treated here as a promising hypothesis rather than an established result.

1.2 Objectives and Structure

The review has four objectives: (i) to trace the field's evolution; (ii) to organise generation techniques into a taxonomy and subject each family to critical scrutiny; (iii) to describe the evaluation metrics, datasets, and engines that make up the field's shared infrastructure; and (iv) to identify the open challenges that motivate the dissertation. Section 2 sets out the historical and conceptual foundations and places the chess work within computational creativity and PCG. Section 3 presents the four-family taxonomy and weighs the strengths and documented weaknesses of each family. Section 4 treats evaluation, and Section 5 datasets and tools. Section 6 analyses the generation-evaluation loop, Section 7 offers a comparative synthesis,

and Section 8 sets out the open challenges and positions the thesis, whose problem analysis and methodology Chapter 2 develops.

2. Historical Background and Conceptual Foundations

2.1 From Hand Composition to Computer-Assisted Composition

Chess composition is an art form several centuries old. It is governed by conventions (economy of force, absence of duals, purity of mate) that the WFCC codifies today [15]. For most of that history the only computation involved was verificational: solving engines and, later, endgame tablebases let composers confirm soundness and uniqueness. The tablebase tradition is a strand of AI history in its own right. It begins with Thompson's retrograde analysis of endgames [53], continues through space-efficient indexing schemes [54], and arrives at the compressed Syzygy bases now standard in practice [55]. Syzygy itself has no peer-reviewed description and is citable only as software and community documentation [55], a small but telling sign of how much of the field's infrastructure sits outside the formal literature. The verificational phase mattered: it removed soundness errors, and it freed human effort for the creative core of composition.

2.2 Making Aesthetics Computable: The Chesthetica Programme

The next step was to make aesthetic judgement itself computable. The idea predates the chess-specific work: Margulies proposed empirical "principles of beauty" for chess as early as 1977, in Psychological Reports, not, as it is sometimes miscited, in a popular magazine [14]. From the mid-2000s, Iqbal and collaborators formalised classical themes such as the fork, the pin, and zugzwang, and they assigned these themes weighted values grounded in established chess metrics rather than arbitrary scores [9, 10]. The model was embodied in the program Chesthetica and validated against the judgements of human experts across roughly twelve thousand combinations; the validation reported statistically significant differences between compositions and game positions, with the compositions scoring higher [10].

The contribution of this line of work is real: for the first time, a generator had an objective function for beauty that it could optimise. Its evidential standing must nevertheless be stated with care. First, the model operationalises beauty as a weighted count of recognised themes, and whether that measures aesthetic value or merely tallies motifs remains an unresolved question; the evaluation of computational creativity is itself recognised as difficult and has dedicated evaluation frameworks of its own [14, 60, 61, 77]. Second, and more material for a

literature review, the entire programme is concentrated in a single research group, and its central artefacts have not been independently replicated. The more ambitious creativity claims (Section 2.3) were repeatedly declined by high-impact venues and circulated chiefly as preprints and trade-press articles [11, 12]. Chesthetica is best regarded, then, as a proof of concept for computable aesthetics, not as a settled or externally validated standard.

2.3 Computational Creativity and the DSNS Approach

Chesthetica was later extended from evaluator to composer through the Digital Synaptic Neural Substrate (DSNS). The method extracts numerical attributes from objects in unrelated domains (chess games and problems, but also paintings, computer-evolved art, photographs, and music) and recombines them as strings that seed new problems [11]. The principal reported result deserves attention: a low-quality chess source combined with photographs of people reportedly produced three-move problems of quality comparable to, or better than, those seeded from expert compositions, and it did so more efficiently [11]. A widely cited demonstration was the automatic composition of a rare four-knights-versus-queen problem, hard even for human composers because the win cannot rely on a complete tablebase [12].

The conceptual contribution is substantive. DSNS reframed generation as a problem of machine creativity, moved part of the creative burden from human to machine, and connected chess to the broader computational-creativity tradition [16, 19]. The empirical claim at its heart warrants caution, though. The authors themselves describe the reason cross-domain information helps as "an open question" [11]; the result has not been independently reproduced; and the natural sceptical hypothesis, that the cross-domain string works as an elaborate random seed whose specific provenance is incidental, is not ruled out by the published experiments. Assessed against established frameworks and criteria for attributing creativity to a program [17, 18, 20], DSNS scores well on novelty and on Colton's "imagination" leg of the creative tripod. The value of its output relative to strong human composition, on the other hand, remains weakly evidenced. This is a single-source, unreplicated claim.

2.4 The Wider Context: Procedural Content Generation

Independently of chess, the games-research community developed PCG, the algorithmic creation of levels, maps, rules, and puzzles, and PCG is the natural home for the present problem. Puzzle generation outside chess is an established strand of this work in its own right: solvable Sokoban levels and physics-based puzzle levels, for instance, are produced either by construction or by verifying candidates with a solver [70, 71]. Surveys organise the field around

what content is generated, how it is represented, and how it is evaluated [25, 30, 59]. Two framings are directly relevant here. Search-based PCG explores a content space with an evolutionary or other metaheuristic guided by an evaluation function [24]. PCG via machine learning (PCGML) instead trains generative models (autoencoders, GANs, recurrent and transformer networks) on existing content [26]; recent deep-learning [27] and reinforcement-learning [28] variants extend the paradigm, as do, most recently, LLM-based methods [32, 78, 79]. Standard PCG practice judges a generator on the diversity and expressive range of its output as well as on the quality of individual artefacts [57, 58]. Arguably the strongest single precedent for the dissertation lies outside chess altogether. Browne and Maire's Ludi system evolved complete board games and evaluated them automatically through self-play quality metrics [31]; it ran the generation-plus-evaluation loop a decade before its chess analogues. Framed this way, chess puzzle generation is a specialised and unusually well-constrained instance of PCG (fixed rules, engine-based ground truth, and large human-authored datasets). That makes it close to an ideal laboratory. It also means the field has not consistently held itself to PCG's standards of evaluation and reproducibility [29, 58].

3. A Taxonomy of AI Techniques for Puzzle Generation

The techniques reported in the literature fall into four families, summarised in Table 1.1. The families are not mutually exclusive; the strongest contemporary systems are hybrids that pair a generator with an engine-based evaluator [1, 28]. Before the families are reviewed, one cross-cutting design decision deserves separate treatment.

3.0 Position Representation as a Design Decision

How a position is represented constrains which algorithms apply and which operations (mutation, interpolation, sampling) come naturally. Three representations dominate. Forsyth-Edwards Notation (FEN) is a compact string, the native input for sequence models such as transformers, which consume a tokenised FEN directly [3, 33]. The spatial tensor encodes the board as a stack of binary planes; it suits convolutional networks that exploit two-dimensional locality and is the representation AlphaZero-style systems use [4]. The piece-list, an explicit set of (piece, square) pairs, suits evolutionary methods, since adding, removing, or relocating a piece is then a single well-defined mutation [24]. The three are interconvertible, but in practice they are not equivalent. A generator that emits raw FEN can readily produce positions that are syntactically valid yet illegal (two kings in check, nine pawns), so this representation must be paired with a legality filter; a piece-list makes some constraints trivial by construction

but complicates spatial modelling. Consistent with the PCG literature's treatment of representation [25], representation and evaluation are co-designed: the easier it is for a representation to express degenerate positions, the more work falls to the evaluator.

Table 1.1: A four-family taxonomy of AI techniques for chess puzzle generation, with the evaluator that typically drives each family. The families are complementary, not competing (Section 7).

Family	Core idea	Typical evaluator	Representative work
Search & extraction	Scan real games / search the position space for critical moments	Engine (Stockfish) depth analysis	Lichess Puzzler [48, 50]; engine-based tactic finders
Evolutionary & metaheuristic	Evolve positions under a fitness function over many generations	Engine + heuristic fitness	Search-based PCG [24]; GA puzzle generators
Computational creativity	Recombine cross-domain attributes; emphasise aesthetic conventions	Aesthetics model (Chesthetica)	DSNS [11]; Chesthetica composer [9, 10]
Deep generative & RL	Learn the distribution of puzzles; fine-tune with engine-derived rewards	Engine-search statistics as RL reward	Transformer / diffusion generators; RL fine-tuning [1]

3.1 Search-Based Generation and Extraction from Games

The approach most widely deployed in practice invents no positions at all; it extracts them from games already played. The canonical example is the Lichess Puzzler. It scans hundreds of millions of analysed games, flags the moments where one side blundered, and re-analyses the candidates with a strong engine to confirm a single best line [48, 50]. The defining technical idea is the only-move criterion: every solving move must be uniquely best, which is established by comparing the engine's evaluation of the best move against the second-best, recursively along the principal variation, with an explicit exception for mate-in-one [50]. Practice adds a complementary signal, the behaviour of the evaluation as search depth changes. In a genuine tactic the evaluation shifts sharply as the engine searches deeper; a quiet advantage is already stable at shallow depth [50].

The strengths of this family are reliability, realism, and direct access to human-relevant positions. Its limitations are more substantial than the descriptive literature admits, and three deserve precise statement. First, the method is bounded by what has been played. It cannot produce novel material or rare configurations, and its creative yield is correspondingly low: of the roughly one million puzzles Lichess generates per year, only about 2.1% meet a stringent counter-intuitiveness criterion [1] (reported in that study's Appendix B, Figure 6). Second, the evaluation is not reproducible by default. Multithreaded Stockfish is non-deterministic [51], so the only-move classification can change from run to run unless analysis is restricted to a single

thread, at a substantial cost in speed. Third, the only-move judgement is engine- and threshold-relative. Whether a move is "uniquely best" depends on the engine version, the search depth, and a chosen centipawn cutoff, so the same position can be a valid puzzle under one configuration and not under another. None of these points undermines the approach. Together, they explain why extraction alone cannot carry a research contribution aimed at novelty.

3.2 Evolutionary and Metaheuristic Generation

Evolutionary algorithms attack generation directly in position space. A population of candidates is scored by a fitness function; the weakest are replaced by mutated or recombined copies of the strongest; the cycle repeats. Mutation adds, removes, or relocates pieces, and fitness typically combines engine-verified solubility and uniqueness with heuristic terms that reward difficulty or aesthetics. This is the search-based PCG paradigm [24] instantiated for chess, and it inherits that paradigm's well-studied behaviour. Practical genetic-algorithm generators have coupled the loop to Stockfish and sought out positions where shallow analysis misleads but deep analysis reveals the truth. Because positions are constructed rather than extracted, such generators can produce large volumes of candidate puzzles on commodity hardware. Two genuine advantages set the family apart: it can reach material configurations that essentially never occur in real games, and the same machinery transfers with little modification to chess variants, for which game databases do not exist.

The family's difficulties are structural; they are built into the approach. The central unsolved problem is the design of a fitness function that faithfully captures "puzzle quality", and in the strong sense it is unsolvable as long as the target itself is undefined [1]. A hand-designed fitness encodes the designer's assumptions about what a good puzzle is, and those assumptions are exactly what is in dispute. The most commonly reported failure mode is premature convergence: populations stall after a few generations unless mutation is adapted. This is an instance of the deception problem that motivated novelty search and quality-diversity methods, which are now a standard way to keep a population varied [21, 22, 23, 72, 73, 81]. The output cannot be specified in advance, which complicates debugging and falsification, and evaluation with an engine in the loop is computationally heavy. Finally (a point the broader PCG community treats as standard but the chess literature neglects), there is no shared benchmark for evolutionary puzzle generation, and output is rarely assessed for diversity or expressive range, so reported throughput and quality figures cannot be compared across studies [29, 30, 57, 58, 80].

3.3 Computational-Creativity and Knowledge-Based Composition

The computational-creativity family, exemplified by the Chesthetica composer and DSNS (Section 2), treats generation as the construction of objects that satisfy aesthetic and conventional criteria, not as the discovery of tactically critical positions. An explicit, experimentally tuned model of beauty guides generation, alongside composition conventions that can be emphasised during search. One reported refinement takes the position satisfying the most selected conventions as a seed for further composition, instead of rejecting outright any position that fails a single convention; this was found to improve composing efficiency though not necessarily quality [13].

The conceptual contribution connects the chess work to mainstream computational creativity: a generated puzzle should be judged by creative value in addition to correctness, and cross-domain information can be injected productively [16, 17, 19, 20]. The critical assessment from Section 2.3 carries through here as well. The aesthetic model is single-group and unreplicated; the cross-domain claim lacks independent validation and a ruled-out null hypothesis; and the family's own literature concedes that computer compositions have not consistently reached the level of master human composers, largely because the most refined conventions valued by expert judges are esoteric and resist formalisation [11, 13]. Independent frameworks for evaluating computational creativity call for multi-component, value-oriented criteria rather than a single aggregate score [18, 60, 61, 77]. Measured against them, the family is strong on novelty and on process but weak on demonstrated value, and demonstrated value is the property a thesis on generation must ultimately deliver.

3.4 Deep Generative Models and Reinforcement Learning

The most recent family learns the distribution of well-formed puzzles from data and samples new ones. Its feasibility rests on evidence that chess positions are tractable data for large networks. A 270-million-parameter transformer, trained by supervised learning on roughly fifteen billion engine-annotated data points and using no explicit search at test time, plays at grandmaster strength and solves hard puzzles [3]. The architecture can evidently internalise enough positional structure for non-trivial generalisation. That result concerns playing and solving, however, not composing, and the step from the former to the latter is precisely what remains to be demonstrated. The architectural lineage should be credited explicitly: the transformer [33], the generative-adversarial framework whose mode-collapse

pathology recurs below [34], denoising diffusion [35], and the masked-generative [36] and latent-diffusion [37] variants are the general-purpose machinery these chess systems specialise.

The study most directly relevant to this dissertation benchmarks four such architectures for puzzle creation: an autoregressive transformer, a masked-generative image-transformer variant [36], a latent diffusion model [37], and a masked diffusion model, all trained from scratch on several million puzzles [1]. The finding is instructive and, in part, negative. No single architecture dominates across the quality metrics, and although every generative model beats a baseline of positions sampled from ordinary games, pure supervised generation yields genuinely creative puzzles at a very low absolute rate; the autoregressive transformer produced puzzles that were simultaneously unique and counter-intuitive in only about 0.22% of samples, below even the best Lichess-trained baseline [1] (the supervised result, Section 3.4). To raise this yield, the authors add a reinforcement-learning fine-tuning stage. It optimises a composite reward, computed from engine-search statistics, that rewards uniqueness, counter-intuitiveness, diversity, and realism; counter-intuitiveness is quantified by the critical depth at which the engine first finds the winning idea. The fine-tuning raised the counter-intuitive rate roughly tenfold, to about 2.5%, which exceeds the rate in the human-authored training data itself [1] (the reinforcement-learning result, Section 3.4).

These numbers are among the strongest quantitative evidence in the field, but their provenance bounds how far they can be trusted. They come from a single 2025 study by one industrial group, whose companion expert review reports that the models showed reward-hacking tendencies "such as entropy collapse, producing unrealistic samples, or exploiting chess engine weaknesses," and concedes that "achieving a high reward does not automatically equate to high human-perceived creativity"; final validation rested on a curated booklet judged by only three experts [2]. The failure modes are themselves a contribution. Naïve reward maximisation produces entropy collapse, in which the generator emits one high-reward puzzle repeatedly, and out-of-distribution drift, with unrealistic piece counts that exploit reward loopholes [1, 2]; this is a textbook case of reward hacking [39, 40, 41]. The reported remedies (a replay buffer, novelty filters on board and solution distance, Kullback-Leibler regularisation toward the pretrained model, and explicit piece-count constraints [1]) amount, in PCG terms, to a quality-diversity scaffold [22, 23, 72] applied to a learned generator, and they give this dissertation a practical blueprint. Deep generation, then, brings scale and learned structure, but on present evidence it delivers only a thin, metric-sensitive margin of genuine creativity, and it holds that margin only through careful regularisation against its own optimisation pathologies.

4. Automatic Evaluation of Generated Puzzles

Evaluation is the mechanism that makes generation possible, and it operates at four levels.

4.1 Verification: Legality, Solubility, Uniqueness

An evaluator's first duty is to confirm that a candidate is a valid puzzle. Legality follows from the rules and is straightforward. Solubility, that the intended solution achieves the goal, is established by engine analysis or, for few-piece positions, by an endgame tablebase, which supplies perfect information [53, 54, 55]. Uniqueness is the subtlest requirement, and it is the property that separates a puzzle from an arbitrary position. The standard operationalisation, inherited from the Lichess methodology and reused in recent generative work, recursively compares the best move's winning chances with the second-best's and accepts the move only when the gap exceeds a threshold, with mate-in-one excepted [50, 1]. As Section 3.1 noted, the apparent objectivity of this test is qualified by its dependence on engine version, depth, and threshold, and that caveat should travel with the metric wherever it is used.

