UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

DÉPARTEMENT D'INFORMATIQUE ET D'INGÉNIERIE

Analyse comparative des performances des modèles de langage dans le jeu d'échecs

RAPPORT FINAL

PRÉSENTÉ À

ALAN DAVOUST, SUPERVISEUR

KARIM EL GUEMHIOUI, COORDONNATEUR

COMME EXIGENCE PARTIELLE

DU COURS

INF4173 PROJET SYNTHÈSE

PAR

MUHOZA OLIVIER TUYISHIME

WILLIAM MCALLISTER

25 AVRIL 2025

Contenu

Résumé

Ce travail présente une évaluation de divers Grands Modèles de Langage (LLMs) sur des tâches de résolution de puzzles d'échecs. Nous avons développé un système d'évaluation automatique qui utilise le classement Glicko-2 pour comparer les performances de cinq modèles. L'évaluation a consisté à tester ces modèles sur un ensemble diversifié de puzzles d'échecs issus de la base de données Lichess, répartis selon trois thèmes : *End Game*, *Strategic* et *Tactic*.

Nos résultats révèlent qu'ils ont tous obtenu un classement inférieur au niveau moyen d'environ 1500. Le modèle le plus performant (nvidia/llama-3.1-nemotron-ultra-253b-v1) n'atteignant qu'environ 705 ± 81 points. Nous avons observé que la taille et l'architecture du modèle ne corrélaient pas systématiquement avec la performance, comme le démontre meta/llama-3.1-405b-instruct qui a obtenu des résultats comparables au modèle nemotron malgré des nombres de paramètres différents.

Cette recherche met en lumière les limitations actuelles des LLMs dans les tâches de raisonnement aux échecs et suggère que ni la taille du modèle ni l'entraînement par chaîne de pensée (Chain-of-Thought) ne sont pas des indicateurs de performance dans les jeux stratégiques complexes. Notre méthodologie d'évaluation fournit une base pour les travaux futurs d'évaluation des systèmes LLMs pour des tâches de prise de décision et planification en plusieurs étapes qui sont accessibles par API.

Introduction

Historique et état de la question

Les grands modèles de langage (LLMs) ont démontré des capacités impressionnantes dans divers domaines: génération de texte, raisonnement logique et programmation. Leurs performances s'articulent autour de trois facteurs principaux: raisonnement, compréhension et modélisation linguistique [1]. Ces capacités sont traditionnellement évaluées par des tests de performance (benchmarks) standardisés comme GSM8k pour les mathématiques [2].

Le jeu d'échecs, depuis la victoire de Deep Blue contre Kasparov en 1997, constitue un terrain d'évaluation emblématique pour l'IA. Il offre un cadre pour mesurer les capacités de raisonnement et de planification des différents modèles.

Problématique et portée de l'étude

Le jeu d'échecs, avec ses règles formelles et sa profondeur stratégique, a historiquement servi de terrain d'évaluation pour l'intelligence artificielle, depuis les premiers programmes comme celui d'Alan Turing *Turochamp* [3]. Bien que les LLMs possèdent des capacités générales de raisonnement et de compréhension, leur application aux échecs soulève des questions spécifiques. Une étude récente indique que les LLMs, notamment les chatbots, peinent à jouer des parties complètes en raison d'hallucinations fréquentes qui les conduisent à proposer des coups illégaux [4]. Cette difficulté à respecter les règles strictes du jeu met en lumière les limites actuelles de ces modèles dans des domaines exigeant une logique formelle rigoureuse.

Il existe donc une nécessité claire de développer une méthode d'évaluation standardisée, capable de mesurer leurs performances aux échecs de manière comparable à l'échelle humaine. Ce projet s'attaque à cette problématique en se concentrant sur l'évaluation des LLMs non pas dans des parties complètes, mais à travers la résolution de puzzles d'échecs spécifiques.

Objectifs de la recherche

Objectif principal

Évaluer et comparer systématiquement les performances de plusieurs LLMs dans la résolution de puzzles d'échecs de différents niveaux de difficulté.

Objectifs spécifiques

1. Estimer un niveau de jeu comparable à l'échelle humaine pour chaque LLM testé
2. Identifier les forces et faiblesses de chaque modèle, notamment en analysant les types de puzzles où ils excellent ou échouent, et en quantifiant la fréquence des hallucinations.
3. Développer un système d'évaluation automatisé

Méthodologie et approche

Notre approche repose sur l'évaluation des LLMs par des puzzles d'échecs de complexité variable (tactiques, positionnels, finaux), présentés sous forme textuelle. Cinq modèles seront évalués, dans les limites d'un budget de 40$, sur 1400 puzzles.

Le système Glicko-2 sera utilisé pour estimer le classement des modèles, prenant en compte le taux de réussite, la difficulté des problèmes résolus et la consistance des performances.

La base de données des puzzles de Lichess servira de référence, avec ses caractéristiques détaillées: PuzzleId, FEN (configuration de l'échiquier), Moves (séquence de solutions), Rating (difficulté), RatingDeviation (fiabilité du classement), et catégorisation thématique [4, 5, 6].

La base de données filtrée est composée de trois thèmes, *End game*, *Tactic* et *Strategic*. Ces thèmes globaux représentent les types de positions que l'on retrouve tout au long d'une partie d'échecs. Dans la base de données de Lichess, il existe une multitude de sous thèmes correspondant au thème parent choisi. Vu que nous sommes contraints de choisir que 1400 puzzles. Les sous-thèmes que nous avons retenus pour le thème *End game* sont *MateIn1, MateIn2, MateIn3* et *MateIn4*. Pour le thème *Tactic*, nous avons retenu Démarche proposée, *pin, fork, deflection, sacrifice* et *discoveredAttack*. Pour le thème *Strategic*, nous avons retenu *queensideAttack, kingsideAttack, defensiveMove, advantage* et *quietMove*.

Voici une vue d'ensemble de notre approche.

1. Création de 3 bases de données de Puzzles pour chaque catégorie thématique évaluée.
2. Mise en place d'un protocole de test
3. Développement d'un environnement de test automatisé
4. Évaluation systématique des LLMs sélectionnés
5. Analyse des résultats et estimations du classement Glicko-2
6. Interprétation des résultats dans le contexte plus large de l'IA appliquée aux jeux stratégiques

Cadre théorique

Principes pertinents au projet

Afin de saisir pleinement les mécanismes sous-jacents de notre système, il est important de se familiariser avec un ensemble de concepts fondamentaux :

Notations d'échecs :

- **FEN (Forsyth-Edwards Notation)**
    - Il s'agit d'une notation standardisée pour décrire une position sur un échiquier [5].
    - Exemple : "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
    - Explication
        - Les premières lettres avant le slash représentent les pièces occupant cette rangée de l'échiquier de gauche à droite.
        - Les lettres minuscules représentent les pièces noires, les majuscules représentent les pièces blanches.
        - Les chiffres représentent le nombre de cases vides consécutives.
        - Chaque rangée est séparée par un slash.
        - "w" ou "b" indique qui a le trait.

        - Les lettres "KQkq" indiquent les droits de roque.
        - Le tiret indique si la prise en passant est possible.
        - Les deux derniers chiffres représentent le nombre de demi-coups et le numéro du coup.
    - **Raison d'utilisation :** Ce format est utilisé pour représenter précisément une position d'échiquier avec un minimum de caractères, ce qui est essentiel pour l'optimisation de la transmission des données d'échecs.

- **SAN (Standard Algebraic Notation)**
    - Exemple : "Cf3"
    - Explication : Cette notation décrit un mouvement de pièce en indiquant le nom de la pièce (en abrégé) et la case d'arrivée. Dans cet exemple, le cavalier va à la case f3 [5].
    - **Raison d'utilisation :** Ce format est le plus utilisé par les joueurs humains pour enregistrer leurs coups et aussi celui qui offre les meilleures performances pour les modèles de langage. C'est pourquoi il est également utilisé pour ces derniers.

Système de Classement Glicko-2

Il estime le niveau de jeu d'un joueur en utilisant une distribution de probabilité. Cette approche diffère du système Elo, qui utilise une valeur unique. La distribution de probabilité dans Glicko-2 est caractérisée par deux valeurs : le classement et la déviation. Le classement représente le niveau de jeu estimé du joueur, similaire au classement Elo, mais basé sur une distribution de probabilité. La déviation indique l'incertitude associée à ce classement. Une déviation faible signifie une certitude dans l'évaluation du niveau de l'agent. Glicko-2 combine ces deux valeurs pour fournir une estimation plus précise et fiable du niveau de jeu d'un joueur. Un joueur avec un classement de 1500 et une déviation de 100 à 95 % de chances d'avoir un niveau réel entre 1300 et 1700 [6]. Cette approche permet de situer le niveau du joueur dans un intervalle de confiance élevé, même sans connaître sa valeur exacte.

Les Tokens

Ils sont des morceaux de texte utilisés par les modèles de langage pour comprendre et générer du texte. Un token peut être un mot, mais aussi une partie de mot ou un signe de ponctuation. Par exemple, le mot "incroyablement" pourrait être divisé en trois tokens : "in", "croy", "ablement" [7]. C'est important de comprendre cela, car le coût d'utilisation de certains modèles de langage est calculé en fonction du nombre de tokens et non du nombre de mots.

Les hallucinations

Les LLMs sont excellents dans la manipulation du langage naturel, mais rencontrent des défis dans les domaines exigeant un raisonnement logique strict et le respect de règles formelles, comme les échecs [4]. Les "hallucinations", où le modèle génère des informations incorrectes ou des coups illégaux, sont un problème notable que notre système cherche à quantifier.

Le poids des paramètres

Le poids des paramètres dans les LLMs fait référence au nombre de paramètres ajustables. Un poids plus élevé améliore généralement la compréhension syntaxique et sémantique, mais augmente également le coût d'inférence et l'utilisation des ressources.

Le Raisonnement Formelle (Chain-of-Thought)

Cette technique incite le LLM à décomposer son raisonnement étape par étape avant de fournir la réponse finale. Techniquement, tous les LLMs peuvent être amenés à produire un raisonnement CoT en utilisant des techniques de requête enchaînée complexes. Cependant, il existe une différence entre les modèles qui sont spécifiquement entraînés à générer un raisonnement CoT automatiquement et ceux qui ne le sont pas. Dans ce rapport, lorsque nous parlons de "modèle de raisonnement CoT", nous faisons référence aux modèles qui ont été explicitement entraînés pour produire un raisonnement CoT. Notre système permet d'évaluer l'impact de cette technique en comparant les performances des mêmes modèles avec et sans CoT.

Architecture et Implémentation

Le système d'évaluation a été implémenté sous la forme d'une application Python, conçue selon une architecture modulaire pour faciliter la maintenance, l'extensibilité et les tests. L'architecture générale s'appuie sur des composants distincts interagissant avec des interfaces définies, avec une base de données centrale pour la persistance des données.

Architecture Globale

Le système est organisé autour d'une série de modules Python interdépendants, chacun responsable d'une partie spécifique du processus d'évaluation (voir Figure 1 pour une représentation schématique des classes principales et de leurs relations). Les composants clés incluent la gestion des agents, la simulation de l'environnement d'échecs, l'orchestration de l'évaluation, la sélection des puzzles, l'interaction avec la base de données, et la communication avec les API externes des LLMs. La base de données SQLite sert de référentiel central pour les puzzles, les configurations d'agents, les résultats des évaluations (parties et coups individuels), et l'historique des classements Glicko-2.

Composants Principaux

Les fonctionnalités du système sont réparties entre plusieurs classes clés, conformément au diagramme de classes (Figure 1) et au code source fourni :

Agent (Abstrait)

Définis l'interface commune pour tous les types d'agents capables de jouer aux échecs (LLM, Stockfish, Aléatoire). Elle impose l'implémentation des méthodes essentielles que tous les agents doivent avoir : get_move (pour choisir un coup dans une position donnée) et retry_move (pour tenter de fournir un coup valide après un échec initial). Les attributs is_reasoning et is_random permettent de caractériser le type d'agent.

