IMT Atlantique
Technopôle de Brest-Iroise
29238 Brest Cedex 3
www.imt-atlantique.fr

IMT Atlantique – UE CALC

Plateforme d'échecs distribuée avec RabbitMQ

Marion Boulin

Date d'édition : Décembre 2025
Version : 1.0

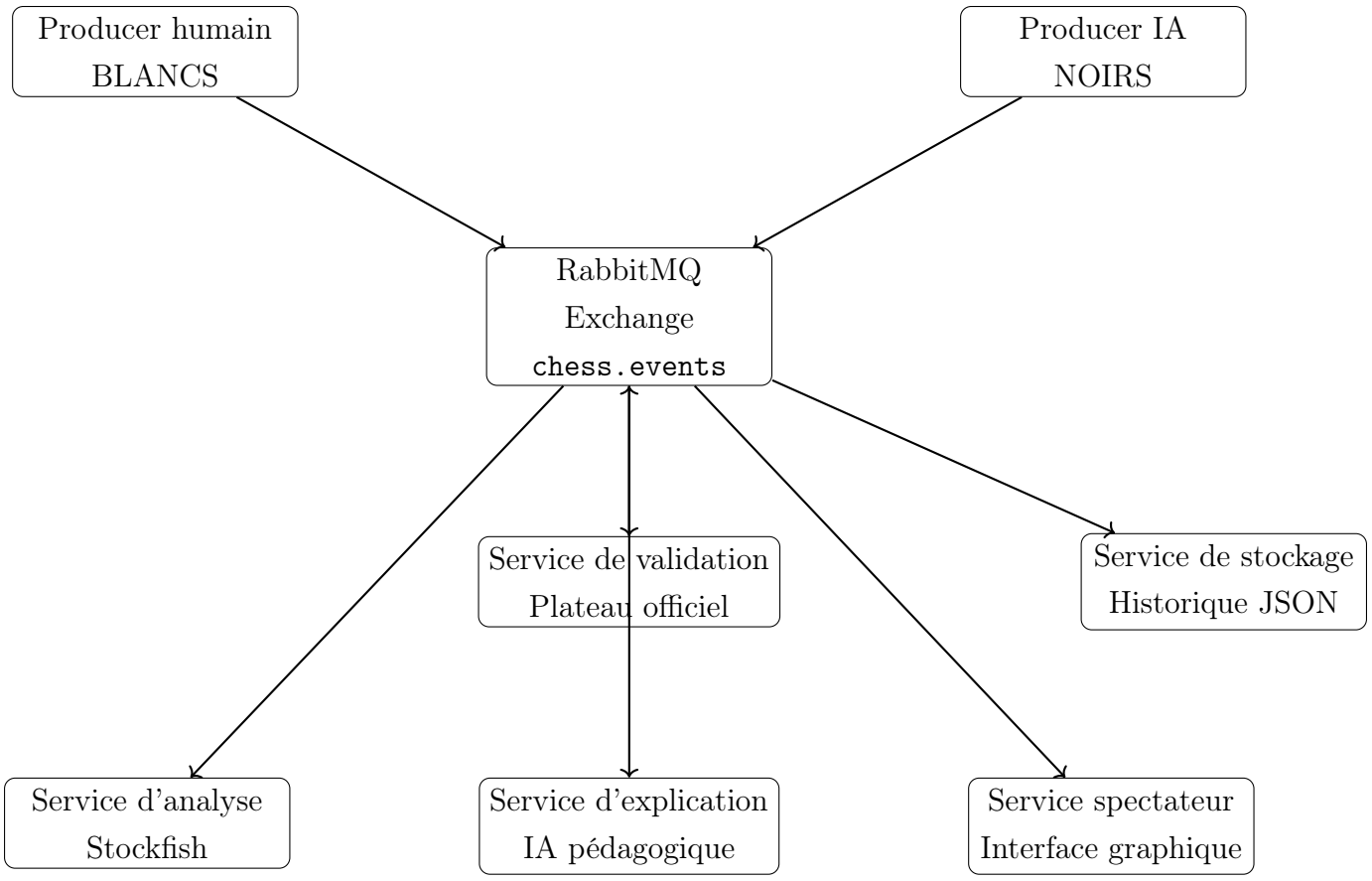
Sommaire

1. Introduction

1.1. Présentation du projet

Dans le cadre de l'UE CALC, ce projet consiste à concevoir une application distribuée reposant sur un middleware de messages. L'application met en œuvre une partie d'échecs simulée à l'aide de plusieurs services indépendants communiquant via RabbitMQ.
Chaque service remplit un rôle précis, allant de la proposition et de la validation des coups à l'analyse, l'affichage et le stockage des événements. Cette architecture permet d'illustrer les principes de communication asynchrone, de découplage des composants et de séparation des responsabilités.

