Procedural Generation of Quests in a Role-Playing Game using Large Language Models

Body of Creative Work

MSc Game Development (Programming)
Jasfiq Rahman – K2365192

Acknowledgements: Dr. Vasilis Argyriou

Table of Contents

Aims

The central appeal of a RPG, as indicated by the genre's nomenclature, lies in its emphasis on the role-playing element. While gameplay mechanics play a significant role in player engagement, the most effective method for immersing players in this aspect is through a compelling narrative. For instance, embodying a character such as Batman (Batman: Arkham Asylum, 2009)[1] involves not only utilizing advanced gadgets but also the overarching mission of safeguarding Gotham. Similarly, the experience of being a Witcher (The Witcher 3: Wild Hunt, 2015)[2] encompasses both the combat mechanics of wielding a sword and the narrative-driven goal of protecting people from monstrous threats. Consequently, one of the foundational components of a robust RPG is its quest design.

For a narrative to be compelling, it often requires substantial effort and meticulous craftsmanship. Current technological capabilities are not yet able to produce content that surpasses the intricacies and depth achieved through human effort. However, not all projects possess the expansive scope of The Witcher or the extensive resources available to CD Projekt Red. For smaller development teams, it may be advantageous to utilize a system capable of generating engaging narrative threads with minimal input. It is preferable to have quests with substantive story elements, even if they lack the refinement of handcrafted narratives, rather than resorting to simplistic tasks such as "Go fetch five non-descript items."

In this project, we aim to introduce a novel solution for Procedural Quest Generation using Large Language Models (LLMs). This system is not designed to replace human involvement in quest design but to enhance it by reducing the effort required to craft engaging narratives. The tool can complement human-crafted main storylines, particularly by generating side quests that are narratively rich but don’t require the same level of complexity or asset development as main quests.

Key Features:

- **Procedural Quest Generation using LLMs**: Introducing a system that leverages LLMs to generate narrative-driven quests with minimal human input, particularly suitable for side-quests.
- **Augmentation, not replacement**: The tool is designed to support human developers by enhancing their ability to create diverse quest narratives efficiently, not to fully replace human storytelling.

- **Adaptability for smaller teams**: The system provides a viable solution for smaller development teams, enabling them to produce engaging narrative content without the extensive resources required by larger AAA titles.
- **Practical limitations considered**: Acknowledging that AI-generated stories are less effective without the proper development pipeline, the tool is intended primarily for supplementary content, like side-quests, rather than intricate main plotlines.

Objectives

1. Design and implement a Procedural Quest Generation (PQG) system using LLMs
Functional Requirements:

- Develop an LLM-based quest generation module capable of creating narrative-rich quests.
- Develop a generation pipeline for Non-Playable Characters (including enemies), their dialogue trees, and quest relevant items.
- Ensure modular integration into existing game engines.

2. Enhance the performance of the LLM by implementing local deployment and exploring different models
Functional Requirements:

- Deploy the model locally to eliminate network dependency.
- Explore multiple locally deployable LLMs (e.g., Llama, Mistral) to balance model complexity and efficiency.

3. Evaluate generated quests for narrative coherence, player engagement, and immersion
Functional Requirements:

- Create metrics for measuring coherence, creativity and engagement.
- Conduct both quantitative and qualitative evaluation.
- Leverage LangSmith to create a robust and easily iterable pipeline for quantitative evaluation.

4. Integrate the PQG system into the game development pipeline as an in-engine tool
Functional Requirements:

- Build an intuitive pipeline for the game development team, supporting drag-and-drop integration with the core Quest Framework.

- Ensure compatibility with major game engines like Unity or Unreal Engine.

5. Optimize workflow for an intuitive and seamless quest deployment process
Functional Requirements:

- Develop an interface for adjusting quest generation parameters like the primary prompt, quest objectives, characters, settings and rewards.
- Develop a user-friendly and decoupled framework for the easy generation and iteration of generated quests, NPCs, NPC dialogues and quest items.

Literature Review

The application of Procedural Content Generation (PCG) in video games is a well-established practice. Prominent examples such as Minecraft (2011)[3], Terraria (2011)[4], and No Man's Sky (2016)[5] demonstrate the centrality of PCG to their core mechanics, creating expansive and varied game worlds. Despite the extensive use of PCG for environmental and gameplay elements, there is a notable scarcity of procedurally generated narrative content, both in practical implementations and academic discourse. This project aims to examine the most exemplary instances of PCG in narrative design to address this gap.

First, we will review the existing literature on this topic, followed by an examination of games that have practically implemented systems for emergent storytelling.

Existing Literature

Lima et all. (2022)[6], propose a method for automatically creating video game quests with branching storylines. The method uses a combination of Genetic Algorithms and Automated Planning to build these quests based on a pre-defined narrative structure. The result is quests that are reported to be similar in quality to those designed by human professionals.

The CONAN system, developed by Breault et al. (2021)[7], utilizes a world state to generate quests. This world state, specified by the designer, comprises locations, items, and non-player characters (NPCs). NPCs possess personality traits that are considered when CONAN generates a quest for an NPC, who then proposes it to the player. This mechanism ensures that NPCs with pacifist tendencies do not request the player to engage in violent actions. The quests are derived from a designer-specified grammar, where literals can be substituted with NPCs, locations, or items.

Prins et al. (2023)[8] present an event-driven simulation of an in-game world that includes elements such as items, locations, characters, and their memories and desires in StoryWorld. Their work aims to address the procedural generation of quests specifically within the RPG genre.

Grey et al. (2011)[9] construct their quest engine around believable social agents. These agents are designed to have emotions, perform actions, perceive actions performed by others, retain memories of these actions, and disseminate information through conversations. The agents'

memories and emotions are intended to provide credible motivations for assigning quests. Furthermore, memories can influence an agent's emotions, creating lasting emotional impacts even after the specific memories have faded.

Al-Nassar et al. (2023)[10] propose a method for generating immersive quests through the application of Natural Language Processing (NLP) techniques, leveraging BERT and GPT-2 models. This approach is demonstrated within the context of the case study game QuestVille. The findings suggest that the integration of BERT and GPT-2 holds promise in crafting captivating narrative content.

Ashby et al. (2023)[11] amalgamate the capabilities of LLMs with a Knowledge-Graph based methodology to devise a comprehensive solution for generating quests and corresponding NPC dialogues. Human evaluation corroborates that quests produced through their framework approximate the quality of manually crafted quests in aspects including fluency, coherence, novelty, and creativity.