4.2 Difficulty Estimation

Matching a puzzle to a solver requires an estimate of its difficulty, and difficulty estimation has become an active subfield. The established approach treats each solving attempt as a rated game and applies an Elo- or Glicko-style rating system, so that the puzzle accrues a rating over many attempts [64, 65]. In the long run this is accurate. For a generator it has a serious weakness, the cold-start problem: a freshly generated puzzle has no solving history, so it would be matched to players essentially at random [45]. History-free predictors have grown out of this problem. Earlier model-based work estimated difficulty by simulating human, knowledge-limited search rather than engine-optimal search [42, 44], and difficulty can be modelled computationally and calibrated against human solving times rather than optimal search [66, 74]. One message recurs across this literature: difficulty is a function of human cognition, not of engine evaluation, and humans make predictable, systematic errors that strong engines do not [42, 43, 46]. The same principle drives adaptive learning systems, in which item difficulty is estimated from learner performance and used to choose what to present next [75, 76]. Recent deep-learning entries, several of them developed for the IEEE BigData 2024 Cup on predicting puzzle difficulty [56], illustrate both the progress and its limits. A CNN-LSTM combines spatial board features with temporal move-and-clock information [47]; a human-inspired architecture predicts the solution and tactical themes as auxiliary tasks and feeds the model's

own uncertainty back into the estimate, and it targets the cold-start case explicitly [45]; transformer entries report competitive accuracy [46]. The same competition data also expose how hard the task is. The Lichess difficulty labels have very large dispersion (mean ≈ 1516, SD ≈ 543), predictions remain unstable at the rating extremes, and the "ground truth" is itself biased by which segment of the online population happens to attempt each puzzle [56]. Difficulty estimation is advancing but far from solved, and its instability is one of the dissertation's concrete openings.

4.3 Aesthetic and Creativity Assessment

Beyond correctness and difficulty, a high-quality puzzle should be elegant or surprising, and two quantitative traditions address this. The older tradition, the Chesthetica computational-aesthetics model (Section 2), scores beauty by detecting formalised themes weighted by chess metrics and validated against human judgement [9, 10]; the replication and construct-validity caveats discussed earlier apply. The newer tradition, which arises from the deep-generative work, decomposes creativity into measurable axes: uniqueness (a single solution), novelty (distance from known positions and solutions, e.g. an edit distance over board and principal variation), and counter-intuitiveness (a correct move that defies shallow evaluation but is vindicated by deeper search) [1]. These axes appeal because they double as evaluation metrics and as components of an RL reward. The same fact is their danger, since a metric that is optimised directly ceases to be a trustworthy measure [40, 41]. Both traditions are in the end proxies for human aesthetic judgement, which even strong experts apply inconsistently [14, 61]. Neither can therefore be a definitive measure, which is consistent with the broader literature on evaluating computational creativity [60].

4.4 System-Level Metrics and Human Studies

A generator must also be judged as a whole. System-level metrics include the yield of valid and creative puzzles per sample, the diversity of the output set, and its novelty relative to the training corpus; these are the quantities the quality-diversity and expressive-range literature was built to measure [21, 22, 23, 57]. Because every automatic metric is in the end a proxy, human expert evaluation remains indispensable, a point emphasised in the computational-creativity evaluation literature [60, 61] and again in the expert review accompanying the most recent chess study [2]. The most encouraging recent evidence is a controlled study in which AI-generated puzzles were compared against human compositions and platform puzzles, and rated players scored realism, difficulty, creativity, enjoyment, and counter-intuitiveness.

Curated AI puzzles were rated more creative and more enjoyable than ordinary platform puzzles, and they approached classic compositions [2]. This is one of the clearer signs that the field has matured. It rests, though, on a small expert panel and on curated outputs, so it should be read as an existence proof rather than a population-level result. Table 1.2 draws the evaluation dimensions together.

Table 1.2: Dimensions of automatic puzzle evaluation, the methods most commonly used, and the principal weakness of each.

Dimension	What it measures	Typical method	Principal caveat
Legality	Position obeys the rules	Rule checker	None of note
Solubility	Solution achieves the goal	Engine search / tablebase	Engine-version dependent for many pieces
Uniqueness	Only-move property along the line	Best vs. second-best gap	Threshold- and depth-relative [1]
Difficulty	Human solving hardness	Rating system; learned predictor	Cold-start; unstable at extremes [45, 56]
Aesthetics	Beauty, thematic richness	Theme-detection model	Single-group, construct-validity disputed [14, 2]
Novelty	Distance from known puzzles	Edit distance on board / solution	Proxy; gameable if optimised [40]
Counter-intuitiveness	Best move defies shallow search	Shallow-vs-deep evaluation gap	Engine- and compute-sensitive
Diversity	Variety across the output set	Pairwise distance; entropy	Collapses under naïve RL [2]

5. Datasets, Engines, and Shared Infrastructure

Progress rests on a small set of shared resources. The dataset nearly every system relies on is the public Lichess puzzle database, an openly licensed collection of several million puzzles. Each entry provides the FEN, the only-move solution sequence, a Glicko-style difficulty rating with its deviation [48, 65], a popularity score, and automatically assigned thematic tags. The puzzles were themselves produced by scanning roughly six hundred million analysed games and re-analysing interesting positions with Stockfish, a process estimated to have required the equivalent of more than a century of CPU time [49]. One caution applies: later snapshots of the dataset quote a smaller figure, so the exact number should be cited to the specific source that reports it. One dataset thus provides the training corpus, the difficulty labels, and the benchmark for novelty and diversity.

On the tool side, Stockfish is the de facto standard for verification, uniqueness checking, and reward computation; modern versions use an efficiently updatable neural-network evaluation [51]. Neural engines are increasingly used alongside it as complementary

references, among them Leela Chess Zero [52] and the human-imitating Maia models [7]. Maia matters in particular because imitating human play yields a notion of surprise better aligned with human perception, and its successors can capture individual stylistic signatures [8]. Endgame tablebases give ground truth for few-piece positions [53, 54, 55]. For dataset construction at scale, the ChessBench resource, ten million games annotated with engine values, illustrates the volume of supervision now available [3]. The broader AlphaZero line [4, 5], and its creative offshoot, the AlphaZero study that generated and assessed new rule-set variants of chess [6], matter less as tools than as evidence that engine-grade systems can be turned from playing toward designing, the same turn this dissertation attempts at puzzle scale. Together, these resources mean that a generation-and-evaluation system can be built almost entirely on open, reproducible foundations. That lowers the barrier to a thesis-scale contribution; it also raises the bar for what counts as novel.

6. The Generation-Evaluation Loop in Practice

Across all four families the recurring architecture is a closed loop: a generator proposes candidates, and an evaluator accepts, rejects, scores, or rewards them. Making the loop explicit clarifies how existing systems are organised and where this dissertation can contribute. The loop runs in four stages. In the proposal stage the generator emits a candidate, whether by a database scan, an evolutionary mutation, a cross-domain recombination, or a sample from a learned distribution; the output is uniformly a position with an intended solution. In the filtering stage, cheap decisive tests come first (legality, piece-count sanity, a rapid shallow engine check), because expensive deep analysis should not be spent on a position that a one-millisecond legality test could reject. In the scoring stage the survivors receive graded assessments: uniqueness margin, estimated difficulty, novelty distance, counter-intuitiveness, aesthetic themes. In the feedback stage the scores return to the generator, as an evolutionary fitness [24], as an RL reward [38], or as an accept/reject decision in a static pipeline.

Two cross-cutting concerns govern the loop's quality, and the wider literature explains both better than the chess work alone does. The first is computational cost. Deep engine analysis is the dominant expense; the reference puzzle database is reported to have required the equivalent of more than a century of CPU time [49]. Practical systems therefore cache evaluations, parallelise, and use shallow proxies where a decisive answer does not require depth. The second concern is the integrity of the feedback signal. Because the generator is optimised against the evaluator, any exploitable weakness in the evaluator is amplified into a systematic distortion of the output. This is the mechanism of reward hacking [39], which has

since been characterised formally [40] and shown empirically to worsen as the optimiser grows more capable [41]. Its chess-specific manifestations, entropy collapse and out-of-distribution drift, are exactly the ones the deep-generative study reports [1, 2]. A loop that withstands this pressure combines multiple, partially redundant criteria, regularises the generator toward realistic output, and validates periodically against human judgement, which no proxy fully replaces [60, 61]. Treating the loop as the central object of study reframes the thesis problem. The contribution is to design a loop whose feedback signal is faithful enough that optimising it yields puzzles that human judges would also rate highly.

7. Comparative Synthesis

Viewed as a whole, the literature follows a clear trajectory: verificational at first, then knowledge-based and creativity-oriented, then search- and evolution-based, and most recently data-driven and learning-based. Comparing the families yields three observations.

First, evaluation and generation have converged. In every modern system the function that judges a puzzle also drives its production, whether as fitness [24], reward [1], or filter [50]. The generator inherits any weakness in the evaluator, so the true bottleneck of the field is the design of evaluation metrics that hold up under optimisation, not generator architecture, and the reward-hacking literature [39, 40, 41] is the relevant body of theory.

Second, the families complement one another, and Table 1.1 reads better as a menu than as a ranking. Search and extraction give realism, at the price of novelty [1]. Evolutionary methods give novelty and support for variants, but they rest on a hand-designed fitness aimed at an undefined target [24]. The computational-creativity methods supply an explicit account of beauty, yet they still lack independent validation and master-level value [11, 13]. Deep generative methods bring scale and learned structure, together with low creative yield and reward-hacking pathologies that only heavy regularisation contains [1]. The strongest reported results come from hybrids, in which a learned generator is constrained and rewarded by engine-based evaluation [1]. This dissertation takes the engine-based evaluator as its implemented core and treats the learned generator as the wider framework it builds toward.

Third, the field has converged on a shared empirical infrastructure: the Lichess dataset, Stockfish, and an increasingly standard vocabulary of metrics [48, 51]. The convergence makes results more comparable and lowers the barrier to entry. It also carries a subtler risk that the field has not confronted. In a monoculture where dataset, engine, and metrics all come from overlapping sources, systematic biases can become invisible, because every system is trained

and judged against the same proxies [29, 40, 58]. A genuinely critical contribution scrutinises that shared infrastructure instead of simply reusing it.

8. Open Challenges and Positioning of the Thesis

Several problems remain open, and together they define the space this dissertation occupies.

- The undefined target. No agreed, operational definition of a "good" or "creative" chess puzzle exists [1, 2]. Every downstream metric and fitness function inherits the ambiguity, so progress on definition comes logically before progress on generation.
- The cold-start difficulty problem. A newly generated puzzle has no solving history, yet it must be rated before it is served. History-free predictors exist, but they remain inaccurate across the rating range and unstable at the extremes [45, 46, 56].
- Imperfect, gameable creativity metrics. Counter-intuitiveness, novelty, and aesthetic scores are proxies for human judgement. They are sensitive to engine version, search budget, and hardware, and a generator that optimises them directly can game them [1, 40, 41].
- Reward hacking and mode collapse. Unless regularised by diversity filters, divergence penalties, and explicit constraints, reinforcement-learning generators tend toward entropy collapse or unrealistic, out-of-distribution output [1, 2, 39].
- Realism versus novelty. Extraction yields realistic but unsurprising puzzles, and free generation yields novelty at the risk of artificiality. Explicit control of this trade-off is an unsolved problem [1].
- Controllability. Generation on demand, with a specified theme, difficulty, and material configuration, has been achieved only in part [1, 32].
- Evaluation standardisation and reproducibility. No agreed benchmark protocol exists, and engine non-determinism and the infrastructural monoculture compound its absence, so cross-study comparison is harder than it should be [29, 30].

Against this background, the dissertation adopts the design this review identifies as best supported: an engine-based generation and evaluation pipeline, with a learned, reinforcement-trained generator as the wider ideal framework. Chapter 2 develops the precise problem analysis, the system requirements, and the proposed methodology, together with an explicit account of what will actually be implemented.

9. Conclusion

The artificial generation and evaluation of chess puzzles has now been surveyed, with an aim that went beyond cataloguing. One theme persists throughout and unifies the rest: generation cannot be separated from evaluation. The metric that judges a puzzle invariably becomes the engine that produces it, and no insight in the literature has proved more useful. The same coupling is also the field's most serious vulnerability. Mature tools and large open datasets now support the principal evaluation requirements (legality, solubility, uniqueness, difficulty, and aesthetic or creative merit), yet the most consequential claims rest on a small number of recent, single-group studies. Difficulty estimation without solving history remains open, as does the design of reliable, non-gameable creativity metrics, and the field's shared infrastructure warrants more scrutiny than it has received. These observations justify the joint treatment of generation and evaluation adopted here and, as set out in Section 8, they motivate the problem analysis and methodology that Chapter 2 develops.

Chapter 2: Problem Analysis and Methodology for an Engine-Based Chess Puzzle Generator

2.1 Introduction

One conclusion organised the survey of Chapter 1: generation and evaluation cannot be separated in this field, since the function that judges a puzzle is the same signal that trains, rewards, or filters the generator producing it. A weak evaluator therefore passes its weakness on to the generator, a mechanism reinforcement learning knows as reward hacking [39, 40, 41]. The survey also found that the field still lacks an agreed, operational definition of a good or creative puzzle [1].

What follows converts that conclusion into a design that can be built and tested within the limits of a master's project. The problem is defined in Section 2.2, the research questions are stated in Section 2.3, the objectives that follow from them in Section 2.4, and the system requirements in Section 2.5. Section 2.6 presents the architecture, and Section 2.7 draws the line between what will be implemented and what belongs to the wider ideal framework. The remaining sections specify the representation (Section 2.8), the generation strategy (Section 2.9), engine-based verification (Section 2.10), the evaluation criteria (Section 2.11), difficulty estimation (Section 2.12), the datasets and tools (Section 2.13), the experimental protocol (Section 2.14), and the evaluation metrics (Section 2.15); the limitations (Section 2.16) and a short conclusion (Section 2.17) close the chapter. The evaluator is treated throughout as the primary object of design. Concrete settings are given so that the design can be tested, and any value that is an engineering choice is marked as a proposed default.

2.2 Problem Definition

A chess puzzle is defined here as a pair (p, sigma): p is a legal position given in Forsyth-Edwards Notation (FEN), and sigma is the intended solution, an alternating sequence of moves that opens with the side to move in p. Four properties are required of the pair. Legality: p obeys the rules, and every move of sigma is legal in the position where it is played. Solubility: playing sigma achieves the stated goal, namely checkmate or a decisive gain, against the best defensive replies found under the fixed engine configuration. Uniqueness: at each move of the solving side the played move is uniquely best, that is, the engine's score for the best move exceeds the second-best by more than a fixed margin, a condition checked recursively along the principal variation, with mate-in-one excepted [50]. Quality: beyond validity, the puzzle should carry a

difficulty suited to a target solver and, where possible, some counter-intuitiveness or aesthetic interest [1, 9].

Two problems follow from this definition. The generation problem is to produce pairs (p, sigma) that satisfy these properties and that, taken as a set, are novel and varied relative to a human corpus. The evaluation problem is to decide, automatically and reproducibly, whether a candidate satisfies the properties, and to score its difficulty and quality. Neither problem can be treated alone, because the evaluator's output doubles as the generator's training or selection signal [24, 1, 28]. The core problem of this dissertation is therefore the following: given the open Lichess corpus [48] and a strong engine [51] as ground truth, build a generation-evaluation loop whose feedback signal is faithful enough that optimising it yields puzzles a human would also rate highly, under a protocol that stays reproducible despite engine non-determinism.

The scope is kept narrow on purpose. The system targets tactical puzzles with a forced, uniquely best line, the one family for which an engine can supply ground truth, and leaves aside long artistic compositions, whose conventions resist formalisation [11, 13]. Working toward a constructed, solvable target also follows established practice in procedural puzzle generation outside chess, where solvability is either guaranteed by construction or verified by a solver [70, 71]. Standard rules are assumed; variants fall outside the scope of the core experiments.

2.3 Research Questions

The open challenges identified in Chapter 1 are restated here as the research questions this dissertation addresses. Each is concrete, and each is paired with an objective (Section 2.4), an experiment (Section 2.14), and a metric (Section 2.15).

- RQ1. Can a candidate puzzle be verified as legal, soluble, and unique automatically and reproducibly, at a useful acceptance rate, using a chess engine as ground truth?
- RQ2. Can puzzle quality be scored along measurable axes (uniqueness, counter-intuitiveness, novelty, and diversity), and does that score, where a limited human check is available, rank candidates in broad agreement with human judgement?
- RQ3. Can the difficulty of a freshly generated puzzle be estimated without any solving history, and how accurate and stable is that estimate?
- RQ4. How sensitive are the evaluation metrics to engine version, search depth, and search budget, and how reproducible are the resulting verdicts?

- RQ5. To what extent can generation be directed toward a target theme, difficulty band, or material configuration, and how should the output set be measured for novelty and diversity?

Two of these questions contain a component that rests on an optional, experimental extension. In RQ2, agreement of the quality score with human judgement requires the optional human study of Section 2.15; in RQ5, the directed-generation aspect (conditioning on theme, difficulty band, or material) requires the optional conditioning mechanism of Section 2.9. RQ2 is therefore answered in full only for its automatic part, and RQ5 only for its novelty and diversity measurement; the human-judgement component of RQ2 and directed generation in RQ5 remain future work.

2.4 Objectives of the Proposed System

The general objective is a system that generates chess puzzles and evaluates them automatically, with the evaluator and generator designed as one loop. It breaks down into the following specific objectives.

- O1. Implement an engine-based verifier that confirms legality, solubility, and uniqueness reproducibly (RQ1, RQ4).
- O2. Implement a scorer that measures uniqueness, counter-intuitiveness, novelty, and diversity, and ranks candidates (RQ2).
- O3. Implement a history-free heuristic difficulty estimate as the core; as an experimental extension, train a learned model against the Lichess difficulty labels and characterise its accuracy (RQ3).
- O4. Generate candidate puzzles by engine-guided extraction and perturbation, and assemble the full generation-evaluation loop (RQ1, RQ5).
- O5. Measure the output set for novelty and diversity; directed generation by theme, difficulty, and material is left as future work (RQ5).