LLMAgent

Spécialisation de l'interface Agent pour les modèles de langage. Ce composant est central pour l'interaction avec les API des LLMs.

- Il construit une requête en incluant la FEN, la liste des coups légaux, et la couleur à jouer.
- Il gère la communication avec les APIs externes des LLMs (ex. OpenRouter, Nvidia Nim, etc.).

- Il utilise un Router pour acheminer les requêtes et respecter les limites de taux globales de requêtes API que certaines plates-formes imposent.
- Il utilise un limiteur (AsyncLimiter) spécifique à l'instance du modèle spécifique pour gérer les appels API concurrents de celui-ci.
- Il parse les réponses textuelles des LLMs pour extraire le coup en notation SAN.
- Il implémente la logique de relance (retry_move) en cas de coup invalide ou d'erreur de formatage.

RandomAgent

Agent simple choisissant un coup au hasard parmi les coups légaux. Cet agent nous permet de modéliser l'aspect de la chance dans les résultats. La pensée est que les modèles qui dépendent fortement de la chance pour leur fonctionnement produiront des résultats similaires à ceux d'un agent qui agit de manière aléatoire. Cela signifie que la performance de ces modèles sera imprévisible et peu fiable, car elle sera largement déterminée par le hasard plutôt que par une logique ou une stratégie sous-jacente. Dans certains cas, ces modèles peuvent produire des résultats acceptables par hasard, mais ils sont tout aussi susceptibles de produire des résultats médiocres ou mauvais.

StockfishAgent

Interface avec le moteur Stockfish. Configure le niveau de difficulté et interroge le moteur pour obtenir le meilleur coup selon son évaluation. Cet agent est conçu pour simuler les performances d'un joueur d'échecs compétent. Il sert de référence pour évaluer les modèles de langage. Si un modèle donné obtient des résultats similaires à ceux de l'agent Stockfish, cela suggère que ce modèle s'appuie davantage sur des stratégies et des tactiques d'échecs solides plutôt que sur le hasard. En d'autres termes, ses performances sont plus déterminées par sa capacité à analyser la position et à prendre des décisions éclairées que par des coups aléatoires ou chanceux.

ChessEnv

Encapsule la logique du jeu d'échecs à l'aide de python-chess. Maintiens l'état de l'échiquier, fournis la liste des coups légaux, valide les coups, applique les coups, et gère les conversions de notation.

DatabaseManager

Implémentée en tant que Singleton, cette approche garantit une connexion unique à la base de données SQLite. Cela permet d'éviter les problèmes de concurrence, le moteur SQLite n'offrant qu'un support limité des opérations simultanées. Gère toutes les opérations CRUD (Create, Read, Update, Delete) pour les tables définies dans le schéma Entité-Relation (Figure 2). Fournis des méthodes pour enregistrer les agents, les parties, les coups individuels (incluant les tokens utilisés), les résultats des benchmarks (classements Glicko-2), et pour récupérer les données nécessaires à l'évaluation et à la génération de rapports.

Evaluator

Orchestre le processus d'évaluation pour un agent donné sur un ensemble de puzzles.

- Reçois un agent et une liste de puzzles (obtenue de PuzzleSelector).
- Pour chaque puzzle, il simule la partie coup par coup en utilisant l'agent et ChessEnv.

- Enregistre le déroulement de la partie, chaque coup joué, y compris les informations sur les tokens et les coups illégaux, ainsi que les résultats de la partie par le DatabaseManager.
- Calcule, met à jour et enregistre la note Glicko-2 de l'agent selon le résultat du puzzle joué.
- Gère la logique d'arrêt de l'évaluation optionnelle (atteindre une déviation cible pour le classement).

PuzzleSelector

Responsable de sélectionner et de fournir les puzzles d'échecs qu'un agent (Agent) doit évaluer. Il s'assure que les puzzles sont disponibles pour l'évaluation, en les chargeant initialement à partir de fichiers CSV distincts contenant des puzzles classés par thèmes (tactique, stratégie, fin de partie) si la base de données est vide. Une fois les puzzles en base de données, le PuzzleSelector interroge celle-ci pour identifier et retourner uniquement les puzzles qui n'ont pas encore été complétés par l'agent spécifique faisant la demande. Il utilise donc l'historique stocké en base pour éviter les répétitions et garantir une évaluation complète par chaque agent.

Router

Gère les clés d’accès aux API et les liens aux points d'accès (base URL) de différents modèles. Il contrôle la procédure d’attente grâce à un limiteur (AsyncLimiter) pour éviter les dépassements de limites de taux de requêtes par minute (RPM) des fournisseurs d’accès. Il gère les requêtes de façon asynchrone. Ceci permet au fil d’exécution de traiter d’autres requêtes des autres modèles en attente d’une réponse.

ReportGenerator

Dédiée à la création de rapports visuels sur les performances des agents d'échecs. Elle utilise le DatabaseManager pour extraire les données pertinentes (résultats des parties, benchmarks, utilisation des tokens, etc.) stockées dans la base de données. Ensuite, elle emploie les librairies pandas pour la manipulation et l'agrégation des données, et matplotlib (aidé par numpy) pour générer une gamme de visualisations statistiques, telles que :

- Les tendances d'évolution des classements Glicko-2 des agents.
- Les tendances d'évolution de la déviation des classements.
- Les taux de succès/échec globaux et par agent, ventilés par type de puzzle.
- La distribution (en pourcentage) des coups illégaux par agent.
- Les classements finaux des agents avec leurs intervalles de confiance.
- Le taux de succès par agent en fonction de la difficulté des puzzles.

Essentiellement, ReportGenerator transforme les données brutes de la base de données en graphiques informatifs pour analyser et comparer les performances des différents agents testés.

Base de Données

Le système s'appuie sur une base de données relationnelle SQLite pour stocker et organiser l'ensemble des données relatives aux évaluations afin de pouvoir générer des rapports. Le

choix de technologie s'explique par sa simplicité de déploiement (fichier unique, pas de serveur dédié) tout en offrant les capacités SQL nécessaires pour ce projet. Le schéma est représenté par le diagramme Entité-Relation (Figure 2).

Technologies Utilisées

Le développement du système s'appuie sur un ensemble de technologies et librairies:

- **Langage :** Python 3.11.11 a été choisi pour son écosystème riche en bibliothèques scientifiques et son support des opérations asynchrones, indispensable pour interroger les APIs des LLMs efficacement.
- **Gestion de l'environnement d'échecs :** La bibliothèque python-chess pour la représentation de l'échiquier, la génération et la validation des coups légaux et la gestion des notations FEN/SAN.
- **Calcul du classement :** Librairie Python implémentant Glicko-2 pour mettre à jour les cotes des agents après chaque partie.
- **Base de données :** SQLite 3 pour stocker de manière simple, mais avec la possibilité de créer des relations entre les données de l'évaluation.
- **Services de point d'accès aux LLMs:** Les services d'API d'OpenRouter et de Nvidia NIM ont été utilisés pour accéder aux modèles de langage. OpenRouter permet d'acheminer les requêtes à différents hébergeurs de modèles (OpenAI, Anthropic, etc.) par une interface unifiée, tandis que NIM est la plateforme de Nvidia qui elle-même héberger et rends accessibles des LLMs open source.
- **Appels API LLM :** La librairie cliente openai pour envoyer les requêtes aux modèles de manière asynchrone et recevoir les réponses. Cela a permis de paralléliser les requêtes et d'optimiser le débit d'évaluation.
- **Programmation Asynchrone :** Utilisation intensive du module asyncio pour orchestrer les tâches concurrentes. Plusieurs agents peuvent jouer en parallèle, et un même agent LLM peut attendre une réponse réseau pendant qu'un autre poursuit son calcul.
- **Manipulation de données et rapports :** Utilisation des bibliothèques pandas pour le traitement des données en DataFrame, numpy pour certaines opérations numériques, et matplotlib pour la création des graphiques utilisés dans le rapport final.
- **Gestion des dépendances :** Un fichier requirements.txt liste les versions des modules Python utilisés, afin de faciliter l'installation reproductible de l'environnement. De plus, un conteneur Docker a été configuré pour automatiquement effectuer l'installation et garantir que toutes les expériences s'exécutent dans un environnement logiciel identique.

Conditions d'Expérimentation

Cette section décrit les conditions dans lesquelles les expériences ont été menées, y compris l'environnement logiciel, le choix des puzzles, la configuration des agents évalués, ainsi que les paramètres et contraintes ayant influencé le déroulement des évaluations.

Environnement d'exécution

Les expérimentations ont été réalisées à l'aide d'un conteneur Docker. Ceci garantit que l'évaluation de tous les modèles fut complétée dans des conditions identiques, éliminant les différences de configuration logicielle. L'utilisation de Docker facilite également la reproductibilité des résultats et l'isolation du projet (évite les conflits de versions de bibliothèques avec d'autres projets). Le matériel sous-jacent utilisé était un PC standard, mais l'essentiel de la charge de calcul étant porté par les appels aux APIs distantes ou par le moteur Stockfish (peu gourmand aux niveaux choisis).

Sélection des puzzles

Un sous-ensemble de puzzles issu de la base de données Lichess a été utilisé pour évaluer les agents, comme décrit dans la méthodologie. Le PuzzleSelector s'occupe de copier les puzzles des fichiers CSV dans la base donnée lors de l'évaluation d'un agent, le PuzzleSelector sélectionne dynamiquement les puzzles non résolus par l'agent pour constituer sa série de tests. L'évaluation de chaque agent se poursuit jusqu'à ce que son classement Glicko-2 se stabilise (déviation en dessous d'un seuil cible, ex. 60–70), ou jusqu'à la fin de l'évaluation de tous les puzzles.

Restrictions budgétaires et limites de requêtes

L'évaluation de LLMs étant coûteuse en termes de requêtes API, nous avons dû prendre en compte les contraintes budgétaires dans la conception de nos expériences. Initialement, nous avions prévu d'utiliser largement le service OpenRouter avec des modèles haut de gamme (comme GPT-4 ou Claude) en estimant, d'après leurs tarifs, que le coût total resterait sous 40 USD pour l'ensemble des tests. Cependant, cette estimation n'avait pas pleinement anticipé le nombre de tokens générés par les modèles utilisant le raisonnement chaîné. Par exemple, lorsqu'un modèle de langage large (LLM) détaille son raisonnement, il génère des réponses plus longues. Comme les plateformes d'IA facturent habituellement selon le nombre de tokens générés, avec un tarif généralement plus élevé pour les tokens de sortie que pour ceux d'entrée, cette approche explicative entraîne des coûts d'utilisation plus importants. Lors de nos tests exploratoires, le budget alloué a été englouti : par exemple, le modèle deepseek-r1 par OpenRouter a consommé à lui seul la quasi-totalité des crédits, sans même couvrir 50 % des puzzles prévus. Face à cette contrainte, nous avons dû réviser notre sélection de modèles et éliminer certains des modèles les plus coûteux ou les plus verbeux. OpenRouter propose bien des accès gratuits à certains modèles performants, mais ceux-ci s'accompagnent de limites draconiennes (20 requêtes par minute et 200 requêtes par jour dans notre cas), incompatibles avec le volume de tests à effectuer pour notre projet.

