ChatGPT

Refactored Adaptive Story & NPC System Design

We reorganize the narrative engine to ensure **fresh, non-repetitive responses** and correct state updates. We separate fixed instructions (system prompt), dynamic game state, and player actions clearly in each LLM request. Each prompt is built from the **latest story context and flags**, and long histories are **truncated or summarized** to fit token limits. We enforce JSON output format via the system prompt to parse responses reliably.

Key improvements: - Dynamic prompt construction with current story, flags, and NPC states.
- Short-term vs long-term memory: keep only recent segments (2–3) in prompt; summarize older events [1] [2].
- System messages include role instructions and JSON schema hints (e.g. “Return ONLY valid JSON”). Modern APIs even support a “JSON mode” or **Structured Outputs** to guarantee valid JSON [3] [4].

Prompt Construction & Context Management

Each turn, Unity rebuilds the prompt using the **latest game state**. The *system* message contains static guidance (tone, output format, NPC rules, background). The *user* message includes dynamic content: recent story snippets, player metrics, NPC emotional states, and the player’s chosen action. For example:

```
System: You are a fantasy game story generator. Your task is to continue the
narrative and NPC dialogues based on the given context. Always respond in
**JSON** with fields: story_segment, npc_dialogues, and choices.
User: Current story: "Alice’s brother was lost in the forest last night."
Player metrics: kindness=2, aggression=8.
NPC states: Alice: trust=0.3, anger=0.7.
Player chose: "Approach Alice and apologize."
Generate the next story segment, NPC responses, and next choices in valid JSON.
```

- **Latest context only:** We prepend only the last few story sentences to the prompt. Older events are stored in a separate summary string. Periodically (e.g. every few turns), we call the LLM to generate a brief summary of the story so far (2–3 sentences) and replace the old summary [5]. This keeps token usage low while preserving long-term context [1] [2].
- **JSON enforcement:** We explicitly instruct the LLM to output **only valid JSON**. For example, using OpenAI’s JSON mode or Structured Outputs guarantees syntactic validity [3] [4]. Even without specialized APIs, including “Return a JSON object with keys ‘story_segment’, ‘npc_dialogues’, ‘choices’” in the system prompt greatly improves reliability.
- **Example prompt refinement:** In practice, we refine wording based on model behavior. We may include a mini-schema or example in the prompt. For example, saying **“Your response MUST be valid JSON”** or even adding an opening brace (`{`) to the assistant message can nudge the model toward clean JSON [6].

This design ensures that each API call sees the *full up-to-date game state*. We never reuse an old system/user prompt verbatim; instead we reconstruct it with `storyContext` + `memorySummary` + `npcStates` + `playerMetrics` + the chosen action. Libraries like Unity's JSONUtility or external JSON schemas can validate outputs before using them. In sum, **prompt = System instructions + dynamic user context**, with clear separation between immutable rules and changing game data [7].

Game Flow and State Logic

The game flow is refactored to **track state centrally** and update it turn-by-turn. Key elements (story text, player metrics, NPC states, available choices) are stored in a Game State object that persists between prompts. On each player decision: 1. **Record choice and update metrics/flags** (e.g. increment aggression if choice was hostile).
2. **Build and send prompt** with updated state.
3. **Parse LLM JSON response**, append the new story segment to `storyContext`, update NPC states, and load new choices.

For example, after a response JSON like:

```
{
  "story_segment": "Alice scowls and steps back, clearly upset.",
  "npc_dialogues": {"Alice": "How could you say that?"},
  "choices": ["Apologize again", "Get defensive"]
}
```

we append `"Alice scowls..."` to the game's story string, set `npcStates["Alice"]` to reflect her increased anger (e.g. trust–=0.2, anger+=0.2), and present "Apologize again" or "Get defensive" as the next UI buttons. Metrics and flags are adjusted either by explicit LLM cues or by predefined rules on key phrases (e.g. if the response text contains "smiles warmly," we might raise trust [8]).

We ensure **consistent state updates** by always reading and writing to the same GameState instance (or session). For multiplayer or persistence, this state can be tied to a session ID or saved to disk. Choices and flags (boolean or numeric) are sent *each turn* as part of the prompt so the LLM "sees" the consequences. In effect, the LLM is stateless but we handle state externally.

Crucially, we **keep the system prompt separate from game state**. The system prompt holds only fixed instructions (e.g. style rules, JSON format). All changing data (story, flags) go into the user prompt. This separation avoids stale content: for example, when a flag changes, only the user prompt content changes, so the system prompt can remain cached or constant. We explicitly clear or regenerate the entire user prompt each turn, never reusing an older user message.