1.2. Schéma d'architecture du système



1.3. Lancement du projet

Le lancement de l'application s'effectue en plusieurs étapes. Chaque service correspond à un script Python indépendant et doit être exécuté dans un terminal distinct.

1. Lancer le serveur de messages RabbitMQ.
2. Exécuter le service de validation `validation_service.py`, qui maintient l'état officiel de la partie.

3. Lancer les services passifs : `analysis_service.py`, `explanation_service.py`, `spectator_service.py` et `storage_service.py`.
4. Exécuter les services producteurs `producer_human.py` et `producer_ai.py` afin de démarrer la partie.

1.4. Comment jouer

Le joueur humain interagit avec l'application via le terminal. Les coups sont saisis manuellement au format UCI, qui indique uniquement la case de départ et la case d'arrivée de la pièce.

Par exemple, le coup `e2e4` correspond au déplacement de la pièce située en `e2` vers la case `e4`. Il n'est pas nécessaire de préciser le type de pièce, car le plateau d'échecs maintenu par le système connaît déjà la position de toutes les pièces et applique automatiquement les règles du jeu.

Après la saisie d'un coup, celui-ci est transmis au service de validation. Si le coup est légal, il est accepté et diffusé à l'ensemble des services. Dans le cas contraire, le coup est rejeté et le joueur est invité à en saisir un nouveau.

La partie se poursuit alternativement entre le joueur humain et l'intelligence artificielle jusqu'à la détection d'une fin de partie.

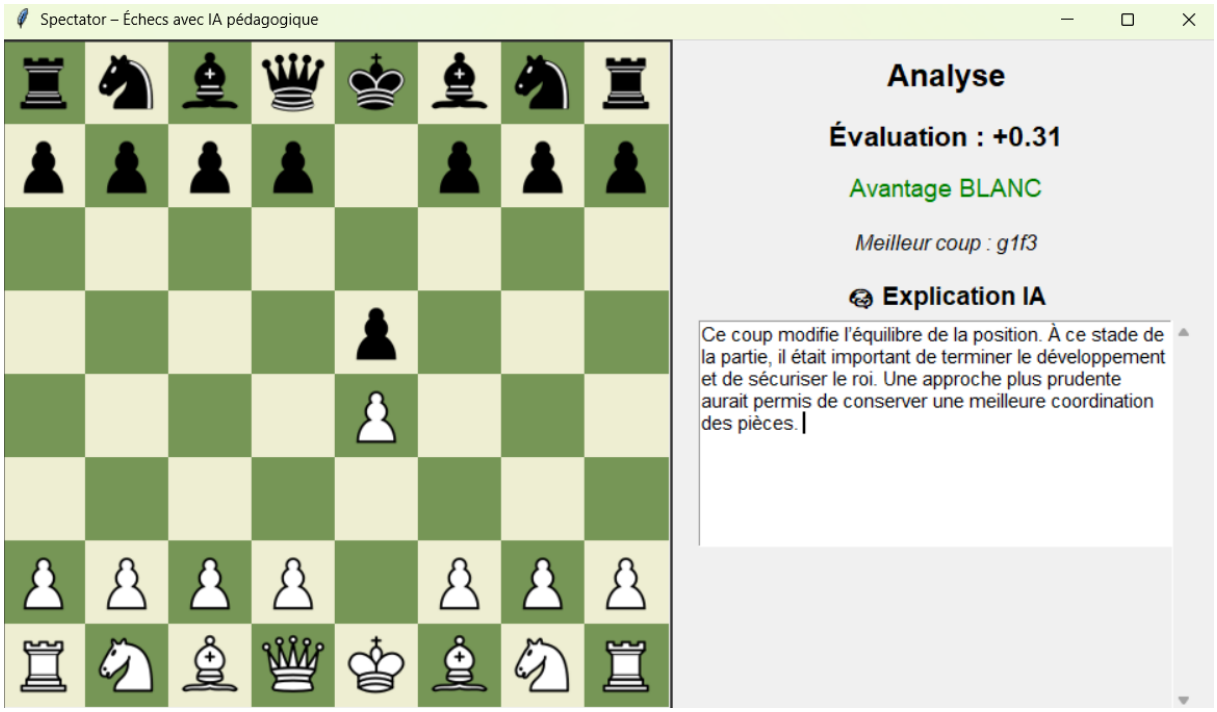


FIGURE 1 – Interface obtenue

Lettre	Pièce (français)	Nom (anglais)
P	Pion	Pawn
R	Tour	Rook
N	Cavalier	Knight
B	Fou	Bishop
Q	Dame	Queen
K	Roi	King

1.5. Comment j'ai construit ce projet

La construction de ce projet s'est faite de manière progressive et itérative. L'objectif initial n'était pas de réaliser immédiatement un jeu d'échecs complet, mais de comprendre et maîtriser les mécanismes fondamentaux d'une architecture événementielle basée sur RabbitMQ.