2.5 Requirements

The objectives refine into functional requirements, which state what the system must do, and non-functional requirements, which state the qualities it must exhibit. Each requirement can be verified on its own.

2.5.1 Functional Requirements

Table 2.1: Functional requirements of the proposed system.

ID	Requirement	Source
FR1	Generate a candidate puzzle (p, sigma) by extracting or perturbing positions from the open corpus.	O4 [48, 50]
FR2	Verify legality of p and of every move in sigma.	O1 [50]
FR3	Verify solubility against the engine's best defensive line by engine search.	O1 [51]
FR4	Verify uniqueness via the only-move criterion (best vs. second-best margin along the line, mate-in-one excepted).	O1 [50, 1]
FR5	Score uniqueness, counter-intuitiveness, novelty, and diversity, and rank candidates.	O2 [1, 57]
FR6	Estimate a difficulty rating for a candidate without solving history.	O3 [45, 56]
FR7	Filter candidates with cheap decisive tests before any deep analysis.	O4
FR8	Export accepted puzzles and the run report, with metadata, as JSON and CSV.	O4 [48]
FR9	Measure novelty and diversity across the output set; optionally condition generation on theme, difficulty, and material.	O5 [57, 58]

2.5.2 Non-Functional Requirements

Table 2.2: Non-functional requirements and their targets.

ID	Quality	Target
NFR1	Reproducibility	A pinned engine configuration (version, single thread, fixed depth and thresholds) yields identical verdicts and metrics [51].
NFR2	Determinism	Engine analysis runs on a single thread to remove the non-determinism of multithreaded search [51].
NFR3	Throughput	The cheap-test filter rejects most invalid candidates before any deep search.
NFR4	Scalability	Evaluation parallelises across positions, even though each position is analysed single-threaded.
NFR5	Modularity	Generator, verifier, scorer, and difficulty estimator are independent modules behind fixed interfaces.
NFR6	Openness	The system uses only open, reproducible foundations: open dataset, open engine, open libraries [48, 51, 67, 68].
NFR7	Auditability	Every reported figure is traceable to a fixed dataset snapshot and engine configuration [49].

2.6 General Architecture of the Proposed Framework

The framework is a closed loop with four stages: proposal, filtering, scoring, and feedback. A candidate proposed by the generator first meets cheap decisive tests, which discard invalid or implausible candidates before any expensive search; the survivors are verified and scored, and the result flows back to the generator. In the implemented core that feedback is nothing more than an accept or reject decision that builds up an output set. Turning the same signal into an evolutionary fitness or a reinforcement-learning reward is an extension and a direction for

future work (Section 2.7). Figure 2.1 shows the loop; Figure 2.2 shows the modules and the data that pass between them.

Figure 2.1 The generation–evaluation loop.

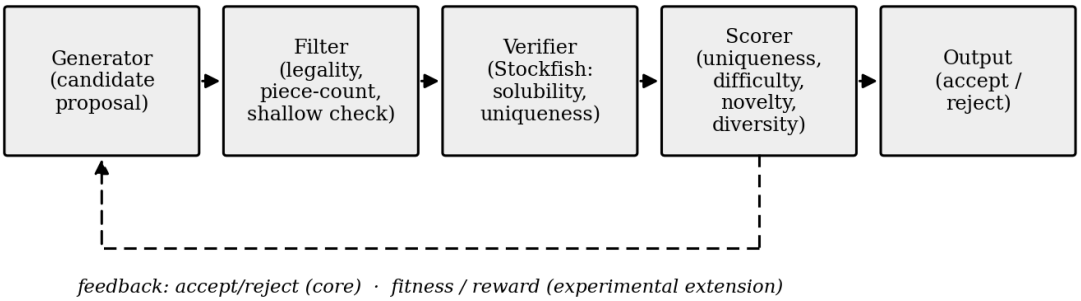


Figure 2.1: The generation-evaluation loop, with its four stages of proposal, filtering, scoring, and feedback.

Figure 2.2 System modules and data flow (shaded = data sources / engine; light = extension).

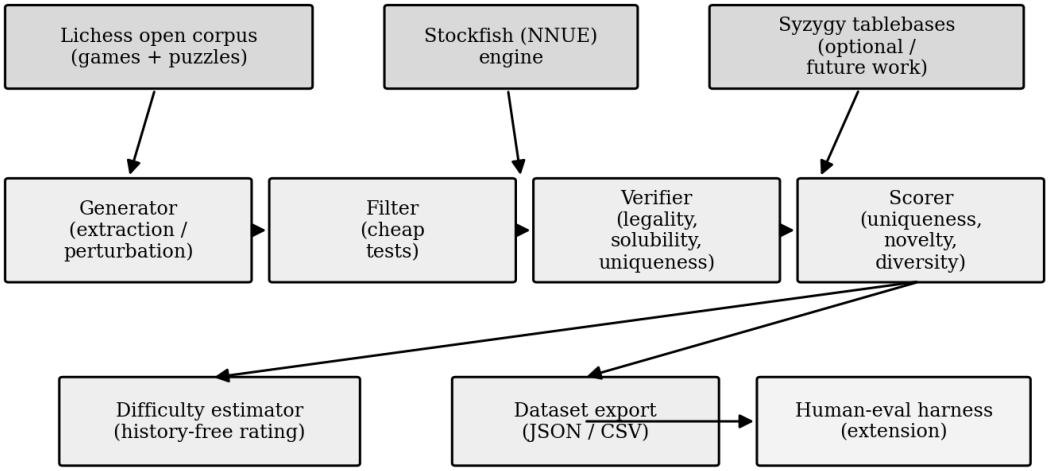


Figure 2.2: The six modules of the framework and the data that flow between them.

The loop is realised by six modules, summarised in Table 2.3. Modularity (NFR5) is enforced in the design: the generator communicates only through the (p, sigma) representation of Section 2.8, while the verifier and scorer expose a single function from a candidate to a structured verdict. The generator can therefore be swapped out, for example for the evolutionary extension of Section 2.9, without touching evaluation. Algorithm 2.2 (Section 2.11) gives the loop itself as pseudocode.

Table 2.3: Modules of the framework, their responsibilities, and their inputs and outputs.

Module	Responsibility	Input to output
Generator	Propose candidate puzzles by extraction or perturbation.	seed positions to candidate (p, sigma)
Filter	Reject invalid or implausible candidates cheaply.	(p, sigma) to keep / discard
Verifier	Confirm legality, solubility, and uniqueness with the engine.	(p, sigma) to validity verdict
Scorer	Compute uniqueness, counter-intuitiveness, novelty, and diversity.	valid (p, sigma) to score vector
Difficulty estimator	Predict a difficulty rating with no solving history.	(p, sigma) to rating, uncertainty
Exporter	Export the accepted puzzles and the full run report as JSON and CSV.	JSON / CSV report

2.7 Scope: Implemented Core, Experimental Extensions, and Future Work

The ideal framework sketched in Chapter 1, a learned generator fine-tuned by reinforcement learning against an engine reward, with diversity and divergence safeguards, is more than a master's project can responsibly implement and validate. This section therefore draws an explicit line between what will be built and what is described only as the wider framework; the three tiers are summarised in Table 2.4. The implemented core is an engine-based generation-evaluation pipeline that is feasible, reproducible, and sufficient to answer RQ1, RQ3, and RQ4 in full, together with the automatic part of RQ2. The human-judgement check in RQ2 and directed generation in RQ5 are not implemented and are left as future work; RQ5 is answered only for its novelty and diversity measurement. The experimental extensions are attempted as time allows and reported as partial. The future-work tier is specified at the level of design alone; it is neither implemented nor claimed as a result of this dissertation.

Table 2.4: Scope of the work. Only the core tier is claimed as an implemented contribution.

Tier	Components	Status
Implemented core	Engine-based generation-evaluation pipeline; Stockfish verification; only-move uniqueness checking; history-free difficulty estimate; novelty and diversity analysis.	Implemented and evaluated in this dissertation.
Experimental extensions	Evolutionary generation in position space; learned difficulty-regression model.	The learned difficulty model is implemented and validated (Chapter 3); evolutionary generation is partial.
Future work	Directed generation (theme, difficulty, material); human-evaluation study; learned transformer generator; PPO fine-tuning; KL regularisation; large-scale reinforcement learning; extensive ablation of safeguards.	Specified as the ideal framework; not implemented here.

The division also settles the feasibility of the project. Because the core rests only on mature, open components, an engine, an open corpus, and standard libraries, it can be built and re-run on commodity hardware. Engine analysis, not large-scale model training, is the cost that bounds the work, and this keeps the implemented system within reach and its results reproducible.

2.8 Puzzle Representation

The choice of representation is a design decision in its own right, because how easily a representation expresses degenerate positions determines how much work falls to the evaluator [25]. The system takes the FEN string as its native representation and keeps an explicit piece-list as an auxiliary view for the evolutionary extension [24]. A candidate is the pair (FEN, solution), the solution being a sequence of moves in long-algebraic (UCI) notation. A syntactically valid FEN can still describe an illegal position, two kings in check, say, or nine pawns of one colour, so the representation is always paired with a legality filter (FR2). The FEN-plus-solution form is also the native input of any sequence model; a future transformer-based difficulty estimator or transformer generator could therefore reuse it without a representation mismatch [3, 33].

2.9 Candidate Generation Strategy

In the implemented core, candidates are generated without learning, from two engine-guided sources. Extraction scans analysed games from the open corpus and, at the moments where one side blundered, re-analyses the position to test whether a single best line exists; this is the established Lichess method [48, 50]. Perturbation starts from an existing position, applies a small legal edit (adding, removing, or relocating a piece), and re-verifies the result, which lets the system reach configurations that extraction alone cannot. Both sources feed the same verifier and scorer, so generation quality is measured in the same way whatever the source.

Two further generators are described here but sit outside the core. The first, an experimental extension, is an evolutionary generator that operates on the piece-list: a population of positions is scored by an engine-based fitness that combines verified uniqueness with a difficulty term, and the weakest individuals are replaced by mutated copies of the strongest [24]. This instantiates the search-based procedural-content-generation paradigm for chess and inherits its behaviour, premature convergence included, unless the mutation is adapted; quality-diversity methods are the standard response and are noted here as the natural

way to keep a population varied [21, 22, 23, 72, 73]. The second, as future work, is a learned transformer generator pretrained on the corpus and fine-tuned by reinforcement learning against the composite reward of Section 2.11. That generator is the ideal framework, not a deliverable of this dissertation.

2.10 Engine-Based Verification

Verification establishes that a candidate is valid, and its output is the ground truth on which scoring depends. Three checks run in order of cost, cheapest first, so that most invalid candidates fall out before any deep search (NFR3). Legality is checked with the chess.js rules engine [67]. Solubility is checked by engine search against the engine's best defensive line. Few-piece positions (at most seven pieces), where an endgame tablebase would give exact ground truth [53, 54, 55], are flagged by the prototype; since no tablebase backend is bundled, they are verified by engine search like any other position, and tablebase probing is left as a later addition (Section 2.16). Uniqueness is checked by the only-move criterion. At each solving move the engine runs with the two best moves retained, and the move passes only if the best exceeds the second-best by more than a fixed margin, applied recursively along the principal variation, with mate-in-one excepted [50, 1]. The verdict depends on engine version, depth, and margin, so these are pinned and reported, and analysis runs single-threaded to keep the verdict reproducible (NFR1, NFR2). The proposed defaults, a verification depth of about 18 to 22 plies and a margin of about 100 to 200 centipawns, are to be calibrated against the Lichess method during implementation [50]. Algorithm 2.1 states the procedure.

```
Algorithm 2.1  Verify(candidate (p, sigma))
 1: if not legal(p) or any move of sigma is illegal then return REJECT
 2: if piece-count or material is implausible then return REJECT
 3: result <- engine_search(p, depth_shallow)   # plausibility gate; a
tablebase probe for <=7 pieces is future work
 4: if result does not reach the stated goal then return REJECT
 5: for each solving move m_i along sigma do
 6:     (best, second) <- engine_search(p_i, depth_full, MultiPV = 2)
 7:     if eval(best) - eval(second) < margin and m_i is not mate-in-one
then
 8:         return REJECT
 9: return ACCEPT with the per-move margins
```

2.11 Puzzle Evaluation Criteria

For every valid candidate the scorer measures four axes, following the decomposition introduced in the most recent chess study [1] and the diversity emphasis of the procedural-content-generation literature [57, 58]. Uniqueness is the centipawn gap between the best and second-best move, averaged along the line and normalised to the unit interval. Counter-

intuitiveness is the critical depth at which the engine first prefers the winning move; a move found only at greater depth thus scores higher than an obvious one. Novelty is an edit distance over the board and the solution line, measured against the corpus and against the set already generated, so that repeating one position cannot earn a good score. Diversity is the spread of the accepted set over board and solution features, the quantity that expressive-range analysis was built to measure [57, 58].

In the implemented core these axes rank candidates and decide acceptance; Algorithm 2.2 gives the loop. Collapsing the axes into a single scalar reward and optimising it directly is left to the future-work tier, for a precise reason: a metric that is optimised directly stops being a trustworthy measure [40, 41]. The implemented system therefore keeps scoring and acceptance apart from any learned optimisation, and final judgement rests on held-out metrics and human ratings (Section 2.15) rather than on the score itself [60, 61, 77].

```
Algorithm 2.2  GenerateEvaluateLoop(N)
 1: A <- empty set                              # accepted puzzles
 2: while |A| < N do
 3:     (p, sigma) <- Generator.propose()       # extraction or perturbation
(core)
 4:     if not Filter(p, sigma) then continue   # cheap legality and sanity
tests
 5:     v <- Verify(p, sigma)                   # Algorithm 2.1
 6:     if v = REJECT then continue
 7:     s <- Score(p, sigma)                    # uniqueness, counter-
intuitiveness, novelty, diversity
 8:     if Accept(s) then A <- A + {(p, sigma, s)}
 9:     # extension: Generator.update(s)        # evolutionary fitness / RL
reward (future work)
10: return A
```

2.12 Difficulty Estimation

A freshly generated puzzle has no solving history, so the usual approach, accruing an Elo- or Glicko-style rating from many attempts, cannot be applied at generation time; this is the cold-start problem [45, 64, 65]. The implemented system therefore begins with a transparent heuristic computed directly from the (p, sigma) representation. It combines simple features (the number of legal moves, the length of the solution, the uniqueness margin between the best and second-best move, whether the solution contains checks or captures, and the engine depth at which the key move is first found) and maps the result to a difficulty band. As an experimental extension, a lightweight learned regression model, gradient-boosted trees or a shallow network rather than a large transformer, can be trained against the corpus difficulty labels [48] and report a calibrated uncertainty, since difficulty predictions are known to be unstable at the rating extremes [46, 56]. Behind the design is a point established independently of chess:

difficulty is a property of human problem-solving, not of engine-optimal search, and it can be modelled and calibrated against human solving behaviour [42, 43, 66, 74]. Adaptive learning systems rest on the same idea, estimating item difficulty from learner performance and using it to select what to present next [64, 75, 76]. The estimator's accuracy, and the bias of its training labels, are treated as objects of study rather than as solved inputs: the Lichess labels are widely dispersed across the puzzle database as a whole (mean about 1516, standard deviation about 543 [56]) and depend on which players attempt each puzzle, while the 40,000-puzzle sample used to train and test the model in Chapter 3 has a mean rating of about 1468.

2.13 Dataset and Tools

The system rests exclusively on open, reproducible foundations (NFR6); Table 2.5 lists them. For each puzzle the Lichess puzzle database supplies the FEN, the only-move solution, a Glicko-style difficulty rating with its deviation, a popularity score, and thematic tags [48]. For the learned-difficulty extension the corpus is split into training, validation, and held-out test partitions, deduplicated by position so that no test position appears in training. Novelty in the implemented core is measured against a small reference sample, not the full database; a corpus-wide novelty benchmark is left as a later addition (Section 2.16). The dataset snapshot is fixed and cited instead of being treated as a moving target, because figures such as the corpus scale vary between snapshots [49].

Table 2.5: Datasets, engine, and software, with their roles.

Resource	Role	Source
Lichess puzzle / games database	Seed corpus, difficulty labels, and novelty reference sample.	[48, 49]
Stockfish (NNUE)	Verification, uniqueness checking, scoring; pinned single-thread.	[51]
Syzygy tablebases (future work)	Exact ground truth for few-piece positions; interface present but no backend bundled in the prototype.	[55]
chess.js	Move generation, legality checking, and FEN/PGN handling.	[67]
Leela Chess Zero / Maia	Optional heterogeneous references for uniqueness cross-checks and human-aligned surprise.	[52, 7]

2.14 Experimental Protocol

Each research question is answered by a specific experiment against a specific baseline, under the reproducibility conditions of Section 2.10; Table 2.6 gives the mapping. The baselines are an extraction pipeline in the Lichess style [48, 50] and, for the extension, the

evolutionary generator of Section 2.9 [24]. Every run fixes the dataset snapshot, the engine version, the search depth, MultiPV, the uniqueness margin, and the random seeds. The compute budget is reported as a total engine node count, deep analysis being the dominant cost [49].

Table 2.6: Experimental protocol: each research question, its experiment, and the comparison.