Storage of flags/choices: In Unity, the GameState object holds lists or dictionaries (e.g. `Dictionary<string,bool> flags; List<string> currentChoices;`). After parsing the LLM JSON, we update those variables in code. If using a backend (Flask), we might store state in memory (e.g. Python dict) keyed by session, or in a simple database. In [Song et al. 2025], dialogue logs (speaker, content, favorability etc.) are saved externally so the system can retrieve the latest 6 turns each time [9] [10]. We

adopt a similar approach: store the last N turns of dialogue (player + NPC lines) and pull them into the prompt for context.

Improved System Architecture

The refactored architecture is modular and linear. At a high level:

- **Unity Client (Front-End):** Captures player input (choice selection), displays story and NPC dialogues, maintains a local GameState. On each choice, it builds a JSON payload with current state and sends it to the backend (or directly to LM Studio).
- **Backend Server (optional):** A Flask (Python) or Node.js proxy that receives Unity's request, calls the local LLM API, and returns JSON. It holds the master GameState (story, flags, NPC states) per session. This layer handles prompt assembly, API calls, and response parsing. If the Unity project can call LM Studio directly, the backend can be a thin proxy to keep Unity code clean.
- **Language Model (LLM):** Hosted locally (LM Studio) or via API. The backend sends a `/v1/chat/completions` request with the system+user messages and JSON schema, and receives a JSON string output.
- **Data Flow:** Unity → Backend (state + choice) → LLM prompt → LLM JSON response → Backend parses JSON → Unity updates UI/state.

File structure example: Unity project contains scripts like `GameState.cs`, `ChatManager.cs`, `NPCState.cs`, and UI prefabs for the dialogue panel and choice buttons. The server project (if used) includes `app.py` or `server.js`, with endpoints `/start` and `/choice`. All code and assets are organized in folders (e.g. `Scripts/` in Unity, `server/` for the API).

Data flow sequence:
1. **/start call**: Unity requests initial story; backend returns first segment, NPC initial states, first choices in JSON. Unity initializes GameState.
2. **Player selects a choice:** Unity calls `/choice` with choice index.
3. **Backend updates state:** Append that choice to state, adjust flags (if scripted), and build LLM prompt with `systemPrompt` + dynamic context.
4. **LLM generation:** The backend calls `POST /v1/chat/completions` (to LM Studio) with `{system: systemPrompt, user: userPrompt}`.
5. **Parse response:** Backend JSON-parses the LLM output (e.g. `parsed = JSON.parse(response)`), updates `storyContext`, NPC states, and prepares new choices.
6. **Return to Unity:** The backend sends a JSON reply `{story_segment, npc_dialogues, choices}`. Unity's ChatManager reads this, updates the displayed story text and NPC dialogue lines, updates GameState (adding `story_segment` etc.), and shows the new choice buttons.

This clear pipeline (Unity → Backend → LLM → Backend → Unity) prevents stale data: each response is built from the **current** GameState.