Après une recherche, nous avons trouvé la plateforme Nvidia NIM, qui héberge des modèles de langage open source et les rend accessibles gratuitement par l'API. Cette solution nous a permis de continuer l'évaluation sans frais directs, au prix de quelques compromis. D'abord, NIM impose une limite d'environ 40 requêtes par minute et par utilisateur, ce qui nous a obligés d'intégrer un mécanisme de contrôle AsyncLimiter), afin de ne jamais excéder la limite. Ensuite, nous avons observé que la disponibilité et la latence des modèles varient. Les modèles récents, dans le catalogue NIM, répondaient rapidement et de manière fiable. Tandis que certains modèles, plus anciens ou très demandés, pouvaient placer les requêtes dans une file d'attente. Cela occasionna des délais ou des échecs sporadiques. Ce fut notamment le cas du modèle *DeepSeek R1* qui s'est avéré inutilisable pendant notre fenêtre d'évaluation dû à un temps d'attente trop élevé. Par conséquent, nous avons dû recentrer notre banc d'essai sur les autres modèles disponibles. Enfin, puisque NIM n'héberge que des modèles open source, notre

sélection finale de LLMs ne comprenait plus certains modèles propriétaires de pointe initialement envisagés.

Au lieu d'évaluer les meilleurs modèles du marché, nous nous sommes concentrés sur des modèles open source de haute performance (certaines itérations récentes de Llama, etc.), ce qui constitue tout de même une étude pertinente, mais rend un peu plus difficile l'extrapolation au très haut niveau. Ces choix ont été dictés par la réalité des contraintes budgétaires et techniques, et font partie des leçons apprises au cours du projet.

Configuration des Agents LLM

Cinq modèles de langage ont été configurés pour l'évaluation. Chaque agent LLM est défini en précisant le service (Router) à utiliser, l'identifiant du modèle, et si le modèle possède la capacité intégrée de raisonnement pas-à-pas (Chain-of-Thought). La liste finale des modèles testés est la suivante :

- nvidia/llama-3.1-nemotron-ultra-253b-v1 : modèle de grande taille (environ 253 milliards de paramètres) de Nvidia, basé sur l'architecture Llama de Méta utilisant un raisonnement explicite (CoT) [8].
- meta/llama-3.1-405b-instruct : modèle de très grande taille de Meta (environ 405 milliards de paramètres), utilisé en mode réponse directe (sans CoT) [9].
- google/gemma-3-27b-it : modèle de taille moyenne (environ 27 milliards de paramètres) de Google, utilisé en mode direct (sans CoT) [10].
- meta/llama-3.1-8b-instruct : modèle de petite taille de Meta (environ 8 milliards de paramètres), utilisé en mode réponse directe (sans CoT) [11].
- meta/llama-4-maverick-17b-128e-instruct : modèle de très grande taille de Meta (environ 17 milliards de paramètres activés, 400B paramètres totaux), utilisé en mode réponse directe (sans CoT) [12].

Tous ces modèles ont été interrogés avec l'API NIM de Nvidia.

Paramètres d'API

Afin d'assurer une cohérence et de minimiser la part de hasard dans les réponses des LLMs, nous avons appliqué des paramètres de requête uniformes pour tous les modèles. La température a été réglée à une valeur relativement basse de 0,6 pour réduire la stochasticité des réponses tout en évitant les réponses trop déterministes ou répétitives. Le paramètre top_p a été fixé à 0,95, ce qui permet au modèle de considérer 95 % des probabilités de sa distribution lors de l'échantillonnage du prochain token [13]. Ces valeurs de température et top_p sont des recommandations usuelles pour l'évaluation de modèles sur des tâches sensibles à la répétitivité par les développeurs [8], [14]. Les autres paramètres d'API par défaut ont été conservés.

Prompting

Le prompt envoyé aux modèles a été conçu pour fournir tout le contexte nécessaire et obtenir une réponse facile à exploiter. Nous utilisons une requête structurée de la manière suivante : le modèle est instruit du rôle qu'il doit jouer *("You are a world-class chess grandmaster and expert analyst…")* afin de le mettre dans le contexte d'un expert en échecs. Ensuite, le prompt fournit

la position initiale du puzzle en notation FEN, ainsi que la liste exhaustive des coups légaux depuis cette position pour le camp devant jouer. Cela vise à réduire les hallucinations, en cadrant le modèle sur les possibilités valides. Le modèle est invité à analyser la position puis à fournir son coup final. Pour faciliter le traitement automatique de la réponse, nous demandons explicitement que le coup choisi soit donné en notation SAN et encadré par des balises spécifiques <FinalMove> dans une nouvelle ligne ne contenant que cela. Par exemple, une réponse attendue pourrait être : <FinalMove>b5</FinalMove>. Il est bien insisté dans l'instruction que *seul* le coup final doit apparaître sur la dernière ligne, sans texte supplémentaire, afin qu'aucune ambigüité ne subsiste. Grâce à ce formatage, notre système peut extraire immédiatement le coup joué en recherchant les balises <FinalMove> dans la réponse du LLM avec une expression régulière. Ce prompt inclut également un léger exemple dans la formulation pour être sûr que le modèle comprenne le format (voir Figure 3). Ce prompt prend en compte qu'un modèle qui utilise le raisonnement CoT aura tendance à produire plusieurs lignes de réflexion avant la balise finale, ce qui est toléré tant que le coup final est correctement isolé. Enfin, si le LLM fournit un coup jugé illégal par ChessEnv on ajoute au prompt, pour chaque tentative illégale précédente, un message contenant le coup illégal précédemment fourni par le modèle (formaté avec les balises <FinalMove>), suivi du message "Invalid move. Please try again.". Cette approche fournit au modèle le contexte de ses erreurs passées, l'incitant à reconsidérer son choix et à proposer un coup différent.

Configuration des agents de référence

Nous avons ajouté deux agents de référence pour nous permettre de mieux analyser nos résultats :

- *RandomAgent* ne nécessite aucune configuration particulière, car il sélectionne, au hasard, un coup pour chaque puzzle. Il sert de référence d'une performance minimale.
- *StockfishAgent* a fait l'objet de calibrations spécifiques. Le moteur Stockfish peut être paramétré à un niveau (Skill Level) entre 0 et 20, ainsi que d'autres réglages comme le temps de recherche ou la profondeur. L'objectif était d'obtenir un agent Stockfish, joueur moyen, c'est-à-dire avec un niveau d'environ 1500 Elo, pour servir de point de comparaison humain. Après plusieurs essais, nous avons déterminé qu'un *Skill Level* de 1 et une limite de temps d'environ 0,01 seconde par coup permettait d'atteindre cet objectif.

Processus d'évaluation

L'évaluation des agents s'effectue en leur faisant jouer une liste de puzzles. Un script Python dédié (evaluation.py) orchestre le lancement des évaluations. Pour chaque agent, on instancie un *Evaluator* qui recevra la liste. Les évaluations sont lancées en parallèle grâce à la bibliothèque asyncio. Cela permet d'évaluer simultanément plusieurs modèles de langage pour profiter pleinement des temps d'attente. Pendant qu'un modèle réfléchit ou que l'API répond, un autre peut être en train de jouer un coup. Le nombre de tâches concurrentes effectives est toutefois limité par les contraintes d'API et les ressources mémoire : nous avons veillé à ne pas lancer plus de requêtes que les limites autorisées (le Router et les AsyncLimiter internes s'en chargent) et à ne pas surcharger le système local. Nous avons pu évaluer les agents avec la limite de 40 requêtes par minute de NIM n'étant jamais dépassées grâce au contrôle logiciel. Chaque *Evaluator* exécute sa boucle d'évaluation de façon autonome.

Paramètres Glicko-2

Tous les agents débutent avec des paramètres identiques, dans le système de classement Glicko-2. Le classement initial est fixé à 1500 (ce qui correspond à une valeur Glicko moyenne par convention). La déviation initiale est fixée à 350. C'est une incertitude élevée qui reflète le fait qu'on ne sait rien du niveau réel de l'agent. La volatilité initiale est fixée à 0,06. Cette valeur est recommandée par le créateur du système Glicko-2 [6]. Avec ces paramètres nous supposons que le classement évoluera de manière mesurée au cours des premières parties, puis se stabilisera graduellement. Après chaque puzzle joué, la formule Glicko-2 est appliquée pour mettre à jour le rating, la déviation et la volatilité de l'agent en fonction du résultat.

Gestion des erreurs

Durant l'évaluation, un coup illégal peut survenir de la part d'un LLM. Par exemple : une pièce qui n'existe pas sur l'échiquier ou un mouvement ne respectant pas les règles. Ce coup est détecté par le module ChessEnv. L'agent bénéficie de cinq tentatives pour rejouer le coup. Au-delà de ce plafond, le puzzle est déclaré échoué et l'agent passe au puzzle suivant. Cette tolérance aux erreurs vise à ne pas pénaliser trop sévèrement un LLM qui aurait commis une erreur. Ceci est inspiré d'interfaces de jeux d'échecs en ligne qui ne permettent pas aux joueurs de commettre des coups illégaux. Un coup légal, mais incorrect (c'est-à-dire un coup valide aux échecs, mais qui n'est pas le coup attendu de la solution du puzzle) entraîne immédiatement l'échec du puzzle pour l'agent. En cas d'erreurs externes, telles que les dépassements de temps, les indisponibilités de service ou les dépassements de quota imprévus, le système enregistre l'exception dans un fichier journal et sur le terminal. Au lieu de réessayer la requête à plusieurs reprises, ce qui pourrait gaspiller le quota de requêtes, la partie est laissée incomplète. Elle sera reprise lors d'une exécution ultérieure du programme, lorsque le système reprendra les puzzles non complétés pour chaque agent. Cette approche garantit que les ressources du modèle sont utilisées de manière efficace et qu'une interruption temporaire du service n'entraîne pas l'échec permanent d'un puzzle.

Traitement des données et flux d'évaluation automatisée

Flux global d'une évaluation de puzzle

Pour chaque agent, l'Evaluator va prendre les puzzles un par un et réaliser la séquence suivante :

1. **Initialisation du puzzle** : L'Evaluator sélectionne un puzzle à partir de la liste fournie par le PuzzleSelector. Il réinitialise le ChessEnv à l'état initial du puzzle (en chargeant la position du FEN correspondante). Il récupère également la solution attendue du puzzle (séquence de coups solutionnaire) dans la base de données, qui servira de référence pour évaluer les réponses de l'agent. À ce stade, le classement courant de l'agent est connu (initialement 1500, puis mis à jour après chaque puzzle).

2. **Demande du coup au modèle** : L'Evaluator formule une requête au module Agent pour le coup suivant. Si l'agent est un LLM (LLMAgent), cela enclenche la construction du prompt avec la position actuelle (en format FEN) et les coups légaux (en format SAN), puis l'envoi de la requête avec le Router approprié (OpenRouter, NIM). Si l'agent est Stockfish, on interroge le moteur d'échecs avec la position. Si c'est RandomAgent, on tire un coup aléatoire. Dans tous les cas, on obtient une proposition de coup (sous forme de notation SAN).

