System Architecture and Mechanics Design: RAG-Powered Text-Based RPG

1. Abstract and Introduction

This document outlines the technical architecture, game mechanics, and development roadmap for a text-based idle RPG. Designed as a practical application of Retrieval-Augmented Generation (RAG) and LangChain, this project transforms a traditional LLM chat interface into a dynamic, stateful game engine. The core innovation lies in utilizing RAG not just for Q&A, but as a "World Engine" that retrieves environmental lore, enemy statistics, and item properties dynamically based on player actions.

2. System Architecture

The application follows a decoupled client-server architecture, allowing the AI to act as both narrative storyteller and strict game state manager.

2.1 Frontend (User Interface & HUD)

The frontend acts as the visual representation of the game state. It does not calculate combat or manage inventory; it merely reflects the state provided by the backend.

- **Chat Window:** Displays the narrative text generated by the LLM and the player's text inputs.
- **Dynamic HUD:** A fixed UI panel displaying:
  - **Vitals:** Health (HP) and Mana (MP) bars.
  - **Core Stats:** Level, EXP, Strength, Defense, Intelligence, Agility.
  - **Economy:** Coins/Scrap.
  - **Time of Day Indicator:** Visual representation of Morning, Afternoon, Evening, and Night.
- **Data Handling:** Parses JSON responses from the backend and triggers UI animations (e.g., flashing red on HP loss, cycling a sun/moon icon for time).

2.2 Backend (FastAPI & State Management)

FastAPI serves as the bridge between the UI and the AI engine.

- **Endpoints:** A primary /chat POST endpoint receiving the user's action and current session ID.
- **State Manager:** A persistent dictionary (or lightweight SQLite/Redis database) mapping Session IDs to the current game state. The backend injects this state into the LLM prompt on every turn and updates its database based on the LLM's structured output.

2.3 AI Engine (LangChain & RAG)

This is the core logic center.

- **Vector Store:** A local database (e.g., Chroma or FAISS) loaded with the "Lore Document". It chunks the text so that when a player explores the "Ash Plains," it retrieves the stats for an "Ash Hound."
- **LLM (Game Master):** Instructed to parse the player's intent, calculate outcomes using the retrieved RAG data and current player state, and return a strict JSON object containing narrative text and state updates.

3. Core Game Mechanics

3.1 Player Attributes

- **Health (HP):** Reaches 0, the player "faints," loses a portion of coins/EXP, and wakes up in the Bastion on the next Morning.
- **Mana (MP):** Consumed by special attacks or magic (scaled by Intelligence).
- **Strength (STR):** Determines base physical damage.
- **Defense (DEF):** Reduces incoming physical damage.
- **Intelligence (INT):** Determines magic damage and maximum Mana.
- **Agility (AGI):** Determines the chance to dodge an enemy attack or flee combat.

3.2 The Temporal Cycle (Time Management)

The game operates on a localized time system, forcing strategic decision-making. The HUD displays four phases: **Morning -> Afternoon -> Evening -> Night**.

- **Time-Consuming Actions:** Exploring a new area, hunting, traveling, or participating in a combat round. These actions advance the clock by one phase.
- **Free Actions:** Checking stats, talking to safe-zone NPCs, buying/selling items, and consuming potions. The clock remains static.
- **Day Reset:** When an action occurs at Night, the state transitions to Morning, the "Day Counter" increments by 1, and the player receives a minor HP/MP restoration ("rest bonus").
- **Combat Pause:** Once the combat_active state is true, the day cycle pauses entirely. Combat operates on a turn-by-turn basis until resolved, after which the time advances by one phase representing the time spent fighting.

3.3 Dynamic Combat & Progression

Combat is resolved by the LLM acting as the physics engine.

1. **Initiation:** Player encounters an enemy. The RAG system provides the enemy's HP, Attack, and Defense.
2. **Resolution:** The LLM calculates: (Player STR + Dice Roll) - Enemy DEF = Damage. It narrates the blow and calculates enemy retaliation simultaneously.
3. **Leveling:** Defeating enemies grants EXP. Hitting predefined thresholds (e.g., 100 EXP for Level 2) triggers a Level Up, increasing base stats and fully restoring HP/MP.

4. Data Schema (The LLM Contract)

To ensure the HUD updates correctly, the LangChain output parser must enforce the following JSON structure from the LLM on every turn:

```
{
  "narrative": "You swing your rusted sword at the Ash Hound, striking true for 12 damage. The beast snarls and bites your leg, dealing 4 damage.",
  "state_updates": {
    "hp_change": -4,
    "mana_change": 0,
    "coins_change": 0,
    "exp_change": 0,
    "time_advanced": false,
    "combat_active": true,
    "current_enemy": "Ash Hound (HP: 18/30)"
  }
}
```

The FastAPI backend takes state_updates, applies them to the current database state, and sends the newly calculated absolute values (e.g., current_hp: 46) to the frontend.

5. Phased Implementation Plan

Phase 1: Frontend UI & HUD (Day 1 - Morning)

- Build the layout: Chat history container, text input bar, and a sidebar/top bar for the HUD.
- Implement state variables in the frontend framework (React state or vanilla JS objects) for HP, MP, Level, and Time.
- Create visual components: Progress bars for HP/MP, icons for the time of day.
- Write mock functions to simulate receiving the JSON schema above and updating the HUD.

Phase 2: FastAPI Backend & State (Day 1 - Afternoon)

- Initialize FastAPI. Create a /chat endpoint.
- Implement the GameState Python class to hold player stats.
- Create the logic that takes state_updates from the mock AI and applies them to the GameState (e.g., preventing HP from dropping below 0).
- Connect the frontend to the backend and ensure data flows cleanly.

Phase 3: RAG & Knowledge Base (Day 2 - Morning)

- Format the Lore Document (provided separately).

- Use LangChain's TextLoader and RecursiveCharacterTextSplitter.
- Embed the chunks into a lightweight vector store (ChromaDB).
- Create a RetrievalQA chain. Test retrieving enemy stats by passing queries like "I walk into the Crystal Forest."

Phase 4: Prompt Engineering & Game Loop (Day 2 - Afternoon)

- Combine the Retriever, the Game State, and the User Input into a comprehensive System Prompt.
- Implement LangChain's StructuredOutputParser to enforce the JSON schema.
- **Vibe Coding Iteration:** Playtest the game. Adjust the prompt if the AI is too generous with EXP, forgets to advance the time of day, or fails to pause time during combat. Add edge-case handling for when the player dies.