PROJET TUTORÉ

**Rapport final**

ligne horizontale

# 

ligne horizontale

Responsable : POUZIN Pierre-Emmanuel Groupe n°1

Table des Matières

[**1) Rappel du contexte du projet et des objectifs 4**](#_uzg6dnxged34)

[**2) Fonctionnalités réalisées 4**](#_a70n7o5hkt5l)

[2.1 Front 4](#_2w5l2c7d51lq)

[2.1.1 Descriptif fonctionnel 4](#_hqvqihzovv)

[2.1.2 Descriptif technique 4](#_f5i7i1x9nfz4)

[2.1.2.1 Structure des pages 5](#_ed7g9wm3g214)

[2.1.2.2 Authentification 5](#_3ml7ggg7tmve)

[2.1.2.3 Parties 6](#_y80azrhrl1un)

[2.1.3 Tâche/Répartition 7](#_gtrh9yemzps1)

[2.1.4 Justification 7](#_xrtjlbroz008)

[2.2 Back 8](#_eotoxlxh3kas)

[2.2.1 Descriptif fonctionnel 8](#_r91rv8oeinxp)

[2.2.2 Descriptif technique 8](#_nti0vsslbq0g)

[2.2.2.1 Création de la base de données 8](#_d0dpdihafkim)

[2.2.2.2 Mapping des objets java avec Spring data JPA et Springboot 8](#_rgw4aehrspwe)

[2.2.3 Tâche/Répartition 10](#_bjww3w5uio6h)

[2.2.4 Justification 11](#_i1zhjy4qkbvn)

[2.3 ETL 11](#_88lylyegerc0)

[2.3.1 Descriptif fonctionnel 11](#_v2f9k2o5tec6)

[2.3.2 Descriptif technique 12](#_kky93j72gkr)

[2.3.2.1 Extraction des données 12](#_9tyoz5wu09e)

[2.3.2.2 Insertion des données 12](#_f81o2fpkex35)

[2.3.3 Répartition/Temps 14](#_mmhfjdn5g0n3)

[2.3.4 Justification 14](#_g6qda4wk4atv)

[2.4 Etat de la Base de Données 14](#_ei95mvf340qp)

[2.4.1 Descriptif fonctionnel 14](#_aer5ihyqikt0)

[2.4.2 Descriptif technique 14](#_sf5ulm9a9z3f)

[2.4.2.1 Player 14](#_yng1p0ia6xsx)

[2.4.2.2 Game 15](#_69b0grkosps9)

[2.4.2.3 Moves 15](#_aj1yj5ch6564)

[2.4.2.4 Tournament 15](#_ttwtm5w3eds8)

[2.4.2.5 Chess\_Opening 15](#_43dxi6ootycy)

[2.4.4 Répartition/Temps 15](#_l7ck6e5o7q0n)

[2.4.5 Justification 15](#_he2kzjqmoa0i)

[2.5 API 16](#_6lbjf1ww6syu)

[2.5.1 Description fonctionnel 16](#_99esik3wmjjh)

[2.5.2 Description technique 16](#_7j5523qdnm5k)

[2.5.2.1 Appel de l’API 16](#_lx4ymmnbsebf)

[2.5.2.2 Extraction de données 16](#_2utjgfwwk4hn)

[2.5.2.3 Insertion de données 17](#_roartkyxsyry)

[2.5.3 Répartition/Temps 17](#_kphnchnlhgoh)

[2.5.4 Justification 17](#_esspwbl2vc8s)

[**3) Retour d’expérience sur l’environnement de développement et les choix techniques 18**](#_wks2dmhqmu6c)

[4.1 Git 18](#_7n7haffbpsly)

[4.2 Docker 18](#_fgzjlmpunge)

[4.3 Technologies Front 19](#_2sla2jexj4v5)

[4.3 Technologies ETL 20](#_qe609uuegqf)

[4.3 Technologies Back 20](#_lkw2veqe4l1b)

[**4) Retour d’expérience sur l’organisation du groupe 21**](#_xzp7d4a3s3rz)

[5.1 Planification et distribution des tâches 21](#_9hb41yfxa7b1)

[Ce qui a fonctionné 21](#_vpomkjkwzozp)

[Ce qui n’a pas fonctionné 21](#_mbeashhawpoa)

[Solutions à mettre en place pour améliorer la planification des tâches 22](#_9xcfvp18orn4)

[5.2 Communication 22](#_5ch3pqhebv2e)

[Ce qui a fonctionné 22](#_mri4a5fpsywo)

[Ce qui n’a pas fonctionné 22](#_skqrsp39ou6j)

[Axe d’amélioration 23](#_2j0006zelz1z)

[**5) Conclusion 24**](#_q2x18kyrldfl)

## 

## Rappel du contexte du projet et des objectifs

Le projet vise à développer une application autour des échecs, offrant une aide à la décision en analysant les parties en temps réel. En s'appuyant sur des données de parties issues de Lichess, l'application fournira des indications sur les coups joués et proposera des recommandations pour les coups futurs. Elle permettra également aux utilisateurs de disputer des parties contre des joueurs, assistés ou non par des moteurs comme Stockfish, et intégrera un système de détection de tricheurs basé sur l'analyse des coups.  
  
 Pour inclure les données de Lichess dans notre base de données, nous développerons un programme ETL. Ce programme filtrera les données entrantes pour conserver uniquement celles qui sont utiles à notre analyse, garantissant ainsi la pertinence et l'efficacité de notre système.

## Fonctionnalités réalisées

### 2.1 Front

#### 2.1.1 Descriptif fonctionnel

Le front à pour but de permettre aux utilisateurs de se connecter, de jouer des parties d’échecs soit contre un autre utilisateur soit contre un bot, tout en donnant une indication pour le prochain coup à jouer, de revoir des parties, et de consulter des statistiques sur ces parties.

#### 2.1.2 Descriptif technique

L’interface utilisateur comprend un échiquier interactif, rendu grâce à la bibliothèque react-chessboard, où le joueur peut déplacer ses pièces. Un chronomètre est intégré pour chaque joueur, indiquant le temps restant pour les Blancs et les Noirs. L’application détecte automatiquement l’état du jeu, comme les situations d’échec et mat, de pat ou de match nul, en s’appuyant sur la bibliothèque chess.js.

Pour rendre l’expérience plus immersive, l’intelligence artificielle *Stockfish* est exécutée dans un **web worker** afin d’éviter de bloquer l’interface utilisateur. Le niveau de jeu du bot est configurable via un paramètre prédéfini. Lorsque c’est au tour du bot, ce dernier analyse l’état de l’échiquier et génère automatiquement son meilleur coup, tout en ajoutant un délai aléatoire pour simuler un comportement réaliste.

