PromptChess – Specifica di Progetto (v1.0)

1. Scopo e obiettivi

1.1 Scopo

PromptChess è un progetto dimostrativo che integra un **LLM conversazionale** (es. ChatGPT) in un ciclo di gioco degli scacchi, con l'obiettivo di:

- simulare una partita **Umano vs AI**,
- ridurre le **hallucinations** dell'LLM tramite verifica locale,
- formalizzare lo scambio dati tramite **JSON** e regole di output strutturate.

1.2 Obiettivi principali

- Generare mosse AI coerenti e **legali** rispetto alle regole scacchistiche.
- Fornire un ciclo robusto "agent-like" (dipendente digitale):
  - **Percepire** stato scacchiera,
  - **Ragionare** mossa migliore,
  - **Agire** applicazione + validazione,
  - **Imparare** tramite correzioni (nel senso di retry e adattamento del prompt, non training).
- Gestire limiti di token e cronologia conversazionale in modo controllato.
- Simulare una partita a scacchi tra un utente umano e un assistente AI,
- Sfruttare un **modello linguistico (LLM)** per generare mosse semanticamente coerenti,
- Integrare **regole scacchistiche formali** per validare ogni mossa proposta,
- Mantenere lo storico delle mosse, lo stato della scacchiera e un commento strategico,
- Evolvere nel tempo verso un **motore auto-consistente** in grado di migliorare il proprio gioco grazie a una knowledge base interrogabile.

1.3 Funzionalità non ancora disponibili

- Un motore scacchistico ottimale tipo Stockfish (minimax/NNUE).
- Training o fine-tuning del modello LLM.
- Persistenza storica e retrieval (RAG) in questa versione (possibile evoluzione).

Funzionalità Principali

1. Gestione della Scacchiera

- La scacchiera è modellata tramite un JSON con coordinate in formato `"a2"`, `"e7"` ecc.
- Le informazioni sono organizzate in due insiemi:
  - `"bianchi"`: pedoni, torri, cavalli, alfieri, regina, re.
  - `"neri"`: stesso schema.

2. Motore di Regole

- Funzioni come `is_valid_move()`, `find_checkers()` e `apply_move()` assicurano la coerenza con le regole ufficiali FIDE.
- Le mosse vengono simulate e verificate prima di essere registrate.
- Supporto per:
  - promozione pedoni,
  - arrocco,
  - scacco e scacco matto (in fase di evoluzione),
  - gestione turni e turnazione bianco/nero.

3. Integrazione LLM (Prompt Engine)

- Ogni turno dei Neri viene gestito tramite chiamata a un **modello linguistico** (es. GPT).
- Il prompt fornisce contesto:
  - stato della scacchiera,
  - ultima mossa avversaria,
  - storia delle mosse precedenti,
  - regole generali e limiti di output.
- L'output atteso dal modello è un JSON del tipo:
- ```
  {
    "mossa_proposta": "e7-e5",
    "commento_giocatore": "Controllo del centro, apertura standard.",
    "messaggio_avversario": "Preparati a una partita serrata!",
    "bianchi": { ... },
    "neri": { ... }
  }
  ```

4. Storia e memoria di gioco

- Ogni mossa è salvata in una collection MongoDB o struttura persistente:
  - `game_id`, `turn`, `board_json`, `last_move_white`, `next_ai_move_black`, `comment`, `timestamp`.
- Questa **memoria delle partite** potrà essere utilizzata per allenare o ispirare nuove risposte del modello.

3. Componenti e responsabilità

3.1 `chess_engine.py` (Orchestratore / Engine)

Responsabilità

- Eseguire il loop principale della partita e gestire i turni.
- Acquisire input umano e validare mosse.
- Costruire prompt (system + user) e inviare richieste al proxy OpenAI.
- Validare la risposta AI:
  - parsing e "riparazione" JSON,
  - rilevazione della mossa,
  ```

- verifica legalità,
- verifica scacco,
- gestione retry e fallback di modello.

Output

- Risultato della partita (es. checkmate, king missing, time limit exceeded) e log della sessione.

- play_match()

- Algoritmo centrale del sistema.
- Coordina i turni, aggiorna la scacchiera, interroga il modello, gestisce errori e logica di fallback.

- play_turn()

- Alternanza tra mosse dell'utente (bianco) e dell'AI (nero).
- Ritorna l'esito della mossa, commento e messaggio di sfida.

3.2 `chess_core.py` (Regole e utilities di dominio)

Responsabilità

- Rappresentazione board:
  - conversione JSON ↔ DataFrame 8×8.
- Applicazione e rilevazione mosse:
  - `apply_move`, `detect_move`.
- Validazione:
  - `is_legal_move` (regole di movimento),
  - `find_checkers` / check state,
  - funzioni di utilità (path clear, ecc.).
- Stato partita:
  - `is_game_active`, `game_result`.

3.3 `openai_proxy_service.py` (Proxy/API client + session manager)

Responsabilità

- Esporre endpoint HTTP per interagire con l'LLM.
- Gestire la sessione conversazionale tramite classe `ChatSession`:
  - storage messaggi con ruoli,
  - messaggi di sistema multipli,
  - sliding window della history,
  - pop/append controllato.
- Gestione token:
  - conteggio token (tiktoken),
  - budget dinamico di risposta (dynamic max tokens),

    - prevenzione sforamento limite modello.
- Persistenza “debug/audit” della history a fine esecuzione.

---

4. Modello dati

4.1 Board JSON (input/output)

La scacchiera è rappresentata come oggetto JSON con due insiemi principali (es. `bianchi` e `neri`), organizzati per tipologia pezzo.

Esempio (semplificato):

