ChatGPT

GPT-4o Prompts for Dynamic Storytelling in Unity with LM Studio

This guide covers setting up a Unity project for a dynamic narrative game driven by a local LLM (via LM Studio's OpenAI-compatible API). We outline a recommended folder structure and give **Cursor/GPT-4o prompt templates** for each code component (Unity C# scripts and optional Python backend). Every component includes a ready-to-use "System/User" prompt (in code blocks) that can be pasted into the Cursor editor to generate working code. Explanations and best practices are fully cited from up-to-date sources.

File Structure Overview

- **Assets/** – Store all game content under Unity's `Assets` folder [1]. Avoid creating extra root-level directories.
- **Assets/Scenes/** – Unity scene files (e.g. main game scene, menu scene).
- **Assets/Scripts/** – C# scripts (game logic, managers, UI controllers).
- **Assets/Prefabs/** – Prefabricated game objects (dialogue boxes, character objects).
- **Assets/UI/** – UI elements (Canvas, Panels, TextMeshPro objects) and related assets.
- **Assets/Resources/** (optional) – To store JSON files or ScriptableObjects if needed for loading content at runtime.
- **Editor/** – (Optional) Editor scripts or custom folder templates.
- **StreamingAssets/** (Optional) – If you need to include JSON files or other data that must remain unchanged.
- **Backend/** (Optional) – A separate folder (at project root or alongside Assets) for the Python Flask proxy: e.g. `backend/app.py` and `requirements.txt`.

Unity best practices emphasize keeping content under `Assets/` and using consistent naming conventions [1]. For example, use CamelCase and avoid spaces in folder names. A top-level Unity "Scripts" folder is customary; you might further organize by subsystem (e.g. `Assets/Scripts/Dialogue`, `Assets/Scripts/State`, etc.).

Unity LLM API Integration

To call LM Studio's chat API at `http://localhost:<PORT>/v1/chat/completions`, use **UnityWebRequest** in a coroutine. The C# script should POST a JSON body with fields like `model`, `messages`, etc., and parse the JSON response. For example, Song's Unity tutorial uses `UnityWebRequest` with `uploadHandler` and `downloadHandler`, and sets headers for Content-Type and Authorization (here LM Studio requires no API key, but the same structure applies) [2]. After sending, check for errors and use a JSON library (e.g. Newtonsoft) to deserialize the reply.

Example: A C# `MonoBehaviour` can have a method `GetChatResponse(string prompt, Action<string> callback)` that sets up and sends the request. Use `yield return`

`request.SendWebRequest()` and then parse `request.downloadHandler.text`. For instance:

```
UnityWebRequest request = new UnityWebRequest(apiUrl, "POST");
request.uploadHandler = new
UploadHandlerRaw(Encoding.UTF8.GetBytes(jsonString));
request.downloadHandler = new DownloadHandlerBuffer();
request.SetRequestHeader("Content-Type", "application/json");
// (Set Authorization header if needed)
yield return request.SendWebRequest();
if (request.result != UnityWebRequest.Result.Success) {
    Debug.LogError(request.error);
} else {
    string responseText = request.downloadHandler.text;
// Parse JSON (see JSON Parser section below)
    var response = JsonConvert.DeserializeObject<OpenAIResponse>(responseText);
    callback(response.choices[0].message.content.Trim());
}
```

This pattern (using `UnityWebRequest` and `JsonConvert`) is demonstrated in recent Unity-ChatGPT integration examples [2] [3].

Prompt (for Cursor): Feed the following prompt into Cursor (GPT-4o) to generate a Unity C# class for calling the LLM API. The system message sets style, and the user message specifies details.

```
System: You are an experienced Unity C# developer. Write clear, commented code
using best practices.
User: Create a Unity C# script named `LLMChatClient.cs`. This script should use
`UnityWebRequest` to send a chat completion request to `http://localhost:5000/
v1/chat/completions` (LM Studio's API). It should send a JSON payload with
properties `model` and `messages`, where `messages` is an array of `{ "role":
"user", "content": <prompt> }`. The script must be a `MonoBehaviour` with a
public method `IEnumerator GetChatResponse(string prompt, Action<string>
callback)` that does the POST. Include headers for content type (application/
json). After sending the request, check for errors. On success, parse the JSON
response to extract the assistant's reply content and invoke
`callback(responseText)`. Show the full C# code with using directives and data
classes for JSON.
```

Game State Manager in Unity

A *Game State Manager* holds persistent data like story flags and dialogue history. Implement this as a Singleton `MonoBehaviour` (or `ScriptableObject`) so it persists across scenes and can be accessed globally. For example, a Unity tutorial shows using a private static instance and a public property for a game manager singleton, with public bool properties (e.g. `GotKeyCard`) for game flags [4] [5]. You should

similarly include fields for each story flag (bools or ints) and a structure to record dialogue history (e.g. a `List<string>`).