RQ	Experiment	Comparison
RQ1	Run the core loop; measure validity and acceptance rates.	Extraction baseline.
RQ2	Score and rank accepted puzzles; the human comparison is left as future validation.	Corpus puzzles of known rating.
RQ3	Predict difficulty with the heuristic; as an experimental extension, train a learned model and test it against held-out Lichess labels.	Corpus difficulty labels [56].
RQ4	Vary engine version, depth, and budget; measure verdict stability.	Same candidates across configurations.
RQ5	Measure novelty and diversity of the output set.	Unconditioned output (directed generation: future work).

Independently of these experiments, an automated test suite checks the implemented components. It validates the engine-free logic: FEN and PGN parsing, the uniqueness calculation against mocked engine outputs, novelty distance, diversity entropy, difficulty-feature extraction, and report generation, together with manual validation of the engine on a set of known puzzles. Chapter 3 reports these tests and their results.

2.15 Evaluation Metrics

Evaluation is primarily automatic, because automatic metrics are necessary but never sufficient [2, 60, 61]. So that the system is not judged by a quantity it was tuned to maximise, the metrics used for final assessment are computed on held-out positions and include diversity measures that acceptance does not directly optimise. Table 2.7 lists them. A human study is specified as future validation but is not implemented in this work, and the system contains no human-evaluation module. Were such a study conducted, a small panel of rated players would score realism, difficulty, and creativity on curated puzzles from the system and from the corpus, blind to the source; agreement would be quantified by Cohen's kappa [68] and differences tested with a non-parametric paired test [69]. Any such result would be a limited validation, not a population-level claim [2].

Table 2.7: Evaluation metrics, what each measures, and its level.

Metric	Definition	Level / direction
Validity rate	Fraction of candidates that are legal, soluble, and unique.	Automatic; higher better

Metric	Definition	Level / direction
Creative yield	Fraction simultaneously unique and counter-intuitive.	Automatic; higher better [1]
Novelty distance	Mean edit distance of accepted puzzles from a reference sample of the corpus.	Automatic; higher better [57]
Diversity / expressive range	Spread of the output set over board and solution features.	Automatic; higher better [57, 58]
Difficulty error	Error of the learned difficulty estimate against held-out Lichess labels (experimental extension).	Automatic; lower better [56]
Human ratings (future work)	Rated players score realism, difficulty, and creativity, blind to source.	Human; not implemented [2]
Inter-rater agreement (future work)	Agreement among raters.	Human; not implemented [68]

2.16 Methodological Limitations

The design has limits, and they bound any conclusion drawn from it; they are listed here. The engine is taken as ground truth for solubility and uniqueness, so every such verdict is relative to an engine and a threshold: a position that is a valid puzzle under one Stockfish version, depth, or margin may not be under another [50]. Pinning the configuration makes the results reproducible, not absolute. Single-thread determinism is paid for in speed, and that cost bounds both the search depth and the scale of the experiments. The difficulty labels inherited from Lichess are widely dispersed and biased by which players attempt each puzzle [56], so the cold-start estimator is calibrated against an imperfect target. No human evaluation was carried out; the human study of Section 2.15 remains future validation, and the present work therefore makes no human-judged claim [2]. Two further components are specified but not realised in the prototype. Endgame tablebase verification is not bundled, so few-piece positions are verified by engine search like any other position. And novelty is measured against a small reference sample instead of the full puzzle database, so the reported novelty is indicative rather than a corpus-wide measure. Finally, the scope is restricted to tactical puzzles with a forced line, so the results do not transfer directly to artistic compositions [11, 13], and the learned-generator framework of the future-work tier is neither implemented nor claimed here.

2.17 Conclusion

The methodology is now fixed. Each of the five research questions is paired with an experiment and a metric (Tables 2.6 and 2.7); the engine configuration is pinned, a single thread and MultiPV 2 with fixed margins, so that verdicts can be reproduced; and the three-tier scope

separates the implemented core from the evolutionary extension and from future work (directed generation, a human study, a learned generator). The metric suite judges the system on held-out evidence rather than on the score it optimises, with human evaluation left as future validation. The main methodological limits, engine-relative ground truth, dispersed and biased difficulty labels, and a sample-based novelty reference, are recorded in Section 2.16 and bound every conclusion that follows. Chapter 3 implements the core and reports the measured results.

Chapter 3: Implementation and Tests

3.1 Introduction

Tactica is the working application of this dissertation; it realises the implemented-core tier of the Chapter 2 methodology as a browser program. The sections below follow the source code from the technologies and architecture through each processing stage to the user interface, the storage layer, the tests, and the measured results. The description stays faithful to the actual source code throughout; wherever a component of the wider methodology was not implemented, the text says so.

3.2 Implemented Scope of the Application

The application implements those parts of the Chapter 2 design that a single engine, an open corpus, and a browser can support on commodity hardware. What is and is not implemented is summarised first, so that the rest of the chapter can be read without ambiguity.

The implemented parts are PGN loading and position extraction; perturbation of seed positions by single legal edits; a three-stage engine-based verifier (legality, a shallow plausibility screen, and a recursive only-move uniqueness test built on MultiPV search); scoring of every accepted puzzle for uniqueness margin, counter-intuitiveness, novelty, and diversity; a history-free heuristic difficulty estimate together with an optional learned regression model; a five-screen user interface for browsing, solving, statistics, run reporting, and settings; client-side persistence with schema-versioned migration; and export of the run report as JSON and CSV.

Not implemented, and described in Chapter 2 as experimental extension or future work, are an endgame-tablebase probe (the interface exists, but no tablebase backend is bundled, so every position is verified by engine search); novelty measured against the full Lichess puzzle database (the implemented metric uses a small reference sample); a human-evaluation study (the application contains no human-rating module); and a learned, reinforcement-trained generator. These boundaries match the three-tier scope of Section 2.7.

Table 3.1 maps each requirement of Chapter 2 to the part of the implementation that satisfies it.

Table 3.1: Mapping from Chapter 2 requirements to their Chapter 3 implementation.

Requirement (Chapter 2)	**Implementation (Chapter 3)**
FR1 generate candidate puzzles	PGN extraction and position perturbation (Sections 3.5, 3.6)

Requirement (Chapter 2)	Implementation (Chapter 3)
FR2 legality	chess.js legality and FEN checks (Section 3.7)
FR3 solubility against best defence	engine search along the forced line (Section 3.7.3)
FR4 uniqueness	recursive only-move test, Stockfish MultiPV = 2 (Section 3.7.3)
FR5 score and rank	uniqueness, counter-intuitiveness, novelty, diversity (Section 3.8)
FR6 difficulty without history	heuristic estimate and optional learned model (Section 3.9)
FR7 cheap filtering first	shallow engine screen before deep search (Section 3.7.2)
FR8 export accepted puzzles	JSON and CSV run report (Section 3.11)
FR9 measure novelty and diversity	computed per run; novelty against a reference sample (Section 3.8)
NFR1 reproducibility	single-thread engine, fixed depth and seed (Section 3.7.1)
NFR7 auditability	exportable report and per-puzzle stored configuration (Section 3.11)

3.3 Technologies Used

Tactica is a self-contained, client-side web application with no server and no external database: a puzzle is generated, analysed, solved, and stored entirely in the user's browser. Table 3.2 lists the main technologies and their roles; the subsections that follow take each in turn.

Table 3.2: Technologies used by the application and their roles.

Technology	Role in Tactica
JavaScript (ES modules), React 18	Application language and user-interface framework.
Vite	Development server and production build tool.
chess.js [67]	Board representation, legality, move generation, FEN and PGN handling.
Stockfish 16 NNUE (WASM) [51]	Position analysis for verification and scoring, run in a Web Worker.
zustand	In-memory state stores for puzzles, progress, settings, and view.
Tailwind CSS	Styling of the dark, responsive interface.
Browser localStorage	Persistence of puzzles, progress, and settings across sessions.

3.3.1 React and Vite

The interface is a single-page React application. No client-side router library is used; a small view field in a state store selects which of the five screens is shown, and this keeps navigation simple and the bundle small. Vite supplies the development server, with hot module replacement while developing, and produces the optimised static bundle for deployment. The whole application is static, so any static host can serve it without special configuration.

3.3.2 chess.js

All chess rules are delegated to chess.js [67], a mature JavaScript library for move generation, legality checking, and the parsing of FEN positions and PGN games. Relying on a well-tested rules library instead of re-implementing chess logic removes a whole class of rule and legality bugs and leaves the application free to concentrate on generation and evaluation. The library reads uploaded games, enumerates legal moves when a position is perturbed, replays solution lines, and validates the user's moves during solving.

3.3.3 Stockfish WASM

Position analysis is carried out by Stockfish 16 with its efficiently updatable neural-network evaluation, compiled to WebAssembly in its single-threaded build [51]. The engine runs inside a Web Worker, so a long analysis does not freeze the interface, and it communicates over the standard UCI protocol through a thin client module that serialises requests and parses the engine's replies. Choosing the single-threaded build was deliberate. It needs no special cross-origin headers, which lets the application run on any static host, and one thread analysing a fixed position returns the same result every time, which makes the verification reproducible. The engine's neural network, about 38 MB, is downloaded once and then cached by the browser.

3.3.4 Local Storage

Persistence goes through a small namespaced wrapper over the browser's localStorage. All keys share a single namespace, and the store records a schema version. On load, a non-destructive migration fills in any fields missing from existing records, so data created by an earlier version keeps working; only a version from the future (a downgrade) triggers a reset. Accepted puzzles, solving progress, and settings are all kept this way, and the user's library and statistics therefore survive a page reload or a browser restart.

3.4 Application Architecture

The application is built from small, single-purpose modules connected in a pipeline, shown in Figure 3.1. A position moves through it from left to right: loaded, screened, analysed by the engine, tested for uniqueness, scored and rated, and at the end stored and displayed. The generator does nothing but propose positions. Every candidate then passes through the same verification and scoring code, so the two concerns stay separate and one generator can be replaced without touching the other.

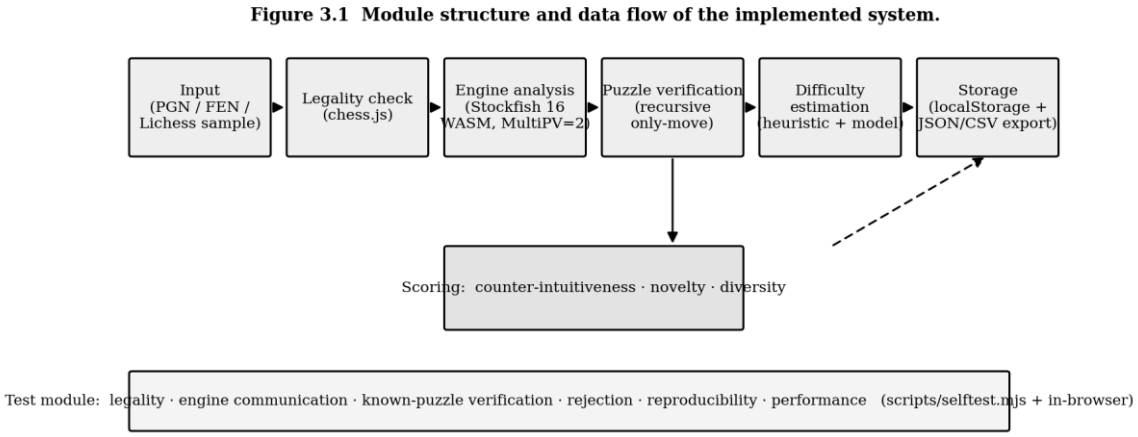


Figure 3.1: The Tactica processing pipeline: a candidate position is loaded, screened, analysed by the engine, tested for uniqueness, scored, rated, and finally stored and displayed.

The modules fall into four layers. The generator layer reads and proposes candidates: pgnParser.js for PGN reading, perturb.js for perturbation, corpus.js for the reference sample. The evaluation layer verifies and scores them: engineClient.js for the engine interface, puzzleFinder.js for verification and the generation loop, counterIntuitiveness.js, novelty.js, and diversity.js for scoring, and difficulty.js with difficultyFeatures.js and difficultyModel.js for rating. The state layer keeps run-time data in four zustand stores: a puzzle store (the generated library and the generation status), a progress store (solved counts and streaks), a settings store (engine depth, generation mode, board orientation), and a small UI store (the current view and selected puzzle). The presentation layer consists of the five React screens described in Section 3.10, with the storage wrapper (localStorage.js) and the run report (report.js) underneath.

3.5 Input Data and PGN Processing

Positions enter the system as games in Portable Game Notation. A small bundled PGN sample ships with the application for demonstration, and an uploaded PGN file serves larger runs. The parser (pgnParser.js), built on chess.js [67], reads games one at a time, replays each, and yields the position before every move together with its ply index and the move played. Parsing is incremental and stops the moment the candidate budget is full, so even a very large game database never freezes the browser tab. Before any engine work, two cheap filters apply: positions inside the opening, below a configured ply threshold, are skipped, and a position is kept only if it contains a forcing move, that is, a capture or a check. Most quiet positions are removed this way before the engine is ever called.

3.6 Candidate Generation

The implemented core generates candidates without any learning. Two engine-guided sources are used, and both feed the same verifier and scorer; the generation mode (extraction, perturbation, or both) is selected in Settings.

3.6.1 Extraction from PGN

Extraction mines real games. As the parser replays a game it offers the position before each move, and whatever survives the opening and forcing-move filters becomes a candidate position. The approach follows the established Lichess method of finding puzzles by re-analysing positions where a player has just gone wrong [48, 50]. What extraction yields is realistic and human-reachable; the price is that it is bounded by what was actually played.

3.6.2 Perturbation of Positions

Perturbation (perturb.js) creates new positions that extraction alone cannot reach. A single small legal edit is applied to a seed position: one piece removed, one piece relocated, or one piece of a given type added. A legality filter then checks each edited position: it must parse, have exactly one king per side, have no pawns on the back ranks, and not leave the side not to move in check. Illegal results are discarded. Edits that survive pass through the same verifier and scorer as extracted positions, so a perturbed puzzle is held to exactly the same standard as a real one.

3.7 Engine-Based Verification

Verification decides whether a candidate is a sound, uniquely solvable puzzle; its output is the ground truth on which scoring depends. Three checks run in order, cheapest first, so that most candidates are rejected before any deep search begins.

3.7.1 Stockfish Configuration

One configuration module holds all engine settings, so the verifier, the scorer, and the run report stay in step and every run is reproducible. The engine is pinned to a single thread; MultiPV is set to two, since the only-move test needs the two best lines; and the search depths and acceptance thresholds are fixed. The constants that govern verification are listed in Table 3.3.

Table 3.3: Engine and verification configuration (from the application's config module).

Setting	Value	Meaning
Engine	Stockfish 16 NNUE, single-thread WASM	Analysis engine [51]
MultiPV	2	Two best lines retained for the only-move test
Screen depth	10 plies	Cheap shallow plausibility gate
Deep-verify depth	16 plies (default), up to 20	Full uniqueness search; 20 is the canonical thesis depth
Only-move margin	150 cp	Best must beat second-best by this much at each solver move
Winning margin	150 cp	Best move must reach at least this advantage
Already-won cap	900 cp	Positions already winning by more are skipped

3.7.2 Shallow Filtering

The first engine step is a cheap shallow analysis at the screen depth. A helper reads the two best lines and passes the position on to deep analysis only when the best move reaches the 150 cp winning margin, beats the second-best line by more than the 150 cp only-move margin, and the position is not past the 900 cp already-won cap (Table 3.3). The screen disposes of most non-tactical positions cheaply; in the measured run it removed 399 quiet positions before any deep verification (Section 3.13). The expensive full-depth search is therefore spent only on positions that could plausibly be puzzles. It is this screen that keeps a useful search depth affordable in the browser.

3.7.3 MultiPV Uniqueness Test

Positions that survive the screen enter the recursive only-move test, the heart of the verifier. The solution is built one move at a time. At each move of the solving side the engine runs with the two best lines retained, and the move is accepted only when the best exceeds the second-best by more than the only-move margin; a forced mate-in-one is treated as unique. The opponent always answers with the engine's best defensive move, so the solution is verified against best defence rather than against an arbitrary line. The line runs on until the tactic resolves in a mate or a decisive material gain, and any solving move that is not uniquely best truncates the line before it. A position therefore becomes a puzzle only if every solving move along the forced line is uniquely best. The per-move margins are stored with the puzzle. Algorithm 3.1 states the procedure; it implements the verification design of Section 2.10 [50, 1].

```
Algorithm 3.1  Verify(position p, solution accumulator)
 1: if p is not legal or has implausible material then return REJECT
```

```
 2: screen <- engine.analyze(p, depth = 10, MultiPV = 2)
 3: if not qualifiesScreen(screen) then return REJECT   # winning, unique,
not already won
 4: repeat for each solving move along the forced line:
 5:     r <- engine.analyze(current, depth = deep, MultiPV = 2)
 6:     if best(r) - second(r) < margin and best is not mate-in-one then
break
 7:     play best (solver), then play engine's best reply (defence)
 8: if the line wins material or mates then return ACCEPT with per-move
margins
 9: else return REJECT
```

3.8 Puzzle Scoring

Every accepted puzzle is scored along four axes, following the decomposition of Section 2.11. The scores are stored with the puzzle and summarised in the Generation Report, and they are kept apart from any single reward, so that no one number is optimised directly.