Rimworld

One of the most expansive examples of Procedurally Generated Narrative is a game called Rimworld (2013)[12]. It is a science fiction colony simulation game characterized by its intelligent AI storyteller. Players begin with three survivors of a shipwreck on a remote planet, tasked with managing various aspects of the colonists' well-being, including their moods, needs, injuries, illnesses, and addictions. As a story generator, RimWorld is designed to co-author narratives that are tragic, twisted, and triumphant, encompassing themes such as imprisoned pirates, desperate colonists, starvation, and survival. The game employs an AI storyteller to control random events, thereby shaping each unique narrative experience.

Fig 1.1: Combat in Rimworld

Strengths:

- **AI-driven storytelling**: The AI adapts to player actions, creating highly personalized, emergent stories.
- **Dynamic narrative flow**: Random events and AI management ensure that no two narratives are the same.

Limitations:

- **Limited narrative depth**: While RimWorld excels at generating emergent short-term stories, it lacks the ability to craft complex, overarching narratives with long-term coherence.

Comparison with our system:
Our LLM-based system aims to surpass RimWorld's short-term emergent narratives by generating more structured, cohesive quests with both short-term and long-term narrative arcs. The use of LLMs enables deeper narrative complexity, while still allowing for dynamic interactions with the world and NPCs.

Middle-earth: Shadow of Mordor

Perhaps the most influential and well-known example of a system for emergent storytelling is found in the action-adventure game, Middle-earth: Shadow of Mordor (2014)[13].

Termed the Nemesis System, the AI storyteller in Middle-earth: Shadow of Mordor employs the game's mechanics in a complex and highly detailed manner to generate diverse narratives. This system allows players to determine their own paths and outcomes, while simultaneously crafting smaller stories within the overarching narrative.

The game is based on J. R. R. Tolkien's seminal work, The Lord of the Rings. The overarching narrative centers on the protagonist, Talion, a human warrior endowed with mystical powers, who battles against the dark forces of Sauron and his army of orcs. The orcs constitute a crucial component of the narrative, which the Nemesis System leverages to generate dynamic and personalized stories.

The Nemesis System orchestrates the appearance and evolution of orcs throughout the gameplay loop by managing their hierarchical structure within the game, akin to the traditional boss lists found in many games. However, unlike static boss hierarchies, the player's interactions and experiences with these NPCs drive their development in diverse and dynamic ways.

For instance, if a player is defeated by a non-descript orc minion, the Nemesis System promotes the orc to the rank of "Captain." This promotion not only grants the orc stat boosts but also a unique title. When the player encounters this orc again, they are likely to be met with taunts. If the player is defeated again, the orc continues to grow stronger, advancing further up the ranks and becoming increasingly difficult to defeat in subsequent encounters.

Mark Brown from Game Maker's Toolkit[14] provides a comprehensive video offering an overview of the system, while Magnusson et al. (2022)[15] contribute a scholarly paper meticulously explaining the operational mechanisms of the system.

Fig 1.2: Orc hierarchy in Middle-earth: Shadow of Mordor

Strengths:

- **Dynamic character development**: Orcs evolve based on player interactions, creating personalized storylines unique to each playthrough.
- **Player-driven narratives**: The system directly responds to player actions, giving players the agency to shape their own stories.

Limitations:

- **Limited to NPC hierarchy**: The Nemesis System focuses primarily on NPC rivalries, limiting its narrative potential to that specific domain.

Comparison with our system:
Our LLM-driven system extends beyond NPC rivalries to generate a broader range of narrative content, including side-quests, character development, and world-building. While the Nemesis System is innovative in NPC progression, our system provides a more comprehensive storytelling solution, generating quests that incorporate dynamic characters, world events, and player actions.

Wildermyth

Wildermyth (2019)[16] is a character-driven, procedurally-generated tactical RPG, serving as a great example for the aspirations of our proposed solution. In contrast to prior instances where emergent storytelling is facilitated through sophisticated systemic design, Wildermyth adopts a distinct approach by employing a system primarily dedicated to narrative generation. Notably, the system prioritizes the development of character arcs for the protagonists, colloquially referred to as "heroes," within the game.

The characters in Wildermyth initially possess rudimentary personality traits and undergo character arcs that unfold organically as the game progresses. Throughout gameplay, events occur based on a predetermined list curated by the developers. These events are algorithmically selected, with careful consideration given to the prevailing circumstances and the available heroes. Subsequently, the personalities of the heroes imbue these events with distinct nuances, adding a layer of individualized expression to the unfolding narrative.

The event selection process adheres to a set of straightforward rules, which involve assigning priorities and weights to each event. Events are not duplicated within a single game session, and their respective weights are subject to adjustment based on their occurrence in the preceding game session.

Subsequently, the selected event is enacted. The event text is crafted with branching narratives, allowing for varied delivery of lines contingent upon the involved actors.
While this system offers significant flexibility in crafting character arcs, the development team encountered a notable challenge in devising an overarching storyline. To address this issue, they opted for a workaround strategy, which involved compiling a curated list of authored main plotlines spanning 3-5 chapters. These plotlines feature designated antagonists and situational contexts designed to furnish the overarching struggle with purpose and coherence.

The developer expounds upon the intricacies of the system at the Procedural Games Conference[17].