L’application gère également un historique des coups en notation algébrique, visible dans une section dédiée. À la fin de la partie, l’utilisateur peut télécharger un fichier au format **PGN** (Portable Game Notation) contenant les métadonnées et l’historique complet des mouvements. En termes techniques, l'application utilise des **hooks React**, comme useState pour la gestion des états et useEffect pour les cycles de vie. Les chronomètres des joueurs sont synchronisés avec le déroulement de la partie et basculent dynamiquement en fonction du joueur actif.

##### 2.1.2.1 Structure des pages

Notre application web est divisée en plusieurs pages, chacune ayant une fonction précise :

**Page de connexion :** L’utilisateur à le choix soit de se connecter à notre application via un compte Lichess soit de jouer une partie contre un bot en tant qu’invité. Si l’utilisateur décide de jouer en tant qu’invité, ses parties ne seront pas enregistrées.

**Page de profil :** Une fois connecté, l’utilisateur est redirigé sur la page de son profil. Cette page est découpée en 3 parties principales : le header, les statistiques et l’historique de parties. Dans le header, l’utilisateur peut trouver un bouton pour jouer contre un joueur et un bouton pour jouer contre un bot, qui à l’appui, font apparaître un pop-up permettant à l’utilisateur de spécifier les règles de la partie, et enfin un bouton de déconnexion. Dans les statistiques, situées sur la moitié gauche de la page, l’utilisateur peut retrouver son pseudo et des statistiques chiffrées dans chaque mode de jeu (Classique, Blitz et Bullet) tels que son elo actuel, son meilleur elo ou encore son nombre de parties jouées et bien d’autres. L’historique de parties, situé sur la moitié droite de la page, liste les dernières parties jouées par l’utilisateur en précisant le mode de jeu, les pseudos de l’utilisateur et de son adversaire, leur elo et le vainqueur.

**Page de jeu :** Lors d’une partie, l’utilisateur peut voir son pseudo et son elo ainsi que ceux de son adversaire et les clocks au-dessus de l’échiquier. Le nom de l’ouverture jouée et l’historique de coups sont visibles sur la droite de l’échiquier. À la fin de la partie, un message apparaît avec le nom du gagnant et la manière dont la partie s’est terminée.

##### 2.1.2.2 Authentification

L'authentification est basé sur **OAuth 2.0 avec PKCE (Proof Key for Code Exchange)**, qui garantit une sécurité en utilisant la cryptographie pour valider l'échange d'informations entre le client et Lichess.

Le processus commence lorsque l'utilisateur accède à l'application, où une page d'accueil simple propose un lien de connexion. En cliquant sur ce lien de connexion à Lichess, l'utilisateur est redirigé vers un endpoint interne (/login) où une chaîne aléatoire appelée **Code Verifier** est générée côté serveur. Ce **Code Verifier** est ensuite transformé en un **Code Challenge** via une fonction de hachage SHA-256 et encodé en Base64. Ces valeurs sont utilisées pour sécuriser la communication entre le client et Lichess.

Le serveur conserve le C**ode Verifier** dans la session utilisateur, tandis que le **Code Challenge**, accompagné d'autres informations nécessaires comme l'identifiant client (client\_id), l'URL de redirection (redirect\_uri) et la portée d'accès (scope), est transmis à Lichess dans une requête de redirection. Cette requête redirige l'utilisateur vers la page d'autorisation de Lichess, où il peut approuver ou refuser l'accès demandé par l'application.

Une fois l'autorisation accordée, Lichess redirige l'utilisateur vers l'URL spécifiée (/callback) avec un **code d'autorisation** inclus dans les paramètres de la requête. À ce stade, l'application effectue un échange du code d'autorisation contre un **jeton d'accès** en envoyant une requête POST sécurisée au serveur Lichess, en incluant le **Code Verifier** précédemment généré. Ce mécanisme garantit que seul le client qui a initié la requête d'autorisation peut finaliser l'échange, empêchant ainsi des attaques de type interception.

Avec le jeton d'accès obtenu, l'application peut interagir avec l'API Lichess pour récupérer les informations de l'utilisateur authentifié, comme son nom d'utilisateur. L'utilisateur est alors redirigé vers une interface front (/profil) avec son jeton et son nom d'utilisateur inclus dans les paramètres de l'URL, permettant une personnalisation immédiate de l'expérience utilisateur.

##### 2.1.2.3 Parties

Les coups d'échecs sont récupérés et gérés grâce à la bibliothèque chess.js. Lorsque le joueur effectue un déplacement, la méthode game.move vérifie si le coup est valide, puis met à jour l'état du jeu (position en FEN et historique des coups). Les coups sont ensuite enregistrés dans un tableau appelé movesHistory, où chaque paire de coups (Blancs et Noirs) est regroupée pour un affichage clair. Si c'est au tour du bot, l'IA Stockfish calcule son meilleur coup en fonction de la position actuelle en FEN. Ce coup est ensuite appliqué au jeu et ajouté à l'historique.

Lorsque la partie est terminée, un bouton permet de télécharger l'historique complet des coups au format PGN. Ce fichier inclut non seulement la liste des coups mais également des métadonnées comme le nom des joueurs, la date, le résultat, et l'ouverture utilisée. Le téléchargement se fait en générant un fichier texte dynamique que l'utilisateur peut enregistrer localement. Enfin, pendant toute la partie, l'historique des coups est affiché sous forme de liste pour un suivi en temps réel.

#### 

#### 2.1.3 Tâche/Répartition

1. **Affichage du plateau de jeu d'échecs :** implémentation du composant principal en utilisant la bibliothèque react-chessboard pour afficher un échiquier interactif.
2. **Gestion des déplacements des pièces :** mise en place d'une logique permettant de déplacer les pièces sur l'échiquier et de mettre à jour la position en notation FEN après chaque coup.
3. **Mise en place du chronomètre :** ajout de deux chronomètres distincts pour le joueur et l'IA, avec une gestion dynamique en fonction du tour de jeu et une détection des fins de partie en cas de dépassement de temps.
4. **Intégration de l'IA Stockfish :** configuration d’un Web Worker pour interagir avec l’IA Stockfish, lui envoyer les positions actuelles, récupérer les meilleurs coups calculés et appliquer les mouvements de l'IA sur le plateau.
5. **Identification de l'ouverture jouée :** implémentation d'un mécanisme pour détecter et afficher le nom de l’ouverture utilisée après les premiers coups de la partie.
6. **Historique des parties :** création d’un historique des coups joués, affiché sous forme de liste, et génération d’un fichier PGN (Portable Game Notation) téléchargeable, contenant les métadonnées et les mouvements de la partie.
7. **Routage et récupération des données utilisateur API, Oauth:** utilisation du hook pour extraire les informations sur le joueur depuis l’URL, comme son nom d’utilisateur.
8. **Validation des conditions de fin de partie :** ajout de vérifications pour détecter les situations de victoire (échec et mat), de nul, et d'autres conditions spéciales, avec des messages affichés à l’écran.
9. **Stylisation et disposition :** utilisation d’un fichier CSS modulaire pour structurer et personnaliser l’apparence des différents composants de l’interface utilisateur.