Dans une première étape, les services échangeaient uniquement des messages JSON très simples. Afin de réduire la complexité fonctionnelle, une ouverture d'échecs fixe (la défense sicilienne) était utilisée, ce qui permettait de se concentrer exclusivement sur la publication et la consommation de messages. Cette phase m'a permis de comprendre le fonctionnement des exchanges, le modèle de diffusion *fanout*, ainsi que la synchronisation de plusieurs services indépendants écoutant les mêmes événements.

Une fois cette base technique maîtrisée, j'ai introduit progressivement la logique métier du jeu d'échecs. J'ai alors séparé clairement les responsabilités des différents services. Un service est chargé de proposer les coups, un service central joue le rôle d'arbitre et valide les coups, tandis que les autres services restent passifs et se contentent d'observer l'évolution de la partie. Cette séparation stricte des rôles a permis de rendre l'architecture plus lisible, plus robuste et plus facile à faire évoluer.

Dans un second temps, j'ai intégré le moteur Stockfish afin d'automatiser la génération et l'analyse des coups. Cette intégration a mis en évidence la nécessité de mécanismes de synchronisation, notamment l'attente explicite de la validation d'un coup avant d'en proposer un nouveau. Sans cette synchronisation, des incohérences pouvaient apparaître entre les différents services. La mise en place de ces mécanismes a renforcé la cohérence globale du système.

Une étape suivante a consisté à rendre le projet interactif. Initialement, l'affichage de la partie reposait sur des visualisations simples réalisées avec `matplotlib`, ce qui était suffisant pour un débogage, mais peu adapté à une interaction en temps réel. J'ai donc progressivement remplacé cette approche par une interface graphique basée sur `Tkinter`, permettant une visualisation claire du plateau, des coups joués et des informations d'analyse.

Enfin, j'ai souhaité enrichir le projet avec une dimension pédagogique, en ajoutant un service chargé de fournir des explications textuelles sur les coups. L'utilisation d'un modèle d'IA générative étant soumise à des contraintes de coût, j'ai prévu un mécanisme

de repli basé sur des textes génériques. Ce choix garantit que le système reste fonctionnel et cohérent, même en l'absence d'accès à une API payante.

2. Producer humain (intéractif) : producer_human.py

Cette section décrit le fonctionnement du service `producer_human.py` en alternant extraits de code et explications détaillées.

2.1. Importation des bibliothèques

```
import pika
import json
import chess
from datetime import datetime, timezone
```

Ces importations permettent respectivement : la communication avec RabbitMQ, l'encodage et le décodage des messages JSON, la gestion du plateau d'échecs, et l'ajout d'un horodatage aux événements.

2.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
HUMAN_COLOR = chess.WHITE
```

Ces constantes définissent la configuration partagée du service. Le joueur humain est explicitement associé aux pièces blanches.

2.3. Plateau local et canal

```
board = chess.Board()
channel = None
```

Le plateau local sert uniquement à suivre l'état de la partie validée. Le canal RabbitMQ est initialisé ultérieurement dans la fonction principale.

2.4. Publication des événements

```
def publish(event_type, payload):
    message = {
        "event_type": event_type,
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "payload": payload
    }

    channel.basic_publish(
```

```
        exchange=EXCHANGE_NAME,
        routing_key="",
        body=json.dumps(message)
    )
```

Cette fonction encapsule l'envoi d'événements. Chaque message contient un type, un horodatage et des données associées, puis est diffusé sur l'exchange commun.

2.5. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Le service écoute les messages RabbitMQ et les décode depuis le format JSON. Le traitement dépend du type d'événement reçu.

2.6. Démarrage d'une partie

```
if event_type == "game_started":
    board.reset()
```

Lorsqu'une nouvelle partie commence, le plateau local est réinitialisé afin de rester cohérent avec l'état officiel.

2.7. Application d'un coup validé

```
elif event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
```

Un coup validé par le service central est appliqué localement. Le producteur humain ne remet jamais en question cette décision.

2.8. Fin de partie

```
elif event_type == "game_ended":
    print("Partie termin e")
    exit(0)
```

Lorsque la partie est terminée, le service s'arrête proprement.

2.9. Fonction principale

```
def main():
    global channel
```

La fonction principale initialise le service et contient la boucle de jeu du joueur humain.

2.10. Connexion à RabbitMQ

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host=RABBITMQ_HOST)
)
channel = connection.channel()
```

Le service se connecte au serveur RabbitMQ et crée un canal de communication dédié.