3. **Validation du coup proposé** : La réponse de l'agent est transmise au module de validation (ChessEnv). On vérifie si le coup proposé est légal sur l'échiquier actuel (par exemple, si le LLM a sorti Nc3 alors qu'il n'y a pas de cavalier pouvant aller en c3, le coup est illégal). Pour Stockfish ou RandomAgent, les coups seront toujours légaux par construction, cette vérification est pour les LLMs.
    - Si le coup est **illégal** : l'Evaluator incrémente un compteur d'échecs pour ce puzzle et déclenche la procédure de réessaie. Le coup est enregistré dans la base de données comme étant illégal. Conformément à la configuration (section 2), l'agent LLM sera sollicité à nouveau pour un coup, éventuellement après avoir été informé que le précédent était invalide. On revient à l'étape 2 sans changer de position, et on redemande une action à l'agent tout en lui indiquant les coups illégaux qu'il a déjà tentés. Ce cycle peut se répéter jusqu'à 6 tentatives totales (la tentative initiale + 5 réessais). Si après 6 essais le modèle n'a toujours pas donné de coup légal, on abandonne le puzzle.
    - Si le coup est **légal** : on passe à l'étape suivante.
4. **Vérification du coup correct** : Si le coup proposé est légal, l'Evaluator enregistre ce coup dans la base de données comme légal et le compare alors au coup attendu dans la solution du puzzle (le prochain coup dans la séquence de solution prédéfinie).
    - Si le coup de l'agent correspond exactement au coup attendu, cela signifie qu'il a joué correctement cette étape du puzzle. Alors, on met à jour l'état de l'échiquier, avançant ainsi d'un coup dans le puzzle. Si le puzzle n'est pas encore terminé (il reste des coups à jouer pour arriver à la solution finale), on retourne à l'étape 2 pour demander le coup suivant dans la nouvelle position.
    - Si le coup de l'agent, bien que légal, n'est pas le coup attendu, alors le modèle s'est trompé de plan ou a raté la solution. Dans le contexte d'un puzzle, cela équivaut à un échec de résolution. On arrête donc le puzzle à ce stade, en le considérant comme échoué pour cet agent.
5. **Fin du puzzle** : Le puzzle se termine soit lorsque l'agent a joué toute la séquence de coups correcte jusqu'à la résolution (cas de réussite), soit lorsqu'il a commis une erreur (coup illégal non corrigé ou coup légal incorrect) épuisant ses chances (cas d'échec). L'Evaluator enregistre alors dans la base de données le résultat final de la partie (réussi ou non), ainsi que tous les détails de chaque coup tenté (avec indication des coups corrects, des éventuels coups illégaux, du nombre de tentatives, etc.). Ces données brutes permettent de retracer précisément l'enchaînement des événements pour chaque puzzle et agent.
6. **Mise à jour du classement** : Une fois le puzzle joué, on évalue la performance de l'agent sur ce puzzle par le système Glicko-2. Pour ce faire, on considère que l'agent a disputé une partie dont l'issue est la résolution ou non du puzzle. Nous modélisons le puzzle lui-même comme un adversaire virtuel de niveau fixe. En cas de succès, l'agent est vainqueur de la partie. En cas d'échec, il est perdant. Le calcul Glicko-2 prend en entrée le rating courant de l'agent, sa déviation, sa volatilité, ainsi que le rating du puzzle et un score de résultat (1 pour une victoire, 0 pour une défaite). Il produit en sortie un nouveau rating mis à jour ainsi qu'une nouvelle déviation et volatilité pour l'agent. Le DatabaseManager enregistre alors ces valeurs dans la table benchmark pour garder la trace de l'évolution du niveau. Cette mise à jour de classement est effectuée après

chaque puzzle dans notre implémentation, de sorte que même si on arrête l'évaluation en cours de route, on a la progression complète.

7. **Itération** : L'Evaluator passe ensuite au puzzle suivant non encore tenté par l'agent et répète le cycle. Ce processus se poursuit ainsi jusqu'à ce que le critère d'arrêt soit atteint (p. ex., déviation Glicko-2 de l'agent < 70, indiquant que son classement est suffisamment précis) ou qu'on a traité tous les puzzles. En fin d'évaluation, le statut de tous les puzzles (réussis/échoués) et le classement final de l'agent sont disponibles dans la base de données pour analyse.

Le fait de stocker chaque étape permet une transparence et une rétractabilité : en cas de doute sur un résultat, on peut inspecter la séquence de coups enregistrée pour voir où le modèle a failli. Par ailleurs, l'approche asynchrone fait que plusieurs agents peuvent progresser en parallèle dans ce flux sans interférence, chacun ayant son propre Evaluator et ChessEnv isolé. Cette approche permet d'optimiser l'utilisation du temps, mais requiert une plus grande utilisation de la mémoire.

Exploitation des données et génération de rapports

Une fois toutes les évaluations terminées, nous disposons d'une base de données qui contient l'historique des performances. Le module ReportGenerator (voir section 1.2) est utilisé pour parcourir ces données et en extraire les indicateurs clés. Par exemple, en agrégeant les champs de la table *move*, on peut calculer le taux de coups illégaux par agent. La table *game* permet de calculer le pourcentage de puzzles résolus par agent. La table *benchmark* fournit l'évolution temporelle du rating et de la confiance. Toutes ces analyses sont effectuées via des requêtes SQL (par le DatabaseManager) et des traitements pandas, puis présentées sous forme de graphiques.

Documentation et Tests du Système

Un effort particulier a été consacré à la documentation interne du code et à la validation du bon fonctionnement du système tout au long du développement. Chaque module principal du projet comporte des commentaires et des docstrings expliquant son rôle et son interface. Les diagrammes UML (diagramme de classes en Figure 1, diagramme Entité-Relation en Figure 2) ont été élaborés en parallèle de l'implémentation afin de clarifier la structure du système et de servir de documentation technique pour d'éventuels utilisateurs et contributeurs. Ces schémas ont notamment aidé à identifier les relations entre composantes.

La validation du système d'évaluation a été menée de manière progressive et itérative pour garantir la fiabilité des composants et du flux de travail global avant de lancer les expérimentations finales à grande échelle.

Tests Unitaires et Manuels

Au fur et à mesure du développement des différents modules (ChessEnv, LLMAgent, DatabaseManager, etc.), des tests manuels et des validations sur des cas simples ont été effectués. Par exemple, nous avons vérifié qu'un LLM répondait correctement à un prompt pour un puzzle trivial (mat en un coup), que le système parvenait à extraire le coup des balises <FinalMove> sans erreur de parsing, et que la détection des coups illégaux fonctionnait comme prévu en utilisant un agent simulant des réponses invalides.

Tests Préliminaires des Points d'Accès

Avant d'intégrer pleinement les appels API dans le système principal, une phase exploratoire a été menée (notamment avec des notebooks comme llm.ipynb) pour se familiariser avec les différents points d'accès des LLMs (OpenRouter et potentiellement d'autres fournisseurs) et les librairies d'interaction (comme openai et requests). Ces tests préliminaires ont été essentiels pour :

- Évaluer la disponibilité et la fiabilité de certains modèles sur les plateformes envisagées. Nous avons par exemple constaté que certains endpoints, comme celui pour DeepSeek R1, présentaient parfois une disponibilité limitée.
- Itérer sur le formatage de la réponse attendue des LLMs. Initialement, nous avions demandé une réponse sous la forme "Final Move: move". Cependant, les tests ont révélé que certains modèles ajoutaient du formatage supplémentaire comme le Markdown (par exemple, **Final Move:** move), ce qui compliquait l'extraction fiable du coup. Face à ce constat, nous avons pivoté vers l'utilisation de balises de type XML (<FinalMove>move</FinalMove>), beaucoup plus robustes face au formatage Markdown non désiré et permettant une extraction simple et fiable avec une expression régulière.

Tests d'Intégration et Détection de Bugs

Des évaluations préliminaires en utilisant un échantillon de puzzles et des modèles LLM plus petits (<= 3 milliards de paramètres) et à faible coût par OpenRouter ont été réalisées pour vérifier l'enchaînement correct du flux. Ces tests ont permis de découvrir certains bugs fonctionnels. Par exemple, nous avons identifié un problème dans la logique de sélection des puzzles (PuzzleSelector) qui pouvait ramener des puzzles déjà joués par un agent, créant ainsi des doublons dans les résultats. Ce bug a été détecté en observant un nombre d'enregistrements dans la table benchmark supérieur au nombre de puzzles uniques qui auraient dû être évalués. Pour corriger cela et garantir l'intégrité des données, des contraintes d'unicité (UNIQUE sur les colonnes puzzle_id et agent_name de la table game) ont été ajoutées au schéma de la base de données.

Journalisation

L'utilisation intensive de la journalisation (logging, par notre module logger.py) a été primordiale durant les tests sur des volumes plus importants. Le script principal evaluation.py produit des logs détaillés pour chaque puzzle joué, incluant les requêtes, les réponses, les erreurs éventuelles et les mises à jour de classement. Cela nous a permis de suivre l'exécution, de diagnostiquer rapidement des problèmes (par exemple, identifier des points d'accès LLM spécifiques qui devenaient surchargés ou répondaient avec des erreurs fréquentes) et de corriger des cas imprévus.

Validation des Visualisations

Enfin, la pertinence et la clarté des graphiques produits par le notebook d'analyse (reports.ipynb) ont également fait l'objet d'une validation. Les retours et suggestions de notre superviseur, Prof. Alan Davoust, ont permis d'affiner la sélection des visualisations pour s'assurer qu'elles communiquent les résultats de manière efficace et compréhensible.

Discussion et interprétation des résultats

Cette section analyse les résultats obtenus lors de l'évaluation comparative des modèles de langage (LLMs) dans la résolution de puzzles d'échecs, en s'appuyant sur les visualisations générées (Figure 4 à 10). Nous interprétons les performances, discutons des erreurs observées, évaluons la fiabilité des mesures, reconnaissons les limites de l'étude, et comparons nos observations avec la littérature existante et les objectifs initiaux du projet.

Analyse des performances globales et des classements Glicko-2

La Figure 4 : Tendances d'évolution des ratings des agents illustre, pour chaque agent, la trajectoire de son classement au fil des puzzles résolus. Tous les agents débutent à une note de 1500, puis voient leur classement varier fortement lors des premières dizaines d'évaluations avant de se stabiliser. Cette phase initiale traduit l'impact d'une forte déviation (incertitude) dans les premières parties. Les courbes montrent ensuite un aplatissement, indiquant que les évaluations suivantes apportent moins d'information nouvelle sur le niveau des agents :

- **random_agent** décroît progressivement pour atteindre des niveaux très faibles (-74 ± 159), confirmant un comportement purement aléatoire sans réelle capacité de résolution.
- **LLMs** se positionnent entre environ 100 et 750 points :
    - *meta/llama-3.1-8b-instruct* termine autour à 140 ± 171.
    - *google/gemma-3-27b-it* progresse jusqu'à 273 ± 105.
    - *meta/llama-3.1-405b-instruct* termine à 518 ± 91.
    - *nvidia/llama-3.1-nemotron-ultra-253b-v1* s'arrête à 705 ± 81.
    - *meta/llama-4-maverick-17b-128e-instruct* s'arrête à 438 ± 102.
- **stockfish-1_agent** reste le plus performant avec un classement stabilisé à (1367 ± 76), proche du niveau visé (~1 500) pour un joueur humain amateur débutant.

Stabilité et incertitude

La Figure 5 : Tendances d'évolution des déviations des agents montre que pour la plupart des LLMs, la déviation (RD) décroît rapidement puis se stabilise entre 80 et 100, signe d'une confiance dans les classements finaux. Cependant, la RD de meta/llama-3.1-8b-instruct, comme celle de random_agent, augmente indéfiniment après environ 100 évaluations. Cette augmentation reflète une incertitude croissante face aux puzzles. stockfish-1_agent affiche régulièrement la plus faible RD (~75), attestant de sa consistance et de sa prédictibilité.

Difficulté moyenne

La Figure 9 : Classements finals des agents compare chaque rating final au rating moyen pondéré des puzzles. Aucun LLM n'atteint la cote moyenne pondérée des puzzles (~1500). Le meilleur LLM (nvidia/llama-3.1-nemotron-ultra-253b-v1) plafonne à (~705 ± 81), un niveau équivalent à un joueur débutant. meta/llama-3.1-8b-instruct termine très bas (~140 ± 171), presque au niveau de l'agent aléatoire, confirmant son incapacité à progresser significativement malgré une RD relativement basse.

Taux de réussite par thème

Aucun LLM ne présente de variation marquée de son taux de réussite selon le thème (fin de partie, stratégie ou tactique). La ressemblance des digrammes de la Figure 7 à la Figure 6 le confirme. Pour chaque modèle, les proportions de succès sont pratiquement identiques, quel que soit le type de puzzle, et reflètent les performances globales. Les LLMs suivent le même profil quasi plat sur tous les thèmes, ce qui indique que la performance dépend principalement de la difficulté intrinsèque des puzzles plutôt que de leur classification thématique.