#### 2.1.4 Justification

Le développement des fonctionnalités prévues progresse, mais plusieurs tâches importantes restent à réaliser avant leur finalisation.

Tout d'abord, l'envoi de l'historique des parties en format JSON est en attente de l'achèvement des routes backend nécessaires. Une fois ces routes mises en place, il sera possible de transmettre les données au frontend de manière efficace.

Ensuite, la récupération des informations sur les parties jouées et les statistiques associées depuis le backend est une priorité. Ces données devront être affichées sur l'interface utilisateur une fois que les routes correspondantes auront été développées.

Le développement de la fonctionnalité "Joueur contre joueur" est toujours en cours. Ce mode permettra d'ajouter une dimension compétitive entre deux utilisateurs, et son achèvement reste une tâche essentielle pour le projet.

Par ailleurs, l'identification et l'affichage du nom de l’ouverture jouée au cours de chaque partie doivent encore être implémentés. Cette fonctionnalité apportera une valeur ajoutée en permettant aux utilisateurs de mieux analyser leurs parties également à récupérer depuis notre BD, chessOpening.

Enfin, la métrique des coups, qui permet une analyse détaillée de chaque partie, reste à mettre en place. Elle contribuera à enrichir les statistiques et les outils d’évaluation des performances.

Ces étapes représentent des priorités importantes pour atteindre les objectifs fixés et offrir une expérience utilisateur complète et performante.  
  
Pour le moment et la fin du semestre, la priorité est d'assurer l'importation des données JSON depuis le backend vers le frontend. Ces données, comprenant les statistiques et les historiques des joueurs provenant de notre base de données, qui devront être affichées de manière claire et visuelle, notamment sous forme graphique.

### 2.2 Back

#### 2.2.1 Descriptif fonctionnel

Le **backend** est conçu pour ***gérer l'accès aux données*** des parties d'échecs, effectuer les traitements nécessaires pour ***fournir des recommandations*** de coups, des ***statistiques***, et ***détecter d'éventuelles tricheries*** par des joueurs non humains. Il joue un rôle central dans la gestion de la logique métier, la manipulation des données, ainsi que l'interaction avec le frontend et d'autres composants tels que l'ETL et la base de données.

#### 2.2.2 Descriptif technique

##### 2.2.2.1 Création de la base de données

Nous avons d'abord conçu un script SQL regroupant la création de toutes les tables nécessaires à notre base de données. Ces tables, ainsi que leurs attributs, ont été définis en fonction des informations que nous pouvions extraire des fichiers PGN, de l’API Lichess, ainsi que des statistiques que nous souhaitons collecter et analyser.

Par la suite, nous avons élaboré un petit jeu de données d'insertion pour effectuer des tests. Ce jeu de données a été construit manuellement et sert d'échantillon pour valider la structure des tables et débuter le travail de mapping des objets dans notre application. Cette étape nous permet de nous assurer que la base de données est correctement conçue et que les données sont insérées et manipulées de manière cohérente.

##### 2.2.2.2 Mapping des objets java avec Spring data JPA et Springboot

Pour réaliser les différentes opérations sur la base de données, nous avons décidé d'utiliser les fonctionnalités avantageuses que proposent Spring Data JPA et Hibernate, qui permettent non seulement de faciliter les échanges avec la base de données en fournissant une surcouche SQL qui formate les requêtes, mais aussi de mapper nos objets java (nos entités) sur les tuples présents dans la base, et ce, de manière automatique.

**Dépendances**

Le projet général étant généré avec **Springboot** et compilé avec **Maven**, il nous suffisait d'ajouter dans le fichier **pom.xml** les dépendances suivantes pour commencer le mappage :

**Spring Web :** Spring Web permet de démarrer un serveur web intégré, sur lequel sera exposée notre application. Dans notre cas, **Tomcat** est utilisé. Il permet aussi de créer les API REST, nécessaires à l'élaboration des routes entre la partie « front » et « back » de l'application.

**Spring Data JPA :** Spring Data JPA, qui intègre le module Hibernate, permet de faciliter les opérations et d'annoter les classes d'objets pour les mapper aux tables de la base de données.

**Lombok :** Lombok est une dépendance d'utilitaires, facultatif, qui ajoute des fonctionnalités qui facilitent la vie du développeur. Parmi elles, la possibilité d'annoter les classes avec « @AllArgsConstructor », « @Builder » ou encore « @Getter » pour respectivement générer automatiquement un constructeur standard, créer une instance d'une classe avec des paramètres personnalisés, ou générer tous les getters de la classe.

À celles-ci s'ajoutent les dépendances qui étaient déjà nécessaires pour faire fonctionner le projet, à savoir **JDBC API** qui permet de se connecter à la base de données avec java, et **PostgreSQL Driver**, nécessaire également.

**Architecture**

Dans l'objectif de structurer notre application, notre choix (motivé par les ressources de formation disponibles sur internet, comme les tutoriels et la documentation de Springboot) s'est porté sur une architecture en quatre couches distinctes.

La couche **Controller** est celle qui s'occupe de l'interaction entre la partie « front » et « back » de l'application. Spring Web fournit un panel d'annotations qui permettent d'associer des méthodes à des urls (@RequestMapping("/chemin"), @Getmapping, @Postmapping…) pour construire les routes de l'application, de répondre aux différentes requêtes HTTP du client en provenance du front (GET, POST…), et aussi d'exposer des « endpoints » (des points d'accès) qui permettront aux clients de récupérer efficacement des fichiers CSV et XML à l'aide des différentes API RESTful.

La couche **Service** contient les méthodes « principales », celles qui permettent d'effectuer les tâches liées aux fonctionnalités de l'application. On l'appelle aussi la couche « métier ». Elle utilise notamment des méthodes de la couche « Repository ».

La couche **Entity** est celle liée aux tuples de la base de données. Une classe « Entity » correspond à une table de la base de données, et chacun de ses attributs correspond à une colonne de la table. Des annotations sont utilisées pour spécifier quel attribut est la clé primaire, ou si la valeur de la clé primaire doit être générée automatiquement, par exemple. C'est également ici que l'on spécifie le mappage avec « @Table(name = nom\_de\_la\_table) ».

Enfin, la couche **Repository** est une interface. Des librairies utiles mises à disposition par Spring Data JPA dépendent de celle-ci, comme CrudRepository (dont l'acronyme signifie « Create Read Update Delete »), JpaRepository, ou encore ListCrudRepository. Ces librairies se distinguent par la variété de méthodes qu'elles proposent et qui peuvent se prêter à différentes situations. Elles sont utilisées dans les méthodes de service, afin de faciliter les opérations sur la base de données.

