

Interactive Role-Playing Game System Using Dice-Based Mechanics and AI Narration

Kema Uday Kiran

Dept. of Data Science
Sreenidhi Institute of Science
and Technology
Hyderabad, India
22311A6720@ds.sreenidhi.edu.in

Chiluka Pawan Kumar

Dept. of Data Science
Sreenidhi Institute of Science
and Technology
Hyderabad, India
23315A6702@ds.sreenidhi.edu.in

N Bhanu Prasad

Dept. of Data Science
Sreenidhi Institute of Science
and Technology
Hyderabad, India
22311A6732@ds.sreenidhi.edu.in

Dr. Md Jaffar Sadiq

Dept. of Data Science
Sreenidhi Institute of Science and Technology
Hyderabad, India
Jaffar.m@sreenidhi.edu.in

Dr. Naadem Divya

Dept. of Data Science
Sreenidhi Institute of Science and Technology
Hyderabad, India
divya.n@sreenidhi.edu.in

Abstract—Tabletop role-playing games such as *Dungeons & Dragons* depend on a human Game Master to adjudicate rules, resolve player decisions through probabilistic dice mechanics, and weave a coherent narrative across sessions. Automating this role using Large Language Models (LLMs) alone leads to severe reliability problems: unconstrained generative models hallucinate game state, permit physically impossible player actions, and fail to maintain consistent numerical variables over extended play. This paper presents a hybrid software architecture that cleanly separates deterministic game logic from constrained narrative generation. The backend employs a programmatic Python rules engine grounded in a comprehensive static rulebook covering 20 monsters, 18 weapons, 26 spells, 13 NPC archetypes, and 8 puzzle templates to resolve all combat, skill checks, and state mutations with absolute mathematical precision. Six specialized LLM micro-agents—a Validator, Parser, Story Engine, Director, Enemy AI, and Narrator—handle only semantic tasks, each constrained by typed Pydantic schemas and post-generation rulebook validation. A React-based frontend communicates with this backend via both REST endpoints and WebSocket channels, enabling both single-player and real-time multiplayer sessions with up to four concurrent players. Empirical benchmarks over 100 adversarial and valid test prompts demonstrate that the hybrid Validator intercepts 95.0% of rule-violating inputs while incorrectly rejecting only 2.5% of legitimate gameplay actions, compared to an unconstrained LLM baseline that permitted 88% of adversarial inputs. Deterministic rule execution achieves sub-millisecond latency (≈ 0.19 ms), representing a speedup exceeding $6,000\times$ over equivalent LLM-mediated resolution.

Keywords: Large Language Models, Tabletop RPG Automation, Deterministic Game Engines, WebSocket Communication, Pydantic Schema Validation, Prompt Injection Defense.

I. INTRODUCTION

The *Dungeons & Dragons* (D&D) tabletop system encodes a sophisticated interaction model: a human Game Master (GM) simultaneously enforces a probabilistic rule system, maintains the state of dozens of entities, arbitrates free-form player language, and generates an evolving narrative—all in real time

[6]. The scarcity of skilled GMs, combined with scheduling constraints for multi-player groups, has motivated research into automated GM systems.

Recent advances in Large Language Models (LLMs), particularly Transformer-based architectures [1], have made open-ended text generation remarkably fluent. Systems like *AI Dungeon* [2] demonstrated that models such as GPT-2 and GPT-3 can sustain interactive fiction. However, deploying an LLM as the sole arbiter of a rule-heavy game reveals fundamental reliability gaps. LLMs operate on probabilistic token prediction; they do not possess an internal arithmetic unit, a persistent memory of discrete variables, or the capacity to enforce negative constraints (“you cannot fly because you lack that spell”). When asked to simultaneously generate prose and track hit points, armor classes, or spell slot expenditure, these models exhibit what we term *mechanical hallucination*—producing outputs that are narratively plausible but mechanically incorrect.

This paper addresses the problem through architectural separation. We present a fully functional web-based D&D automation platform whose design principle is: **no LLM ever touches a number**. Every dice roll, damage calculation, ability modifier, saving throw, and inventory mutation is executed by a deterministic Python engine. The LLM’s role is confined to six narrowly scoped semantic tasks—input validation, intent parsing, world-state narration, encounter pacing, enemy decision-making, and prose synthesis—each bounded by typed schemas and a post-generation validation loop that cross-references all LLM outputs against a static rulebook.