Example Code Snippets

Unity (C#): Below are examples of key parts. A full production code would include error handling and UI updates, but here is the core logic.

```
// Holds game state (can be a singleton or component)
public class GameState : MonoBehaviour {
    public string storyContext = "";
    public float playerKindness = 5f;
    public float playerAggression = 0f;
    public Dictionary<string,NPCState> npcStates = new
Dictionary<string,NPCState>();
    public List<string> currentChoices = new List<string>();
    void Awake() {
        // Initialize NPCs
        npcStates["Alice"] = new NPCState(0.5f, 0f);
        // ... add other NPCs
    }
}
public class NPCState {
    public float trust, anger;
    public NPCState(float trust=0.5f, float anger=0f) { this.trust=trust;
this.anger=anger; }
}
```

```
// Manager handling LLM calls and dialogue flow
public class ChatManager : MonoBehaviour {
    public GameState gameState;
    public Text storyText, aiReplyText;
    public Button[] choiceButtons;

    void Start() {
        // On start, call the backend to initialize story
        StartCoroutine(SendChoice(-1)); // special case for initial call
    }

    // Called when player clicks a choice (set index in UI)
    public void OnChoiceSelected(int choiceIndex) {
        StartCoroutine(SendChoice(choiceIndex));
    }

    IEnumerator SendChoice(int choiceIndex) {
        string playerChoice = (choiceIndex>=0) ?
gameState.currentChoices[choiceIndex] : "";
        // Build prompt parts
        string systemPrompt = "You are a story and NPC dialogue generator. ...
return valid JSON.";
        string userPrompt = $"Current story: {gameState.storyContext}\n" +
                            $"Player metrics:
kindness={gameState.playerKindness}, aggression={gameState.playerAggression}\n"
+
```

```
                            "NPC states:";
        foreach(var kv in gameState.npcStates) {
            userPrompt += $"\n- {kv.Key}: trust={kv.Value.trust},
anger={kv.Value.anger}";
        }
        if(choiceIndex>=0) userPrompt += $"\nPlayer chose: \"{playerChoice}\"";
        userPrompt += "\nGenerate the next story segment, NPC dialogues, and
next choices in JSON format.";

        // Create JSON payload
        var payload = new {
            model = "gpt-4o",
            messages = new [] {
                new { role="system", content=systemPrompt },
                new { role="user", content=userPrompt }
            }
        };
        string jsonPayload = JsonUtility.ToJson(payload);

        // Send request to local LM API (assuming LM Studio at localhost:5000)
        using(var req = new UnityWebRequest("http://localhost:5000/v1/chat/
completions","POST")) {
            byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(jsonPayload);
            req.uploadHandler = new UploadHandlerRaw(bodyRaw);
            req.downloadHandler = new DownloadHandlerBuffer();
            req.SetRequestHeader("Content-Type", "application/json");
            yield return req.SendWebRequest();
            if(req.result == UnityWebRequest.Result.Success) {
                string respText = req.downloadHandler.text;
                // Parse JSON response (using a JSON parser or Unity's
JsonUtility + helper classes)
                LLMResponse resp = JsonUtility.FromJson<LLMResponse>(respText);
                // Update game state
                gameState.storyContext += resp.story_segment;
                foreach(var npc in resp.npc_dialogues) {
                    // Update NPC states based on simple rules (example)
                    if(npc.Key == "Alice" && npc.Value.Contains("grins")) {
                        gameState.npcStates["Alice"].trust += 0.1f;
                    }
                    // ... handle other NPC reactions
                }
                gameState.currentChoices = resp.choices;
                // Update UI
                storyText.text = gameState.storyContext;
                aiReplyText.text =
resp.npc_dialogues["Alice"]; // example: show Alice's line
                for(int i=0;i<choiceButtons.Length;i++) {
                    if(i<resp.choices.Count) {
```

```
                        choiceButtons[i].GetComponentInChildren<Text>().text =
resp.choices[i];
                        choiceButtons[i].gameObject.SetActive(true);
                    } else choiceButtons[i].gameObject.SetActive(false);
                }
                // Reset input if needed
            } else {
                Debug.LogError("LLM request failed: " + req.error);
            }
        }
    }

    [Serializable]
    public class LLMResponse {
        public string story_segment;
        public Dictionary<string,string> npc_dialogues;
        public List<string> choices;
    }
}
```

Notes: In practice, Unity's `JsonUtility` doesn't natively handle `Dictionary<string,string>`. You may use a JSON library (e.g. JSON.NET) or adjust with helper classes. The code above shows intent. We set `temperature=0` or low to reduce random repetition.

Backend (Python Flask): Alternatively, Unity can send JSON to a Flask endpoint. Example structure:

```
from flask import Flask, request, jsonify
import requests, json

app = Flask(__name__)
game_state = {
    "story": "",
    "player_metrics": {"kindness":5, "aggression":0},
    "npc_states": {"Alice": {"trust":0.5, "anger":0.0}},
    "choices": []
}

@app.route('/start', methods=['POST'])
def start():
    # Reset game_state if needed
    # (Build initial prompt similarly and call LLM)
    response = call_llm(game_state, choice=None)
    update_state_from_response(response)
    return jsonify(response)

@app.route('/choice', methods=['POST'])
```

```
def choice():
    data = request.json
    choice = data['choice']
    # Update flags/metrics based on choice here if needed
    response = call_llm(game_state, choice)
    update_state_from_response(response)
    return jsonify(response)

def call_llm(state, choice):
    # Assemble prompts (system + user as above)
    system_prompt = "You are a story and NPC dialogue generator..."
    user_prompt = f"Current story: {state['story']}\nPlayer metrics:
kindness={state['player_metrics']['kindness']},
aggression={state['player_metrics']['aggression']}\nNPC states:"
    for name, st in state['npc_states'].items():
        user_prompt += f"\n- {name}: trust={st['trust']}, anger={st['anger']}"
    if choice:
        user_prompt += f"\nPlayer chose: \"{choice}\""
    user_prompt += "\nGenerate the next story segment, NPC dialogues, and next
choices in JSON format."
    payload = {
        "model": "gpt-4o",
        "messages": [
            {"role":"system", "content": system_prompt},
            {"role":"user", "content": user_prompt}
        ]
    }
    headers = {"Content-Type": "application/json"}
    resp = requests.post("http://localhost:5000/v1/chat/completions",
                         json=payload, headers=headers)
    return resp.json()['choices'][0]['message']['content']

def update_state_from_response(response_json):
    data = json.loads(response_json)
    # Append to story
    game_state['story'] += data['story_segment']
    # Update NPC states (example: if anger increased)
    # Developer might parse data['npc_dialogues'] content or use rules here
    game_state['choices'] = data['choices']
```