3.8.1 Uniqueness Margin

The uniqueness margin is the centipawn gap between the best and second-best move, recorded at each solving move during verification and summarised per puzzle. A large margin means a sharply forced solution; a small one means the position only just qualifies. The margin is the same quantity the verifier thresholds on, which ties this score directly to the acceptance decision.

3.8.2 Counter-Intuitiveness

Counter-intuitiveness captures how hard the key move is to see. The scorer (counterIntuitiveness.js) probes the position at increasing search depths until it finds the critical depth, the smallest depth at which the engine's own top choice first becomes the solution's first move. A move that only a deep search prefers therefore scores higher than one an immediate search already likes. The critical depth is then combined with the evaluation swing between the shallow and deep searches and folded into a value in the unit interval, weighted about seventy percent toward depth and thirty percent toward the eval swing. This operationalisation of counter-intuitiveness comes from the recent generative work [1].

3.8.3 Novelty

Novelty measures how unlike known material a puzzle is. The scorer (novelty.js) computes the distance from the puzzle to its nearest neighbour in a reference set, over the board and the solution alike: the board distance is a Hamming distance over the sixty-four squares, and the solution distance compares the move sequences. A value near one means nothing in the

reference resembles the puzzle; a value near zero means the puzzle is already there. Because the reference set in the implemented system is a small sample of the full Lichess database, the reported novelty is indicative rather than a corpus-wide measure; a full-database novelty benchmark is noted as future work in Section 3.14.

3.8.4 Diversity

Diversity describes the spread of a whole run rather than of a single puzzle. The scorer (diversity.js) computes the normalised Shannon entropy of the accepted set across motif, difficulty band, material signature, and solution length, together with the mean pairwise distance between puzzles. Values near one mean the run produced an even, varied set; low values mean it concentrated on a few kinds of puzzle. Diversity is exactly the property the expressive-range and quality-diversity literature was built to measure, and it is included so that a generator cannot score well merely by repeating one good idea.

3.9 Difficulty Estimation

Each accepted puzzle receives an estimated difficulty. A freshly generated puzzle has no solving history, so a rating accrued from many human attempts cannot be applied at generation time; this is the cold-start problem. The implemented core therefore uses a transparent heuristic computed directly from the position and its solution (difficulty.js). Starting from a base rating, it adds for a longer forced line, a quiet first move, a sacrifice or discovered attack, a wide uniqueness margin, and lower mobility, and it maps the result to one of four bands: Easy, Medium, Hard, and Expert. An optional learned model (difficultyModel.js) refines this estimate, following work on predicting puzzle difficulty without solving history [45, 46]. A small one-hidden-layer network over thirteen history-free features predicts a rating and a calibrated uncertainty; when its model file is present it is used in place of the heuristic, with the heuristic as the fallback.

For reproducibility the learned model is specified as follows. Its inputs are thirteen history-free features extracted from the position and its verified solution: the number of solver moves, the material swing, the piece count, the side-to-move mobility, whether the first move is quiet, the number of checks and of captures in the solution, whether the solution is a mate, and binary flags for fork, pin, skewer, discovered attack, and sacrifice. The features and the target rating are standardised to zero mean and unit variance. The network has a single hidden layer of eight rectified-linear units and is trained by gradient descent for 250 epochs at a learning rate of 0.02. The data are shuffled with a fixed pseudo-random seed (42) and split eighty-five to fifteen into

34,000 training and 6,000 held-out test puzzles. The split is by puzzle, each Lichess puzzle being a distinct position with its own rating, and no test puzzle is seen during training. The four rating bands are Easy below 1150, Medium below 1650, Hard below 2150, and Expert above. The training script (scripts/trainDifficulty.mjs, with the shared scripts/difficultyTrainLib.mjs) is included in the project and regenerates the model file from the Lichess CSV.

For this work the model was trained on the real Lichess Open Puzzle Database [48], where each puzzle carries a Glicko-based difficulty rating derived from many real solving attempts [56]. Training used 34,000 puzzles; evaluation used a held-out set of 6,000 puzzles the model had never seen. Table 3.4 reports its accuracy against two baselines, predicting the global mean rating and the rule-based heuristic, and Table 3.5 gives the confusion of predicted against actual difficulty band.

Table 3.4: Difficulty prediction on 6,000 held-out Lichess puzzles.

Predictor	**MAE (rating points)**	**RMSE**	**Pearson r**	**Spearman rho**
Learned model	309	390	0.72	0.73
Baseline: predict the mean (1468)	457	548	-	-
Baseline: rule-based heuristic	468	594	-	-

On the held-out puzzles the learned model reaches a mean absolute error of about 309 rating points and a Pearson correlation of 0.72 with the real Lichess difficulty; the two baselines sit at 457 and 468 rating points of error. Its predicted band is exactly right 51.8 percent of the time and within one band 94.6 percent of the time. As an ordering, then, it tracks real difficulty well, even though an absolute error of a few hundred rating points is large for any single puzzle. In the confusion matrix, almost all of the errors fall between neighbouring bands; the hardest bands to separate are Medium and Hard, which is also where human ratings sit closest together.

Table 3.5: Band confusion matrix (row = predicted band, column = actual band; 6,000 held-out puzzles).

Band	**Easy**	**Medium**	**Hard**	**Expert**
Easy	1,380	510	92	9
Medium	471	745	368	77
Hard	129	619	813	522
Expert	1	16	76	172

3.10 User Interface

The interface consists of five screens, selected from a navigation bar and rendered inside a shared dark, responsive layout. Every screen reads from the in-memory stores, so a change such as a newly generated puzzle or a solved result appears at once across the application. The screens are described below, and the accompanying figures show each one. The screenshots in this section come from an illustrative live session on the bundled sample at a user-chosen deep-verify depth of eighteen. They are therefore distinct from the measured evaluation of Section 3.13, which used a fixed depth of twelve on the Lichess Elite sample; the small run-level figures they show, such as the acceptance rate on the Generation Report, belong to that short demonstration and not to the results reported there.

3.10.1 Puzzle Library

The Library is the home screen. It shows the generated puzzles as a grid of cards, and a filter bar narrows the set by motif, difficulty band, the piece that delivers the blow, and solved or unsolved status. Generation is also launched here: the user generates from the bundled sample or uploads a PGN file, and during a run a progress banner shows the engine status, a progress bar, the number of positions scanned, and the number of puzzles found so far, with a button to cancel. Selecting a card opens the puzzle in the solving screen.

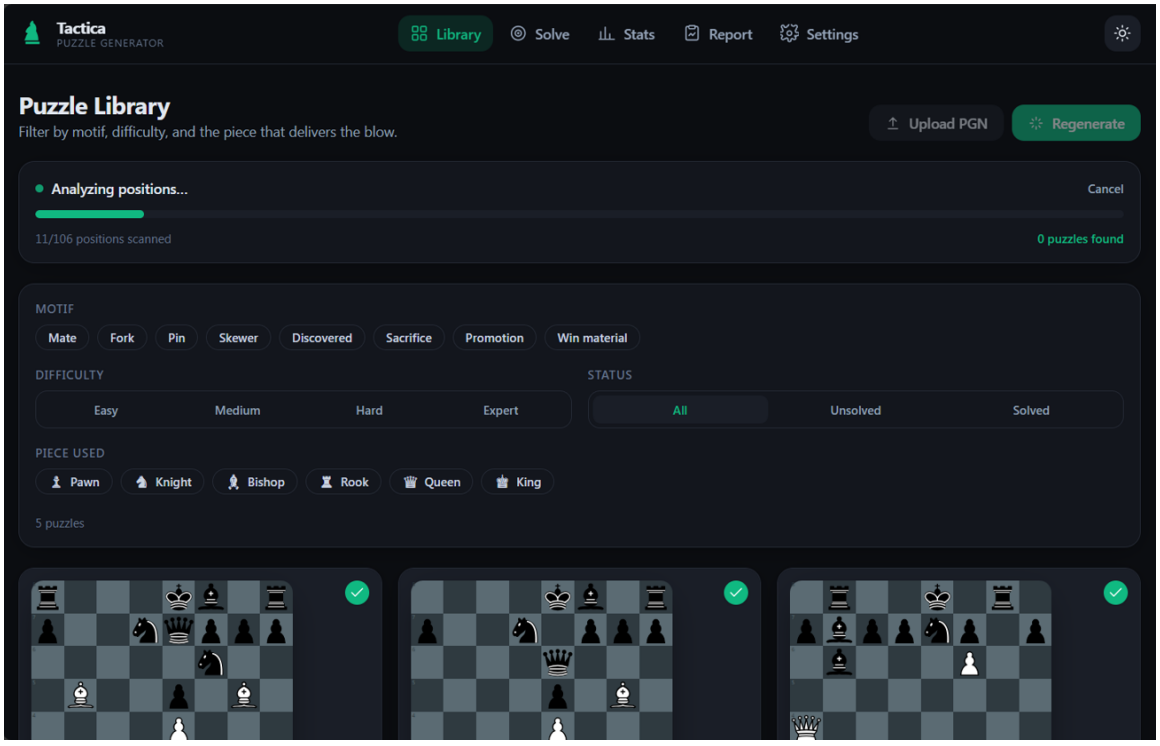


Figure 3.2: The Puzzle Library during a generation run: the filter bar (motif, difficulty band, piece, and solved status), the Upload-PGN and Regenerate controls, the live progress banner, and the grid of generated puzzles below.

3.10.2 Solving Interface

The Solve screen presents a single puzzle on an interactive board. The user plays the solution by dragging or clicking pieces. Each move is validated against the verified line, correct and incorrect moves receive immediate feedback, and after a wrong move the opponent's reply is played out so the position stays consistent. A hint marks the piece to move, the full solution can be revealed, and the result feeds the streak counter and the statistics. Board orientation follows the user's setting, and tags above the board show the puzzle's motifs and its difficulty band.

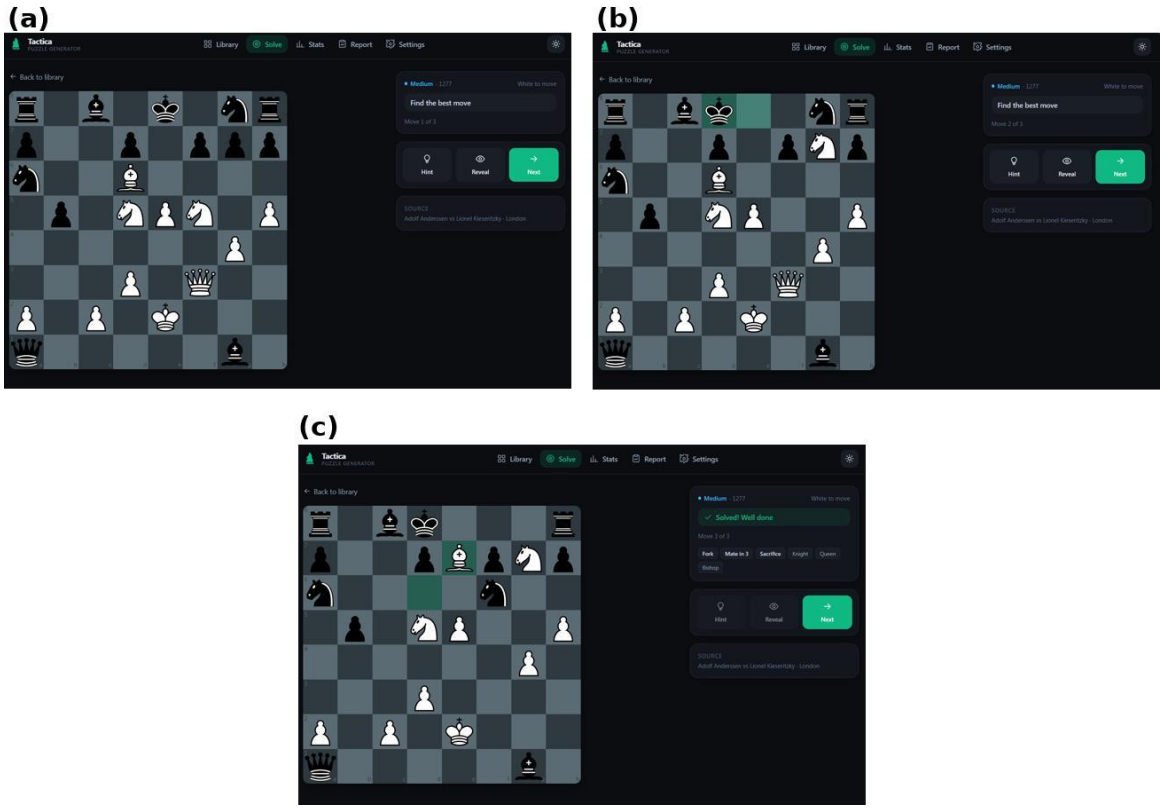


Figure 3.3: Solving an example puzzle (the Immortal Game, Anderssen versus Kieseritzky, London 1851): (a) the puzzle as presented, with the side to move, difficulty, and the hint, reveal, and next controls; (b) after the first correct move, the opponent's reply played out; (c) on completion, with the detected motifs revealed.

3.10.3 Statistics

The Statistics screen summarises the user's solving record: total attempts, total solved, accuracy, current streak, and best streak appear as summary cards. Coverage bars below them break down solved against available puzzles by motif and by difficulty band, so the user can see which kinds of tactic they have worked through and which they have not.

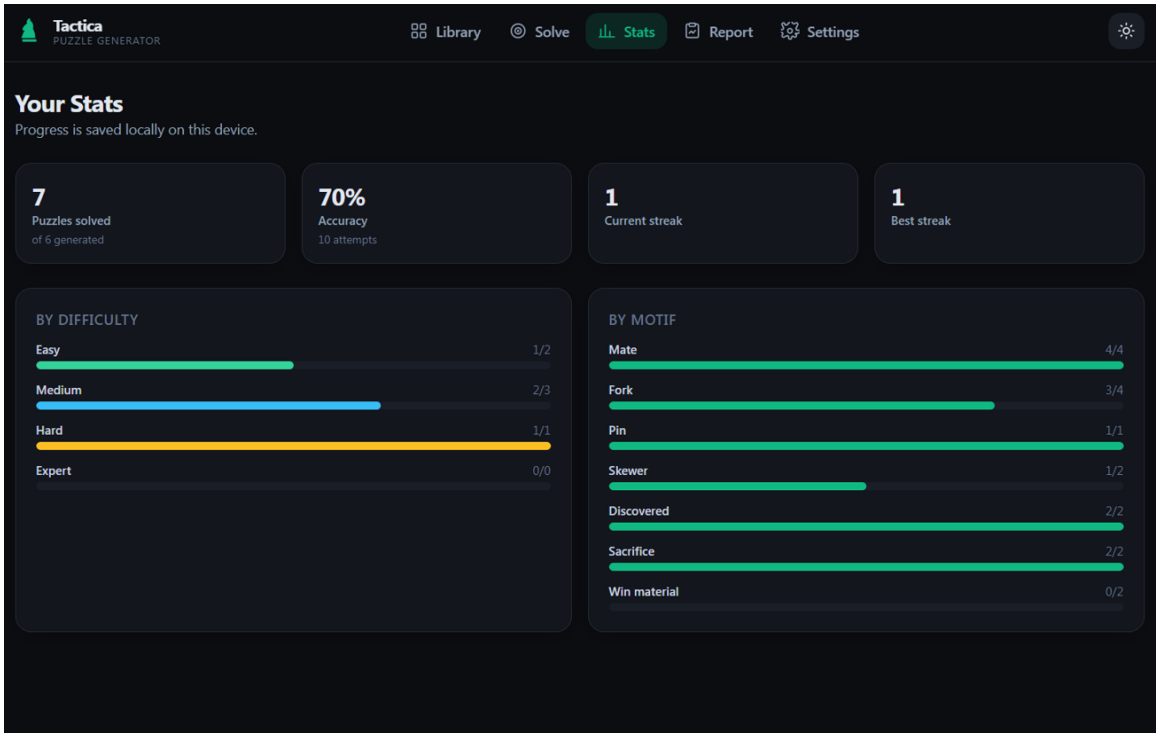


Figure 3.4: The Statistics screen: the puzzles-solved count, accuracy, and current and best streaks, with solving coverage broken down by difficulty band and by motif.

3.10.4 Generation Report

The Generation Report is the analyst's view of the most recent run, and it is kept separate from the solver's statistics. It shows the acceptance rate, the creative yield (the fraction of examined positions that were both unique and counter-intuitive), the mean uniqueness margin, mean novelty, and the diversity measures, together with the difficulty distribution of the accepted puzzles and the full run configuration that produced them. The whole report can be exported as JSON or CSV for inclusion in an analysis or, as here, a dissertation.