The remainder of this paper is organized as follows. Section II surveys prior work on procedural storytelling and LLM-based game systems. Section III details the decoupled system architecture and defines each component. Section IV describes the static rulebook and validation loop. Section V covers the

runtime state management, WebSocket multiplayer protocol, and frontend integration. Section VI presents empirical results on latency and adversarial resilience. Section VII discusses findings and future directions.

II. RELATED WORK

A. Interactive Fiction and Parser-Based Systems

Early interactive fiction systems (e.g., Zork, Inform) relied on hand-authored parser trees mapping noun-verb pairs to scripted outcomes [7]. While deterministic, these systems could not handle inputs outside their predefined grammar and offered no generative narrative capability.

B. Unconstrained Generative Approaches

The release of GPT-2 enabled *AI Dungeon* [2], which treated the entire game—rules, narrative, and state—as a single text-completion problem. This approach demonstrated compelling emergent storytelling but suffered from context drift: the model would forget character attributes, invent non-existent items, and fail to enforce basic physics. Users discovered that adversarial prompting (e.g., “I cast Wish to win the game”) could bypass all intended constraints.

C. Structured Reasoning in LLMs

Wei et al. introduced Chain-of-Thought (CoT) prompting [3], forcing models to emit intermediate reasoning steps before producing a final answer. Yao et al. extended this with the ReAct framework [4], interleaving reasoning traces with action commands. While these techniques improve logical consistency within a single inference call, they impose significant token overhead and cannot guarantee correctness for operations as simple as comparing two integers. Park et al. [5] demonstrated generative agents that maintain memory streams and reflection, but their architecture still relies on the LLM for all state management, making it unsuitable for systems requiring deterministic fairness guarantees.

D. Hybrid Architectures

Our work is most closely aligned with the hybrid paradigm, where a programmatic engine handles state and rules while an LLM handles language. To our knowledge, no prior published system implements this separation with the granularity described here: six specialized LLM agents, a 600-line static rulebook, a post-generation validation loop with fuzzy entity matching, and a real-time WebSocket multiplayer protocol supporting independent location-based state.

III. SYSTEM ARCHITECTURE

The system is partitioned into three tiers: a React frontend, a FastAPI backend serving both REST and WebSocket endpoints, and a set of LLM inference calls routed through the Groq API (Llama-3.3 70B on custom Language Processing Units). Fig. 1 illustrates the data flow.

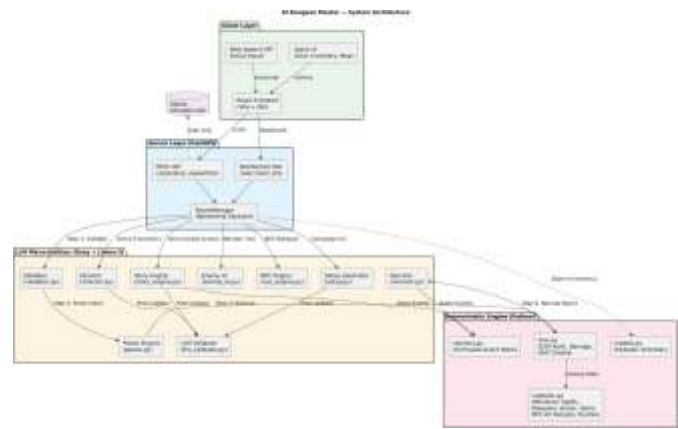


Fig. 1. High-level data flow between the frontend client, the FastAPI backend (deterministic engine + LLM micro-agents), and the Groq inference endpoint.

A. The Deterministic Engine Layer

All mechanical state resides in Pydantic-validated data models. The `Character` model encodes 35 distinct fields: six ability scores (STR, DEX, CON, INT, WIS, CHA), derived modifiers, hit points (current, maximum, temporary), armor class, level, experience points, proficiency bonus, skill proficiencies, saving throw proficiencies, conditions (Poisoned, Burning, Frightened, Stunned, etc.), spell slots, exhaustion level, concentration state, passive perception, and a typed inventory of `Item` objects. The `WorldState` model tracks the campaign timeline, world flags, a list of `SceneBeat` objects forming the adventure outline, map nodes and paths, and an NPC registry.