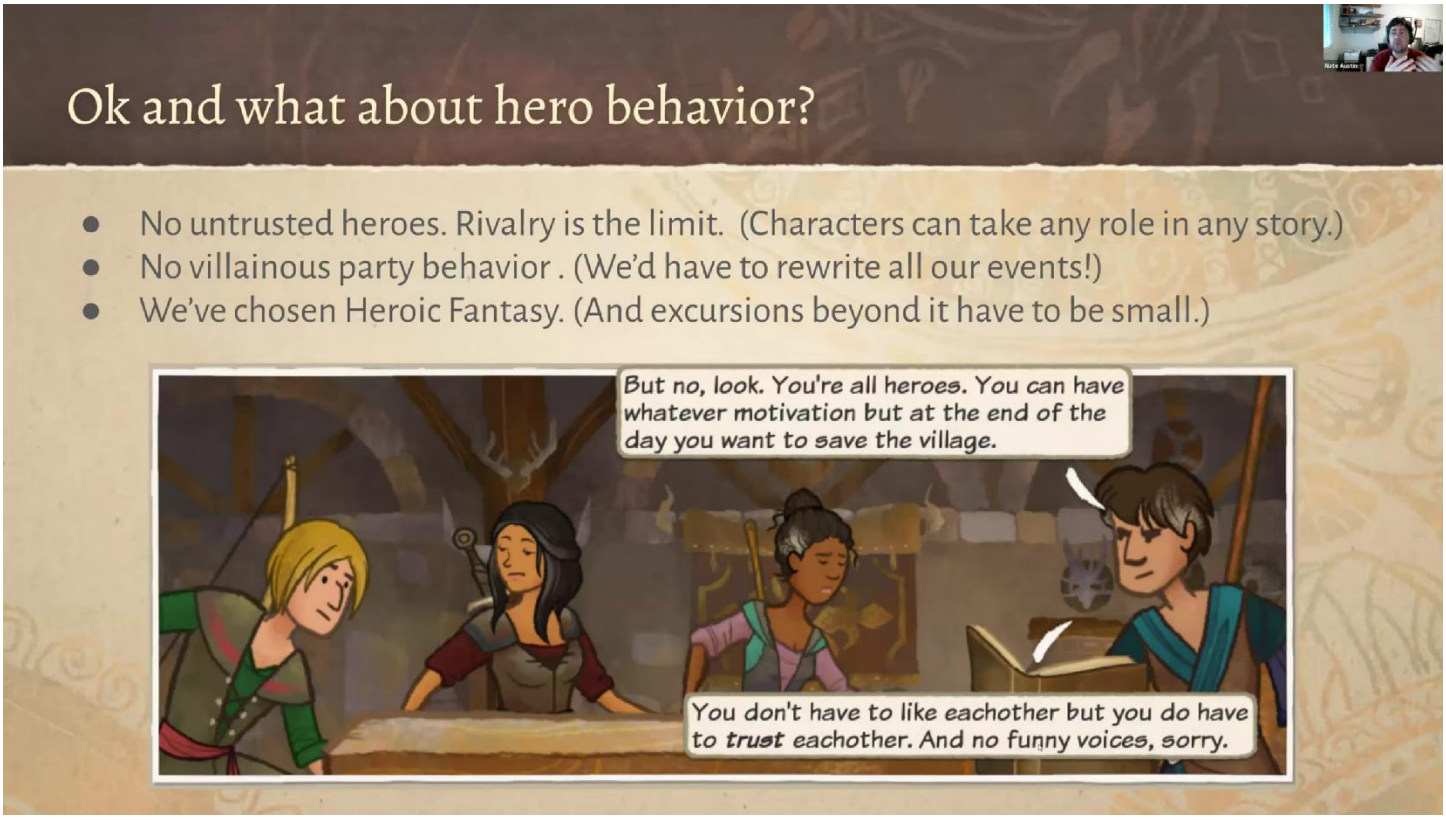

Fig 1.3: Limitations in Hero behaviour in Wildermyth

Strengths:

- **Character-driven narratives**: Heroes develop distinct character arcs, making the storytelling feel personal and unique.
- **Branching events**: Events adapt based on the characters involved, adding layers of variation.

Limitations:

- **Curated plotlines**: While character arcs are procedurally generated, the overall narrative is guided by pre-written main plotlines, limiting the system's generative capabilities.

Comparison with our system:
While Wildermyth excels at character-driven storytelling, our LLM-based system offers more narrative diversity and scope. Our system can generate both character arcs and overarching plotlines dynamically, ensuring that each quest is unique and coherent within the broader game world, without relying on pre-written main stories.

Conclusion

In comparing the procedural narrative systems reviewed, our proposed LLM-based PQG system stands out due to its flexibility, ability to generate both short-term and long-term narratives, and seamless integration into game worlds. Unlike systems dependent on predefined structures or rules, LLMs offer the potential for dynamic, context-aware storytelling that adapts in real-time to player interactions. This opens up new possibilities for emergent gameplay, interactive storytelling, and player-driven narratives across diverse game genres.

Method and Workplan

Technologies and Resources

In this project, the large language models (LLMs) utilized include Llama3.1[18], Gemma2[19], and Mistral-Nemo[20], with the pipeline primarily developed using Llama3.1. These models present several advantages, particularly their open-source nature and capability for local deployment. This selection not only reduces costs but also mitigates potential processing delays associated with network-dependent operations, thereby ensuring uninterrupted and efficient model performance.

The procedural quest generation pipeline is developed in Python. The project employs Ollama[21] and LangChain[22] to create a modular and decoupled pipeline, allowing for seamless integration with various LLMs. For quantitative evaluation, LangSmith[23] is utilized, with GPT-4 Turbo[24] serving as the backend for assessing the pipeline's responses.

The final product is a package designed for integration with Unity[25], written in C#. The tool takes advantage of Unity's Python Scripting[26] to create a wrapper around the Python-based pipeline, which is then made accessible to end users through Unity's Editor scripting[27] framework. The foundational Quest Framework was developed based on content from GameDev.tv's[28] series of courses on creating a Quest System[29] in Unity.

Visual Studio[30] and Visual Studio Code[31] were utilized as the primary IDEs. GitHub[32] was utilized for version control and source code management. ChatGPT[33] was used for the generation of the Evaluation dataset.

Workplan

A brief overview final workplan that was followed during development is outlined below. The sections to follow delve further into the intricacies of the development process.

Phase 1: Initial Research and Planning (1 Iteration)

1. **Quest Design Research and Initial Requirements:**
    - Conducted a review of best practices in quest design.

- Defined the initial criteria for quest generation, including basic narrative structures and various quest objective types.

2. **Technology Exploration and Setup:**
   - Established the development environment using Llama3.1 and Ollama.
   - Conducted exploratory experiments with core functionalities to understand the basic capabilities of the tools.
3. **Evaluation and Iteration Planning:**
   - Refined the agentic framework based on insights from the initial research and exploration.
   - Planned the scope and objectives for the next phase of development.

Phase 2: Iterative Design and Implementation (Multiple Iterations)
Each Iteration:

1. **Design Focused Algorithm Component:**
   - Developed specific components of the procedural quest generation algorithm, such as quest generation and dialogue tree creation.
   - Integrated the generation pipeline with the quest system to produce the Unity package.
2. **Implementation and Testing:**
   - Tested the designed tool within Unity.
   - Assessed the functionality of newly implemented components to ensure they operate as intended within the system.