2.11. Déclaration de l'exchange et abonnement

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

L'exchange de type fanout permet de diffuser chaque événement à tous les services abonnés.

```
queue = channel.queue_declare(queue="", exclusive=True).method.
   queue
channel.queue_bind(exchange=EXCHANGE_NAME, queue=queue)
```

Une file temporaire exclusive est créée pour ce service.

2.12. Démarrage de la partie

```
publish("game_started", {})
```

Le producteur humain déclenche le début de la partie en envoyant l'événement correspondant.

2.13. Boucle principale

```
while True:
    connection.process_data_events(time_limit=0.1)
```

La boucle principale traite les événements RabbitMQ sans bloquer l'exécution du programme.

2.14. Saisie et validation locale du coup

```
if board.turn == HUMAN_COLOR:
    move_uci = input("Ton coup : ").strip()
```

Le joueur ne peut jouer que lorsque c'est son tour.

```
try:
    move = chess.Move.from_uci(move_uci)
except ValueError:
    print("Format invalide")
    continue
```

Le format du coup est vérifié selon la notation UCI.

```
if move not in board.legal_moves:
    print("Coup ill gal")
    continue
```

La légalité du coup est vérifiée localement avant toute transmission.

2.15. Proposition du coup

```
publish("move_proposed", {"uci": move_uci})
```

Le coup est proposé au service de validation, qui est le seul à pouvoir l'accepter ou le refuser.

2.16. Point d'entrée

```
if __name__ == "__main__":
    main()
```

Ce bloc garantit que le service démarre uniquement lorsque le fichier est exécuté directement.

3. Producer IA : producer_ai.py

Cette section décrit le service `producer_ai.py`. Ce service représente un producteur de coups automatique basé sur le moteur Stockfish. Il joue les pièces noires et propose ses coups au service de validation.

3.1. Importation des bibliothèques

```
import pika
import json
import time
import chess
import chess.engine
from datetime import datetime, timezone
import os
```

Ces importations permettent respectivement : la communication avec RabbitMQ, la manipulation des messages JSON, la gestion des temporisations, la représentation du plateau, l'interfaçage avec Stockfish, la gestion des horodatages, et l'accès au système de fichiers.

3.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
STOCKFISH_PATH = r"...stockfish.exe"
AI_COLOR = chess.BLACK
```

Ces constantes définissent l'environnement du service. L'intelligence artificielle joue explicitement les pièces noires. Le chemin vers Stockfish doit être valide pour permettre l'analyse.

3.3. État interne

```
board = chess.Board()
waiting_validation = False
game_over = False
channel = None
```

Le plateau local permet de reconstruire la partie. Les variables d'état indiquent si l'IA attend une validation et si la partie est terminée.

3.4. Publication des événements

```
def publish(event_type, payload):
    message = {
        "event_type": event_type,
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "payload": payload
    }
```

Chaque coup proposé par l'IA est encapsulé dans un événement contenant un type, un horodatage et une charge utile.

```
channel.basic_publish(
    exchange=EXCHANGE_NAME,
    routing_key="",
    body=json.dumps(message)
)
```

L'événement est diffusé à l'ensemble des services abonnés.

3.5. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Le service écoute passivement les événements diffusés par le système.

3.6. Synchronisation du plateau

```
if event_type == "game_started":
    board.reset()
    waiting_validation = False
    game_over = False
```

Lors du démarrage d'une nouvelle partie, le plateau local est réinitialisé et l'état interne est remis à zéro.

```
elif event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
    waiting_validation = False
```

Chaque coup validé est appliqué localement. L'IA peut alors rejouer lorsque c'est son tour.

3.7. Fin de partie

```
elif event_type == "game_ended":
    game_over = True
```

Lorsque la partie est terminée, l'IA sort de sa boucle principale.

3.8. Fonction principale

```
def main():
    global channel, waiting_validation, game_over
```

La fonction principale initialise le moteur Stockfish et lance la boucle de jeu de l'intelligence artificielle.

3.9. Initialisation de Stockfish

```
if not os.path.exists(STOCKFISH_PATH):
    raise FileNotFoundError(...)
```

Le service vérifie la présence du moteur Stockfish avant de démarrer.

```
engine = chess.engine.SimpleEngine.popen_uci(STOCKFISH_PATH)
```

Le moteur Stockfish est lancé via l'interface UCI.

3.10. Connexion à RabbitMQ

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host=RABBITMQ_HOST)
)
channel = connection.channel()
```

L'IA se connecte au serveur RabbitMQ et crée son canal de communication.

3.11. Abonnement aux événements

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

L'exchange fanout diffuse chaque événement à tous les services.

```
queue = channel.queue_declare(queue="", exclusive=True).method.
    queue
channel.queue_bind(exchange=EXCHANGE_NAME, queue=queue)
```

Une file temporaire exclusive est créée pour ce service.