At runtime, when events occur (e.g. player choices), update these flags via the singleton (e.g. `GameState.Instance.someFlag = true`). Other parts of the game can then check these flags to influence dialogue or outcomes. Ensure thread safety if needed, and mark `DontDestroyOnLoad` if using scenes.

Prompt (for Cursor): Generate a Unity C# singleton GameState manager.

```
System: You are an expert Unity developer. Provide concise C# code.
User: Write a Unity C# class named `GameStateManager` using the Singleton
pattern. This `MonoBehaviour` should persist across scenes
(`DontDestroyOnLoad`). It must track an example set of story flags (e.g. public
booleans `HasKey`, `CompanionSaved`, etc.) and a dialogue history list (e.g.
`public List<string> dialogueHistory`). Include methods to add a dialogue line
to history and to reset or query flags. Provide the full C# code.
```

Dialogue UI (TextMeshPro and Unity UI)

The dialogue user interface typically uses Unity UI elements. Create a `Canvas` with a panel for the dialogue box. Inside, use **TextMeshProUGUI** components for the speaker name and dialogue text for crisp rendering [6]. Also include `Button` objects (with TextMeshPro text) for player choices. A common pattern is a `DialogueManager` script with fields like:

```
public TextMeshProUGUI speakerNameText;
public TextMeshProUGUI dialogueText;
public Button[] optionButtons;
public GameObject dialoguePanel;
```

This is shown in recent tutorials [6]. At the start, the dialogue panel can be hidden. To display lines, populate these UI fields: e.g. `speakerNameText.text = currentLine.speakerName;` `dialogueText.text = currentLine.text;`. Enable the relevant option buttons and set their text using `optionButtons[i].GetComponentInChildren<TextMeshProUGUI>().text = choices[i]` [7]. Hide unused buttons to match the number of choices. When a button is clicked, advance the dialogue based on the chosen option.

Prompt (for Cursor): Create a Dialogue UI controller script.

```
System: You are a Unity UI expert. Output well-structured C# code with comments.
User: Write a Unity C# script called `DialogueManager.cs`. It should reference
`TextMeshProUGUI` fields for the speaker name and dialogue text, plus an array
of `Button` for choice options, and a `GameObject dialoguePanel`. In `Start()`,
```

```
hide the panel. Write a public method `StartDialogue(DialogueLine[] lines)` that shows the panel, resets index, and displays the first line. Write a method `DisplayNextLine()` that advances through the `DialogueLine[]`. For each `DialogueLine`, set `speakerNameText.text` and `dialogueText.text`, then handle its choices: if there are choices, activate buttons and set their text (`optionButtons[i].GetComponentInChildren<TextMeshProUGUI>().text = ...`) and their onClick listeners to call a `ChooseOption(int index)` method. Show disabling unused buttons. Provide the full script and the `DialogueLine` and `DialogueOption` data class definitions.
```

Prompt Builder in Unity

The Prompt Builder script formats the game context and history into a single prompt string to send to the LLM. Typically, you concatenate the dialogue history, current game state, and any relevant scene description. In C#, you can use `StringBuilder` for efficiency or simple string interpolation. For example, prepend system instructions (like "You are an NPC" context) and then append each previous line:

```
StringBuilder sb = new StringBuilder();
sb.AppendLine("Game state: ...");  // e.g. flags
foreach (string line in dialogueHistory) {
    sb.AppendLine(line);
}
sb.AppendLine("Current prompt: " + currentPlayerInput);
string finalPrompt = sb.ToString();
```

Include any state flags and recent narrative. Keep it clear and human-readable. In Unity, this builder can live in a static helper class or on the GameStateManager.