Phase 3: Evaluation and Validation (1 Iteration)

1. **Metric Identification:**
   - Define appropriate metrics for evaluating the system's performance.
2. **Quantitative Evaluation:**
   - Develop an evaluation pipeline using LangChain and LangSmith, where the LLM (GPT 4 Turbo) serves as the evaluator.
   - Generate a relevant dataset for evaluation.
   - Deploy the evaluation pipeline on the Quest Generation framework using different models, and compare the results.

Phase 1: Initial Research and Planning

The initial phase focused on researching quest design, as it was essential for the system to incorporate best practices in order to generate engaging and enjoyable quests. This research involved reviewing relevant resources, primarily YouTube videos from channels such as *Game Maker's Toolkit*[34], *Design Doc*[35], *Extra Credits*[36], and *Adam Millard – The Architect of Games*[37], among others.

Key insights revealed that multi-part questlines, which build upon previous stages, tend to be more engaging for players compared to standalone, single-objective quests. Following this, a basic development pipeline was established using Llama3.1 and Ollama.

```
import ollama

def generate_quest(template_info, questline_example, objective_info):
    brainstorming_system_prompt = (f"You are a Quest Designer at a
Game studio.\n"
    f"You have been tasked with creating compelling Side-Quests for a
Role-Playing Game.\n"
    f"The game is set in a Fantasy setting.\n"
    f"Create engaging and creative questlines that enhance the
player's experience and provide meaningful content.\n"
    f"You should create multi-part questlines.\n"
    f"Try to compelling narratives that deviate from the norms.\n"

    f"\n###\n"

    f"The questline generated should follow the \"template\" given
below:\n"

    f"{template_info}\n"

    f"Given below is an example. Use it for reference only:\n"

    f"{questline_example}\n"

    f"\n###\n"
```

```
    f"Each quest of the questline should be of a type otlined in the
\"quest_objectives\" below:\n"

    f"{objective_info}\n"

    f"\n###\n"

    f"\nGive a name to the questline as a whole.\n"

    f"\nDescribe each quest in the format given:\n"
    f"Name:\nType:\nGoal:\nDescription:\n")

    response = ollama.chat(model="llama3", messages=[
        {
            "role": "system",
            "content": brainstorming_system_prompt
        }
    ], options={"temperature": 2})

    return response["message"]["content"]
```

Fig 2.1: Initial Quest Generation Agent

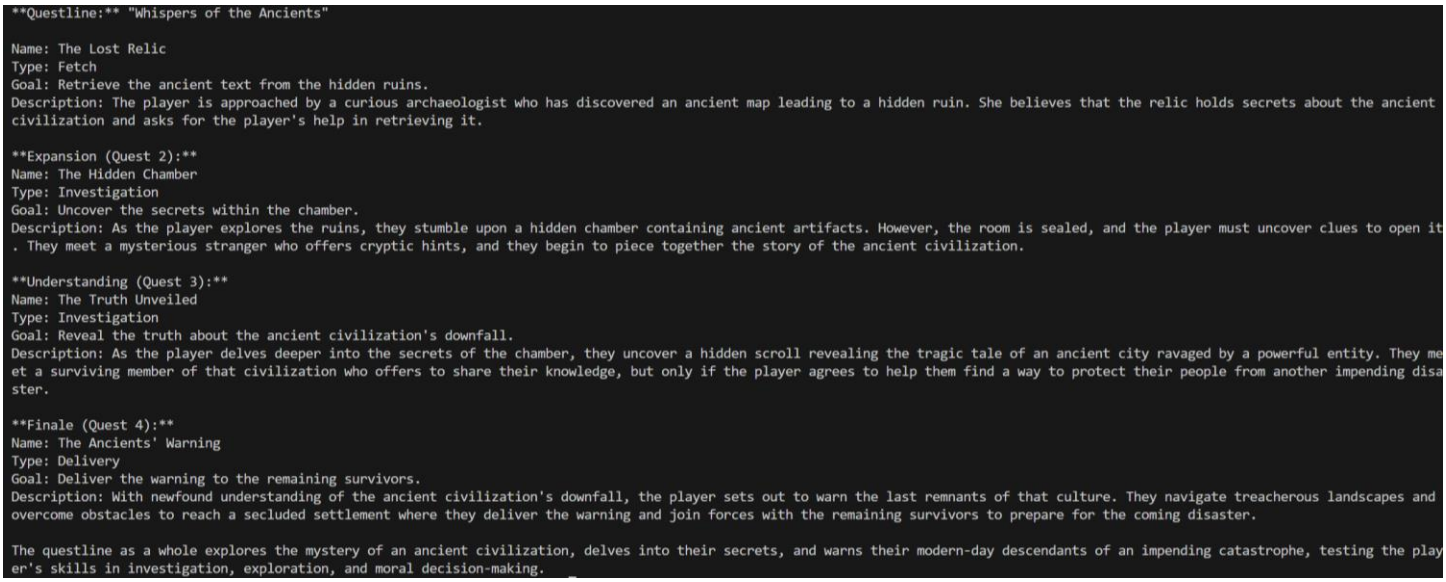

Fig 2.1: Initial Output Example

The Python pipeline was subsequently dubbed Campbell-Quest.

After completing the preliminary assessment of quest design requirements and conducting basic testing of the agentic framework, significant time was dedicated to enhancing our understanding of LangChain through the Udemy course titled "LangChain – Develop LLM Powered Applications with LangChain[38]." However, during a subsequent meeting with the project supervisor, it was determined that it would be more effective to prioritize the integration of the pipeline within Unity in order to begin developing the tool. As a result, further research into LangChain was postponed, and Phase 2 begun.

Phase 2: Iterative Design and Implementation

The first iteration of Phase 2 focused on a straightforward objective: executing a basic print function in Python through the Unity Editor. Once this was successfully implemented, attention shifted to integrating the Campbell-Quest generation pipeline into Unity.