3.12. Boucle principale de l'IA

```
while not game_over:
    connection.process_data_events(time_limit=0.1)
```

La boucle principale traite les événements et contrôle le moment où l'IA peut jouer.

3.13. Conditions de jeu

```
if waiting_validation:
    time.sleep(0.1)
    continue
```

L'IA attend la validation de son coup précédent.

```
if board.turn != AI_COLOR:
    time.sleep(0.1)
    continue
```

L'IA ne joue que lorsque c'est son tour.

3.14. Calcul et proposition du coup

```
result = engine.play(
    board,
    chess.engine.Limit(time=0.7)
)
move = result.move
```

Stockfish calcule le meilleur coup selon la position actuelle.

```
publish("move_proposed", {"uci": move.uci()})
waiting_validation = True
```

Le coup est proposé au service de validation. L'IA attend ensuite la décision officielle.

3.15. Arrêt du moteur

```
engine.quit()
```

Le moteur Stockfish est arrêté proprement à la fin de la partie.

4. Service de validation : validation_service.py :

Le service `validation_service.py` est le cœur logique du système. Il est le seul service autoritaire sur l'état officiel de la partie. Tous les autres services dépendent exclusivement de ses décisions.

4.1. Importation des bibliothèques

```
import pika
import json
import chess
```

La bibliothèque `pika` permet la communication avec RabbitMQ. La bibliothèque `json` est utilisée pour décoder et encoder les messages. La bibliothèque `python-chess` fournit les règles officielles du jeu.

4.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
```

Ces constantes définissent l'adresse du serveur RabbitMQ et l'exchange commun utilisé par tous les services.

4.3. Plateau officiel

```
board = chess.Board()
channel = None
```

Le plateau contenu dans ce service est le plateau officiel. Il représente l'unique source de vérité sur l'état de la partie.

4.4. Publication des événements

```
def publish(channel, event_type, payload):
    message = {
        "event_type": event_type,
        "payload": payload
    }
```

Cette fonction construit un message décrivant un événement à partir de son type et de ses données associées.

```
channel.basic_publish(
    exchange=EXCHANGE_NAME,
    routing_key="",
    body=json.dumps(message)
)
```

L'événement est ensuite diffusé sur l'exchange commun et reçu par tous les services abonnés.

4.5. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Chaque message reçu est décodé depuis le format JSON. Le traitement dépend ensuite du type d'événement.

4.6. Réception d'un coup proposé

```
if event_type == "move_proposed":
    uci = payload.get("uci")
```

Lorsqu'un coup est proposé par un producteur, le service récupère sa notation UCI.

4.7. Validation du format

```
try:
    move = chess.Move.from_uci(uci)
except ValueError:
    print("Coup invalide (format) :", uci)
    return
```

Le service vérifie que le coup respecte le format UCI. Un coup mal formé est immédiatement rejeté.

4.8. Validation de la légalité

```
if move in board.legal_moves:
    board.push(move)
```

La légalité du coup est vérifiée selon les règles officielles des échecs. Seuls les coups légaux sont appliqués au plateau officiel.

4.9. Diffusion du coup validé

```
publish(channel, "move_validated", {"uci": uci})
```

Une fois validé, le coup est diffusé à l'ensemble du système. Tous les autres services mettent alors à jour leur état local.

4.10. Détection de la fin de partie

```
if board.is_game_over():
    result = board.result()
```

Après chaque coup validé, le service vérifie si la partie est terminée.

```
publish(channel, "game_ended", {"result": result})
```

En cas de fin de partie, un événement spécifique est diffusé contenant le résultat officiel.

4.11. Gestion des coups illégaux

```
else:
    print("Coup ill gal :", uci)
```

Les coups illégaux sont ignorés. Ils ne produisent aucun événement valide.

4.12. Fonction principale

```
def main():
    global channel
```

La fonction principale initialise la connexion RabbitMQ et lance la boucle d'écoute du service.

4.13. Connexion à RabbitMQ

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host=RABBITMQ_HOST)
)
channel = connection.channel()
```

Le service se connecte au serveur RabbitMQ et crée son canal de communication.

4.14. Déclaration et abonnement

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

L'exchange fanout permet la diffusion globale des événements.

```
queue = channel.queue_declare(queue="", exclusive=True).method.
   queue
channel.queue_bind(exchange=EXCHANGE_NAME, queue=queue)
```

Une file temporaire exclusive est créée pour ce service.

4.15. Démarrage de l'écoute

```
channel.basic_consume(
    queue=queue,
    on_message_callback=on_message,
    auto_ack=True
)
channel.start_consuming()
```

Le service entre dans une boucle d'écoute bloquante et traite les événements au fur et à mesure.