```
{
  "neri": {
    "pedoni": ["a7","b7"],
    "torri": ["a8","h8"],
    "cavalli": ["b8","g8"],
    "alfieri": ["c8","f8"],
    "regina": ["d8"],
    "re": ["e8"]
  },
  "bianchi": {
    "pedoni": ["a2","b2"],
    "torri": ["a1","h1"],
    "cavalli": ["b1","g1"],
    "alfieri": ["c1","f1"],
    "regina": ["d1"],
    "re": ["e1"]
  }
}
```

4.2 Risposta AI attesa (schema)

L’assistente deve rispondere con JSON strutturato (senza testo extra), contenente:

- `mossa_proposta` (es. `"e7-e5"`)
- `neri` e `bianchi` aggiornati
- opzionali: `commento_giocatore`, `messaggio_avversario`
- in caso di errore: campo `errore`

Esempio:

```
{
  "mossa_proposta": "e7-e5",
  "neri": { "...": "..." },
  "bianchi": { "...": "..." },
  "commento_giocatore": "Controllo il centro...",
  "messaggio_avversario": "Tocca a te."
}
```

4.3 Conversazione (ChatSession)

Ogni messaggio è un dict:

```
{"role": "system|user|assistant", "content": "<text>"}
```

- I system messages contengono: regole, vincoli di formato, obiettivi.
- I user messages contengono: board JSON + last move.
- Le risposte assistant contengono: JSON di risposta.

5. Flusso operativo

5.1 Setup

1. Avvio del proxy:
   - `python openai_proxy_service.py`
2. Avvio engine:
   - `python chess_engine.py`
3. Inizializzazione sessione LLM con system messages (`/chat/init`).

5.2 Turno umano

1. Input CLI: `<piece> <from> <to>` (es. `P e7 f8`)
2. Verifica:
   - esistenza pezzo in `from`,
   - legalità movimento,
   - assenza di auto-scacco.
3. Applicazione mossa e passaggio turno all'AI.

5.3 Turno AI

1. Creazione prompt:
   - system messages (regole, schema JSON, obiettivi),
   - user message: board JSON + last move bianco.
2. Invio a proxy (`/chat`) con modello e temperatura.
3. Parsing e "repair" JSON se necessario.
4. Conversione JSON → DataFrame.
5. `detect_move` e verifica:
   - legalità della mossa,
   - non lasciare il proprio re sotto scacco.
6. Retry loop:
   - fino a `max_retries`,
   - eventualmente cambio "gear" del modello.
7. Applicazione mossa e ritorno turno all'umano.

5.4 Fine partita

Condizioni:

- re mancante / checkmate logico (a seconda delle funzioni implementate),
- time limit superato.
  Output: stringa `game_result(...)`.

5. Gestione hallucinations e resilienza

5.1 Tipologie di errori attesi

- AI non aggiorna lo stato board (JSON invariato).
- AI propone mosse del Bianco (move-repeat / role confusion).
- JSON malformato o schema diverso.
- Mossa illegale o auto-scacco.
- Risposte vuote / timeout / rate limit.

5.2 Contromisure

- **Schema rigido** imposto via system message.
- **Repair** del JSON.
- **Validazione locale** completa su ogni mossa.
- **Retry loop** con:
  - feedback correttivo al modello,
  - rollback dell'ultimo assistant,
  - eventuale upgrade modello.
- **Token budgeting dinamico** lato proxy.

6. Requisiti non funzionali

6.1 Affidabilità

- Ogni mossa AI deve essere validata localmente prima dell'applicazione.
- In caso di fallimento reiterato: fallback modello o terminazione controllata.

6.2 Prestazioni / costi

- Necessità di minimizzare token:
  - sliding window history,
  - system messages concisi e modulari,
  - budgeting max tokens per modello.
- Compatibilità con modelli economici, con la consapevolezza che:

- contesti lunghi portano ad errori di limite token,
- modelli piccoli hanno maggiore rischio di hallucinations.

6.3 Tracciabilità

- Salvataggio history di sessione per debug/audit.

7. Estensioni future (roadmap)

1. **Memoria persistente + MongoDB Vector Search**
   - Salvare stati/mosse e recuperare esempi simili (RAG) per ridurre chiamate LLM.
2. **Training mode / coach**
   - commenti didattici e obiettivi di apprendimento (scacchi come tutor).
3. **Vision input**
   - acquisizione board da immagine con VLM + conversione a JSON.
4. **Metriche**
   - tasso mosse illegali, retry per mossa, costo token per partita, qualità (valutazione Stockfish).

5. Prompt Engine dinamico: adattamento della temperatura in base allo stato.
6. Gestione multi sessione, definizione profilo utente, accesso tramite credenziali, avvio modalità di gioco singola partita, Modalità tornei e salvataggio partita.
7. Integrazione frontend interattivo (scacchiera grafica).

8. Limiti Attuali

- Capacità del modello limitata dal contesto massimo (token).
- Nessun meccanismo di auto-apprendimento sul lungo periodo.
- Alcune mosse speciali non sono ancora perfettamente implementate (en passant, matto verificato, ecc.).
- Non supporta nativamente logiche asincrone o multiplayer.