The pipeline was adjusted to prioritize the creation of one-shot quests instead of multipart questlines. This modification aimed to reduce the complexity of the generated quests, thereby facilitating a more efficient integration process within Unity.

```
from .questAgents import quest_brainstormer, quest_refiner,
quest_formatter

def generate_initial_quest(quest_prompt, objective_info,
location_info, character_info):
    initial_generated_quest =
quest_brainstormer.generate_quest(objective_info, quest_prompt,
location_info, character_info)
    return initial_generated_quest

def generate_quest_with_objectives(initial_generated_quest,
location_info, character_info):
    quest_with_objectives =
quest_refiner.define_quest_objectives(initial_generated_quest,
location_info, character_info)
```

```
    return quest_with_objectives

def generate_quest_reward(initial_generated_quest, rewards):
    quest_reward =
quest_refiner.define_quest_reward(initial_generated_quest, rewards)
    return quest_reward

def get_formatted_quest(quest, schema):
    return quest_formatter.format_quest(quest, schema)

def get_formatted_quest_with_rewards(quest, reward, schema):
    return quest_formatter.format_quest_with_rewards(quest, reward,
schema)
```

Fig 2.3: questGenerator.py

This code defines a modular pipeline for generating and refining quests using agents from the questAgents module. Key components include:

1. **generate_initial_quest()**: Utilizes the quest_brainstormer to create an initial quest draft based on provided prompt, objective, location, and character information.
2. **generate_quest_with_objectives()**: Uses the quest_refiner to enhance the quest by defining objectives.
3. **generate_quest_reward()**: Adds rewards to the quest via the quest_refiner.
4. **get_formatted_quest()** and **get_formatted_quest_with_rewards()**: Employ the quest_formatter to format the quest and rewards based on a predefined schema.

The pipeline enables structured quest generation, refinement, and formatting for use within a game development environment.

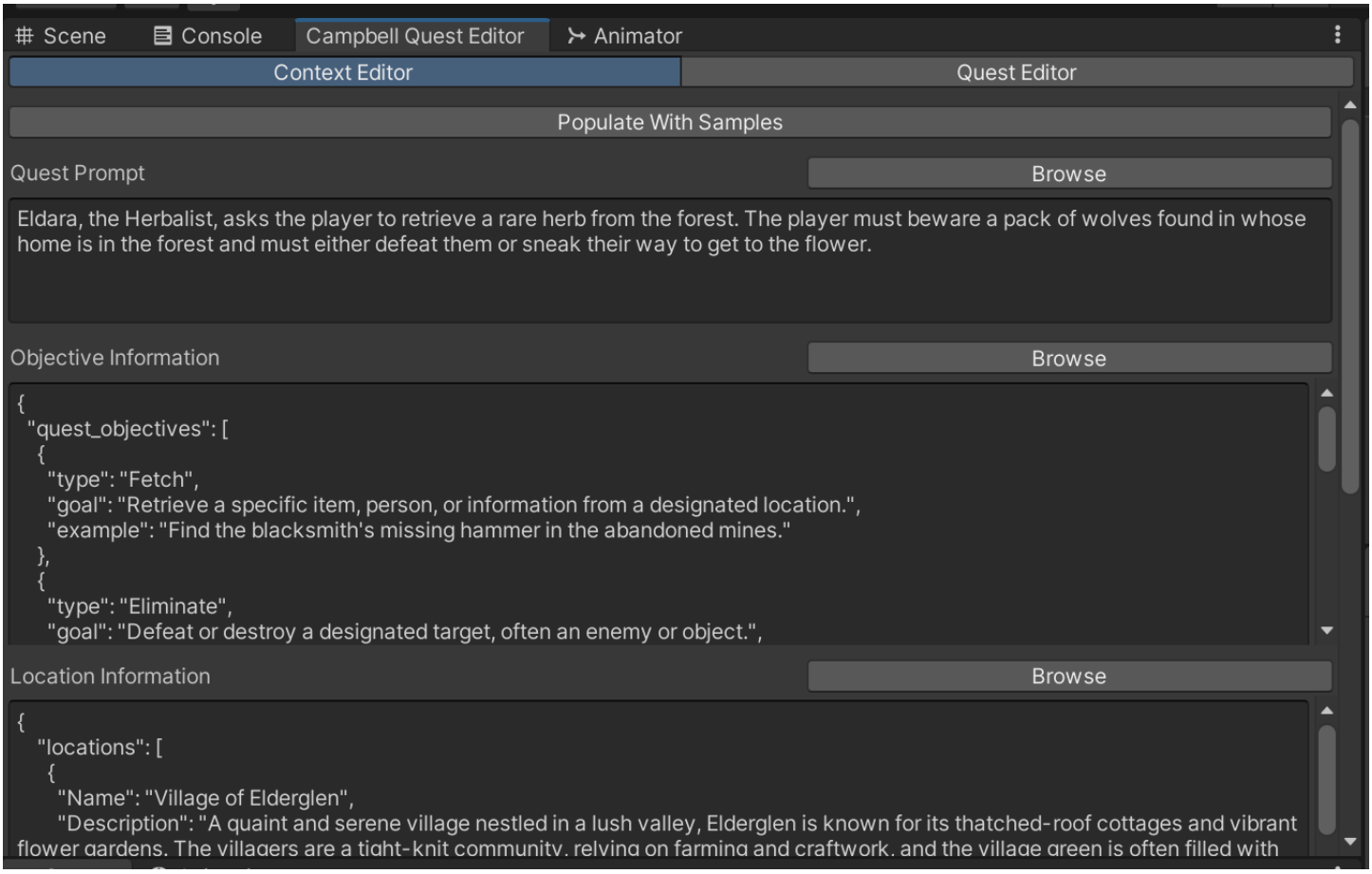

Fig 2.4: Initial Editor Window

A *CampbellEditorWindow* script was developed, enabling users to assign the various parameters necessary for quest generation. These parameters are then passed to the *GenerateQuest()* function, which executed a Python script interfacing with the quest generation pipeline.

```
private void GenerateQuest()
{
    if (GUILayout.Button("Generate Quest"))
    {
     string prompt =
UtilityLibrary.FormatStringForPython(_questPrompt);
     string objectives =
UtilityLibrary.FormatStringForPython(_objectiveInformation);
     string locations =
UtilityLibrary.FormatStringForPython(_locationInformation);
     string characters =
UtilityLibrary.FormatStringForPython(_characterInformation);
```

```
      string rewards =
UtilityLibrary.FormatStringForPython(_rewardInformation);

      string questSchema = UtilityLibrary.LoadSchema("quest");

      string pythonScript = "import UnityEngine;\n" +
      "from campbell_quest import quest_generator\n" +
      "\n" +
      $"prompt = \"{prompt}\"\n" +
      $"schema = \"{questSchema}\"\n" +
      $"objectives = \"{objectives}\"\n" +
      $"locations = \"{locations}\"\n" +
      $"characters = \"{characters}\"\n" +
      $"rewards = \"{rewards}\"\n" +
      "quest = quest_generator.generate_quest(prompt, schema,
objectives, locations, characters, rewards)\n" +
      "print(quest)\n";

      using StringWriter stringWriter = new StringWriter();
      using (Py.GIL())
      {
       dynamic sys = Py.Import("sys");
       sys.stdout = new CampbellTextWriter(stringWriter);
       PythonRunner.RunString(pythonScript);
      }

      _generatedQuest = stringWriter.ToString();
     }
}
```