5. Service d'analyse : analysis_service.py

Le service `analysis_service.py` est un service passif chargé d'analyser la position d'échecs après chaque coup validé. Il utilise le moteur Stockfish pour fournir une évaluation objective.

5.1. Importation des bibliothèques

```
import pika
import json
import chess
import chess.engine
import os
```

Ces bibliothèques permettent respectivement : la communication avec RabbitMQ, le traitement des messages JSON, la gestion du plateau d'échecs, l'interface avec le moteur Stockfish, et l'accès au système de fichiers.

5.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
STOCKFISH_PATH = r"...stockfish.exe"
```

Ces constantes définissent l'environnement d'exécution du service. Le chemin vers Stockfish doit être correct pour permettre l'analyse.

5.3. État interne du service

```
board = chess.Board()
channel = None
engine = None
```

Le plateau local permet de rejouer la partie. Le moteur Stockfish est initialisé une seule fois au démarrage.

5.4. Publication des résultats d'analyse

```
def publish(event_type, payload):
    message = {
        "event_type": event_type,
        "payload": payload
    }
```

Cette fonction encapsule l'envoi d'un événement d'analyse.

```
channel.basic_publish(
    exchange=EXCHANGE_NAME,
    routing_key="",
    body=json.dumps(message)
)
```

Les résultats sont diffusés à tous les services abonnés.

5.5. Analyse de la position

```
def analyse_position():
    info = engine.analyse(
        board,
        chess.engine.Limit(time=0.5)
    )
```

Le moteur Stockfish analyse la position actuelle avec une limite de temps afin de garantir la réactivité.

5.6. Interprétation du score

```
score = info["score"]
```

Le score retourné par Stockfish est ensuite interprété.

```
if score.is_mate():
    mate = score.mate()
```

Le service distingue explicitement le cas d'un mat forcé.

```
value = score.white().score() / 100.0
```

Dans le cas général, l'évaluation est convertie depuis les centipawns vers une valeur lisible.

5.7. Calcul du meilleur coup

```
best = engine.play(
    board,
    chess.engine.Limit(time=0.3)
).move.uci()
```

Stockfish est également utilisé pour déterminer le meilleur coup recommandé dans la position actuelle.

5.8. Diffusion de l'analyse

```
publish(
    "analysis",
    {
        "score": value,
        "best_move": best,
        "text": text
    }
)
```

L'analyse complète est diffusée sous forme d'événement, comprenant l'évaluation numérique et le meilleur coup.

5.9. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Le service écoute passivement les événements du système.

5.10. Nouvelle partie

```
if event_type == "game_started":
    board.reset()
```

Lorsqu'une nouvelle partie commence, le plateau local est réinitialisé.

5.11. Coup validé

```
elif event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
    analyse_position()
```

Chaque coup validé est rejoué localement, puis immédiatement analysé.

5.12. Fin de partie

```
elif event_type == "game_ended":
    print("Analyse termin e")
```

Lorsque la partie se termine, le service cesse logiquement son activité.

5.13. Fonction principale

```
def main():
    global channel, engine
```

La fonction principale initialise le moteur Stockfish et la connexion RabbitMQ.

5.14. Initialisation de Stockfish

```
if not os.path.exists(STOCKFISH_PATH):
    raise FileNotFoundError(...)
```

Le service vérifie que le moteur Stockfish est disponible.

```
engine = chess.engine.SimpleEngine.popen_uci(STOCKFISH_PATH)
```

Le moteur est lancé via l'interface UCI.

5.15. Connexion et abonnement

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host=RABBITMQ_HOST)
)
channel = connection.channel()
```

Le service se connecte au serveur RabbitMQ.

```
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

L'exchange fanout permet de recevoir tous les événements.

```
queue = channel.queue_declare(queue="", exclusive=True).method.
   queue
channel.queue_bind(exchange=EXCHANGE_NAME, queue=queue)
```

Une file temporaire exclusive est créée pour le service.

5.16. Démarrage de l'écoute

```
channel.basic_consume(
    queue=queue,
    on_message_callback=on_message,
    auto_ack=True
```

```
)
channel.start_consuming()
```

Le service entre dans une boucle d'écoute bloquante et analyse chaque coup validé en temps réel.

6. Service d'explication : explanation_service.py

Le service `explanation_service.py` est chargé de produire une explication textuelle pédagogique après chaque coup validé. Il combine l'analyse Stockfish et une IA de génération de texte.

6.1. Importation des bibliothèques

```
import pika
import json
import chess
import os
from openai import OpenAI, RateLimitError, OpenAIError
```

Ces bibliothèques permettent : la communication avec RabbitMQ, le traitement des messages JSON, la reconstruction du plateau d'échecs, l'accès aux variables d'environnement, et l'utilisation de l'API OpenAI.

6.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
MODEL = "gpt-4o-mini"
```

Ces constantes définissent l'environnement du service et le modèle d'IA utilisé pour la génération de texte.