Impact de la difficulté

La Figure 10 : Pourcentage de succès par thème, segmenté par fourchette d'évaluation présente, pour chaque agent, le taux de réussite moyen dans cinq tranches de difficulté. Ces tranches ont été créées automatiquement en répartissant uniformément la distribution de tous les puzzles rencontrés par chaque agent selon leur note Glicko. On observe un déclin régulier du taux de succès lorsque la difficulté augmente, sans lien significatif aux thèmes (fin de partie, stratégie ou tactique) :

- **500–884** : taux de réussite le plus élevé, jusqu'à ~60 % pour les meilleurs LLMs et près de 90 % pour Stockfish.
- **885–1267** : baisse modérée de 10 à 15 %, traduisant la transition vers des puzzles de complexité intermédiaire.
- **1268–1670** : poursuite du déclin, signe que la résolution devient nettement plus exigeante.
- **1671–2072** et **2073–2497** : effondrement progressif vers 5 % pour la quasi-totalité des LLMs, tandis que Stockfish conserve un succès résiduel (25–48 %).

Cette uniformité de la pente de déclin renforce l'idée que la difficulté intrinsèque des puzzles est le facteur dominant expliquant la performance, sans rupture de comportement propre à un thème ou à un modèle particulier. La similarité de la baisse par tranche de rating confirme l'absence d'effet thématique supplémentaire au-delà de la difficulté générale.

Hallucinations et mouvements illégaux

La Figure 8 : Pourcentage des coups illégaux par modèle met en évidence le taux d'hallucination de certains modèles. Bien que la liste des coups légaux ait été fournie aux modèles, ils ont tous suggéré au moins une fois un coup qui n'était pas inclus dans la liste.

- **meta/llama-4-maverick-17b-128e-instruct :** 46,9 % de coups invalides, le taux d'hallucination le plus élevé parmi les modèles testés.
- **meta/llama-3.1-8b-instruct :** 35,8 % de coups illégaux.
- **google/gemma-3-27b-it :** 30,6 % de suggestions hors liste
- **nvidia/llama-3.1-nemotron-ultra-253b-v1 :** 0,5 % de coups illégaux
- **meta/llama-3.1-405b-instruct :** 0,1 % d'illégalités
- **stockfish-1_agent : 0,0 %**, et **random_agent :** N'émettent aucun coup illégal. (par construction, ils choisissent uniquement parmi les coups fournis).

Cette mesure met en évidence que la taille du modèle n'est pas suffisante pour garantir le respect des règles : les gros modèles sans raisonnement CoT (Llama-4, Gemma-3) hallucinent davantage.

Forces et faiblesses spécifiques des LLMs

Absence de corrélation taille et hallucination

Malgré le fait qu'il soit le plus petit modèle évalué, avec 8 milliards de paramètres, meta/llama-3.1-8b-instruct ne présente pas le taux d'illégalité le plus élevé, ce qui suggère un taux d'hallucination inférieur à celui de meta/llama-4-maverick. Ce résultat est surprenant, car on pourrait penser qu'un modèle aussi petit et de génération inférieur aurait pu avoir des difficultés à comprendre le contexte et les attentes en raison d'une capacité de distinction sémantique des tokens plus faible.

Absence de corrélation avec le raisonnement

Malgré ses 405 milliards de paramètres, meta/llama-3.1-405b-instruct se débrouille aussi bien que le modèle avec la capacité de raisonnement nvidia/llama-3.1-nemotron-ultra-253b-v1 sur le classement final et affiche même un taux d'illégalité meilleur (0,1 % vs 0,5 %). Malgré un avantage théorique lié à son poids de paramètres plus important, qui devrait favoriser une meilleure performance, meta/llama-3.1-405b-instruct ne se distingue que légèrement de nvidia/llama-3.1-nemotron-ultra-253b-v1.

Imitation d'agents de référence

La plupart des modèles présentent un profil stable, similaire à celui de stockfish-1_agent en termes d'écarts, ce qui indique une politique de jeu stable (performance prévisible) malgré un taux de réussite inférieur à celui de l'agent stockfish.

En revanche, le modèle meta/llama-3.1-8b-instruct affiche une courbe de déviation croissante, proche de celle de random_agent. Cela suggère que Llama-3.1-8b choisit ses coups de manière quasi aléatoire parmi les coups légaux, sans réelle compréhension de leur valeur stratégique. En d'autres termes, il peut identifier les coups illégaux, mais pas évaluer la qualité ou la pertinence des options disponibles. Cette similarité avec random_agent se reflète également dans leur taux de réussite, qui sont presque identiques.

Corrélation validité vs performance

Les modèles qui ont un faible taux de coups illégaux (meta/llama-3.1-405b, nvidia/nemotron-ultra-253b, meta/llama-4-maverick) se positionnent en tête des LLMs dans les notes Glicko. Cela indique que même avec une marge de 5 essais, les modèles qui ont du mal à donner des coups légaux rencontrent également des difficultés à choisir le coup gagnant, ce qui souligne une incompréhension fondamentale de l'environnement.

En résumé, les résultats dévoilent que ni la taille du modèle, ni un entraînement orienté CoT ne sont des garants absolus de performance. Cette diversité de comportements souligne la nécessité d'une évaluation fine au-delà des seules métriques de paramètres ou d'architecture.

Fiabilité des mesures et limites de l'étude

Aides externes

Fournir la liste des coups légaux modifie considérablement la tâche par rapport à un jeu libre, car cela élimine l'étape de validation autonome des coups avant leur soumission. Dans un jeu libre, le joueur doit non seulement déterminer un coup potentiellement bon, mais aussi vérifier

qu'il est conforme aux règles des échecs avant de le jouer. Cette validation autonome exige une connaissance approfondie des règles et une attention constante, ce qui peut être cognitivement exigeant.

En fournissant la liste des coups légaux, on élimine cette charge cognitive, permettant au modèle de se concentrer uniquement sur la sélection du meilleur coup parmi les options disponibles.

Par conséquent, les performances mesurées dans un contexte où la liste des coups légaux est fournie ne sont pas nécessairement représentatives de celles qu'un joueur obtiendrait dans un vrai jeu d'échecs sans assistance. Dans un vrai jeu, le joueur devrait consacrer une partie de ses ressources cognitives à la validation des coups, ce qui pourrait affecter négativement ses performances globales. Ainsi, les résultats obtenus dans un environnement assisté par la liste des coups légaux peuvent être considérés comme optimistes par rapport à un vrai jeu d'échecs sans assistance.

Prompting et dépendance aux paramètres

Il est important de noter que les résultats obtenus sont étroitement liés aux paramètres spécifiques utilisés lors de la génération, notamment la température et le top_p. Ces paramètres contrôlent le degré de créativité du modèle. Ainsi, en modifiant ces valeurs, il est possible d'obtenir des résultats sensiblement différents en termes de cohérence, de pertinence.

De plus, la structure du prompt, c'est-à-dire la manière dont la requête est formulée, joue également un rôle crucial dans la qualité des résultats. Un prompt clair et précis, contenant des instructions explicites, guidera le modèle vers la réponse souhaitée. À l’inverse, une requête qui manque de clarté et qui laisse place à l'interprétation quant à la méthode pour obtenir le résultat peut produire des résultats incohérents ou non pertinents.

Par conséquent, il est recommandé d'expérimenter avec différentes combinaisons de paramètres et de structures de prompt afin d'optimiser les performances du modèle et d'obtenir les résultats les plus pertinents pour chaque cas d'utilisation spécifique. Il est également important de garder à l'esprit que les modèles de langage sont en constante évolution, et que de nouvelles techniques de prompt sont régulièrement développées pour améliorer leurs performances.

Généralisabilité

La généralisabilité de cette étude est limitée par plusieurs facteurs. Premièrement, seuls les modèles open source hébergés sur Nvidia NIM ont été inclus dans les tests. Il est possible que les modèles propriétaires fermés, tels que GPT-o3 et Claude Sonnet, présentent des comportements, des performances et des biais différents en raison de leurs architectures, de leurs données d'entraînement et de leurs stratégies d'optimisation spécifiques. Deuxièmement, le fait que l'environnement de test soit limité à la plateforme Nvidia NIM pourrait ne pas refléter les performances de ces modèles dans d'autres configurations logicielles d’autres hébergeurs, comme avec différents niveaux de quantisation. Par conséquent, il est essentiel d'étendre cette recherche à d'autres modèles et plateformes afin d'obtenir une image plus complète et plus précise du paysage actuel des modèles de langage.

Qualité du système d’évaluation et perspectives

Le système d'évaluation automatisé mis en place a démontré une robustesse et une flexibilité remarquables, permettant non seulement de tester de multiples agents et puzzles simultanément, mais aussi de gérer efficacement les erreurs potentielles, tout en générant un

corpus de données conséquent et propice à l'analyse approfondie. Les améliorations et extensions futures envisageables pour ce système sont nombreuses et variées :

- **Intégration de modèles propriétaires de haute performance** : Bien que ces modèles puissent entraîner des coûts plus élevés, leur inclusion permettrait de mieux généraliser les résultats aux capacités actuelles des LLMs.
- **Diversification des stratégies de prompting** : L'exploration de variantes telles que le multi-shot prompting ou le zero-shot prompting sans liste de coups fournirait une évaluation plus complète des capacités du système face à différentes méthodes de formulation des requêtes.
- **Évaluation sur des parties complètes** : Au lieu de se limiter à des puzzles isolés, l'évaluation du système sur des parties entières offrirait un aperçu de ses performances dans des conditions de jeu réelles.
- **Inclusion de tests sur des joueurs humains** : Plutôt que de déduire les notes humaines à partir des notes des puzzles, l'intégration de tests sur des joueurs humains permettrait une comparaison directe entre les performances du système et celles de joueurs réels.
- **Utilisation de la stabilité de la déviation pour l'arrêt précoce** : En surveillant la stabilité de la déviation des métriques au fil du temps, le système pourrait être configuré pour s'arrêter automatiquement lorsque les métriques se stabilisent, évitant ainsi le gaspillage de temps, d'argent (dans le cas de l'utilisation d'une plateforme payante) et de ressources de calcul.

Respect des contraintes

Malgré l'allocation d'une partie importante du budget à un modèle unique, Deepseek-R1, dont les résultats se sont avérés infructueux, le budget global du projet n'a pas été dépassé. Cette situation est principalement due à l'utilisation stratégique des ressources gratuites de NVIDIA NIM, qui ont permis de compenser les dépenses liées à Deepseek-R1 et de poursuivre les recherches dans le cadre du budget initialement prévu.

En ce qui concerne le respect de l'échéancier du calendrier final, nous avons initialement suivi le plan établi dès le début du projet. Cependant, lors de la phase de développement, nous avons rencontré des retards significatifs. Cette phase a nécessité plus de temps que prévu initialement, car nous avons dû revoir notre conception en raison d'un manque d'expérience dans la construction d'un programme similaire. De plus, nous avons été confrontés à des problèmes imprévus en cours de route, tels que le coût inattendu des modèles de raisonnement et les restrictions parfois draconiennes sur le taux de requêtes. Ces difficultés ont eu un impact en cascade sur les phases subséquentes, notamment les tests et la collecte de données, qui dépendaient de l'achèvement du développement. Le tableau suivant présente une comparaison détaillée entre le calendrier prévu et le calendrier réel, mettant en évidence les écarts et les retards accumulés au cours du projet :

Nom de la tâche	**Période d'exécution estimée**	**Période d'exécution réelle**

Recherche préliminaire	20 jan - 3 fév	20 jan - 3 fév
Conception de l’environnement	3 fév - 3 mars	3 fév - 3 mars
Développement de l’environnement	17 fév - 24 mars	17 fév - 11 avril
Rédaction du rapport de progrès	17 fév - 7 mars	17 fév - 7 mars
Tests et collecte des données	17 mars - 24 mars	31 mars - 23 avril
Analyse des résultats	24 mars - 7 avril	18 avril - 23 avril
Rédaction du rapport final	17 mars - 14 avril	4 avril - 23 avril
Production de la vidéo de présentation	14 avril - 25 avril	23 - 25 avril

Conclusion

Notre étude visait à évaluer systématiquement les capacités des Grands Modèles de Langage (LLMs) dans le domaine des échecs à travers la résolution de puzzles spécifiques. L’objectif principal consistait à mesurer leurs performances à l'échelle humaine. Les objectifs secondaires étaient d’identifier leurs forces et faiblesses et de développer un système d'évaluation automatisé.