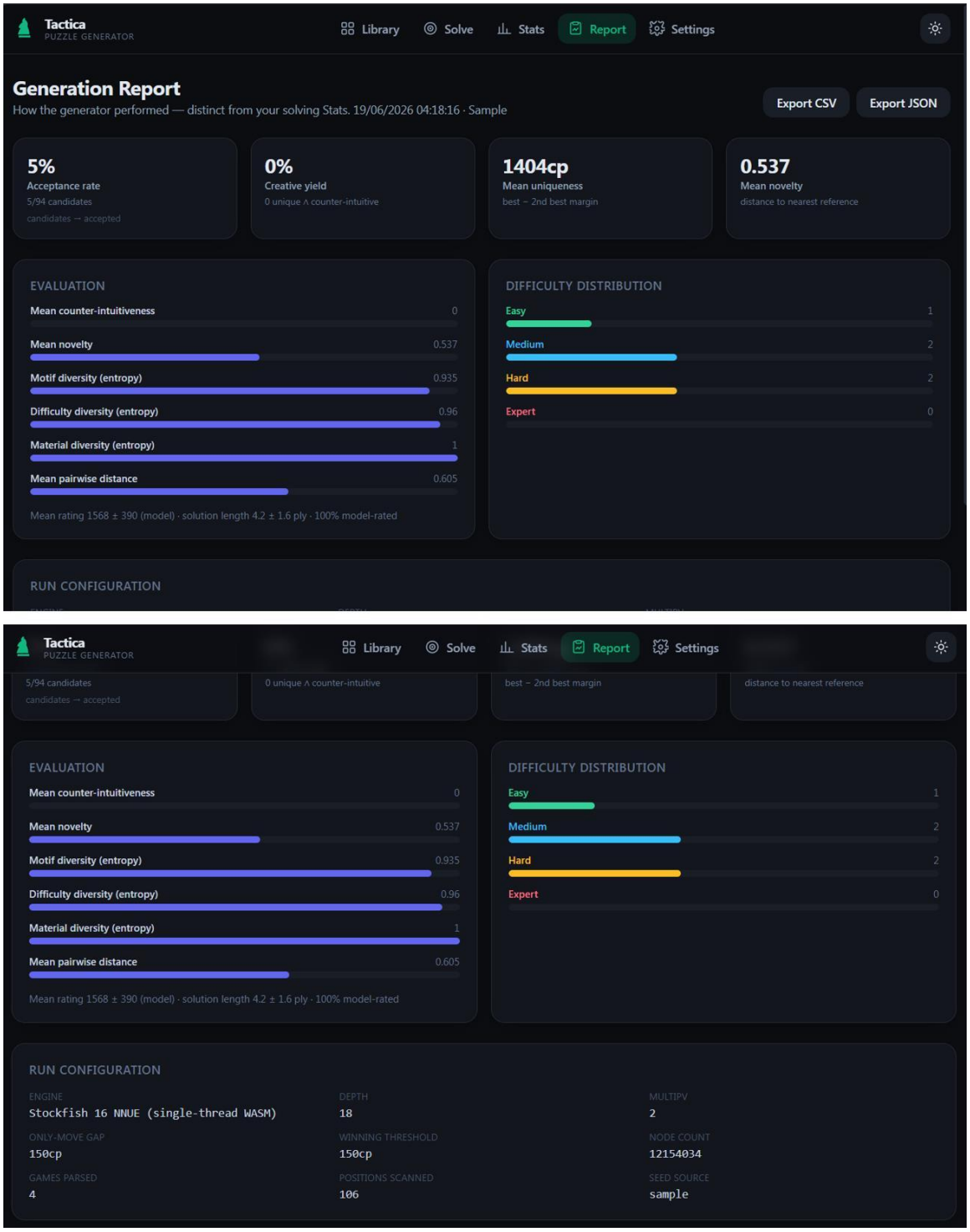


Figure 3.5: The Generation Report: the acceptance rate and creative yield, the evaluation and difficulty-distribution panels, and the JSON and CSV export buttons (top), together with the full run configuration that produced the run (bottom).

3.10.5 Settings

The Settings screen holds the controls that affect generation and solving: the engine's deep-verify depth, the generation mode (extraction, perturbation, or both), and the board orientation. Maintenance actions to clear the generated library and to reset the solving progress live here as

well. The configuration constants are centralised, so a change made here carries through verification, scoring, and the run report consistently.

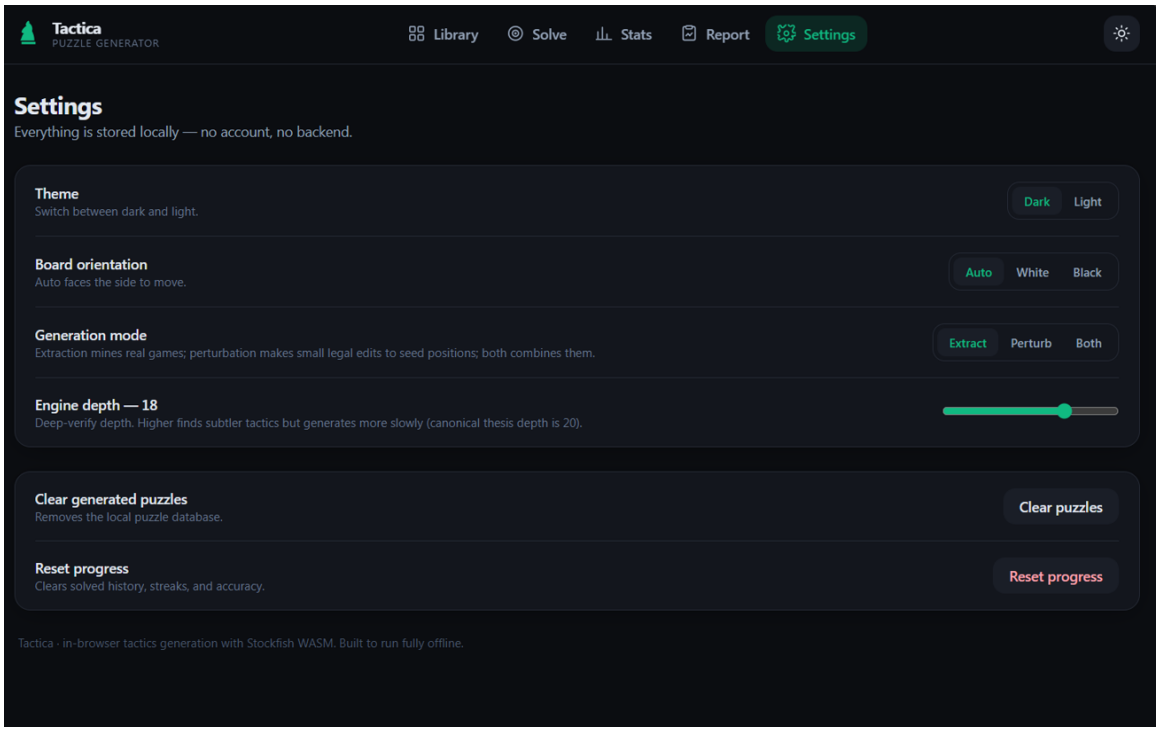


Figure 3.6: The Settings screen: the theme toggle, board orientation, generation mode (extraction, perturbation, or both), the engine-depth control, and the clear-puzzles and reset-progress actions.

3.11 Storage and Export

All persistent data lives in the browser, behind the namespaced, schema-versioned localStorage wrapper of Section 3.3.4. Three kinds of data are stored. The generated puzzle library comes first: each record holds the position, the full solution line with the per-move uniqueness margins, the detected motifs, the scores, the estimated difficulty, and the engine settings that produced it. The solving progress, including which puzzles are solved and the streak counters, and the user settings make up the rest. Schema migration is non-destructive, so a library generated by an earlier version keeps working after an update.

For export, the Generation Report can be written to a downloaded file in two formats: a JSON object with the full report, and a flat key-and-value CSV suited to a spreadsheet or a thesis appendix. The export happens entirely in the browser, by building the text in memory and triggering a download; no server is involved. What is exported is the run report, not the full Lichess puzzle-database schema, and producing a drop-in Lichess CSV of the puzzles themselves is left as a straightforward extension.

3.12 Tests

The application is checked at several levels. An automated test suite that runs without a browser covers the engine-free logic, and a measurement run with the real engine exercises the engine-dependent behaviour. To keep the automated tests fast and repeatable, the decision logic is tested against deterministic engine doubles, small stand-ins that return scripted analysis; the behaviour of the real engine is shown by the measured run of Section 3.13. The suite runs without a browser on any machine with Node.js (npm run test:run), and the measurement harness behind Section 3.13 runs with npm run evaluate; Appendix A lists every test and its result. The categories are summarised here.

3.12.1 Build Test

The production build is checked by compiling the application with the build tool. When the build succeeds, every module imports correctly, no syntax or dependency errors remain, and the application bundles into the static assets that are actually deployed.

3.12.2 Module Tests

Unit tests take the engine-free modules in isolation: FEN handling and PGN parsing, the only-move uniqueness calculation against mocked engine outputs, the novelty distances, the diversity entropies, the difficulty-feature extraction and the difficulty model's forward pass, and the run-report and CSV generation. These modules are pure functions, so the tests are deterministic and fast, and they pin down the behaviour that the rest of the system depends on.

3.12.3 Known Puzzle Tests

Acceptance tests feed the pipeline positions whose intended solution is known in advance, such as a forced mate-in-one, a forced mate-in-two reached through the recursive uniqueness test, and a clear material win, and confirm that each is accepted and that the expected key move is found. Negative tests do the complement. They feed positions that should be rejected, a quiet equal middlegame, a position with two equally good replies, an already-won position, a dead-drawn endgame, and an unanalysable FEN, and confirm that each is rejected for the expected reason.

3.12.4 Reproducibility Tests

A reproducibility test runs the same positions twice under the same fixed settings and confirms that the two runs return identical decisions, counts, identifiers, solutions, and

searched-node totals. The determinism rests on the engine being pinned to a single thread; a multi-threaded search would give no such guarantee.

3.12.5 Performance Tests

The run harness measures performance. It records the number of positions examined, the number accepted and rejected, the average analysis time per position, the total execution time, and the total searched nodes. Section 3.13 reports these measurements.

3.13 Results

The figures below come from a measured run of the application. The run was executed on an Intel Core i5-8350U machine with 17 GB of memory under Windows, driving the real Stockfish engine through a headless browser. The engine was fixed at a search depth of twelve plies, a single thread, and MultiPV set to two, with the thresholds left at their configured values. Twelve plies keep a run of this size within a practical time on this machine; the same procedure reproduces at the configured default of sixteen or at the canonical depth of twenty. The difficulty ratings come from the learned model validated in Section 3.9. The automated test suite behind the implementation is summarised first, the generation figures after it.

Table 3.6: Automated test results, by suite.

Test suite	Tests	Passed	Failed
Unit (pure logic, legality, reconciliation)	27	27	0
Integration (end-to-end pipeline)	1	1	0
Acceptance (known puzzles accepted)	3	3	0
Negative (non-puzzles rejected, by reason)	6	6	0
Reproducibility (identical repeated runs)	1	1	0
Total	38	38	0

All thirty-eight automated tests pass. The generation run processed a sample of the Lichess Elite database [82]: 2,000 candidate positions were drawn, and 1,969 of them were examined after the forcing-move screen and de-duplication. Tables 3.7 to 3.11 report the outcome.

Table 3.7: Number of analysed positions and outcome.

Outcome	Count	Percentage
Positions examined	1,969	100%
Accepted as puzzle	40	2.0%
Rejected	1,929	98.0%

Table 3.8: Reasons for rejection (deep-verification stage).

Rejection reason	Count
Invalid or unanalysable FEN	0
No clearly best move	1,878
Insufficient advantage / not a real tactic	39
Already decided (too trivial)	12
Engine error	0

The forcing-move screen had already removed 399 quiet positions before deep verification; these are not part of the 1,969 examined and therefore do not appear among the rejection reasons in Table 3.8. The deep-stage reasons in Table 3.8 (1,878 with no clearly best move, 39 with an insufficient advantage, and 12 already decided) sum exactly to the 1,929 rejected, and the 40 accepted plus the 1,929 rejected equal the 1,969 examined. A low acceptance rate is expected for games between strong players, where one-sided forced tactics are uncommon, and almost every rejection comes from the absence of a uniquely best move rather than from any engine failure.

Table 3.9: Difficulty distribution of the accepted puzzles.

Difficulty band	Count	Percentage
Easy	17	42.5%
Medium	16	40.0%
Hard	7	17.5%
Expert	0	0.0%

Table 3.10: Timing and throughput.

Metric	Value
Average analysis time per position	550 ms
Total positions examined	1,969
Total execution time	1,091 s (about 18 minutes)
Accepted puzzles produced	40
Total nodes searched	30,435,701
Engine throughput	about 28,100 nodes per second

Table 3.11 shows a selection of the accepted puzzles, each with its FEN, the side to move, the solution in algebraic notation, the motifs detected, and the estimated difficulty.

Table 3.11: Selected accepted puzzles.

FEN	Side	Solution (SAN)	Motifs	Difficulty
6k1/p2R1pp1/Q3p2p/3pPq2/5P2/6BP/P5P1/1r4bK b - - 6 32	Black	Bf2+ Kh2 Rh1+ Kxh1 Qb1+ Qf1 Qxf1+	discovered attack, skewer, sacrifice	Hard (1898)
5r1r/1b2qpk1/p3pR2/3pP1Pp/1p1n1Q2/1P5B/P1P4P/5RK1 w - - 0 34	White	Qxd4	winning material	Medium (1328)
8/1p6/p1pK4/2Pb4/PP1P4/7k/5rR1/8 w - - 1 54	White	Rxf2	winning material	Easy (981)

Figures 3.7 and 3.8 show two of these puzzles as board diagrams, with the side to move, the key move, and a short explanation. In Figure 3.7 it is Black to move. The position looks quiet, but 1...Bf2+ begins a forced mating sequence (2.Kh2 Rh1+ 3.Kxh1 Qb1+ and mate follows); the model rates the puzzle Hard because the decisive first move is a quiet sacrifice that only deeper search rewards. In Figure 3.8 it is White to move, and 1.Qxd4 wins a piece outright, a short forcing tactic the model rates Medium.

Figure 3.7 — Black to move; 1...Bf2+ forces mate (Hard, rated ~1898)

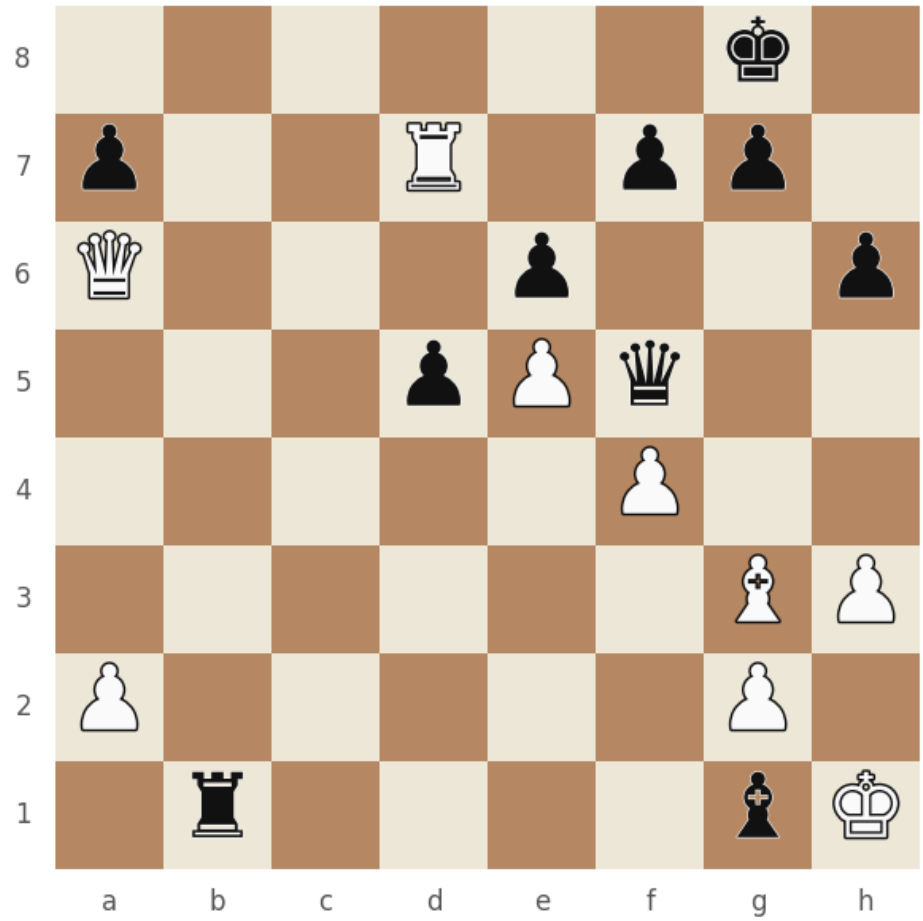


Figure 3.7: A Hard puzzle (Black to move): the quiet 1...Bf2+ opens a forced mating sequence, which is why the learned model rates it Hard.