Fig 2.5: GenerateQuest()

The subsequent iteration focused on automating the serialization of Quest ScriptableObjects[39] from the generated data strings. The Quest object served as a core data class within the quest system, responsible for storing and managing quest-related information during gameplay.

```
public class QuestGenerator
{
```

```
    [System.Serializable]
    public class QuestData
    {
            public string name;
            public string description;
            public string goal;
            public List<ObjectiveData> objectives;
            public List<RewardData> rewards;
    }

    [System.Serializable]
    public class ObjectiveData
    {
            public string reference;
            public string description;
    }

    [System.Serializable]
    public class RewardData
    {
            public int number;
            public string item;
    }

    public static void CreateQuestFromJson(string jsonString, string
savePath)
    {
            QuestData questData =
JsonConvert.DeserializeObject<QuestData>(jsonString);

            Quest quest = ScriptableObject.CreateInstance<Quest>();

            quest.QuestDescription = questData.description;
            quest.QuestGoal = questData.goal;

            foreach (ObjectiveData objective in questData.objectives)
```

```
            {
                quest.AddObjective(objective.reference,
objective.description);
            }

            foreach (RewardData reward in questData.rewards)
            {
                InventoryItem item =
Resources.Load<InventoryItem>(reward.item);
                if (item != null)
                {
                    quest.AddReward(reward.number, item = item);
                }
                else
                {
                    Debug.LogError($"InventoryItem '{reward.item}' not
found in Resources.");
                }
            }

            if (!Directory.Exists(savePath))
            {
                Directory.CreateDirectory(savePath);
            }

            string path = savePath + "/" + questData.name + ".asset";
            UnityEditor.AssetDatabase.CreateAsset(quest, path);
            UnityEditor.AssetDatabase.SaveAssets();
    }
}
```

Fig 2.6: Initial QuestGenerator.cs

The *QuestGenerator* class automates the creation of quest assets in Unity using JSON input. It defines three key data structures:

1. **QuestData**: Stores quest information, including the name, description, goal, objectives, and rewards.

2. **ObjectiveData**: Defines individual quest objectives with a reference and description.
3. **RewardData**: Specifies rewards with item references and quantities.

The *CreateQuestFromJson* method deserializes the JSON into QuestData, creates a Quest object, and assigns objectives and rewards by loading assets from Unity's Resources[40]. The completed quest is saved as an asset file using Unity's AssetDatabase[41]. This approach streamlines procedural quest generation by integrating data-driven content into Unity's asset management system.

With the large language model generation pipeline and its integration with Unity and the Quest System successfully implemented, the basic functionality of the quest generation tool was completed. This provided a foundation for further system expansion. Consequently, focus shifted to developing a Dialogue Generation pipeline for NPCs, a core component of RPGs and essential for quest delivery to players.

The *Campbell-Quest* package was extended to support the generation of dialogue trees, and the *CampbellEditorWindow* class was modified to include dialogue tree generation, mirroring the functionality of the quest generation pipeline. Additionally, a new *DialogueGenerator* class was created to generate dialogue ScriptableObjects.

Subsequent iterations enhanced the system by incorporating the generation of NPC and item prefabs, crucial for the effective implementation of quests. At the conclusion of each iteration, the tool underwent testing in Unity, with any identified bugs addressed to ensure the system's proper functionality.

Once the core functionality of the pipeline was established, the next step involved refactoring the Unity tool to enhance code organization and decoupling. The *CampbellEditorWindow* was restructured into four distinct windows: *QuestEditorWindow*, *DialogueEditorWindow*, *ItemEditorWindow*, and *NpcEditorWindow*. Each window is responsible for exposing the necessary parameters of the Campbell-Quest LLM pipeline to the user and managing the editor's front-end functionality.

These windows transmit the relevant information to their corresponding processors—*QuestProcessor*, *DialogueProcessor*, *ItemProcessor*, and *NpcProcessor*. The processors interface with the Campbell-Quest package and execute the appropriate Python scripts. If the

generated content meets user approval, it is then forwarded to the *QuestGenerator*, *DialogueGenerator*, *ItemGenerator*, and *NpcGenerator* scripts. These generators handle the decoding of the LLM's responses and the serialization of the appropriate ScriptableObjects or MonoBehaviour[42] prefab assets.