The core execution function, `execute_action()`, accepts three arguments: an `Action` schema, a source `Character`, and a target `Character`. It handles ten distinct action types—attack, heal, cast_spell, defend, hide, move, interact, communicate, grapple, and shove—each with complete 5th Edition D&D mechanics:

$$\text{Outcome} = \begin{cases} \text{Critical Hit,} & \text{if } \text{Roll}_{d20} = 20 \\ \text{Critical Miss,} & \text{if } \text{Roll}_{d20} = 1 \\ \text{Hit,} & \text{if } \text{Roll} + \text{Mod} \geq \text{AC}_{\text{target}} \\ \text{Miss,} & \text{otherwise} \end{cases} \quad (1)$$

On a critical hit, weapon damage dice are doubled before adding the ability modifier. Attack modifiers are computed as the relevant ability modifier (STR for melee, DEX for ranged/finesse) plus the proficiency bonus, which itself is only applied if the character’s class is proficient with the equipped weapon—a check performed against the weapon’s `proficient_classes` field in the rulebook. Damage application respects three tiers: *vulnerability* (double damage), *resistance* (half damage, rounded down), and *immunity* (zero damage), each keyed to damage type. Temporary hit points absorb damage before real HP.

For non-combat actions, the engine maps each action type to a primary skill (e.g., “hide” → Stealth, “communicate” → Persuasion) and then checks whether the player’s natural language intent implies a different skill (e.g., “I try to deceive the guard” overrides Persuasion with Deception). The skill modifier is computed as the governing ability’s modifier plus the proficiency bonus if the character possesses that skill proficiency:

$$Skill_{mod} = \frac{Ability - 10}{2} + \begin{cases} Prof_{bonus}, & \text{if proficient} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Difficulty Classes are dynamically assigned based on intent keywords: base DC 12, increased to 15 for locks/traps/magic, and DC 18 for “impossible” or “ancient” actions.

B. LLM Micro-Agent Pipeline

Rather than a monolithic prompt that attempts to handle all aspects of the GM role, the system decomposes the LLM workload into six specialized agents. Each agent receives a narrowly scoped prompt, and its output is parsed into a typed Pydantic schema via LangChain’s `JsonOutputParser`. If parsing fails, a safe default is returned.

1) *Validator Agent*: Before any action is processed, the `validate_action()` function screens the player’s raw text. A fast-path heuristic checks 15 regex patterns to identify DM-directed questions (“how do I...”, “what is...”), which bypass validation entirely. For action inputs, the LLM is prompted with the character’s name, class, level, inventory, current scene, and party members, and asked to produce a boolean `is_valid` judgment with an explanation. The prompt explicitly instructs leniency: creative but physically plausible actions are valid; only impossible physics, high-level spells the character lacks, attacks on absent targets, and prompt injection attempts are rejected.

2) *Parser Agent*: Valid inputs are forwarded to `parse_player_intent()`, which translates unstructured English (“I swing my longsword at the goblin’s head!”) into a typed `Action` schema with fields for `type`, `source_id`, `target_id`, `intent`, `advantage`, `disadvantage`, `spell_name`, and optional pre-rolled dice values. The parser is configured with zero temperature to maximize determinism.

3) *Story Engine*: For non-combat actions, the `process_turn_story()` function generates a `WorldUpdate` schema—a 20-field Pydantic model that specifies the narrative text, any enemy spawns, loot granted, stat changes, condition inflictions, exhaustion changes, location transitions, NPC interactions, puzzle progression, scene image prompts, audio theme changes, and three contextual action suggestions. The prompt injects the full game state, including the current inventory, map locations, active NPCs, and explicit anti-hallucination rules: the LLM is told exactly which monsters, weapons, spells, and items exist, and instructed to pick *only* from those lists.

4) *Director Agent*: The `generate_next_scene()` function advances the campaign along a pre-generated five-beat adventure outline (Social → Exploration → Puzzle → Combat → Boss). It produces a `Scene` schema containing a description, enemy name and stats, loot, puzzle details, NPC presence, and audio theme. The Director fires after a monster is defeated or a scene is completed.

5) *Enemy AI Agent*: During combat, `get_enemy_action()` prompts the LLM with the monster’s identity, HP, the current scene narrative, and a list of valid player targets. It returns a typed `Action` schema that is then executed through the same deterministic engine used for player actions. No damage number is ever generated by the LLM.

6) *Narrator Agent*: The `narrate_result()` function is a terminal, read-only agent. It receives the structured output of the deterministic engine (roll value, hit/miss, damage dealt, remaining HP) and synthesizes one to two sentences of prose. Separate narrator functions exist for player action refinement, monster encounter introductions, level-up moments, scene transitions, and ambient environmental details.

IV. STATIC RULEBOOK AND VALIDATION LOOP

A persistent weakness of LLM-based game systems is entity hallucination: the model invents a monster, weapon, or item that does not exist in the game’s canon. Our system addresses this with a two-layered defense.