6.3. Initialisation du client OpenAI

```
api_key = os.getenv("OPENAI_API_KEY")
if not api_key:
    raise RuntimeError(...)
client = OpenAI(api_key=api_key)
```

Le service récupère la clé OpenAI depuis les variables d'environnement. Il refuse de démarrer si la clé est absente.

6.4. État interne

```
board = chess.Board()
last_analysis = {
    "score": 0.0,
    "best_move": "    "
```

```
}
channel = None
```

Le plateau local sert à reconstruire la position. La dernière analyse reçue est mémorisée pour fournir du contexte à l'IA.

6.5. Publication des explications

```
def publish(event_type, payload):
    channel.basic_publish(
        exchange=EXCHANGE_NAME,
        routing_key="",
        body=json.dumps({
            "event_type": event_type,
            "payload": payload
        })
    )
```

Cette fonction permet de diffuser une explication générée sous forme d'événement.

6.6. Génération du texte explicatif

```
def call_llm(move, score, best_move, fen):
```

Cette fonction encapsule l'appel à l'IA de génération de texte.

```
prompt = f"""
Tu es un professeur d' checs  p dagogue.
Position actuelle (FEN) : {fen}
Coup jou  : {move}
Meilleur coup recommand  : {best_move}
 valuation  Stockfish : {score:+.2f}
"""
```

Le prompt fournit à l'IA le contexte nécessaire pour produire une explication pédagogique.

6.7. Gestion des erreurs de l'IA

```
except RateLimitError:
    return "Ce coup modifie  l  quilibre  de la position..."
```

Des textes de secours sont utilisés en cas de dépassement de quota ou d'erreur API, garantissant la continuité du service.

6.8. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
    payload = event.get("payload", {})
```

Le service écoute passivement les événements du système.

6.9. Mise à jour de l'analyse

```
elif event_type == "analysis":
    last_analysis["score"] = payload.get("score", 0.0)
    last_analysis["best_move"] = payload.get("best_move", "    ")
```

La dernière analyse Stockfish est stockée pour être utilisée lors de l'explication.

6.10. Génération après coup validé

```
elif event_type == "move_validated":
    move = chess.Move.from_uci(payload["uci"])
    board.push(move)
```

Chaque coup validé est appliqué localement.

```
explanation = call_llm(
    payload["uci"],
    last_analysis["score"],
    last_analysis["best_move"],
    board.fen()
)
```

L'explication est générée à partir du coup joué, de l'évaluation et de la position.

6.11. Diffusion de l'explication

```
publish(
    "move_explained",
    {
        "uci": payload["uci"],
        "text": explanation
    }
)
```

L'explication est diffusée à l'ensemble du système.

6.12. Initialisation et écoute

```
connection = pika.BlockingConnection(...)
channel = connection.channel()
```

Le service se connecte à RabbitMQ.

```
channel.exchange_declare(...)
channel.basic_consume(...)
channel.start_consuming()
```

Le service entre dans une boucle d'écoute bloquante.

7. Service spectateur : spectator_service.py

Le service `spectator_service.py` fournit une interface graphique permettant de suivre une partie d'échecs en temps réel. Il est entièrement passif et se contente d'afficher les informations reçues.

7.1. Importation des bibliothèques

```
import pika
import json
import chess
import tkinter as tk
from tkinter import Text, Scrollbar
from PIL import Image, ImageTk
import os
```

Ces bibliothèques permettent respectivement : la communication RabbitMQ, le traitement JSON, la gestion du plateau, la création de l'interface graphique, l'affichage des images, et l'accès au système de fichiers.

7.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
PIECES_PATH = r"..."
SQUARE_SIZE = 64
```

Ces constantes définissent la configuration réseau et les paramètres d'affichage du plateau.

7.3. État interne

```
board = chess.Board()
piece_images = {}
current_score = 0.0
best_move = "    "
```

Le plateau local permet de représenter la position courante. Les variables stockent la dernière analyse reçue.

7.4. Chargement des images

```
def load_images():
    img = Image.open(path).convert("RGBA")
    img = img.resize(...)
```

Les images des pièces sont chargées depuis le disque, redimensionnées, puis stockées pour un affichage rapide.

7.5. Dessin du plateau

```
def draw_board():
    canvas.delete("all")
```

Le plateau est redessiné entièrement après chaque mise à jour.

```
canvas.create_rectangle(...)
canvas.create_image(...)
```

Les cases sont dessinées puis les pièces sont affichées en fonction de l'état du plateau.

7.6. Panneau d'informations

```
def update_side_panel():
    score_label.config(...)
```

Cette fonction met à jour l'évaluation, le camp avantagé et le meilleur coup recommandé.