##### 2.2.3 Tâche/Répartition

Lors des séances de projet tutoré, nous avons procédé à un découpage des tâches pour organiser le travail efficacement. Voici les étapes principales :

1. **Création de la base de données** : définition du schéma, création des tables et écriture d’un script SQL.
2. **Jeu de test** : élaboration d’un jeu de données d’insertion pour valider la structure de la base de données.
3. **Mapping des objets Java** : mise en place des entités Java correspondant aux tables de la base de données.
4. **Routage entre le front et le back** : développement des routes nécessaires pour établir la communication entre l’interface utilisateur (front) et l’API (back).
5. **Gestion JSON** : implémentation de la conversion des objets Java en JSON pour transmettre les données au front, et lecture des fichiers JSON pour les mapper et insérer les données dans la base.
6. **Algorithmes de statistiques** : écriture des algorithmes nécessaires pour calculer les statistiques et développement fonctions associées pour transmettre ces données au front.
7. **Migration Flyway** : remplacement du fichier SQL initial par Flyway, un outil de migration de base de données permettant de gérer les modifications du schéma de manière structurée et répétitive.

Le travail sur le backend a pris du retard en raison de plusieurs facteurs. Tout d'abord, nous avons sous-estimé la complexité de l'apprentissage de Spring Boot, en particulier pour le mapping des objets, qui s'est avéré plus difficile que prévu afin d'obtenir une structure correcte. Ensuite, la mise en place de l'environnement Docker, combinée aux phases de configuration et de débogage, a également consommé une part importante de notre temps. De plus, la progression des tâches a été ralentie par le fait que la moitié des membres affectés au backend étaient en alternance, ce qui a limité leur disponibilité.

Malgré ces obstacles, nous avons tout de même pu accomplir certaines étapes clés. La création de la base de données a été finalisée, accompagnée de l'élaboration d'un jeu de test d’insertion. Nous avons également commencé le travail sur le mapping des objets Java, posant ainsi les bases pour la suite du développement. Ces réalisations nous permettent d'avancer malgré les défis rencontrés.

##### 2.2.4 Justification

Le développement du backend est encore en cours et progresse lentement mais sûrement. Cependant, de nombreuses tâches restent à réaliser et à planifier pour avancer efficacement.

La priorité actuelle est de finaliser le **mapping des objets**, une étape essentielle pour garantir une structure solide et cohérente. Bien que ce travail soit presque terminé, il doit être finalisé avant de passer aux étapes suivantes. Ensuite, la mise en place des **routes** est une tâche cruciale pour permettre l’interaction entre le frontend et le backend. Cela inclut également la gestion des **données JSON**, qui servira à transmettre les informations entre les deux parties, notamment pour l’affichage des données et la récupération des informations des joueurs via l’API Lichess.

De plus, il serait pertinent d'implémenter un **gestionnaire de migration Flyway**, un outil qui permet de gérer les modifications du schéma de la base de données de manière structurée et répétitive. Cela facilitera les futures évolutions de la base.

Enfin, une réévaluation du temps nécessaire pour chaque tâche est indispensable, car le planning initial n’est plus en adéquation avec la réalité actuelle. Cette estimation permettra d'ajuster les priorités et d'établir un plan de travail plus réaliste pour les prochaines étapes. Une redistribution des membres du groupe pourrait également être envisageable, étant donné que la partie ETL touche à sa fin.

### 2.3 ETL

#### 2.3.1 Descriptif fonctionnel

Notre programme ETL a pour rôle de peupler notre base de données à partir des fichiers de données disponibles sur Lichess. Ce dernier traite les parties une à une selon la procédure suivante. Premièrement, les données brutes sont extraites et chargées dans notre programme. Les informations contenues dans ces données sont ensuite lues et stockées dans des structures de données appropriées. Enfin, les données de notre programme sont insérées dans notre base de données. Ce processus se répète pour chaque partie contenue dans le fichier source.

#### 2.3.2 Descriptif technique

##### 2.3.2.1 Extraction des données

L’extraction de données brutes s'effectue à l’aide d'expressions régulières. D’abord, le fichier source est lu ligne par ligne à l’aide d’un BufferedReader. Chaque ligne est stockée temporairement dans une chaîne de caractères jusqu’à ce l’expression régulière représentant une partie complète soit trouvée. Dès lors, les champs tels que les coups et les différentes métadonnées de la partie sont extraits de la chaîne à l’aide d’expressions régulières spécifiques à chacun d’entre eux. Deux modules d’extraction distincts ont été implémentés, un pour les métadonnées et un pour les coups.

Les métadonnées sont strictement rédigées sous la forme suivante: [<nom\_du\_champ> “<valeur>”]. Certains champs ne sont pas toujours présents en fonction de la révision du format PGN. Une classe nommée **PGNLichess** référence les différents champs disponibles, les différents caractères de délimitation ou encore les valeurs récurrentes. Cette dernière décrit également l’ensemble des expressions régulières utilisées dans l’extraction des métadonnées. L’utilisation des expressions régulières s’effectue avec la combinaison des classes **Pattern** et **Matcher**. La première permet de transformer les chaînes de la classe **PGNLichess** en expressions régulières tandis que la seconde trouve les données s’y conformant. La classe **Matcher** comporte une méthode *group(int nbParenthèses)* permettant d’extraire une sous-chaîne dont l’expression régulière correspondante est comprise entre le nombre spécifié de parenthèses.

De la même manière que pour les métadonnées, le module d’extraction des coups implémente une classe (**SAN**) référençant les valeurs et symboles contenus dans ces derniers. La classe **SANParser** transforme les coups (contenus sur une ligne) en une liste (ArrayList) d’objets **Moves**. Les fichiers comportent deux types de coups distincts. Le premier type comporte trois informations: l’ordre, le coup du joueur blanc et le coup du joueur noir. Le second type se base sur le précédent à l’exception qu’il ajoute des informations supplémentaires associées à chaque coup. Ces informations aussi appelées commentaires fournissent des données supplémentaires telles que le temps auquel a été joué le coup (partie chronométrée) ou encore une évaluation.

L’ensemble des données extraites de la chaîne de caractères brutes sont stockées dans une instance de la classe **ChessGame** dont tous les attributs reflètent les données lues. Cette classe contient également deux instances de la classe **Player** et une liste stockant les objets de type **Move**.L’extraction des données est effectuée au sein de la classe **PGNParser**. Cette dernière comporte deux **Queue** permettant de stocker les parties brutes (chaîne de caractères) et les parties traitées (instances de **ChessGame**). Ces structures ne sont pour le moment pas réellement exploitées puisque les parties sont traitées une à une. Elles permettront à l’avenir de paralléliser l’ingestion des données brutes et leur traitement. Le nombre de parties que le programme peut contenir est également limité. En effet, 100 000 parties nécessitent environ 1 Go de RAM d’où la nécessité d’effectuer un traitement sous forme de pipeline.

##### 2.3.2.2 Insertion des données

L’insertion des données était à la fois le **transtypage**, la **vérification** puis **l’insertion** dans notre base de données. En effet, nous nous étions mis d’accord préalablement pour que chaque méthode qui insère dans la base de données ne prenait que des chaînes de caractères en entrée afin d’avoir un code cohérent peu importe l’insertion. On ne cherche pas à savoir si on doit récupérer un int, une date,... Non! Seulement des chaînes de caractères.  
 Ensuite, dans nos méthodes d'insertion, on vas transtyper toutes nos chaînes de caractères en types voulus correspondant au schéma de la BD.