This pattern ensures the backend always builds the prompt from `game_state` and returns the updated JSON for Unity.

Example Scenarios and Prompts

Below is an illustrative branching example. Suppose **Alice's friend was not saved**, and Alice is now hostile. GameState might have `storyContext="Alice's brother was lost in the forest."`, and `npcStates["Alice"].trust=0.2`, `anger=0.8`.

Prompt:

```
System: You are a fantasy RPG story and dialogue engine. Output must be valid
JSON with keys: story_segment, npc_dialogues, choices.

User: Current story: "Alice's brother was lost in the forest and could not be
saved."
Player metrics: kindness=3, aggression=7.
NPC states:
- Alice: trust=0.2, anger=0.8.
Player chose: "Approach Alice calmly and explain what happened".
Generate the next story segment, NPC dialogues, and next choices in JSON format.
```

Here, the **context** (failed rescue) and Alice's low trust/high anger are explicitly given. The LLM should respond with a JSON like:

```
{
  "story_segment": "Alice glares at you, fists clenched. She remembers how her
brother's fate was in your hands.",
  "npc_dialogues": {
    "Alice": "You did this! My brother is gone because of you!"
  },
  "choices": ["Defend yourself", "Apologize profusely", "Stay silent"]
}
```

This output reflects the updated context: Alice's reaction is angry and accusing. The available choices now include apology or defense, based on that hostility.

By contrast, if Alice's trust had been high, her response and choices would differ. **Branching** works because each prompt carries forward the *current* variables. If later the player *defends themselves*, the next prompt will include Alice's response and possibly a change in her anger metric. Our code then again updates state (e.g. `anger+=0.2`) and proceeds.

We test such scenarios to confirm no stale repeats: if the same prompt is sent twice, the LLM now sees an updated story string (since we appended the last `story_segment`), so it cannot repeat the previous segment. Also, summarization ensures the model doesn't forget major events even over many turns [1] [10].

Citations

Our approach follows best practices in context management and prompt design. For example, we **summarize older interactions** to handle token limits [1]. Similar game-NPC systems store recent dialogue externally and feed only necessary context into prompts [10] [7]. Enforcing structured JSON output is recommended (OpenAI's JSON mode/Structured Outputs) to reliably parse responses [3] [4]. The Unity LLM integration code reflects patterns shown in [51], where an on-submit handler disables UI and calls `llm.Chat(input, callback)` to process the LLM's reply [11].

Overall, this refactored design and code will produce an adaptive, stateful narrative engine. Each Unity choice triggers an up-to-date prompt, and responses are parsed into game state. This prevents repetition (since prompts change each turn) and ensures **flags and context are always current**. The result is a production-ready framework for LLM-driven Unity storytelling.

---

[1] GenAI — Managing Context History Best Practices | by VerticalServe Blogs | Medium
https://verticalserve.medium.com/genai-managing-context-history-best-practices-a350e57cc25f

[2] [5] [8] Adaptive Story Arc & NPC Relationship Engine with LLM.pdf
file://file-KzXQmC2MbcbgLaA8vKZNhb

[3] [4] Introducing Structured Outputs in the API | OpenAI
https://openai.com/index/introducing-structured-outputs-in-the-api/

[6] Stop begging for JSON - by Charlie Guo
https://www.ignorance.ai/p/stop-begging-for-json

[7] [9] [10] (PDF) LLM-Driven NPCs: Cross-Platform Dialogue System for Games and Social Platforms
https://www.researchgate.net/publication/390990348_LLM-Driven_NPCs_Cross-Platform_Dialogue_System_for_Games_and_Social_Platforms

[11] How to Use LLMs in Unity | TDS Archive
https://medium.com/data-science/how-to-use-llms-in-unity-308c9c0f637c