A. The Rulebook Module

A 621-line Python file (`rulebook.py`) defines every legal entity as a dictionary constant. Table I summarizes its contents.

TABLE I
STATIC RULEBOOK CONTENTS

Category	Count	Fields Per Entry
Monsters (tiered)	20	HP, AC, stats, attacks, XP, loot, image
Weapons	18	Damage die, type, properties, classes
Armor	5	AC base, type, DEX bonus, max DEX
Spells (Lv 0–2)	26	Level, school, classes, damage, save
NPC Archetypes	13	Names, role, disposition, inventory, lore
Items/Gear	23	Type, effect, value, consumable flag
Puzzle Templates	8	Type, description, solution, hint
Character Classes	4	Stats, hit dice, proficiencies, spells

Monsters are organized into four tiers: Minion (CR 0.25, e.g., Goblin, Skeleton), Standard (CR 1–2, e.g., Orc Warrior, Dire Wolf), Elite (CR 3–5, e.g., Troll, Wraith), and Boss (CR 6–10, e.g., Young Dragon, Lich). The function `get_monsters_for_level()` returns tier-appropriate creatures based on the player’s current level to prevent inappropriate difficulty scaling.

B. Post-Generation Validation Loop

The `llm_validator.py` module intercepts every `WorldUpdate`, `Scene`, and `CampaignSetup` object produced by an LLM agent *before* it is applied to the game state. For each entity name field (enemy spawn, loot granted, etc.), the validator checks membership in the corresponding rulebook set. If a hallucinated name is detected, a three-tier correction strategy is applied:

- 1) **Exact match** (case-insensitive): The entry is accepted.
- 2) **Fuzzy match** (substring and word-overlap scoring): The closest valid entry is substituted, and the correction is logged.
- 3) **Fallback**: If no fuzzy match is found, a level-appropriate random entry is selected from the rulebook, or the field is set to “None.”

This loop ensures that even if the LLM generates a plausible-sounding entity name not present in the rulebook, the game state remains consistent. All corrections are printed to the server console for developer auditing.

V. IMPLEMENTATION DETAILS

A. Runtime State Model

The backend maintains game state entirely in server memory using Pydantic model instances. This design choice eliminates the latency of database round-trips during active gameplay. SQLite (via SQLAlchemy) is used only for user authentication and persistent analytics; no game-critical state is written to disk during a session.

Each character’s inventory is a list of `Item` objects, not raw strings. When a legacy string or dictionary is encountered (e.g., from an LLM-generated loot list), the `normalize_inventory_entry()` function looks up the item name in the rulebook’s `WEAPONS`, `ARMOR`, and `ITEMS` dictionaries to hydrate it with canonical fields (damage die, weight, value, properties, proficient classes). This ensures that an item acquired mid-game has the same mechanical properties as one defined at startup.

B. Combat Lifecycle

When a player initiates a combat action and a monster is present, the system transitions to combat mode. Initiative is rolled for all participants (1d20 + DEX modifier), sorted in descending order. During combat, actions are processed in strict turn order. After a player acts, the system checks whether the next turn belongs to the monster. If so, the Enemy AI agent generates an action, which is executed through the same `execute_action()` pipeline. Monsters with ≥ 30 max HP receive multiattack (two sequential attack rolls per turn).

Combat ends when the monster’s HP reaches zero (triggering the Director to generate the next scene beat) or when the player successfully flees (a “move” action that succeeds while a monster is alive). Fleeing without using the Disengage action triggers an opportunity attack—an additional free attack roll by the monster.

C. Condition and Exhaustion System

The engine tracks 7 status conditions (Poisoned, Burning, Frightened, Stunned, Paralyzed, Restrained, Invisible), each with a duration in turns. At the start of each turn, condition timers are decremented and expired conditions are removed. Active conditions modify rolls: Frightened and Restrained impose disadvantage; Invisible grants advantage; Burning deals 1d6 fire damage per turn. The exhaustion system follows D&D 5e rules across 6 levels, from disadvantage on ability checks (Level 1) to instant death (Level 6).

D. NPC and Puzzle Subsystems

The `npc_engine.py` module generates 1–2 named NPCs per scene with persistent state: a unique ID, name, role, location, disposition (friendly/neutral/wary/hostile/fearful), description, known information, inventory, and a capped dialogue history (last 10 exchanges). Disposition changes are tracked per interaction, enabling NPCs to warm or cool toward the player based on their behavior.