Prompt (for Cursor): Generate a C# prompt-building function.

```
System: You are a Unity developer who writes clear utility code.
User: Write a Unity C# method `BuildPrompt` that takes a `List<string> dialogueHistory` and a `Dictionary<string,bool> storyFlags`, and returns a single string. It should start by appending key story flags (e.g. "FlagName: true/false") on separate lines, then all lines from `dialogueHistory`, then a final line like "Next input: ". Use `StringBuilder` and include comments.
```

JSON Parser (LLM Response)

The LLM (LM Studio) will return a JSON response. We assume it contains fields `"narrative"`, `"choices"`, and `"state_updates"` as specified by our prompt. In C#, parse this with `JsonUtility` (Unity's built-in) or a JSON library. For example, define matching C# classes:

```
[System.Serializable]
public class LLMResponse {
    public string narrative;
    public string[] choices;
    public Dictionary<string,bool> state_updates;
}
```

Then use `JsonUtility.FromJson<LLMResponse>(jsonText)` (or `JsonConvert.DeserializeObject<LLMResponse>` if using Newtonsoft) to get the data. Unity's `JsonUtility` requires serializable types (e.g. arrays instead of List). Similar parsing was shown in tutorials [3]. Once parsed, you can update game flags from `state_updates` and use `choices` to populate UI.

Prompt (for Cursor): Create a JSON parsing class.

```
System: Provide complete, concise Unity C# code.
User: Write a Unity C# class named `LLMResponseParser`. It should define data
classes/structs to match a JSON structure with string `narrative`, string array
`choices`, and a dictionary `state_updates` of flag names to booleans. Include a
method `ParseResponse(string jsonText)` that uses
`JsonUtility.FromJson<YourClass>` to parse and return the parsed object. Show
all necessary classes and using directives.
```

Save/Load Manager

To save and load game state (flags and dialogue history), use JSON files. Unity's `JsonUtility` can serialize simple classes to JSON. For example, create a `SaveData` class that holds your flags and history. When saving, do:

```
string savePath = Application.persistentDataPath + "/save.json";
string json = JsonUtility.ToJson(saveData);
File.WriteAllText(savePath, json);
```

When loading, check `File.Exists(savePath)`, then:

```
string json = File.ReadAllText(savePath);
saveData = JsonUtility.FromJson<SaveData>(json);
```

This approach is demonstrated in Unity tutorials [8] [9]. Note that `Application.persistentDataPath` is a safe location for cross-platform saves [10]. Include error checks (e.g. file exists) and possibly allow multiple save slots by different filenames.

Prompt (for Cursor): Generate a save/load system.

```
System: You are a Unity expert in file I/O and serialization.
User: Write a Unity C# script named `SaveLoadManager.cs` that can save and load
game state. Define a `[System.Serializable]` class `GameData` containing example
fields (e.g. an int `level`, a bool `companionAlive`, and a `List<string>
dialogueHistory`). In `SaveLoadManager`, create a method `SaveGame()` that
converts a `GameData` instance to JSON (using `JsonUtility.ToJson`) and writes
it to `Application.persistentDataPath + "/savegame.json"`. Also create
`LoadGame()` that reads this file (if it exists) and uses
`JsonUtility.FromJson<GameData>`. Include `using System.IO;`. Show the full
code.
```

Flask Python Backend (Optional)

If a direct HTTP call from Unity is not desired, you can use a lightweight Flask proxy. The Flask app listens on a port and forwards requests to LM Studio. For example, using Python's `requests` library inside Flask:

```python
from flask import Flask, request, Response
import requests
app = Flask(__name__)

LM_HOST = "http://localhost:5000"  # LM Studio host

@app.route('/api/chat', methods=['POST'])
def proxy():
    # Forward the Unity request to LM Studio
    res = requests.post(LM_HOST + "/v1/chat/completions",
                        json=request.get_json(),
                        headers={"Content-Type": "application/json"})
    return Response(res.content, status=res.status_code,
content_type=res.headers['Content-Type'])
```

This pattern (Flask route forwarding with `requests.request`) is demonstrated in community examples [11]. It essentially replaces the host URL and streams back the response. Make sure to handle CORS if needed.