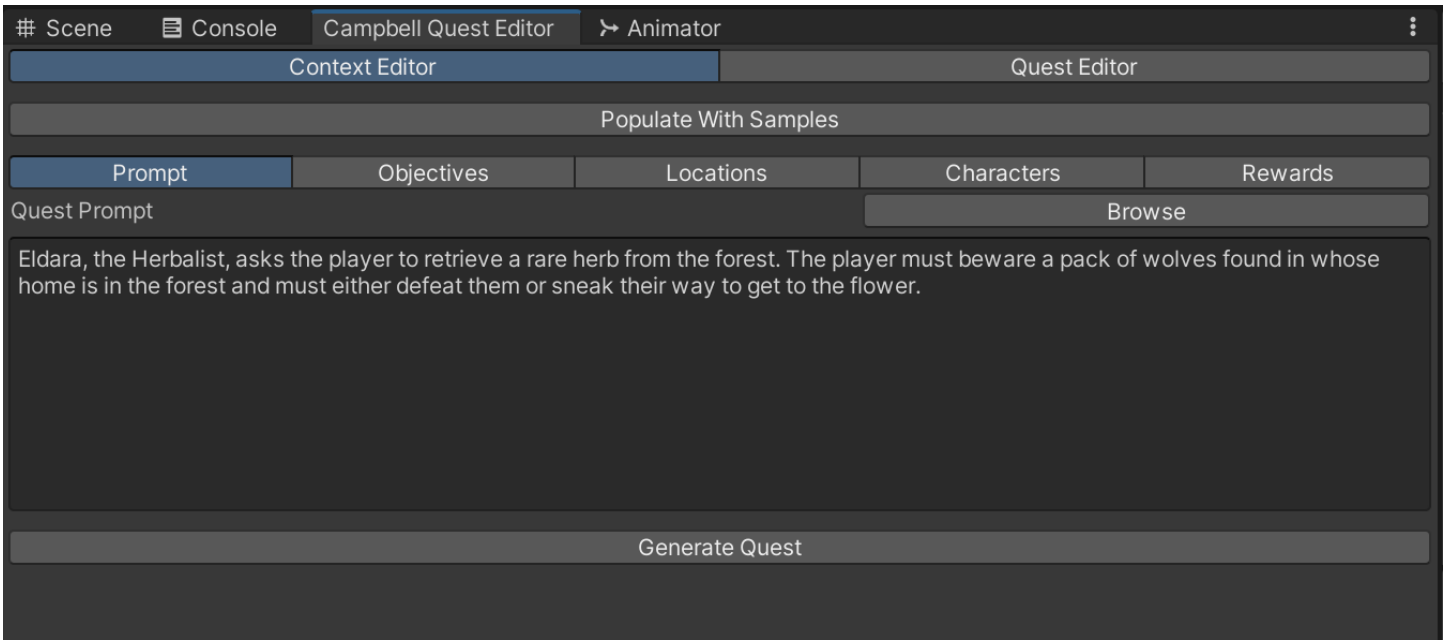

Fig 2.7: Final Quest Editor Window

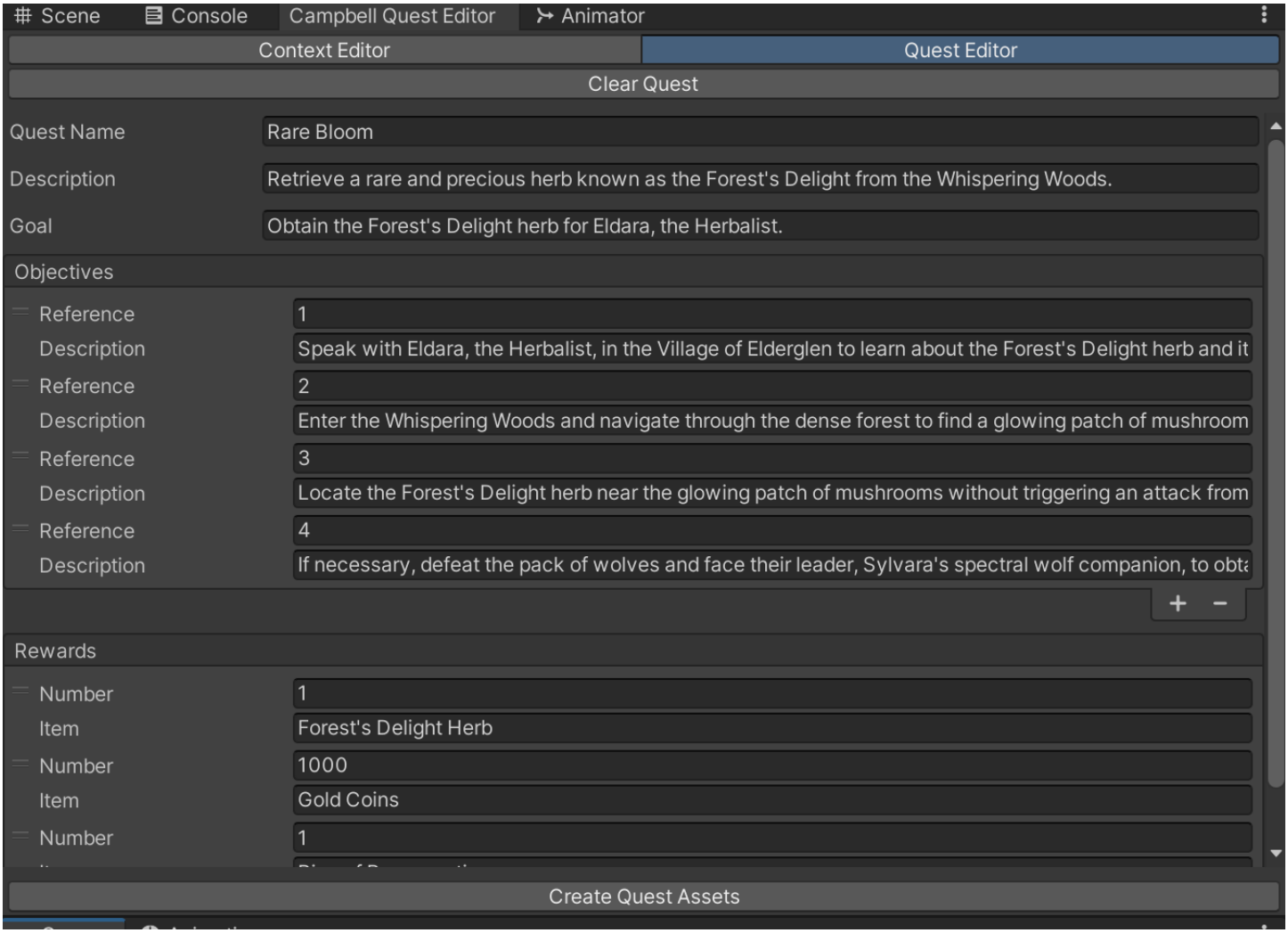

Fig 2.8: Example Generated Quest

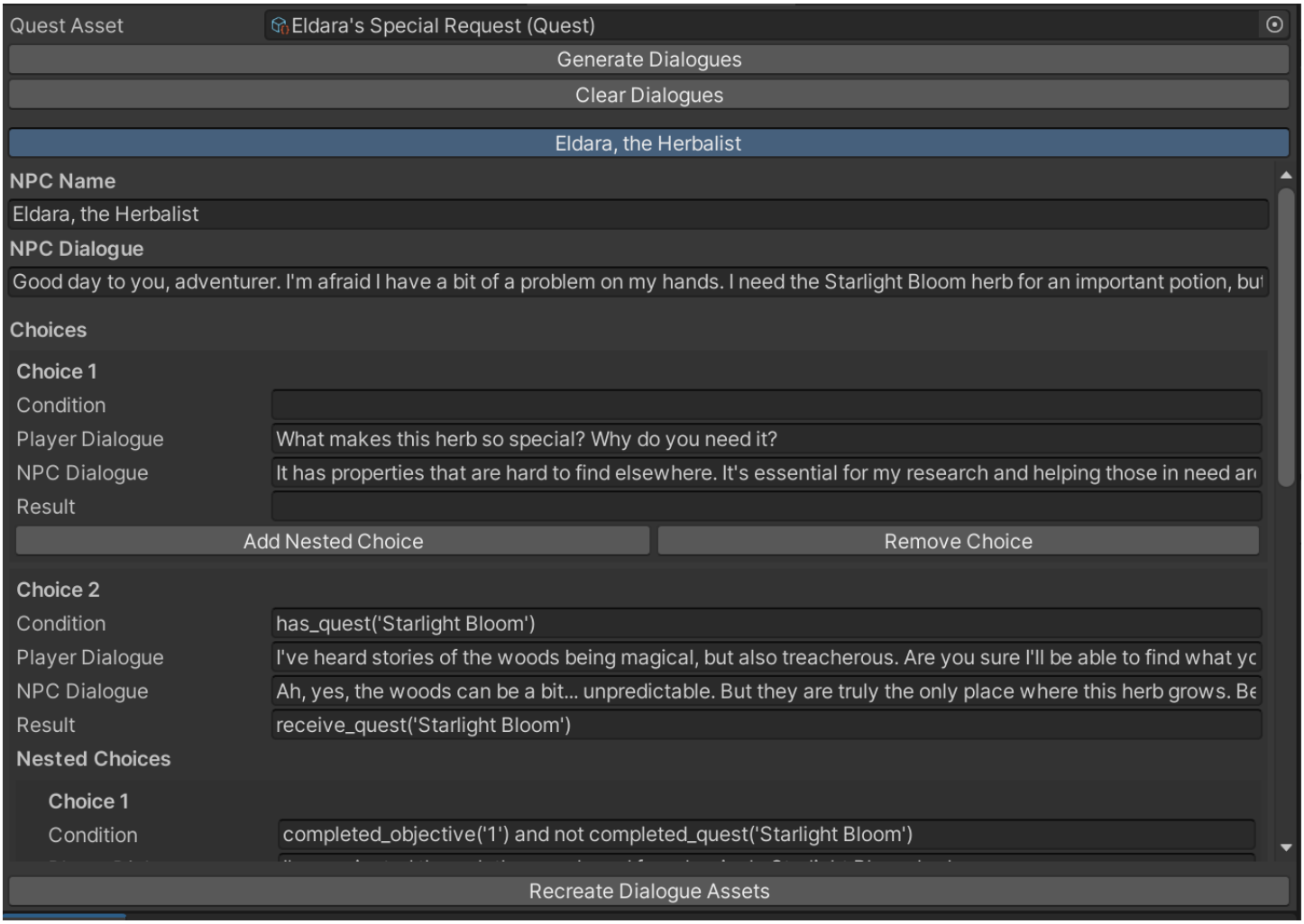

Fig 2.9: Final Dialogue Editor Window and Example Generated Dialogue

Phase 3: Evaluation and Validation

In this phase, a set of rigorous evaluation metrics was established to assess the quality and effectiveness of the generated quests. The metrics include:

- **Conciseness:** This metric evaluates the quest's ability to communicate essential information clearly and succinctly, without unnecessary elaboration. A concise quest effectively conveys objectives and instructions in a direct manner.
- **Coherence:** This criterion assesses the logical structure and flow of the quest, ensuring that it follows a clear and consistent progression. A coherent quest is easy to follow, with all narrative elements logically connected.

- **Relevance:** This metric examines the alignment of the quest with the defined parameters, ensuring it remains focused on the specified objectives, characters, and locations. A relevant quest avoids deviation into unrelated topics.
- **Engagement:** Engagement is measured by the quest's ability to capture and maintain the player's interest throughout. An engaging quest should motivate players to progress by incorporating compelling objectives, characters, and challenges.
- **Creativity:** Creativity assesses the originality and innovation within the quest, highlighting the use of unique ideas, solutions, or narrative elements. A creative quest presents players with novel and imaginative content.
- **Narrative Complexity:** This metric evaluates the depth and richness of the quest's storyline by incorporating multiple elements such as characters, locations, and objectives. A quest with high narrative complexity offers a multi-layered and engaging experience for the player.

Following the identification of the relevant metrics, a pipeline for quantitative evaluation was developed. This pipeline was designed to systematically assess the generated quests against the established metrics, using automated methods to ensure consistency and objectivity in the evaluation process. The pipeline leveraged LangChain and LangSmith to create an LLM-based evaluation system, allowing for a detailed comparison of quest quality across different models and iterations. The generated dataset served as the input for this evaluation, providing a structured approach to measuring the system's performance across the key metrics.