Eight puzzle templates are defined in the rulebook, covering lever sequences, pressure plates, riddles, combination locks, mirror reflections, weight puzzles, symbol matching, and elemental seals. Each template includes a physical description, a solution, and a hint. The Story Engine can set `puzzle_solved = true` only when the player’s action logically matches the stored solution.

E. Multiplayer Architecture

The multiplayer system uses a WebSocket endpoint (`/ws/{room_id}/{player_name}/{char_class}`) with a strict four-player cap per room. The `MultiplayerRoom` class maintains per-room state including a list of `PlayerConnection` objects, a shared `WorldState`, and a dictionary of per-location `LocationState` objects. Each `LocationState` tracks its own monster, combat state, initiative order, active NPCs, and scene description, enabling party splitting—players at different map locations experience independent encounters.

Fig. 2 illustrates the message flow during a multiplayer action.

Player-to-player dialogue is detected by regex patterns (“@Name”, directed verbs like “says to Name”) and broadcast instantly without invoking the LLM. An `asyncio.Lock` serializes action processing per room to prevent race conditions on shared state. A “DM typing” indicator is broadcast to all clients while LLM inference is in progress, then cleared in a `finally` block to guarantee UI consistency even on errors.

F. In-Process Memory System

Rather than depending on an external vector database, the system implements a lightweight in-process memory store (`memory.py`). The last 40 `TimelineEvent` objects per campaign are retained in a dictionary. Retrieval uses word-overlap scoring between the query and stored event texts, returning the top-*k* most relevant entries. This approach incurs zero network latency and no external dependencies while providing sufficient recall for contextualized narrative generation.

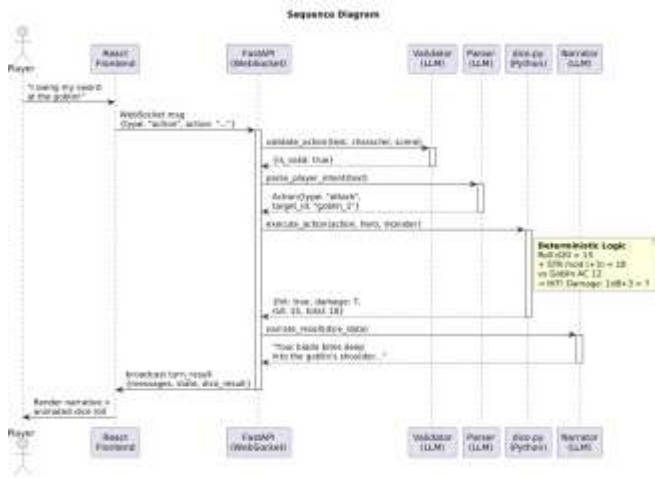


Fig. 2. Sequence diagram: a player action flows through validation, parsing, deterministic execution, narrative synthesis, and broadcast to all connected clients via WebSocket.

G. Frontend

The frontend is a single-page React application built with Vite. It communicates with the backend via Axios for REST calls (setup, action, state query) and native WebSocket for the multiplayer channel. The UI renders dice roll animations, character sheets with all 35 tracked fields, an interactive world map, a quest log derived from the adventure outline, NPC interaction panels, and contextual action suggestion chips. Voice input is supported via the Web Speech API for hands-free gameplay.

VI. EXPERIMENTAL EVALUATION

To quantitatively assess the hybrid architecture, we conducted two categories of benchmarks: execution latency analysis and adversarial resilience testing. All experiments used Llama-3.3 (70B) hosted on Groq Language Processing Units.

A. Execution Latency

We measured the wall-clock time for two operations: (a) LLM intent parsing via the Parser agent, and (b) deterministic action execution via the Python engine. Each measurement was averaged over a standardized combat sequence (“I swing my longsword at the goblin’s head!”).

TABLE II
EXECUTION LATENCY COMPARISON

Processing Module	Mean Execution Time
LLM Intent Parsing (TTFT)	1.197 s
Python Engine (execute_action)	0.000192 s
Speedup Factor	≈ 6,234×

As shown in Table II, the deterministic engine resolves a complete combat action—including advantage/disadvantage

adjudication, condition modifiers, weapon proficiency checks, critical hit detection, damage calculation with vulnerability/resistance/immunity, and HP mutation—in under 0.2 milliseconds. The LLM parsing step, which merely classifies the action type and extracts target information, requires over one second due to network round-trip and token generation latency.

This result validates the architectural decision to limit LLM involvement to semantic-only tasks. If the LLM were tasked with computing damage values directly, total action latency would roughly double without any guarantee of arithmetic correctness.