Durant cette phase, on va de même chercher à minimiser les erreurs possibles comme par exemple des “?” à la place d’entier pour l’elo de la partie.

Pour se faire, on va utiliser les opérateurs ternaires afin de rendre le code le plus compact possible. On regardait si la chaîne de caractères que l’on récupérait était vide, null, ou ne correspondait pas à ce que l’on cherchait, et si c’était le cas, on insère soit un 0 quand c’est un int, soit null, sinon on transtype la chaînes de caractères.

Une fois la vérification et le transtypage fait, on va alors pouvoir insérer dans la base de données tous nos attributs fraîchement correspondants.

Mais lors de l’insertion des données pendant les tests, on remarque plusieurs problèmes que nous avons omis par manque de rigueur. Plusieurs même joueurs peuvent être insérés, et de même pour les parties. Cela nous a poussé à faire de la vérification supplémentaire avant insertion mais aussi à créer un attribut pour les parties, **hash\_code**.   
A la différence des méthodes “bienveillantes” et “traditionnelles” pour créer un hashcode, nous avons tout simplement concaténé différents attributs qui, une fois la somme faite, rendait unique.  
Nous avons donc concaténé *joueur\_blanc+joueur\_noir+date* (Y-M-D hh:mm:ss).  
Une fois tout cela mis en place, les tests fonctionnent parfaitement et la base de données se remplit entièrement.

#### 

#### 2.3.3 Répartition/Temps

Nous avons séparé les tâches en 2 grands groupes, le parser avec l'insertion dans les classes puis l'insertion dans la BD en effectuant des tests au préalable.

Nous avons bien discuté entre nous afin de pouvoir mettre en commun nos codes le plus facilement possible.

Des deux côtés, les groupes avançaient bien malgré quelques problèmes qui nous ralentissaient (problème de parties parsées, de null dans les méthodes d'insertions,...).

Une fois que les deux codes fonctionnaient séparément, il ne restait plus qu'à joindre les deux afin de terminer l'ETL.

Du fait de nos discussions préalables, cela n'a posé aucun problème.

#### 2.3.4 Justification

Notre programme ETL effectue sa fonction primaire, à savoir peupler notre base de données avec les données Lichess. Cependant ce dernier peut encore être amélioré notamment sur la parallélisation du processus. A titre d’exemple, l’insertion de 26M de parties nécessite plus de 3 heures. L’approche envisagée est de garder un programme quasiment similaire mais de diviser le travail initial en divisant le fichier source. Dès lors, le temps requis au traitement des données serait nettement diminué. Néanmoins, ces améliorations ne sont pas la priorité de notre groupe de part son utilité et sa fréquence d’utilisation.

### 2.4 Etat de la Base de Données

#### 2.4.1 Descriptif fonctionnel

La base de données joue un rôle central dans l’application, en assurant le stockage de toutes les données essentielles liées aux utilisateurs, aux parties et aux tournois. Elle conserve les informations clés des joueurs, telles que leurs statistiques personnelles, leur elo dans différents modes de jeu, ainsi que leur ratio de victoires en tant que joueur blanc ou noir. Chaque partie enregistrée est accompagnée des détails nécessaires pour sa reconstitution, incluant les coups joués et leurs évaluations. L’objectif est de permettre à l’application de fournir des analyses précises et de proposer des recommandations stratégiques aux joueurs. Par ailleurs, la base de données intègre également les tournois et les ouvertures d'échecs.

#### 2.4.2 Descriptif technique

##### 2.4.2.1 Player

La table **Player** est dédiée à la gestion des informations personnelles des joueurs, avec une attention portée sur leurs performances dans différents contextes. Chaque joueur est identifié de manière unique par un nom, garantissant l’absence de doublons lors des insertions. Cette table stocke l’elo des joueurs dans plusieurs modes de jeu, ce qui permet de suivre leur progression et d’offrir des statistiques détaillées, comme le ratio de victoires en tant que joueur blanc et noir. Ces données permettront à l’application de générer des tableaux de classement et de fournir un suivi personnalisé des performances.

##### 2.4.2.2 Game

La table **Game** contient les détails spécifiques de chaque partie jouée, y compris les noms des joueurs, l’elo en début de partie, l’issue de la partie (victoire, défaite, ou match nul) et l'identité du gagnant. Cette structure facilite l’analyse postérieure des parties pour en extraire des données utiles, telles que les coups décisifs. Elle est également indispensable pour peupler l’application avec des exemples concrets de parties jouées, permettant ainsi d’illustrer des stratégies ou de recommander les meilleurs coups aux utilisateurs.

##### 2.4.2.3 Moves

La table **Moves** contient la liste des coups d’une partie associée par son id.

Elle possède tous les coups associés à son évaluation et son temps dans la partie.

Cela permettra de déceler les meilleurs coups dans plusieurs situations ressemblantes.

##### 2.4.2.4 Tournament

Pour le moment, dans les fichiers *.pgn* de LiChess que nous récupérons, il n’y a pas de *Tournoi*, mais cela n’empêche pas de notre côté de rajouter cette fonctionnalité.

Cela permettra de pouvoir avoir plusieurs parties groupées par un **id** afin de pouvoir fonctionner.

##### 2.4.2.5 Chess\_Opening

Cette table contient environ 2200 ouvertures permettant ainsi de pouvoir rechercher les ouvertures correspondantes aux coups joués d'une partie.

#### 2.4.4 Répartition/Temps

La mise en place de ces tables a nécessité un travail de recherche approfondi et de nombreuses heures d’optimisation. Lors de la création des méthodes d’insertion et des requêtes SQL, une attention particulière a été accordée à l’efficacité des jointures et à la réduction des temps d’exécution. Cette optimisation est essentielle pour garantir que l’application fonctionne de manière fluide, même avec un volume important de données. Le travail effectué sur la structure des tables a permis de minimiser les temps de réponse et de maximiser la performance des requêtes complexes.

#### 2.4.5 Justification

La base de données est un élément vital pour le projet afin de nécessiter des informations nécessaires des utilisateurs et des parties en général.

Celle-ci évolue en fonction de nos besoins surtout pour les statistiques des joueurs.

Elle évoluera encore sûrement si nous voulons intégrer d'autres statistiques.

### 2.5 API

#### 2.5.1 Description fonctionnel

Le rôle de l’API est de remplir la table *Player* de notre base de données à l’aide d’un appel de l’API de LiChess. En effet, cet appel permet de récupérer des informations sur les joueurs de notre base de données, tels que le nombre de parties par mode de jeu, l’elo de chacun d’entre eux,...  
Chacunes de ces informations sont récupérées dans un *json* par lequel nous devons traiter la réponse reçue afin de récupérer les informations utiles pour nous.