7.7. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
```

Le service réagit aux événements reçus.

```
elif event_type == "analysis":
    current_score = payload.get("score", 0.0)
```

L'évaluation Stockfish est affichée dès réception.

```
elif event_type == "move_explained":
    explanation_box.insert(...)
```

L'explication pédagogique est affichée dans la zone de texte.

7.8. Intégration RabbitMQ et Tkinter

```
def rabbitmq_loop():
    connection.process_data_events(...)
    root.after(100, rabbitmq_loop)
```

Cette fonction permet d'intégrer RabbitMQ dans la boucle événementielle de Tkinter sans bloquer l'interface.

8. Service de stockage : storage_service.py

Le service `storage_service.py` assure la persistance des événements de la partie d'échecs. Il enregistre chronologiquement les événements importants dans des fichiers JSON afin de permettre une relecture ultérieure.

8.1. Importation des bibliothèques

```
import pika
import json
import os
from datetime import datetime
```

La bibliothèque `pika` permet la communication avec RabbitMQ. Le module `json` sert à sérialiser les événements. Le module `os` permet de manipuler le système de fichiers. Le module `datetime` est utilisé pour générer des horodatages.

8.2. Configuration globale

```
RABBITMQ_HOST = "localhost"
EXCHANGE_NAME = "chess.events"
STORAGE_DIR = "games_storage"
```

Ces constantes définissent l'adresse du serveur RabbitMQ, l'exchange commun à tous les services, et le dossier dans lequel les parties seront stockées.

8.3. Initialisation du dossier de stockage

```
if not os.path.exists(STORAGE_DIR):
    os.makedirs(STORAGE_DIR)
```

Le dossier de stockage est créé automatiquement s'il n'existe pas encore. Cela garantit que le service peut démarrer sans configuration manuelle.

8.4. Fichier de partie courant

```
current_file = None
```

Cette variable globale référence le fichier correspondant à la partie en cours. Elle est initialisée à `None` lorsqu'aucune partie n'est active.

8.5. Ouverture d'un nouveau fichier de partie

```
def open_new_game_file():
    filename = datetime.now().strftime(
        "game_%Y%m%d_%H%M%S.json"
    )
```

Lorsqu'une nouvelle partie commence, un nom de fichier unique est généré à partir de la date et de l'heure.

```
path = os.path.join(STORAGE_DIR, filename)
current_file = open(path, "w", encoding="utf-8")
current_file.write("[\n")
```

Le fichier est ouvert en écriture et initialisé comme un tableau JSON afin de stocker une liste d'événements.

8.6. Fermeture du fichier de partie

```
def close_game_file():
    if current_file:
        current_file.write("\n]\n")
        current_file.close()
```

À la fin de la partie, le tableau JSON est correctement refermé et le fichier est fermé proprement.

8.7. Stockage d'un événement

```
def store_event(event):
    json.dump(event, current_file, ensure_ascii=False, indent=2)
    current_file.write(",\n")
```

Chaque événement reçu est écrit dans le fichier sous forme JSON lisible. Une virgule est ajoutée pour séparer les événements successifs.

8.8. Réception des événements

```
def on_message(ch, method, properties, body):
    event = json.loads(body)
    event_type = event.get("event_type")
```

Le service écoute les messages RabbitMQ et décode chaque événement reçu.

8.9. Début de partie

```
if event_type == "game_started":
    close_game_file()
    open_new_game_file()
    store_event(event)
```

Lorsqu'une nouvelle partie commence, un nouveau fichier est créé. L'événement de démarrage est immédiatement enregistré.

8.10. Coup validé

```
elif event_type == "move_validated":
    store_event(event)
```

Chaque coup validé par le service de validation est ajouté à l'historique de la partie.

8.11. Fin de partie

```
elif event_type == "game_ended":
    store_event(event)
    close_game_file()
```

Lorsque la partie se termine, l'événement final est enregistré et le fichier est fermé définitivement.

8.12. Fonction principale

```
def main():
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host=RABBITMQ_HOST)
    )
```

La fonction principale initialise la connexion à RabbitMQ.

```
channel = connection.channel()
channel.exchange_declare(
    exchange=EXCHANGE_NAME,
    exchange_type="fanout",
    durable=True
)
```

Le service se connecte à l'exchange commun afin de recevoir tous les événements.

8.13. Abonnement et écoute

```
queue = channel.queue_declare(queue="", exclusive=True).method.
    queue
channel.queue_bind(exchange=EXCHANGE_NAME, queue=queue)
```

Une file temporaire exclusive est créée pour le service.

```
channel.basic_consume(
    queue=queue,
    on_message_callback=on_message,
    auto_ack=True
)
channel.start_consuming()
```

Le service entre dans une boucle d'écoute continue et stocke les événements en temps réel.