Nous avons réussi à atteindre nos objectifs en nous intéressant à cinq modèles. Leur classement sur une échelle humaine était plus bas que le niveau d’un joueur moyen et il n’y a pas de force ou de faiblesse significative, lorsqu’on s’intéresse aux thèmes des puzzles.

Nos résultats sont concluants avec les autres études qui portent sur la capacité des LLMs à jouer aux échecs. Ils ne sont pas très bons et ils ont encore tendance à faire des hallucinations.

Notre travail présente plusieurs limitations importantes. Le biais potentiel dans la sélection des puzzles de la base Lichess pourrait influencer les résultats. L'aide externe fournie sous forme de liste de coups légaux modifie la tâche par rapport à un jeu libre, rendant les résultats plus optimistes. La dépendance aux paramètres spécifiques de prompting (température, top_p) et à la structure du prompt pourrait également affecter la généralisation des résultats.

La généralisabilité est également limitée par la restriction aux modèles open source hébergés sur Nvidia NIM, excluant les modèles propriétaires fermés comme GPT-o3 et Claude Sonnet, qui pourraient présenter des comportements différents. Enfin, l'environnement de test limité à la

plateforme Nvidia NIM pourrait ne pas refléter les performances de ces modèles dans d'autres configurations logicielles.

À la lumière de ces résultats et limitations, plusieurs pistes de recherche futures se dégagent :

1. **Amélioration du système d'évaluation** : Tester les modèles sans fournir la liste des coups légaux pour évaluer leur capacité à respecter les règles du jeu de manière autonome. Explorer différentes stratégies de prompting, y compris le multi-shot et le zero-shot, pour évaluer leur impact sur les performances.
2. **Élargissement de l'étude** : Intégrer des modèles propriétaires de haute performance pour une évaluation plus complète du paysage actuel des LLMs.
3. **Analyse approfondie** : Examiner plus en détail les causes des erreurs et des hallucinations pour identifier les mécanismes sous-jacents qui limitent les performances.
4. **Optimisation des ressources** : Utiliser la stabilité de la déviation comme critère d'arrêt précoce pour économiser du temps et des ressources de calcul.
5. **Agent de référence humain** : Comparer directement les performances avec celles de joueurs humains de différents niveaux pour une calibration plus précise.

Bibliographie

[1] R. Burnell, H. Hao, A. R. A. Conway, et J. H. Orallo, « Revealing the structure of language model capabilities », 14 juin 2023, *arXiv*: arXiv:2306.10062. doi: 10.48550/arXiv.2306.10062.

[2] K. Cobbe *et al.*, « Training Verifiers to Solve Math Word Problems », 18 novembre 2021, *arXiv*: arXiv:2110.14168. doi: 10.48550/arXiv.2110.14168.

[3] M. Stezano, « In 1950, Alan Turing Created a Chess Computer Program That Prefigured A.I. », HISTORY. Consulté le: 25 avril 2025. [En ligne]. Disponible à: https://www.history.com/articles/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i

[4] M.-T. Kuo, C.-C. Hsueh, et R. T.-H. Tsai, « Large Language Models on the Chessboard: A Study on ChatGPT's Formal Language Comprehension and Complex Reasoning Skills », 29 août 2023, *arXiv*: arXiv:2308.15118. doi: 10.48550/arXiv.2308.15118.

[5] Interested readers of the Internet newsgroup rec.games.chess, « Standard: Portable Game Notation Specification and Implementation Guide ». Consulté le: 3 mars 2025. [En ligne]. Disponible à: https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt

[6] P. M. E. Glickman, « Example of the Glicko-2 system », mars 2022.

[7] « What are tokens and how to count them? | OpenAI Help Center ». Consulté le: 25 avril 2025. [En ligne]. Disponible à: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

[8] « llama-3.1-nemotron-ultra-253b-v1 Model by NVIDIA », NVIDIA NIM. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/nvidia/llama-3_1-nemotron-ultra-253b-v1

[9] « llama-3.1-405b-instruct Model by Meta », NVIDIA NIM. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/meta/llama-3_1-405b-instruct

[10] « gemma-3-27b-it Model by Google », NVIDIA NIM. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/google/gemma-3-27b-it

[11] « llama3-8b-instruct Model by Meta », NVIDIA NIM. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/meta/llama3-8b

[12] « llama-4-maverick-17b-128e-instruct Model by Meta », NVIDIA NIM. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/meta/llama-4-maverick-17b-128e-instruct

[13] M. de la Vega, « Understanding OpenAI's "Temperature" and "Top_p" Parameters in Language Models », Medium. Consulté le: 23 avril 2025. [En ligne]. Disponible à: https://medium.com/@1511425435311/understanding-openais-temperature-and-top-p-parameters-in-language-models-d2066504684f

[14] « deepseek-r1 Model by Deepseek-ai », NVIDIA NIM. Consulté le: 25 avril 2025. [En ligne]. Disponible à: https://build.nvidia.com/deepseek-ai/deepseek-r1

Annexes

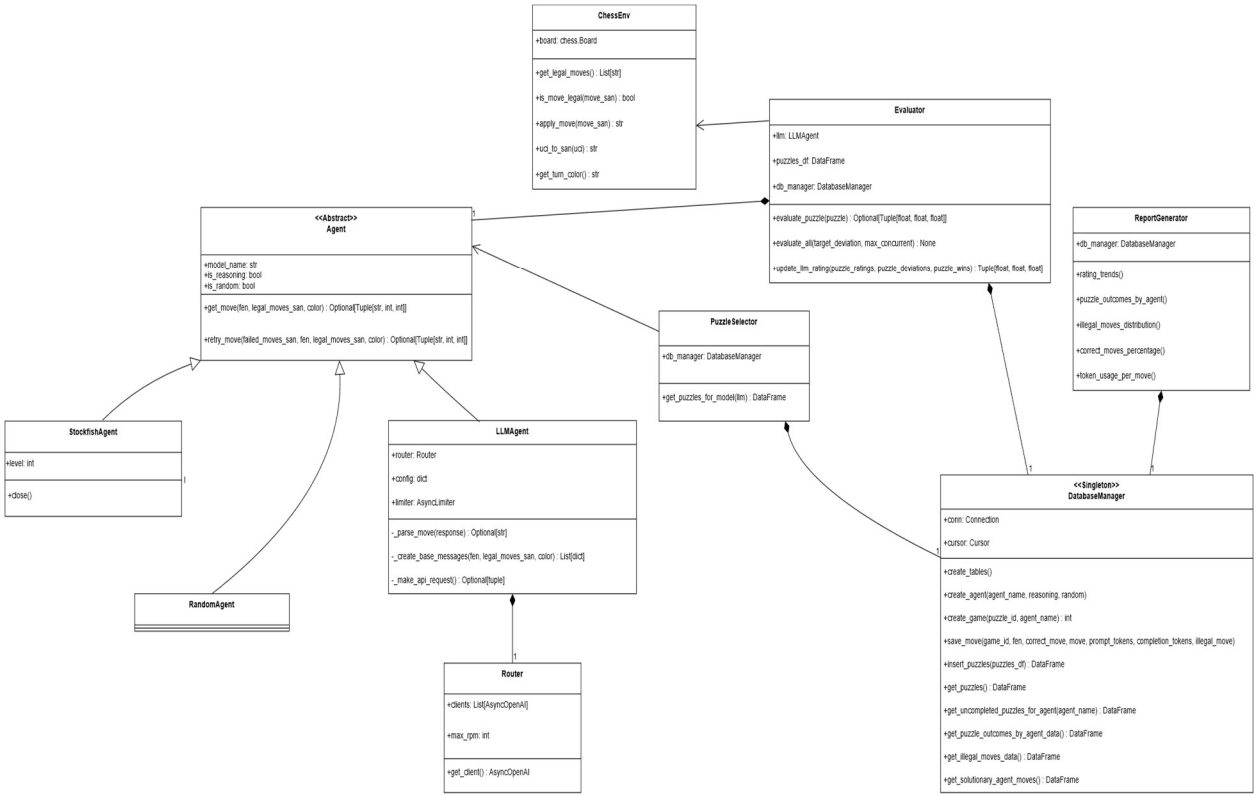

Figure 1 : Diagramme de classes de l'architecture système

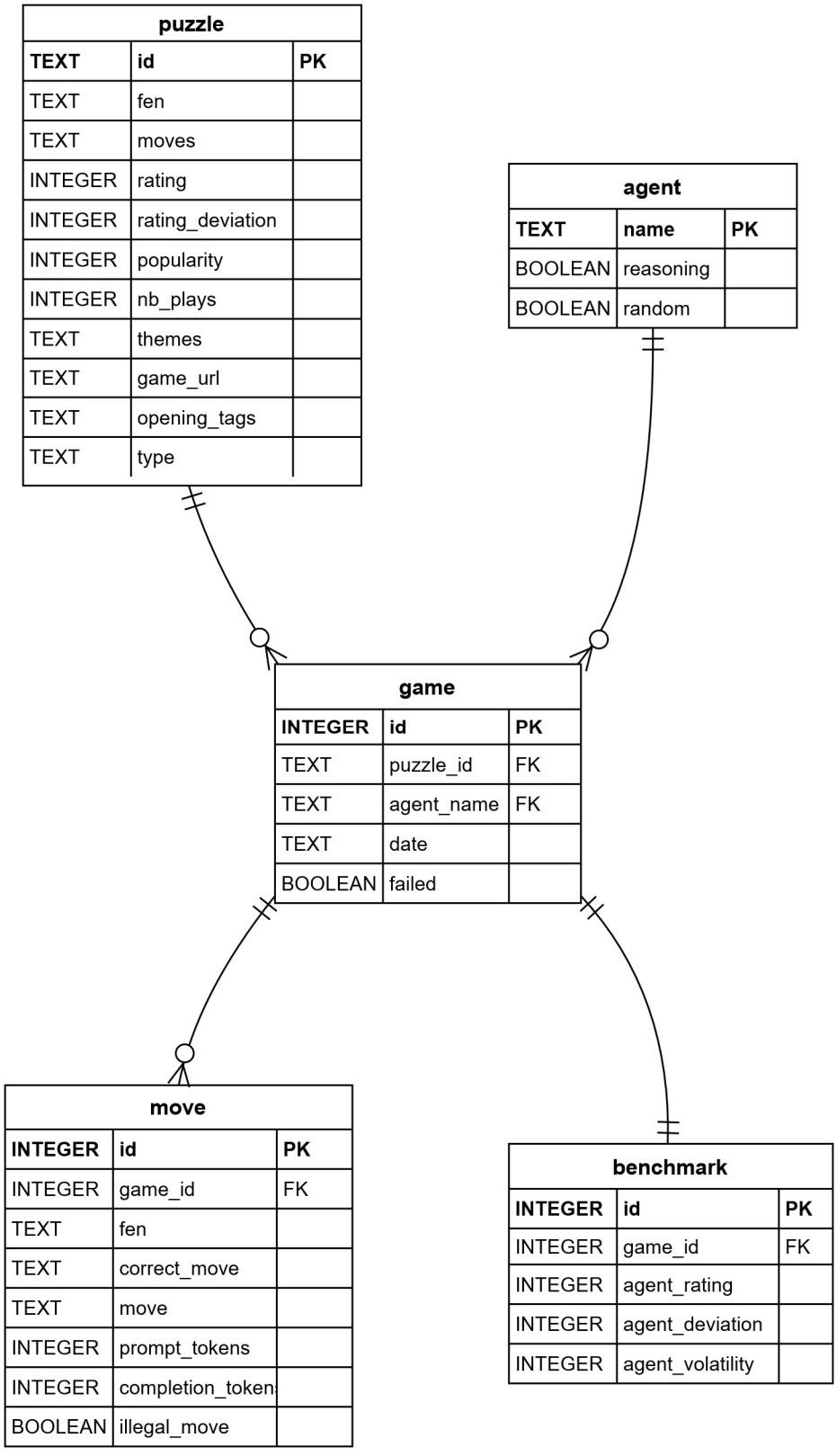

Figure 2 : Diagramme Entité-Relation de la base de données SQLite

```
"""
You are a world-class chess grandmaster and expert analyst. You are presented with the following chess puzzle:

Chess board position (FEN): {fen}
Legal moves for {color}: {', '.join(legal_moves_san)}

Analyze the position. Then, on a new line, output your final chosen move in Standard Algebraic Notation (SAN) enclosed in <FinalMove> tags. For example:

<FinalMove>b5</FinalMove>

Ensure that the final output line contains only the <FinalMove> tag with the move and no additional text, markdown, or formatting.
"""
```