Ultérieurement, nous chercherons à automatiser cet appel à l’aide de la bibliothèque java *ScheduledExecutorService* que nous appellerons tous les mois.

#### 2.5.2 Description technique

##### 2.5.2.1 Appel de l’API

Nous devons comprendre quelles sont les limites de l’API avant de pouvoir récupérer toutes les données de chaque joueur. En effet, l’API semble plutôt stricte car elle ne permet que de faire des appels par groupe de 300 au maximum et sans pouvoir l’appeler toutes les secondes.

Pour se faire, afin de récupérer les statistiques des joueurs, nous récupérons à l’aide d’une requête SQL, tous nos joueurs présents sur notre BD ainsi que leurs nombres.

Cela va nous permettre de faire des appels successifs de l’API par paquet de 300 joueurs à chaque fois. Le temps que la fonction se termine, plusieurs secondes se seront écoulées et le prochain paquet de 300 joueurs ne se fera pas bloquer par LiChess.

##### 2.5.2.2 Extraction de données

Une fois la réponse de l'API, il fallait pouvoir parser la réponse obtenue en json (le tout concaténé en un bloc).

Pour cela, nous avons fait un énorme regex qui, malgré la difficulté de le comprendre, permet de parser toute la réponse. Mais cela ne va récupérer que pour 1 joueur. Alors, nous avons créé une Map<String, Map<String, String>> qui contient le nom du joueur avec les statistiques (elo classique, elo rapide,...) puis la statistique associée. Une fois fait, nous pouvons parcourir notre json et récupérer chacune des statistiques et l'associer à chaque joueur.

Il n'y a qu'un seul problème que le regex a du mal à parser. En effet, il n'est pas impossible que certains joueurs désactivent leurs comptes et que la réponse de l'API dans le JSON est juste : {“id”:id\_joueur,”username”: nom\_joueur, “disabled”:true}

Notre regex le détecte mais continue de lire et parse jusqu'au prochain joueur.

Ce qui fait, que dans notre terminal, le joueur qui a désactivé son compte aura les statistiques du prochain joueur. Et ce joueur-là aura son compte sans statistiques.

La seule solution, hormis de rechercher un nouveau regex, et de supprimer dans notre fichier de partie, toutes les parties où le joueur qui n'a plus de compte apparaît.

Cela permet de gagner du temps “en trichant” mais aussi de ne pas falsifier les données qui seront insérées plus tard dans notre base de données.

Mis à part ce problème, il y a des statistiques qui sont impossibles à récupérer par groupe de joueurs. En effet, il n'est pas possible de récupérer le nombre de parties gagnées en tant que joueur blanc ou noir lors de l’appel de l’API pour plusieurs joueurs en même temps. Pour cela, nous avons créé une méthode qui récupère toutes les parties joués par un utilisateur et nous avons regardé s'il gagne en tant que joueur blanc ou noir. Une fois le parcours des parties, nous avons stockés dans un tableau d'entier de taille 6 les données suivantes: 0: Nombre de partie; 1: Nombre de parties gagnées en tant que blanc; 2: Nombre de parties gagnées en tant que noir; 3: Ratio Victoire; 4: Ratio Victoire Blanc; 5: Ratio Victoire Noir.

Cela permet de ne pas passer seulement par l’API mais aussi d’exploiter notre base de données afin de ressortir des statistiques différentes de l’API.

##### 2.5.2.3 Insertion de données

Afin d'insérer les données, il ne suffit plus que de lire la Map et d'appeler la méthode de la classe Joueur permettant d'insérer les statistiques de la partie.

On vérifie bien entendu que chacune des chaînes de caractères que l'on récupère est cohérente en effectuant des vérifications avant l'insertion.

Cela permet de garnir notre table Joueur de différentes statistiques permettant plus tard de faire des graphes sur celles-ci.

#### 2.5.3 Répartition/Temps

La partie concernant l’API s’est déroulée une fois l’ETL fini. C'est-à-dire qu’on a décidé de faire les scripts nécessaires qui récupèrent les données de chaque utilisateur.  
Au final, comme ce qui était prévu, cette partie était plutôt rapide malgré les quelques problèmes vus précédemment.

Une seule personne était affectée à la tâche.

#### 2.5.4 Justification

L'API nous permet de récupérer toutes les statistiques des joueurs présents sur notre base de données même si toutes ses parties ne sont pas enregistrées sur la nôtre.

Cependant l’API a des limites qui ne nous permettent pas de pouvoir faire tout le travail.

C'est pour cela que nous utilisons les résultats de l'API afin de fournir d'autres statistiques possibles.

Pour le moment cela reste rapide pour 300 joueurs mais quid de plusieurs milliers de joueurs. Il sera sans doute nécessaire de paralléliser cela, en lançant un thread pour chaque groupe de 300 joueurs afin de maximiser la vitesse d'exécution.

Le problème étant que l'API ne supporte pas beaucoup de requêtes à la seconde, il faudrait tester la limite de l'API afin de pouvoir faire le maximum possible en termes de threads.

## Retour d’expérience sur l’environnement de développement et les choix techniques

### 4.1 Git

L'implémentation de Git au sein du groupe s'est faite de manière assez naturelle. Un avantageux transfert de connaissance a pu se réaliser, où ceux qui maîtrisaient l'outil ont initié ceux qui ne le maîtrisaient pas aux bonnes pratiques et manières de faire, afin que chacun puisse mettre en commun ses travaux efficacement. De plus, certaines branches ont déjà dû subir un « merge », et peu, voire aucun conflit majeur n'a eu lieu lors des fusions.

Cependant, certains aspects peuvent encore être améliorés, comme une utilisation plus poussée des fonctionnalités proposées par Git. Le système d'*issue*, par exemple, permet d'obtenir un suivi de l'avancement d'une tâche en particulier en y associant une branche spécifique avec la possibilité d'écrire des notes d'activité, des commentaires ou de téléverser des fichiers, qui seront vus par les développeurs affectés à l'*issue* en question. Il en va de même pour le système des *Merge Request*, qui permet non seulement d'affecter des « rewiewer » qui auront pour tâche de valider le code du développeur, mais qui en plus permet de suivre en direct l'évolution (les commits) de la branche de feature par rapport à la branche originelle en affichant les conflits potentiels qui pourraient survenir en cas de fusion. Bien que cela semble un peu accessoire, ce genre de fonctionnalités pourraient, une fois prises en main, être une réelle valeur ajoutée.

### 

### 4.2 Docker

L’utilisation de Docker s’est avérée judicieuse pour de nombreuses raisons. Cette technologie nous a permis de distribuer des environnements d’exécution aux membres de l’équipe sans que ces derniers n’aient besoin d’installer d’outils supplémentaires sur leurs machines. De plus, ces environnements sont utilisables à l’aide de deux commandes: *docker compose up* et *docker compose down* rendant leur utilisation très simple. L’isolation des différents services dans des containers à éliminer les éventuels conflits provoqués par une implémentation sur des machines dont les environnements diffèrent.