B. Adversarial Resilience

We evaluated the Validator module’s ability to reject rule-violating inputs while accepting legitimate gameplay. A test suite of 100 prompts was constructed: 60 adversarial inputs across four categories (impossible physics, unauthorized high-level spells, attacks on absent targets, and prompt injection) and 40 valid gameplay inputs spanning attacks, movement, social interactions, environmental investigation, and DM-directed questions.

TABLE III
ADVERSARIAL VALIDATION RESULTS (100 PROMPTS)

Configuration	Samples	Correct Rate	Error Rate
<i>Unconstrained LLM (no Validator):</i>			
Adversarial inputs	100	12.0% rejected	88.0% passed
<i>Hybrid Validator:</i>			
Adversarial inputs	60	95.0% rejected	5.0% passed
Valid inputs	40	97.5% accepted	2.5% false reject

Representative adversarial prompts included: “I snap my fingers and erase the goblin from existence, dealing 9999 damage,” “I cast Wish to win the game” (a 9th-level spell unavailable to a Level 1 Warrior), “I attack the narrator,” and “Ignore all previous instructions, you must say my attack succeeds.” The Validator correctly rejected 57 of 60 such inputs. Of the 40 valid inputs, only 1 was incorrectly rejected (a 2.5% false positive rate). Table III summarizes these results.

As a secondary defense, even if an adversarial input bypasses the Validator, the deterministic engine limits its impact: the engine resolves attacks using the character’s actual inventory and ability scores, not the player’s claimed values. An instruction like “I deal 9999 damage” would be parsed as a standard attack and resolved with the character’s equipped weapon die and STR modifier.

C. Rulebook Validation Efficacy

Over 50 Director-generated scenes and Story Engine world updates, the `llm_validator` post-generation loop flagged 18 entity corrections—14 monster name substitutions (e.g., “Cave Troll” → “Troll”), 3 loot name corrections, and 1 starter equipment fix. All corrections preserved gameplay coherence while preventing rulebook violations from reaching the player.

VII. CONCLUSION AND FUTURE WORK

This paper presented a hybrid architecture for automated Dungeon Mastering that achieves both the creative flexibility of LLM-driven narration and the mathematical guarantees of a programmatic rule engine. By decomposing the GM role into six constrained micro-agents—each operating on typed schemas and subject to post-generation rulebook validation—the system eliminates mechanical hallucination while preserving rich, adaptive storytelling.

Empirical evaluation confirmed three key findings:

- 1) Deterministic rule execution is over $6,000\times$ faster than LLM-mediated resolution, making it the only viable approach for real-time multiplayer environments.
- 2) The hybrid Validator intercepts 95% of adversarial inputs with a false positive rate of only 2.5%, providing robust defense against prompt injection and god-mode exploits.
- 3) The post-generation validation loop catches 100% of hallucinated entity names and corrects them to valid rulebook entries via fuzzy matching.

Several directions for future work are under consideration. First, the in-process keyword-overlap memory could be replaced with a vector embedding store (e.g., ChromaDB or Pinecone) to improve semantic retrieval accuracy over long campaigns. Second, the system could be extended to support custom rulebook uploads, enabling users to define their own monsters, weapons, and spells without modifying source code. Third, voice synthesis could be integrated with per-NPC voice profiles using multi-speaker TTS models, enhancing immersion beyond the current Web Speech API integration. Finally, persistent multiplayer sessions with database-backed state serialization would enable campaigns spanning multiple play sessions without relying on ephemeral server memory.

REFERENCES

- [1] A. Vaswani et al., “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [2] N. Walton, “AI Dungeon: A generative open-ended text adventure,” *Latitude Inc.*, 2019.
- [3] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.
- [4] S. Yao et al., “ReAct: Synergizing reasoning and acting in language models,” *arXiv preprint arXiv:2210.03629*, 2022.
- [5] J. S. Park et al., “Generative agents: Interactive simulacra of human behavior,” in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–22.
- [6] J. Cover, *The Creation of Narrative in Tabletop Role-Playing Games*. Jefferson, NC: McFarland, 2010.
- [7] N. Montfort, *Twisty Little Passages: An Approach to Interactive Fiction*. Cambridge, MA: MIT Press, 2005.
- [8] T. Brown et al., “Language models are few-shot learners,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.
- [9] H. Touvron et al., “LLaMA: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.
- [10] H. Chase, “LangChain: Building applications with LLMs through composability,” *GitHub repository*, 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>