Prompt (for Cursor): Create a Flask forwarder.

```
System: You are a Python developer specializing in web services.
User: Write a Python Flask app (`app.py`) that defines a POST endpoint `/api/
chat`. This endpoint should read the incoming JSON from Unity and forward it to
`http://localhost:5000/v1/chat/completions` using the `requests` library
(maintaining the JSON payload). The response from LM Studio should be returned
```

```
verbatim to the Unity client. Include error handling for connection failures.
Provide the complete Flask code.
```

Prompt Engineering

To drive narrative branching, craft *system* and *user* prompts carefully. Use a **system prompt** to establish the LLM's role and output format. For example:

> *System:* "You are a narrative engine for an interactive story game. You will output **only** a JSON object with the keys `narrative` (string continuation of the story), `choices` (an array of possible player choices), and `state_updates` (an object mapping game flag names to booleans). Do not output any additional text or explanations."

This style of instruction (asking the model to reply *only* with JSON) is known to improve parseable output [12]. Then, use **user prompts** to describe the current scene and context. For instance:

- *User:* "The player just failed to save their companion in the last chapter. Based on the game flags (`CompanionSaved: false`), continue the story. Output JSON with the next narrative, exactly 3 choices, and any flag changes (like `CompanionAlive: false`)."

- *User:* "Now the player explores the dark forest alone. Generate the next narrative JSON with branching choices."

Always remind the model of the JSON schema. For example: "**Output format:** `{ \"narrative\": ..., \"choices\": [...], \"state_updates\": {...} }`". This enforces a consistent structure. Recent studies recommend reinforcing format in system/user messages to avoid stray text [12].

Example Prompt (Cursor):

```
System: You are an interactive-fiction narrative AI. Respond only with JSON:
keys must be "narrative", "choices", and "state_updates".
User: The current story context is that the player's companion was not saved.
Given game state { "CompanionSaved": false }, write the next part of the story.
Provide a "narrative" string, exactly 3 choice strings in the "choices" array,
and any updates to state flags in "state_updates". Ensure valid JSON output.
```

Performance Advice

- **Warm up the LLM.** On startup, send a dummy request (e.g. a simple prompt) to LM Studio to load the model into memory. This "warm-up" run eliminates one-time loading costs [13]. Subsequent real queries will then respond faster.
- **Manage context size.** Language models have limited context windows (e.g. a few thousand tokens). Include only the most relevant recent dialogue and essential flags in each prompt. If history grows long, summarize older parts to keep prompts concise.

- **Batch and cache where possible.** If using many API calls, reuse an active model session if LM Studio supports it, and avoid re-sending unchanged context.
- **Asynchronous requests.** Perform network calls in Unity coroutines (as above) to prevent blocking the main thread.

By following these guidelines and using the provided prompts, you can generate complete, working code for each component via Cursor/GPT-4o. This modular approach yields a flexible storytelling system: Unity handles UI and game logic, while LM Studio supplies dynamic narrative content.

Sources: Unity's official project organization guide [1], UnityWebRequest examples [2] [3], Unity singletons [4] [5], TextMeshPro dialogue UI guide [6] [7], Unity JSON save/load tutorial [10] [14] [9], Flask proxy example [11], and prompt engineering best practices [12] [13].

---

[1] Best practices for organizing your Unity project | Unity
https://unity.com/how-to/organizing-your-project

[2] [3] Unity and ChatGPT Integration Demo: How to Use OpenAI API | by Hwanuk Song | Medium
https://medium.com/@olgaphila40/unity-chatgpt-integration-demo-ab5bce5b4168

[4] [5] Implementing a game manager using the Singleton pattern | Unity | by Fernando Alcantara Santana | Nerd For Tech | Medium
https://medium.com/nerd-for-tech/implementing-a-game-manager-using-the-singleton-pattern-unity-eb614b9b1a74

[6] [7] Create a Dynamic Dialogue UI with TextMeshPro in Unity - Wayline
https://www.wayline.io/blog/dynamic-dialogue-ui-textmeshpro-unity

[8] [9] [10] [14] Save and Load Games using JSON in Unity | Pondering Pixel
https://ponderingpixel.com/tutorials/unity/save-and-load-games-using-json-in-unity/

[11] python - Proxying to another web service with Flask - Stack Overflow
https://stackoverflow.com/questions/6656363/proxying-to-another-web-service-with-flask

[12] How to Structure JSON Responses in ChatGPT with Function Calling
https://www.freecodecamp.org/news/how-to-get-json-back-from-chatgpt-with-function-calling/

[13] Optimizing latency – Hamel's Blog
https://hamel.dev/notes/llm/inference/inference.html