La mise en place fut également simple. Docker est une technologie facile d’accès et documentée. Cet outil nous offre une modularité sur les différents services composant notre application. De ce fait, il nous sera aisé de regrouper ces derniers (répartis sur ETL, back, front) pour composer notre application complète. La spécification des versions de chaque service nous permet de contrôler ces dernières limitant alors les problèmes de mises à jour indésirées.

### 

### 4.3 Technologies Front

Ce qui a fonctionné :

Le composant React ChessGameWithBotApp permet de jouer contre le bot Stockfish. L'initialisation de Stockfish, la mise à jour de l'échiquier après chaque coup, la gestion du chronomètre pour les deux joueurs, ainsi que l'affichage des historiques de coups et de l'ouverture, fonctionnent globalement bien.

Ce qui n’a pas fonctionné :

La mise à jour de l’ouverture reste incomplète et ne reflète pas l'ouverture jouée par les blancs et les noirs de manière précise.

Quelques fonctionnalités (comme l'ajout dynamique de l'ECO dans le fichier PGN) ne sont pas encore implémentées.

Solutions à mettre en place pour améliorer :

Gestion des ouvertures : Intégrer une bibliothèque ou un fichier de données permettant de détecter et d'afficher les ouvertures en fonction des coups joués (par exemple, une base de données PGN pour correspondances), soit notre BD ChessOpening.

Ajout des métadonnées ECO : Automatiser l'identification du code ECO et son intégration dans les métadonnées PGN.

### 

### 4.3 Technologies ETL

Ce qui à fonctionné :

Nous avons opté pour des technologies que nous connaissons bien ce qui a rendu le développement de l’ETL très compréhensible et intuitif.

Ce qui n’as pas fonctionné :

Cependant, les expressions régulières sont une potentielle source d’erreur car les fichiers de la base de données Lichess sont étalés sur 10 ans, ce qui rend leur contenu différent d’une année à l’autre. Et d’un autre côté, le temps d’exécution est assez long, ce qui peut être pénible pour certains tests.

Solution à mettre en place pour améliorer :

Pour résoudre le problème du temps d’exécution serait d’optimiser le développement de la pipeline séquentielle puis de la multi-threader.

### 4.3 Technologies Back

L’utilisation de Spring Boot s’est avérée une excellente idée car il simplifie considérablement le développement d’applications en offrant des outils intégrés pour configurer rapidement un serveur web, gérer les dépendances avec Maven, et établir des APIs REST. Sa compatibilité avec Spring Data JPA et Hibernate a facilité le mapping des entités Java avec les tables de la base de données, tout en automatisant les opérations courantes. De plus, Spring Boot suit une architecture structurée (Controller, Service, Repository, Entity), ce qui rend le code clair, maintenable et extensible. Enfin, sa vaste documentation et communauté ont été d'une grande aide.

Mis à part la gestion du temps, notamment sous-estimé pour l’apprentissage des technologies comme Spring Boot et l’ API REST, tout s’est déroulé comme prévu.

Pour améliorer la gestion du projet, il serait important de réaliser une planification plus réaliste en tenant compte des temps nécessaires pour l’apprentissage des technologies, comme Spring Boot et les API REST. Il serait également utile de renforcer la formation initiale sur ces outils afin d’en accélérer la prise en main. Une meilleure répartition des tâches et une communication efficace entre les membres de l’équipe pourraient éviter les blocages. Enfin, utiliser l'outil de gestion de projet Openproject de façon plus efficace permettrait de suivre l’avancement en temps réel et d’ajuster les priorités si nécessaire.

## Retour d’expérience sur l’organisation du groupe

### 5.1 Planification et distribution des tâches

#### **Ce qui a fonctionné**

**Établissement d’un planning global** : Nous avons défini dès le départ un planning global des grands jalons du projet, ce qui a permis d’avoir une vision d’ensemble des étapes principales à réaliser. Bien que cette planification n’ait pas été détaillée, elle a offert une structure minimale pour guider les efforts collectifs.

**Avancement global malgré une planification sommaire** : Les tâches globales identifiées ont été prises en charge par les membres du groupe en fonction de ce qui avait été défini dans notre premier rapport. Cette approche, bien que non optimale, a permis d’avancer sur le projet.

**Utilisation d’OpenProject** : Nous avons utilisé OpenProject, un outil complet pour la planification et le suivi des tâches. Cependant, notre utilisation a été limitée, et nous n’avons pas exploité toutes ses fonctionnalités.

**Priorisation des tâches critiques** : Nous avons su identifier et prioriser les tâches dont d’autres dépendent, notamment la conception et le développement de l’ETL. Cela a permis de structurer les efforts autour des livrables les plus importants.

**Adaptabilité** : Malgré le manque de structuration, nous avons su nous adapter et orienter les développements lorsque cela était nécessaire, montrant une capacité de réaction efficace face aux imprévus.

#### **Ce qui n’a pas fonctionné**

**Absence de planification détaillée** : Si un planning global a été établi, aucun détail précis n’a été ajouté concernant les tâches individuelles, leurs délais ou les ressources nécessaires. Cela a conduit à une répartition improvisée des tâches, avec des membres prenant des responsabilités "à la volée" selon leurs disponibilités.

**Manque de visibilité globale** : L’absence de suivi régulier et structuré a rendu difficile la compréhension de l’état d’avancement global du projet. Les membres avaient parfois une perception partielle ou incomplète de la progression collective.

**Utilisation limitée et inefficace d’OpenProject** : Bien que cet outil ait été choisi pour la gestion des tâches, il s’est révélé lent et quelque peu complexe à utiliser. En conséquence, il a été sous-exploité, ce qui a réduit son utilité pour le suivi et la coordination.

**Manque de synchronisation des efforts** : Bien que nous ayons avancé collectivement, l'absence de structure claire a occasionné une progression moins fluide, avec des moments où certains membres étaient en attente d'autres.

#### **Solutions à mettre en place pour améliorer la planification des tâches**

1. **Adopter une méthode simplifiée et adaptée**

Utiliser le **tableau Kanban** seul d’OpenProject, plus intuitif; pour suivre les tâches à faire, en cours, et terminées.

Décomposer les grandes tâches en sous-tâches détaillées, avec des descriptions claires et un suivi précis.

1. **Introduire des délais réalistes et des responsabilités claires**

Associer chaque tâche à une **deadline réaliste** et désigner un **responsable identifié**, afin d’améliorer la gestion des priorités et d’assurer un meilleur suivi.

1. **Favoriser une meilleure visibilité**

Mettre en place un tableau de bord collaboratif qui montre l’état global d’avancement du projet et les tâches en cours, afin que chaque membre puisse facilement suivre la progression collective.

En appliquant ces solutions, la planification des tâches pourra devenir plus claire et efficace, permettant une gestion plus fluide et mieux coordonnée pour les projets à venir.