Figure 3.8 — White to move; 1.Qxd4 wins material (Medium, rated ~1328)

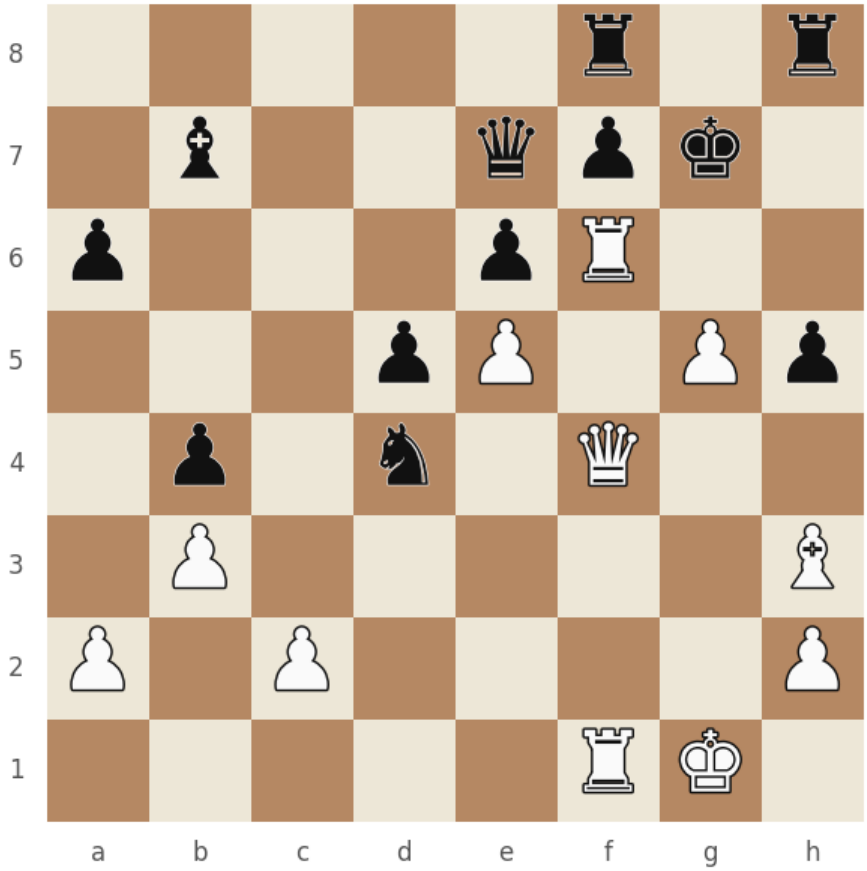


Figure 3.8: A Medium puzzle (White to move): 1.Qxd4 wins a piece by a short forcing tactic, rated Medium by the model.

3.14 Limitations

The implemented system has several limitations.

- Its verdicts depend on the engine version, the search depth, and the uniqueness margin; a position may qualify under one configuration and not under another, and verification is reproducible but not absolute.
- The run reported here used a fixed depth of twelve plies rather than the canonical twenty, to keep the run practical on the test machine.
- Novelty is measured against a small reference sample, not the full Lichess puzzle database, so reported novelty is indicative rather than corpus-wide.
- Endgame-tablebase verification is not bundled; few-piece positions are verified by engine search like any other position.

- The difficulty estimate is approximate: the learned model is trained and tested on real Lichess ratings, but its mean absolute error of about 309 rating points makes it reliable for banding, not as an exact rating.
- No human evaluation of the generated puzzles was carried out; the human study is left as future validation.
- The engine is the single-threaded in-browser build, so deep analysis is slower than with a native multi-threaded engine, which bounds the depth and the number of positions a run can process.

3.15 Conclusion

Tactica implements the Chapter 2 core in full, and the measurements bound what it achieves. All thirty-eight automated tests pass. The measured run accepted 40 puzzles from 1,969 examined positions, at about 550 milliseconds of analysis per position, with 1,878 of the 1,929 rejections caused by the absence of a uniquely best move. The learned difficulty model, validated on 6,000 held-out Lichess puzzles, reaches a mean absolute error of about 309 rating points and a Pearson correlation of 0.72; its band prediction is exact 51.8 percent of the time and within one band 94.6 percent of the time. The limits recorded in Section 3.14, the low extraction acceptance rate, the sample-based novelty reference, and a residual error too large to rate a single puzzle with confidence, carry into the conclusions. The general conclusion weighs these results against the objectives of Chapter 2.

General Conclusion

Two linked problems in the automatic creation of chess puzzles set the agenda for this dissertation: how to generate and verify sound, uniquely solvable tactical positions without human intervention, and how to estimate the difficulty of such a position before any player has attempted it. Its answer is a working system: an engine-based pipeline that keeps generation and evaluation apart and treats the evaluator as the object of greatest care.

Chapter 1 left two findings that shaped the design: the field has no agreed, operational definition of a good puzzle, with the most ambitious aesthetic claims concentrated in a single unreplicated research programme, and difficulty estimation without solving history remains open. Chapter 2 answered with a methodology: the research questions paired one-to-one with experiments and metrics, a pinned engine configuration, and an explicit three-tier scope whose line, drawn in advance, made clear at reporting time which claims the evidence covered. Chapter 3 built the core as Tactica and measured it. The suite of thirty-eight automated tests passes in full, and a generation run on real games from the Lichess Elite database examined 1,969 positions and accepted 40, with 1,878 of the 1,929 rejections caused by the absence of a uniquely best move, the outcome games between strong players should produce.

The difficulty estimator received the most stringent test of any part of the system, because it makes a prediction that can be checked against ground truth. Trained and evaluated on real ratings from the Lichess Open Puzzle Database, with a held-out test set it had never seen, the learned model reached a mean absolute error of about 309 rating points and a Pearson correlation of about 0.72 with the human difficulty labels; it placed a puzzle in the correct difficulty band 51.8 percent of the time and within one band 94.6 percent of the time, and it outperformed both baselines, the global-mean predictor (MAE 457) and the rule-based heuristic (MAE 468). The result cuts two ways. As an ordering of puzzles by difficulty, the model tracks human experience well; as an absolute predictor for any single puzzle, an error of a few hundred rating points remains large. The result is real and useful, and it comes with those limits attached.

The other evaluation axes told a consistent story. The accepted puzzles were uniquely solvable by construction, and the recorded uniqueness margins confirmed that the engine's preferred move exceeded its alternatives by more than the 150 cp margin at each step of the solution. Counter-intuitiveness, measured from the depth at which the key move first becomes the engine's choice, separated the routine captures from the surprising resources, and the novelty and diversity measures gave a compact picture of how varied a generated set was. None

of these axes is a finished instrument; the novelty figure in particular is limited by the small reference set against which it is computed. Together, though, they show that a puzzle can be described by defined, checkable numbers instead of a single unexplained verdict. It is in this sense that the evaluator, and not the generator, is the centre of the work.

Taken together, the work makes a small number of concrete contributions. It delivers a reproducible, fully engine-based framework that unifies three things usually treated separately: the generation of candidate positions, the verification of their soundness and uniqueness, and the estimation of their difficulty. It shows that uniqueness, the property most often asserted but least often checked, can be tested directly and at every step of a solution, by running a two-line engine search recursively along the forced line. It provides a history-free difficulty estimator validated against real human ratings rather than synthetic labels, and it reports that estimator's accuracy in full, error included. And it embodies all of this in an open, self-contained application, so the method and its results can be rerun by anyone. The contribution is a demonstration, carried out in the open, that the core of the problem can be handled rigorously.

One further contribution is methodological. Every figure reported in the dissertation comes from a run that can be repeated on a stated machine with a fixed engine, a fixed search depth, and a fixed random seed, and the reconciliation between the positions examined, accepted, and rejected was checked explicitly. Where a part of the design could not be measured within the project, it was labelled as such. In an area where striking results have sometimes proved hard to reproduce, that discipline is itself a result.

Beyond the research contribution, the system has a practical side worth stating. It runs entirely in the browser and needs neither a server nor an account, which lowers the barrier to generating verified tactical material to almost nothing. A teacher, a club player, or a smaller platform without the resources of the largest sites can produce a fresh, sound set of puzzles from their own games, filter them by motif or difficulty, and inspect the evaluation of the whole run. The property that makes the work reproducible for a researcher, an engine and a code base that are fixed and local, is the same property that makes it usable for a practitioner. Tactica was kept open and self-contained so that the people it is meant for can run and inspect it themselves.

The work has clear limitations. The acceptance rate on elite games is low, which is appropriate for soundness but makes extraction alone an inefficient source of puzzles; the novelty measure is computed against a small reference sample instead of the full puzzle database, so the reported novelty figures are indicative, not corpus-wide; and the difficulty model, though it beats its baselines, still carries an absolute error too large to rate a single puzzle with confidence. Several parts of the wider design were, by deliberate choice, not

implemented or only partly explored: there is no endgame-tablebase backend, so every position is judged by engine search; directed generation toward a chosen theme, difficulty, or material balance was only sketched; the agreement of the quality score with human judgement was not put to a formal human study; and the fully learned, reinforcement-trained generator remains at the design stage. These boundaries follow the three-tier scope set out in Chapter 2, and naming them precisely is part of the contribution.

Each of these limits is also a direction for further work. The most immediate is a corpus-wide novelty benchmark, which would measure each generated puzzle against the full Lichess database instead of a sample, and so turn the novelty score from an indication into a proper measure. A formal human study is the natural next step for the quality score, both to test whether the measurable axes agree with how players actually rate puzzles and to calibrate the difficulty model on human solving times as well as ratings. On the generation side, the perturbation strategy could grow into a directed generator that conditions on a target theme or difficulty band, and beyond that into the learned generator described as future work, trained against the engine as a reward signal with explicit safeguards for novelty and diversity so that it does not collapse onto a handful of easy patterns. Integrating an endgame tablebase would extend exact verification into the endgame, and running the pipeline over larger and more varied game collections would test how well its behaviour holds up at scale. None of this requires abandoning the present design; each item builds on the verified core that this dissertation provides.

Looking further ahead, the framework invites questions that reach beyond chess. The generate-and-test pattern at its core, a permissive generator paired with a strict, automatic verifier, applies wherever candidate artefacts can be checked against a reliable oracle, and the discipline of validating a history-free difficulty estimate against real human performance matters to any system that must match problems to learners. Within chess itself, the most valuable extension may well be the human study that was left for future work, since that is the step that would connect the measurable axes used here to the judgements of real solvers and teachers. The present dissertation does not take those steps, but it is built so that they can be taken without starting again.

No single number carries the argument of this dissertation. Chess-puzzle generation is an area in which ambition has often outrun evidence, and its most interesting questions, about beauty, creativity, and difficulty, are also the hardest to pin down. This work therefore confined itself to properties that can be defined and checked, fixed the engine so that results can be reproduced, validated against real data, and recorded what was and was not achieved. Under

the fixed Stockfish configuration used in the experiments, the accepted puzzles are sound and uniquely solvable; the predicted difficulties track what real players experience; and anyone can run, inspect, and repeat the whole system. The dissertation does not claim to have solved the problem. It shows that this much of it can be built, measured, and checked.

Bibliography

[1] Feng, X., Veeriah, V., Chiam, M., Dennis, M., Pachauri, R., Tumiel, T., Barbero, F., Obando-Ceron, J., Shi, J., Singh, S., Hou, S., Tomašev, N., and Zahavy, T. (2025). Generating Creative Chess Puzzles. arXiv:2510.23881. Google DeepMind. https://arxiv.org/abs/2510.23881

[2] Veeriah, V. et al. (2025). Evaluating In Silico Creativity: An Expert Review of AI Chess Compositions. arXiv:2510.23772. Creative AI Track, NeurIPS 2025. https://arxiv.org/abs/2510.23772

[3] Ruoss, A., Delétang, G., Medapati, S., Grau-Moya, J., Wenliang, L. K., Catt, E., Reid, J., Lewis, C. A., Veness, J., and Genewein, T. (2024). Amortized Planning with Large-Scale Transformers: A Case Study on Chess (arXiv v1 title: Grandmaster-Level Chess Without Search). NeurIPS 2024. arXiv:2402.04494. https://arxiv.org/abs/2402.04494

[4] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

[5] McGrath, T., Kapishnikov, A., Tomašev, N., Pearce, A., Wattenberg, M., Hassabis, D., Kim, B., Paquet, U., and Kramnik, V. (2022). Acquisition of chess knowledge in AlphaZero. PNAS, 119(47), e2206625119. https://doi.org/10.1073/pnas.2206625119

[6] Tomašev, N., Paquet, U., Hassabis, D., and Kramnik, V. (2020). Assessing Game Balance with AlphaZero: Exploring Alternative Rule Sets in Chess. arXiv:2009.04374. (Peer-reviewed version: Reimagining Chess with AlphaZero, Communications of the ACM, 65(2), 60–66, 2022.) https://arxiv.org/abs/2009.04374

[7] McIlroy-Young, R., Sen, S., Kleinberg, J., and Anderson, A. (2020). Aligning Superhuman AI with Human Behavior: Chess as a Model System. In Proc. 26th ACM SIGKDD (KDD '20), 1677–1687. https://doi.org/10.1145/3394486.3403219

[8] McIlroy-Young, R., Wang, R., Sen, S., Kleinberg, J., and Anderson, A. (2021). Detecting Individual Decision-Making Style: Exploring Behavioral Stylometry in Chess. In Advances in Neural Information Processing Systems 34 (NeurIPS 2021). arXiv:2208.01366. https://arxiv.org/abs/2208.01366

[9] Iqbal, A. (2006). Computing the Aesthetics of Chess. In Proc. AAAI Workshop on Computational Aesthetics (Technical Report WS-06-04), 16–23. AAAI Press. https://cdn.aaai.org/Workshops/2006/WS-06-04/WS06-04-004.pdf

[10] Iqbal, A., and Yaacob, M. (2008). Theme Detection and Evaluation in Chess. ICGA Journal, 31(2), 97–109. https://doi.org/10.3233/ICG-2008-31204

[11] Iqbal, A., Guid, M., Colton, S., Krivec, J., Azman, S., and Haghighi, B. (2016). The Digital Synaptic Neural Substrate: A New Approach to Computational Creativity. SpringerBriefs in Cognitive Computation. Springer. (Preprint arXiv:1507.07058.) https://doi.org/10.1007/978-3-319-28079-0

[12] Iqbal, A. (2017). A Computer Composes A Fabled Problem: Four Knights vs. Queen. arXiv:1709.00931. https://arxiv.org/abs/1709.00931

[13] Iqbal, A. (2018). Improving Automatic Chess Problem Composition by Emphasizing Conventions. Journal of Fundamental and Applied Sciences, 10(6S), 1744–1756.

[14] Margulies, S. (1977). Principles of Beauty. Psychological Reports, 41(1), 3–11. https://doi.org/10.2466/pr0.1977.41.1.3

[15] World Federation for Chess Composition (2020). Codex for Chess Composition. WFCC official document. https://www.wfcc.ch/

[16] Boden, M. A. (2004). The Creative Mind: Myths and Mechanisms (2nd ed.). Routledge. (1st ed. 1990.)

[17] Wiggins, G. A. (2006). A preliminary framework for description, analysis and comparison of creative systems. Knowledge-Based Systems, 19(7), 449–458. https://doi.org/10.1016/j.knosys.2006.04.009

[18] Ritchie, G. (2007). Some Empirical Criteria for Attributing Creativity to a Computer Program. Minds and Machines, 17(1), 67–99. https://doi.org/10.1007/s11023-007-9066-2

[19] Colton, S., and Wiggins, G. A. (2012). Computational Creativity: The Final Frontier? In ECAI 2012, Frontiers in AI and Applications, vol. 242, 21–26. IOS Press. https://doi.org/10.3233/978-1-61499- 098-7-21

[20] Colton, S. (2008). Creativity Versus the Perception of Creativity in Computational Systems. In Creative Intelligent Systems: AAAI Spring Symposium (TR SS-08-03), 14–20. AAAI Press.

[21] Lehman, J., and Stanley, K. O. (2011). Abandoning Objectives: Evolution Through the Search for Novelty Alone. Evolutionary Computation, 19(2), 189–223. https://doi.org/10.1162/EVCO_a_00025

[22] Mouret, J.-B., and Clune, J. (2015). Illuminating search spaces by mapping elites. arXiv:1504.04909. https://arxiv.org/abs/1504.04909

[23] Pugh, J. K., Soros, L. B., and Stanley, K. O. (2016). Quality Diversity: A New Frontier for Evolutionary Computation. Frontiers in Robotics and AI, 3:40. https://doi.org/10.3389/frobt.2016.00040

[24] Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-Based Procedural Content Generation: A Taxonomy and Survey. IEEE TCIAIG, 3(3), 172–186. https://doi.org/10.1109/TCIAIG.2011.2148116

[25] Shaker, N., Togelius, J., and Nelson, M. J. (2016). Procedural Content Generation in Games. Springer. https://doi.org/10.1007/978-3-319-42716-4

[26] Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A., and Togelius, J. (2018). Procedural Content Generation via Machine Learning (PCGML). IEEE Transactions on Games, 10(3), 257–270. https://doi.org/10.1109/TG.2018.2846639

[27] Liu, J., Snodgrass, S., Khalifa, A., Risi, S., Yannakakis, G. N., and Togelius, J. (2021). Deep learning for procedural content generation. Neural Computing and Applications, 33(1), 19–37. https://doi.org/10.1007/s00521-020-05383-8

[28] Khalifa, A., Bontrager, P., Earle, S., and Togelius, J. (2020). PCGRL: Procedural Content Generation via Reinforcement Learning. In Proc. AIIDE-20, 16(1), 95–101. AAAI Press. https://ojs.aaai.org/index.php/AIIDE/article/view/7416

[29] Risi, S., and Togelius, J. (2020). Increasing generality in machine learning through procedural content generation. Nature Machine Intelligence, 2(8), 428–436. https://doi.org/10.1038/s42256-020-0208-z

[30] De Kegel, B., and Haahr, M. (2020). Procedural Puzzle Generation: A Survey. IEEE Transactions on Games, 12(1), 21–40. https://doi.org/10.1109/TG.2019.2917792

[31] Browne, C., and Maire, F. (2010). Evolutionary Game Design. IEEE TCIAIG, 2(1), 1–16. https://doi.org/10.1109/TCIAIG.2010.2041928

[32] Farrokhi Maleki, M., and Zhao, R. (2024). Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration. In Proc. AIIDE, 20(1), 167–178. AAAI Press. https://doi.org/10.1609/aiide.v20i1.31877

[33] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention Is All You Need. In NeurIPS 2017. arXiv:1706.03762. https://arxiv.org/abs/1706.03762

[34] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Nets. In NeurIPS 2014. arXiv:1406.2661. https://arxiv.org/abs/1406.2661

[35] Ho, J., Jain, A., and Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. In NeurIPS 2020. arXiv:2006.11239. https://arxiv.org/abs/2006.11239

[36] Chang, H., Zhang, H., Jiang, L., Liu, C., and Freeman, W. T. (2022). MaskGIT: Masked Generative Image Transformer. In CVPR 2022, 11315–11325. arXiv:2202.04200. https://arxiv.org/abs/2202.04200

[37] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2022). High-Resolution Image Synthesis with Latent Diffusion Models. In CVPR 2022, 10684–10695. arXiv:2112.10752. https://arxiv.org/abs/2112.10752

[38] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347. https://arxiv.org/abs/1707.06347

[39] Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete Problems in AI Safety. arXiv:1606.06565. https://arxiv.org/abs/1606.06565

[40] Skalse, J., Howe, N. H. R., Krasheninnikov, D., and Krueger, D. (2022). Defining and Characterizing Reward Hacking. In NeurIPS 2022. arXiv:2209.13085. https://arxiv.org/abs/2209.13085

[41] Pan, A., Bhatia, K., and Steinhardt, J. (2022). The Effects of Reward Misspecification: Mapping and Mitigating Misaligned Models. In ICLR 2022. arXiv:2201.03544. https://arxiv.org/abs/2201.03544

[42] Stoiljkovikj, S., Bratko, I., and Guid, M. (2015). A Computational Model for Estimating the Difficulty of Chess Problems. In Proc. Third Annual Conference on Advances in Cognitive Systems (ACS 2015).