Figure 3 : Image du prompt utilisée pour les requêtes

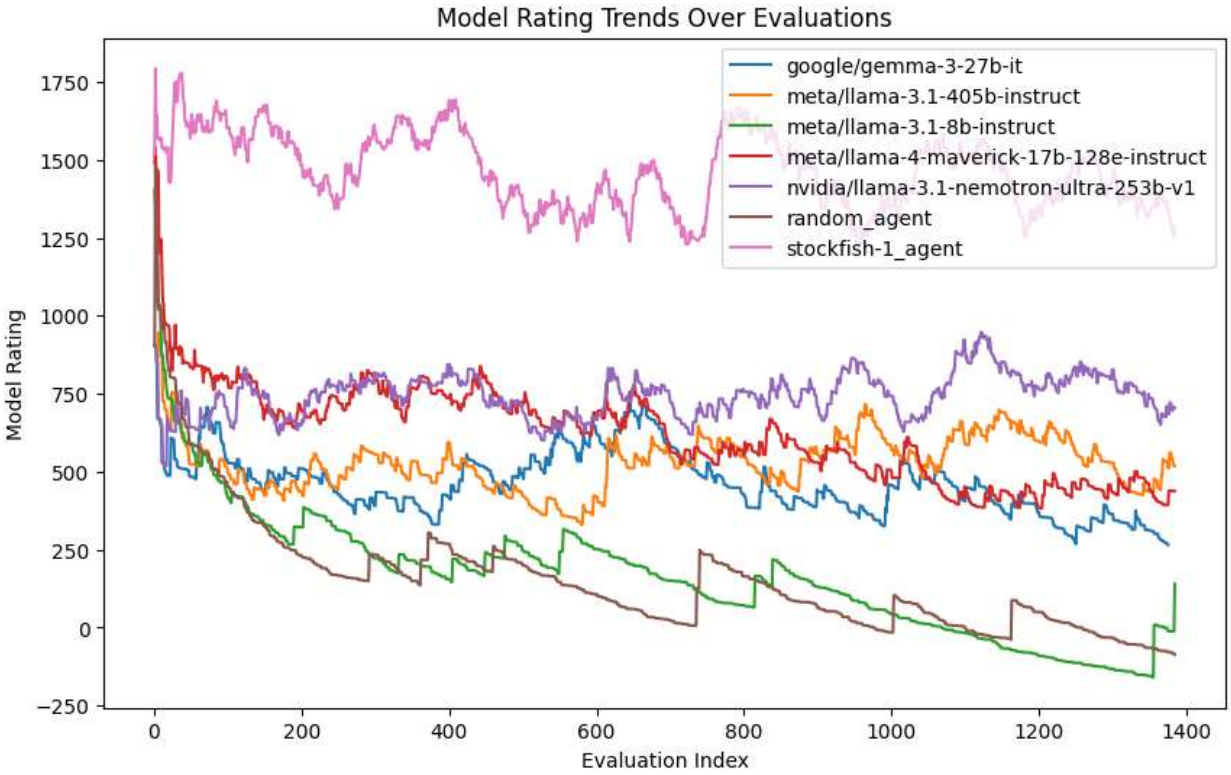

Figure 4 : Tendances d'évolution des ratings des agents

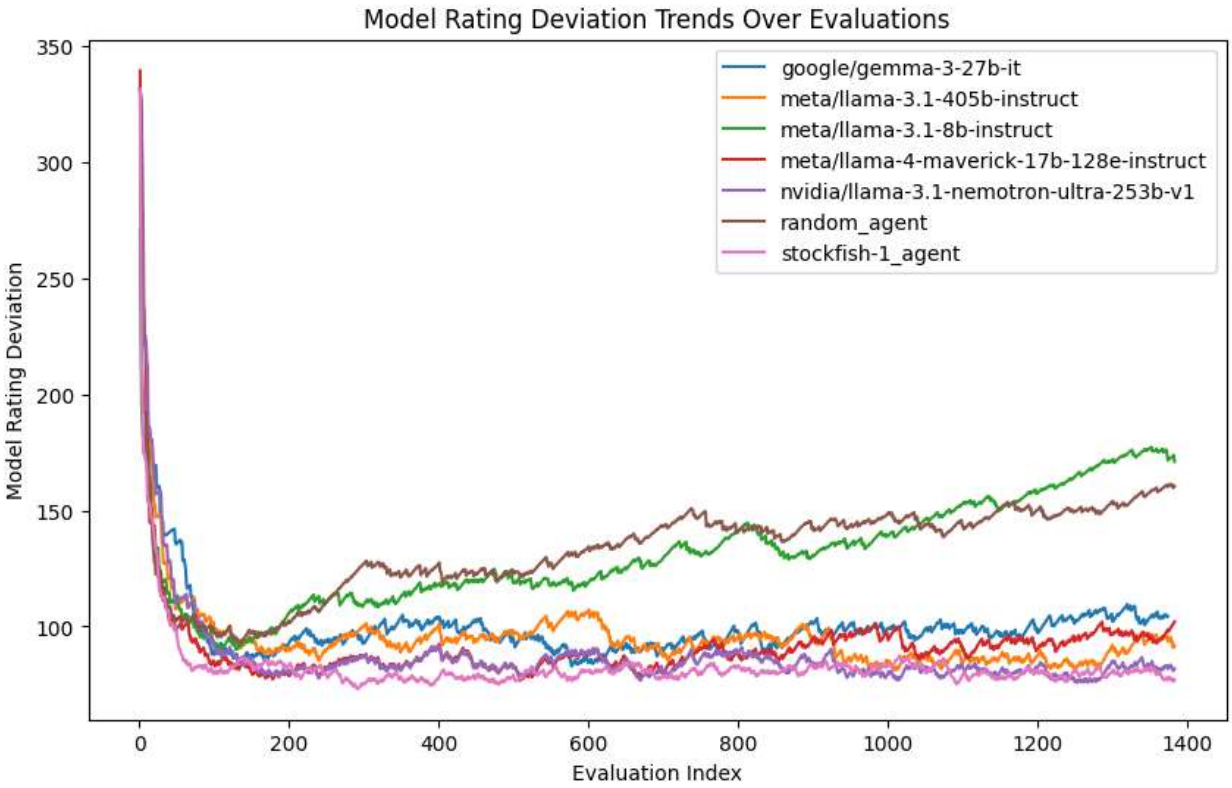

Figure 5 : Tendances d'évolution des déviations des agents

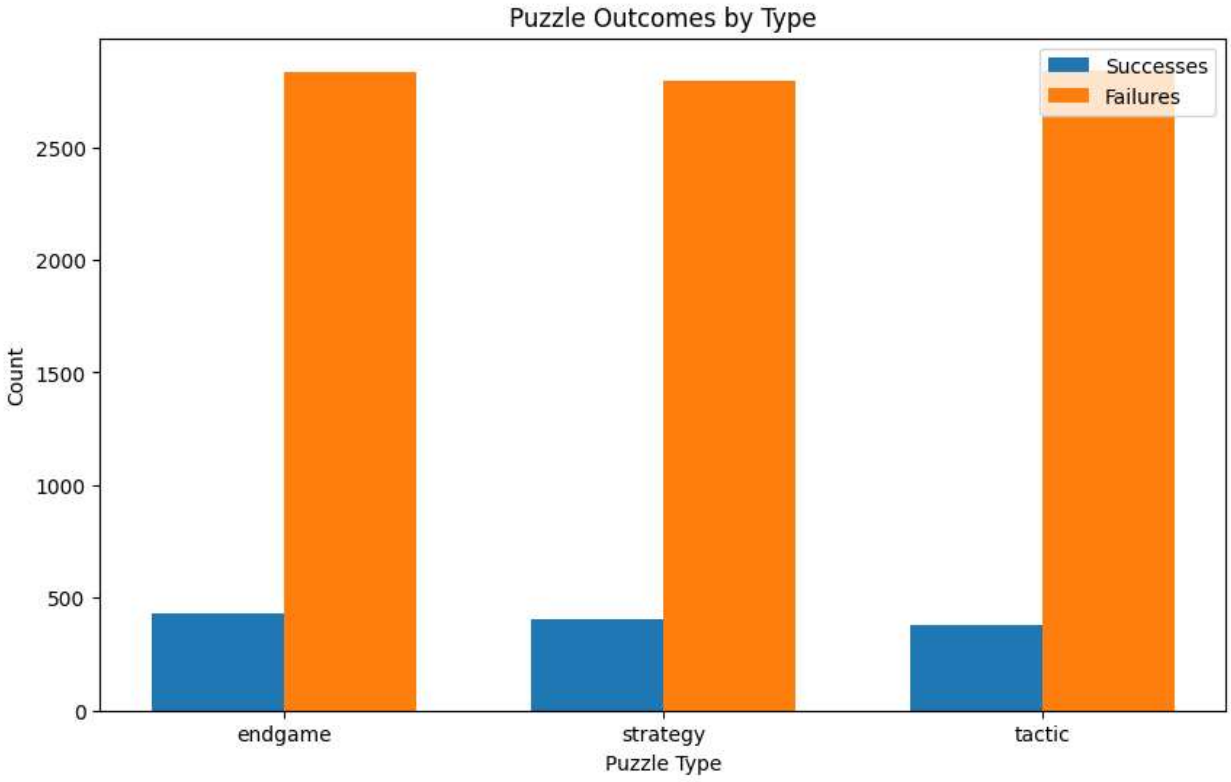

Figure 6 : Résultats agrégés des évaluations par thème

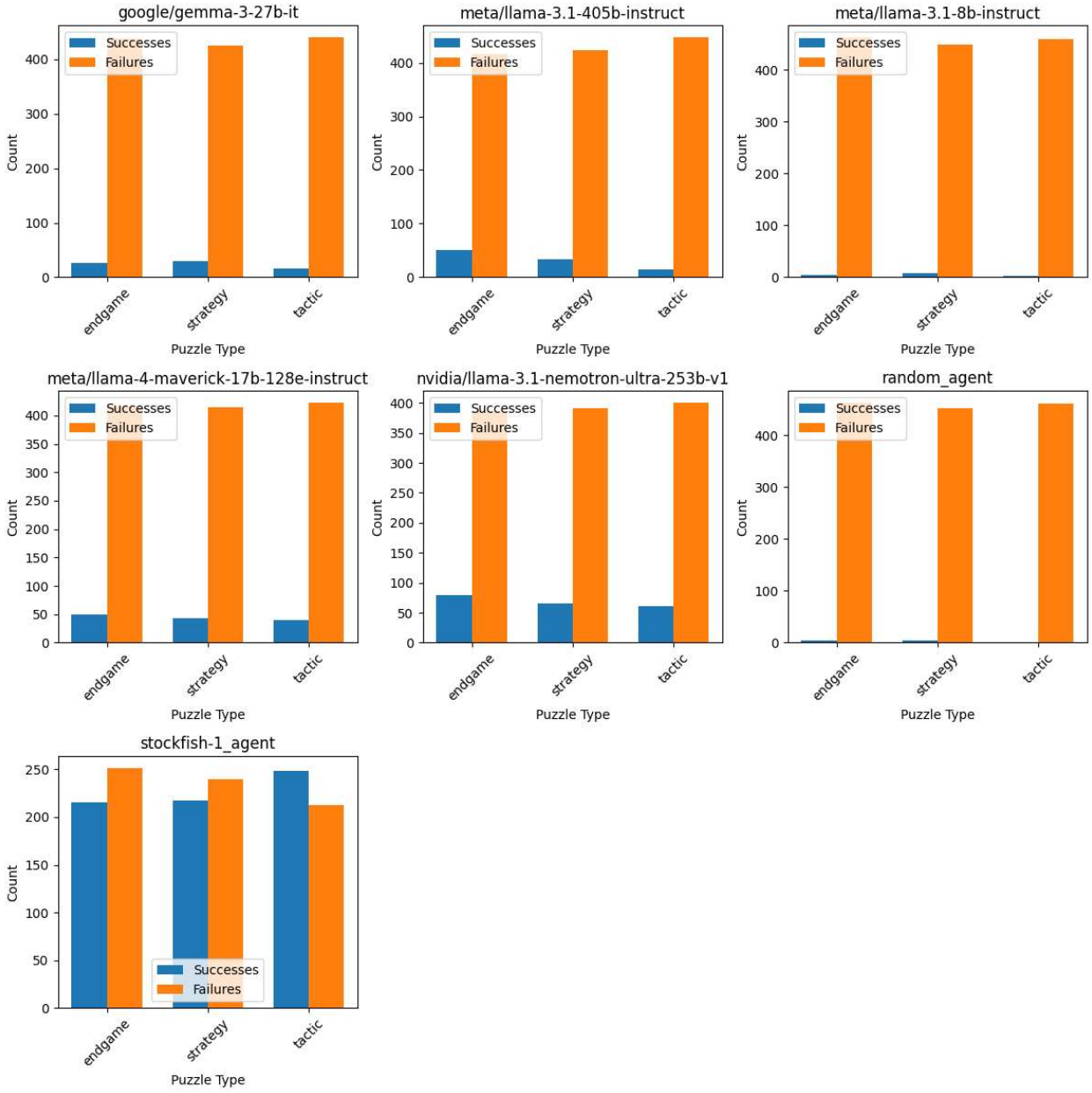