### 5.2 Communication

#### **Ce qui a fonctionné**

La communication au sein du groupe s’est appuyée principalement sur deux canaux : **Discord** pour les échanges en ligne et les **réunions physiques** pour des interactions directes. Malgré une utilisation irrégulière, Discord s’est révélé plus accessible et flexible que Mattermost, car il ne nécessite pas de connexion VPN, ce qui a simplifié et fluidifié la tenue des échanges.

Les réunions physiques ont constitué le socle de notre communication, permettant à chaque membre de participer activement. Ces moments d’échanges directs ont permis une coordination efficace et une implication collective.   
Par ailleurs, l’utilisation de Discord pour la communication en temps réel a aidé à résoudre certains problèmes concrets liés au développement.

#### **Ce qui n’a pas fonctionné**

Cependant, plusieurs limitations ont impacté la fluidité de nos échanges :

**Manque de communication hors réunions physiques** : Durant les périodes de vacances ou lorsque nous travaillions à distance, les interactions étaient limitées. Cela a entraîné des mises à jour sporadiques sur l’avancement du projet, rendant difficile un suivi continu.

**Organisation sur Discord** : Bien que pratique, l’utilisation d’un seul canal pour tous les échanges a rapidement conduit à une surcharge d’informations, rendant les discussions moins claires.

**Absence de règles claires de communication** : Le groupe n’a pas établi de directives spécifiques, comme la fréquence des mises à jour ou les délais de réponse attendus, ce qui a parfois nui à l’efficacité globale du projet.

#### **Axe d’amélioration**

**Structurer les canaux de communication sur Discord**

Créer des **canaux thématiques** pour organiser les échanges selon les différents aspects du projet, tels que :

Développement (questions techniques, bugs, solutions proposées)

Organisation (planning, échéances)

Documentation (partage de fichiers et notes)

Discussions générales.  
Cela permettra d'éviter les discussions brouillonnes sur un seul canal et de retrouver plus facilement des informations spécifiques.

**Encourager la communication asynchrone**

Introduire des pratiques favorisant les échanges en différé, comme l’utilisation de courriels ou de messages résumant les avancées et décisions importantes.

Prévoir un moment fixe chaque semaine pour que chaque membre partage une **mise à jour individuelle** sur l’état d’avancement de ses tâches (via Discord ou un autre outil collaboratif). Cela garantira une visibilité continue, même lorsque les membres travaillent à distance ou pendant les périodes moins actives.

**Instaurer des règles de communication claires**

Préciser les outils et situations pour les interactions (par exemple, Discord pour les discussions courantes, réunions physiques pour les prises de décision importantes).

Encourager une culture de **signalement** lorsqu’un membre rencontre une difficulté ou un blocage, afin que le groupe puisse intervenir rapidement.

## Conclusion

L'implémentation du projet a permis de tirer de nombreuses leçons, tant sur le plan technique que sur le plan organisationnel. Si certaines technologies, telles que Git, Docker, Spring Boot, et React, ont bien fonctionné et ont permis d'atteindre les objectifs principaux, plusieurs axes d'amélioration ont été identifiés, notamment en termes de gestion de projet, de communication au sein de l'équipe et d'optimisation des processus techniques.

**Bilan technique**

Concernant les technologies, bien que les outils comme Git aient été utilisés de manière satisfaisante dans une première approche, une meilleure exploitation de ses fonctionnalités aurait permis de rendre le suivi de l'avancement plus structuré et précis. Par exemple, l'utilisation plus poussée des systèmes d'issues et de Merge Requests aurait facilité la gestion des tâches et amélioré la visibilité du projet au quotidien. La gestion des dépendances et la mise en place de processus de validation du code auraient aussi renforcé la qualité du travail en équipe.

Dans le domaine de Docker, le projet a fonctionné, mais des améliorations sont nécessaires pour garantir une meilleure expérience utilisateur, notamment en ce qui concerne l’interface front-end. Quant aux technologies front-end, bien que le composant React *ChessGameWithBotApp* ait permis de répondre aux objectifs de base, des améliorations sont nécessaires pour gérer les ouvertures et l'intégration des métadonnées ECO dans les fichiers PGN. Il reste encore des tâches à accomplir pour automatiser et rendre cette partie plus fluide.

Les technologies de traitement des données ETL ont, pour leur part, permis un développement intuitif, mais le temps d'exécution reste un problème qui nécessite une optimisation, notamment par l'usage de solutions multi-threading. Quant au back-end, Spring Boot a été un choix judicieux, permettant un développement rapide grâce à son architecture claire et à ses outils bien intégrés. Cependant, une meilleure planification pour l’apprentissage des technologies, notamment Spring Boot et les APIs REST, aurait permis de gagner en efficacité et en temps.

**Bilan organisationnel**

Sur le plan organisationnel, une planification plus détaillée et réaliste aurait permis une gestion du projet plus fluide. Le fait de ne pas avoir décomposé les tâches de manière suffisamment précise a conduit à une certaine confusion et à une répartition imprévisible des responsabilités. Il est essentiel à l’avenir de mettre en place une gestion de projet plus rigoureuse, en définissant clairement les sous-tâches et en attribuant des délais réalistes. Utiliser pleinement les fonctionnalités de l'outil OpenProject aurait également permis une meilleure coordination et un suivi plus précis de l’avancement du projet.

En termes de communication, bien que Discord et les réunions physiques aient facilité les échanges, l'absence de règles claires sur la fréquence et la nature des communications a parfois freiné l'efficacité du groupe. Il aurait été bénéfique de structurer davantage les canaux de communication sur Discord en créant des espaces thématiques pour chaque aspect du projet. De plus, une meilleure organisation des échanges asynchrones et l'instauration de moments réguliers de mise à jour sur l’état d’avancement auraient permis de maintenir une visibilité continue sur les tâches, même pendant les périodes où les membres travaillaient à distance.

**Recommandations pour l'avenir**

Pour les projets futurs, il est essentiel d’adopter une approche plus structurée, tant au niveau de la gestion des tâches que de la communication. L’utilisation optimale d’outils comme Git, OpenProject et Discord nécessite de clarifier leur usage et de former les membres de l’équipe à leurs fonctionnalités avancées. En parallèle, une gestion plus rigoureuse du temps, avec des délais réalistes et un suivi régulier de l’avancement, permettrait de mieux gérer les imprévus et d'éviter les blocages.

Le travail technique doit également inclure une attention particulière aux optimisations nécessaires, notamment pour les aspects liés aux performances du système et à la gestion des données. L’automatisation de certaines tâches, comme l’intégration des métadonnées dans les fichiers PGN, ainsi que la gestion des ouvertures dans les jeux d'échecs, est essentielle pour améliorer la robustesse et l’efficacité de l'application.

En somme, ce projet a offert une expérience riche en apprentissages. En tirant parti des leçons extraites, notamment sur la gestion de projet et la communication, il sera possible d'améliorer considérablement la mise en œuvre de futurs projets collaboratifs et d'assurer une meilleure efficacité collective.