[43] Guid, M., and Bratko, I. (2006). Computer Analysis of World Chess Champions. ICGA Journal, 29(2), 65–73. https://doi.org/10.3233/ICG-2006-29203

[44] van Kreveld, M., Löffler, M., and Mutser, P. (2015). Automated puzzle difficulty estimation. In 2015 IEEE Conference on Computational Intelligence and Games (CIG), 415–422. https://doi.org/10.1109/CIG.2015.7317913

[45] Schütt, A., Huber, T., and André, E. (2024). Estimating Chess Puzzle Difficulty Without Past Game Records Using a Human Problem-Solving Inspired Neural Network Architecture. In 2024 IEEE International Conference on Big Data (BigData), 8396–8402. https://doi.org/10.1109/BigData62323.2024.10826087

[46] Miłosz, S., and Kapusta, P. (2024). Predicting Chess Puzzle Difficulty with Transformers. arXiv:2410.11078 (IEEE BigData 2024 Cup). https://arxiv.org/abs/2410.11078

[47] Omori, M., and Tadepalli, P. (2024). Chess Rating Estimation from Moves and Clock Times Using a CNN-LSTM. In Computers and Games (CG 2024), Springer LNCS. arXiv:2409.11506. https://arxiv.org/abs/2409.11506

[48] Lichess (2020–present). Lichess Open Puzzle Database. https://database.lichess.org/

[49] Lichess (2020). New Puzzles are here! Lichess blog (describes the puzzle-generation pipeline and the ">100 years of CPU time" estimate). https://lichess.org/blog/X-S6gRUAAGjNX4ki/new-puzzles- are-here

[50] Fiekas, N. ("ornicar" et al.) (2020–present). lichess-puzzler (puzzle generator source; only-move criterion). https://github.com/ornicar/lichess-puzzler

[51] The Stockfish Community (2008–present). Stockfish: A Strong Open-Source Chess Engine. https://stockfishchess.org/

[52] The LCZero Authors (2018–present). Leela Chess Zero (Lc0). https://lczero.org/

[53] Thompson, K. (1986). Retrograde Analysis of Certain Endgames. ICCA Journal, 9(3), 131–139. https://doi.org/10.3233/ICG-1986-9302

[54] Nalimov, E. V., Haworth, G. McC., and Heinz, E. A. (2000). Space-Efficient Indexing of Chess Endgame Tables. ICGA Journal, 23(3), 148–162.

[55] de Man, R. (2013). Syzygy Endgame Tablebases. Software and documentation. https://github.com/syzygy1/tb

[56] KnowledgePit / IEEE (2024). IEEE BigData 2024 Cup: Predicting Chess Puzzle Difficulty. Competition (4.2M Lichess puzzles; Glicko-2 difficulty labels). https://knowledgepit.ai/predicting- chess-puzzle-difficulty/

[57] Smith, G., and Whitehead, J. (2010). Analyzing the Expressive Range of a Level Generator. In Proc. 2010 Workshop on Procedural Content Generation in Games (PCGames '10). ACM. https://doi.org/10.1145/1814256.1814260

[58] Summerville, A. (2018). Expanding Expressive Range: Evaluation Methodologies for Procedural Content Generation. In Proc. AIIDE, 14(1), 116–122. AAAI Press. https://doi.org/10.1609/aiide.v14i1.13012

[59] Yannakakis, G. N., and Togelius, J. (2018). Artificial Intelligence and Games. Springer. https://doi.org/10.1007/978-3-319-63519-4

[60] Jordanous, A. (2012). A Standardised Procedure for Evaluating Creative Systems: Computational Creativity Evaluation Based on What it is to be Creative. Cognitive Computation, 4(3), 246–279. https://doi.org/10.1007/s12559-012-9156-1

[61] Lamb, C., Brown, D. G., and Clarke, C. L. A. (2018). Evaluating Computational Creativity: An Interdisciplinary Tutorial. ACM Computing Surveys, 51(2), Article 28. https://doi.org/10.1145/3167476

[62] Gobet, F., and Simon, H. A. (1996). The Roles of Recognition Processes and Look-Ahead Search in Time-Constrained Expert Problem Solving: Evidence From Grandmaster-Level Chess. Psychological Science, 7(1), 52–55. https://doi.org/10.1111/j.1467-9280.1996.tb00666.x

[63] Charness, N., Tuffiash, M., Krampe, R., Reingold, E., and Vasyukova, E. (2005). The role of deliberate practice in chess expertise. Applied Cognitive Psychology, 19(2), 151–165. https://doi.org/10.1002/acp.1106

[64] Pelánek, R. (2016). Applications of the Elo Rating System in Adaptive Educational Systems. Computers & Education, 98, 169–179. https://doi.org/10.1016/j.compedu.2016.03.017

[65] Glickman, M. E. (1999). Parameter Estimation in Large Dynamic Paired Comparison Experiments. Journal of the Royal Statistical Society: Series C (Applied Statistics), 48(3), 377–394. https://doi.org/10.1111/1467-9876.00159

[66] Jarušek, P., and Pelánek, R. (2012). Analysis of a Simple Model of Problem Solving Times. In Intelligent Tutoring Systems (ITS 2012), LNCS 7234, 379–388. Springer. https://doi.org/10.1007/978-3-642-30950-2_49

[67] Hlywa, J. (2014–present). chess.js: a JavaScript chess library for move generation, validation, and FEN/PGN handling. https://github.com/jhlywa/chess.js

[68] Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. Educational and Psychological Measurement, 20(1), 37–46. https://doi.org/10.1177/001316446002000104

[69] Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. Biometrics Bulletin, 1(6), 80–83. https://doi.org/10.2307/3001968

[70] Taylor, J., and Parberry, I. (2011). Procedural Generation of Sokoban Levels. In Proc. 6th North American Conference on Intelligent Games and Simulation (GAMEON-NA 2011), 5–12. EUROSIS. https://ianparberry.com/pubs/GAMEON-NA_METH_03.pdf

[71] Khalifa, A., Perez-Liebana, D., Lucas, S. M., and Togelius, J. (2016). General Video Game Level Generation. In Proc. Genetic and Evolutionary Computation Conference (GECCO '16), 253–259. ACM. https://doi.org/10.1145/2908812.2908920

[72] Gravina, D., Khalifa, A., Liapis, A., Togelius, J., and Yannakakis, G. N. (2019). Procedural Content Generation through Quality Diversity. In 2019 IEEE Conference on Games (CoG), 1–8. https://doi.org/10.1109/CIG.2019.8848053

[73] Cully, A., and Demiris, Y. (2018). Quality and Diversity Optimization: A Unifying Modular Framework. IEEE Transactions on Evolutionary Computation, 22(2), 245–259. https://doi.org/10.1109/TEVC.2017.2704781

[74] Pelánek, R. (2014). Difficulty Rating of Sudoku Puzzles: An Overview and Evaluation. arXiv:1403.7373. https://arxiv.org/abs/1403.7373

[75] Corbett, A. T., and Anderson, J. R. (1994). Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge. User Modeling and User-Adapted Interaction, 4(4), 253–278. https://doi.org/10.1007/BF01099821

[76] Pelánek, R. (2017). Bayesian Knowledge Tracing, Logistic Models, and Beyond: An Overview of Learner Modeling Techniques. User Modeling and User-Adapted Interaction, 27(3–5), 313–350. https://doi.org/10.1007/s11257-017-9193-2

[77] Pease, A., and Colton, S. (2011). On Impact and Evaluation in Computational Creativity: A Discussion of the Turing Test and an Alternative Proposal. In Proc. AISB '11: Computing and Philosophy, 15–22. AISB. ISBN 978-1-908187-03-1.

[78] Gallotta, R., Todd, G., Zammit, M., et al. (2024). Large Language Models and Games: A Survey and Roadmap. IEEE Transactions on Games. arXiv:2402.18659. https://arxiv.org/abs/2402.18659

[79] Sudhakaran, S., González-Duque, M., Glanois, C., et al. (2023). MarioGPT: Open-Ended Text2Level Generation through Large Language Models. In NeurIPS 2023. arXiv:2302.05981. https://arxiv.org/abs/2302.05981

[80] Khalifa, A., Gallotta, R., Barthet, M., et al. (2025). The Procedural Content Generation Benchmark: An Open-source Testbed for Generative Challenges in Games. In Proc. Foundations of Digital Games (FDG 2025). arXiv:2503.21474. https://arxiv.org/abs/2503.21474

[81] Stock, M., Van Hauwermeiren, D., and De Baets, B. (2025). Quality-Diversity Methods for the Modern Data Scientist. WIREs Computational Statistics, 17(4), e70047. https://doi.org/10.1002/wics.70047

[82] nikonoel (2020–present). Lichess Elite Database. A monthly compilation of games between higher-rated Lichess players (2400+ versus 2200+). https://database.nikonoel.fr/

Appendix A: Test Evidence

A.1 How to run the tests

The tests reproduce on any machine with Node.js installed. From the project root, the three commands below install the dependencies, run the automated test suite, and run the measurement harness that produces the result tables of Chapter 3.

```
# 1. install dependencies
npm install

# 2. run the automated test suite (no browser needed)
npm run test:run

# 3. run the measurement harness (drives the real engine, prints the tables)
npm run evaluate
```

The second command runs the thirty-eight automated tests listed in Section A.3 and prints a pass or fail line for each. The third command runs the generator on the bundled sample with the real Stockfish engine and prints the figures reported in Chapter 3. To keep a permanent record of a run, the suite can also write a readable log, a machine-readable report, and a coverage report:

```
npx vitest run --reporter=verbose > test-log.txt 2>&1
npx vitest run --reporter=junit --outputFile=test-results.xml
npx vitest run --coverage
```

A.2 Run environment

The environment recorded for the run reported in Chapter 3 is listed below, so the figures can be reproduced under the same conditions.

- Machine: Intel Core i5-8350U at 1.70 GHz (eight logical cores), 17 GB of memory, Windows (build 10.0.26200).
- Runtime: Node v24.16; the engine was driven through a headless Chromium browser.
- Engine: Stockfish 16 NNUE, single-thread WebAssembly build; fixed search depth 12; MultiPV 2.
- Thresholds (unchanged from the source configuration): only-move margin 150 cp, winning margin 150 cp, already-won cap 900 cp.
- Scaled generation run: 2,000 candidate positions drawn from the Lichess Elite database (June 2020), 1,969 examined after de-duplication, 40 accepted as puzzles.

A.3 Automated test results

All thirty-eight automated tests pass. The tables below list every test case and its result, grouped by suite as produced by the test runner. The decision logic is tested against deterministic engine doubles so the tests are fast and repeatable; the behaviour of the real engine is shown by the measured run in Chapter 3.

Table A.1: Unit tests — test/unit.test.js (26 of 26 passed).

#	Test	Result
1	cpEquiv: passes cp through, folds mate into large signed values	PASS
2	bandFor: maps ratings to bands	PASS
3	fenId is stable across runs (same FEN same id; different different)	PASS
4	screenVerdict: accepts a winning unique move	PASS
5	screenVerdict: rejects quiet / non-unique as noClearBestMove	PASS
6	screenVerdict: rejects already-won as alreadyDecided	PASS
7	screenVerdict: accepts forced mate	PASS
8	evaluateUniqueness: mate-in-1 is unique and flagged	PASS
9	evaluateUniqueness: clear margin unique, tiny margin not unique	PASS
10	evaluateUniqueness: uses the configured onlyMoveGap threshold (150)	PASS
11	pgnParser: parses sample into legal games	PASS
12	pgnParser: gameChunks stops at maxGames	PASS
13	detectMotifs: detects mate-in-1	PASS
14	detectMotifs: detects a knight fork	PASS
15	detectMotifs: no false sacrifice on an unrelated capture	PASS
16	computeDifficulty: returns a valid band and rating	PASS
17	computePieces: records the solver piece	PASS
18	novelty: identical near 0, distinct higher, empty reference 1	PASS
19	novelty: board / solution distance primitives	PASS
20	computeDiversity: entropies in [0,1] and counts correct	PASS
21	extractFeatures: deterministic fixed-length vector	PASS
22	buildReport: counts and rates are correct	PASS
23	reportToCsv: CSV round-trips header and a known metric	PASS
24	perturbations: every output FEN legal and distinct from seed	PASS
25	difficultyModel.predict: rating + band + uncertainty for a known vector	PASS
26	tablebase eligibility: counts pieces, gates on seven or fewer	PASS

Table A.2: Reconciliation test — test/reconciliation.test.js (1 of 1 passed).

#	Test	Result
27	accepted + sum(deep rejections) == examined, and candidate / pre-screen counts balance	PASS

Table A.3: Integration test — test/integration.test.js (1 of 1 passed).

#	Test	Result
28	generatePuzzles end-to-end: legal solutions, ends on solver move, uniquenessMargin length correct, valid band, reconciled counts, timing populated	PASS

Table A.4: Acceptance tests — test/acceptance.test.js (3 of 3 passed).

#	Test	Result
29	accepts a mate-in-1 with the correct key move	PASS
30	accepts a clear material win (free queen) with the correct key move	PASS
31	accepts a forced mate-in-2 via recursive uniqueness	PASS

Table A.5: Negative / rejection tests — test/negative.test.js (6 of 6 passed). Each asserts the expected rejection code.

#	Test	Expected code	Result
32	quiet equal middlegame	noClearBestMove	PASS
33	two equally good replies (not unique)	noClearBestMove	PASS
34	already-won (+15) position	alreadyDecided	PASS
35	dead-drawn king-versus-king endgame	noClearBestMove	PASS
36	single legal move (king in check, only one escape)	noClearBestMove	PASS
37	invalid / unanalysable FEN	invalidFen	PASS

Table A.6: Reproducibility test — test/reproducibility.test.js (1 of 1 passed).

#	Test	Result
38	two identical runs give identical decisions, counts, ids, solutions and node count	PASS

A.4 Summary

Table A.7: Automated test results by suite.

Suite	Tests	Passed	Failed
Unit (pure logic, legality, reconciliation)	27	27	0
Integration (end-to-end pipeline)	1	1	0
Acceptance (known puzzles accepted)	3	3	0
Negative (non-puzzles rejected, by reason)	6	6	0
Reproducibility (identical repeated runs)	1	1	0
Total	38	38	0

The measured generation run that accompanies these tests is reported in Section 3.13 of Chapter 3: 1,969 positions examined from the Lichess Elite sample, 40 accepted as puzzles, the remainder rejected mostly for the absence of a uniquely best move, with an average analysis

time of about 550 milliseconds per position. The difficulty model used for those ratings is validated in Section 3.9.

الجمهورية الجزائرية الديمقراطية الشعبية

Republique Algérienne Démocratique Populaire

وزارة التعليم العالي والبحث العلمي

Ministère de l'Enseignement Supérieur de la Recherche Scientifique

Université 20 août 1955 Skikda

جامعة 20 أوت 1955 – سكيكدة

Faculte des Sciences
Departement d'informatique

كلية العلوم
قسم الإعلام الآلي

الرقم: .... / ق/ إ/ أ/ ك.ع/2026

Autorisation de Dépôt de Mémoire de Master

Je soussigné : TOUIL Ghassen

Certifie que l'étudiant(e) : Mokhtari Rezkallah

Spécialité : IA

Ayant soutenu le projet intitulé : Artificial Intelligence Technique for Chess Puzzle Generation and automated Evaluation

A apporté les corrections nécessaires sur son manuscrit de Master.

Signature de l'encadreur