Figure 7: Résultats des évaluations thématique par modèle

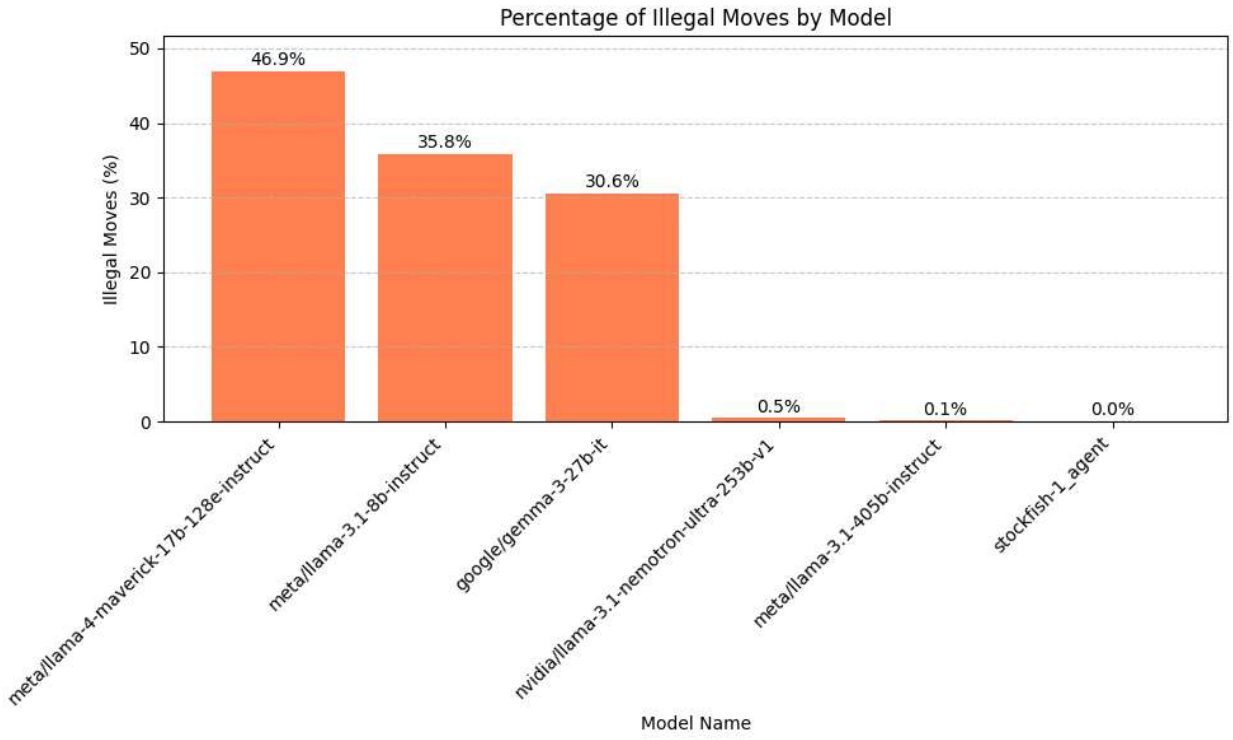

Figure 8 : Pourcentage des coups illégaux par modèle

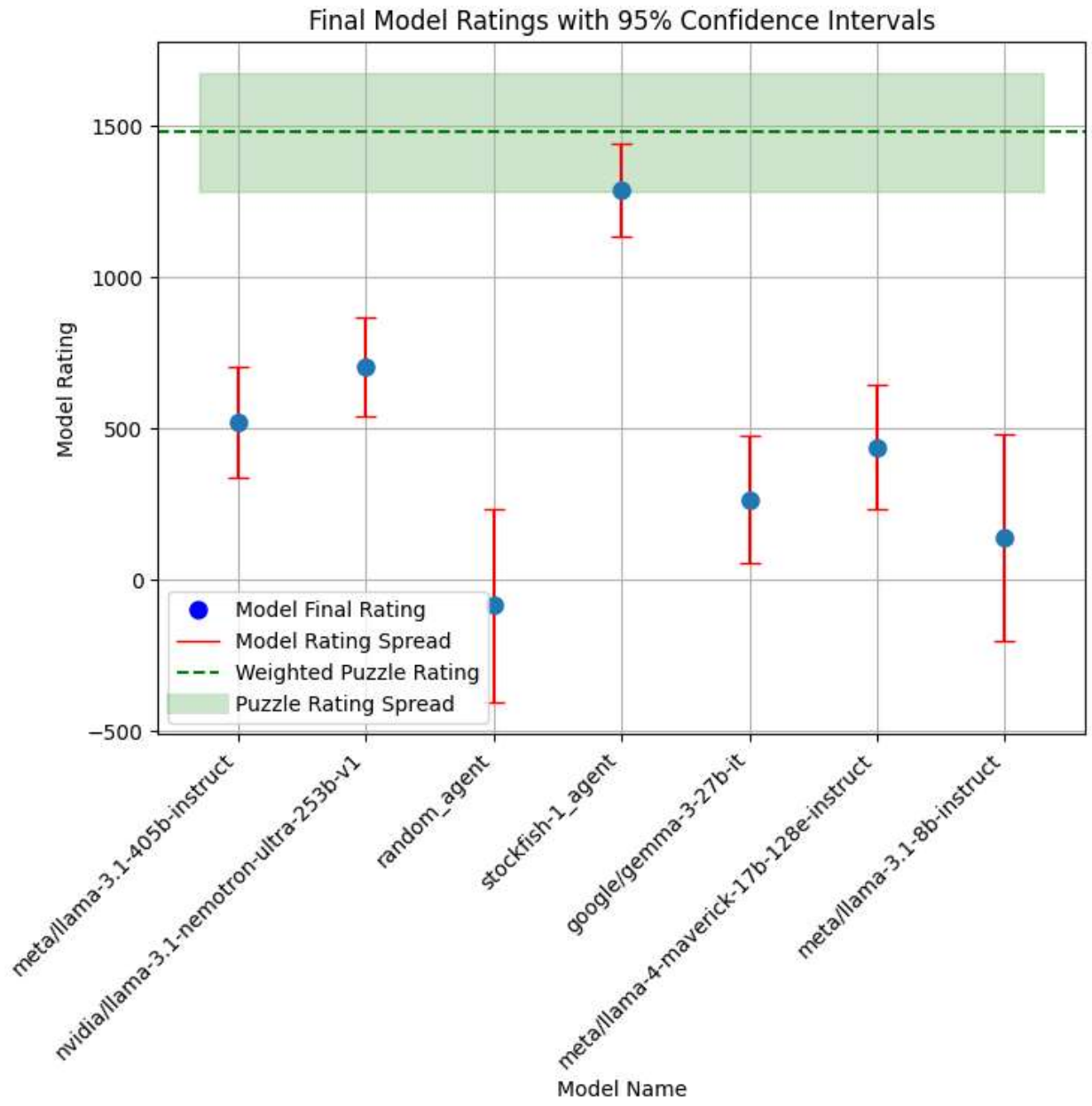

Figure 9 : Classements finals des agents

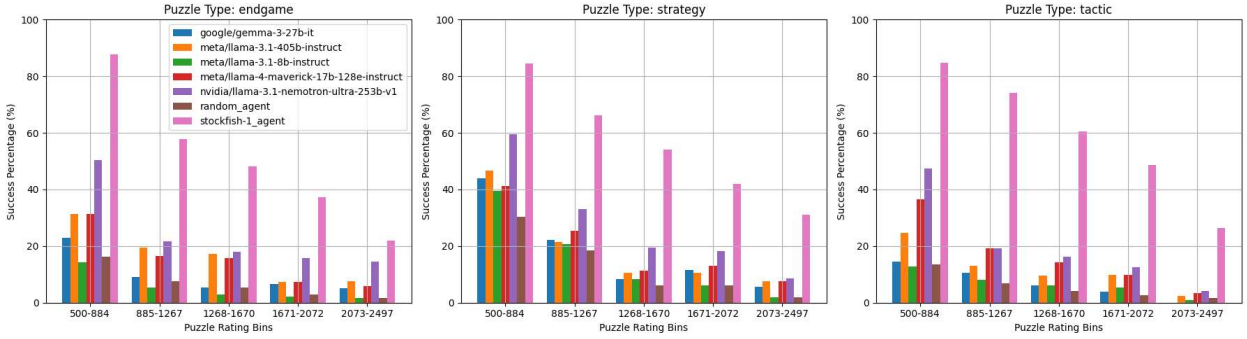

Figure 10 : Pourcentage de succès par thème, segmenté par fourchette d'évaluation