```python
import os
from src.campbell_quest import quest_generator
from dotenv import load_dotenv
import time

from openai import RateLimitError

from langchain_openai import ChatOpenAI
from langsmith.schemas import Example, Run
from langsmith.evaluation import LangChainStringEvaluator, evaluate

load_dotenv()

def load_json(filename):
    try:
        with open(f"{filename}.json", "r") as file:
            info = file.read()
            print(f"{filename}.json read successfully.")
            return info
    except Exception as e:
        print(f"An error occurred: {e}")

############################################################
#####   Quest Generation

def evaluate_quest_generation_llama(inputs: dict) -> dict:
    prompt = inputs["prompt"]

    objectives = load_json("example_objectives")
    locations = inputs["locations"]
    characters = inputs["characters"]

    quest_schema = load_json("quest_schema")
```

```
    initial_generated_quest =
quest_generator.generate_initial_quest(prompt, objectives, locations,
characters)
    quest_with_objectives =
quest_generator.generate_quest_with_objectives(initial_generated_quest
, locations, characters)
    formatted_quest =
quest_generator.get_formatted_quest(quest_with_objectives,
quest_schema)

    return {"quest": formatted_quest}

############################################################
#####   Evaluation Calls

def run_evaluation_llama():
    dataset_name = "ds-campbell-evaluation-50"
    evaluators = [run_clarity_evaluator, run_engagement_evaluator,
run_creativity_evaluator]
    prefix = "llama"

    evaluate(
        evaluate_quest_generation_llama,
        data=dataset_name,
        evaluators=evaluators,
        experiment_prefix=prefix
    )

############################################################
#####   Evaluator Wrappers

def run_clarity_evaluator(root_run: Run, example: Example) -> dict:
    evaluator = get_clarity_evaluator()
    run_evaluator = evaluator.as_run_evaluator()

    max_retries = 12
```

```
    backoff_factor = 2  # Exponential backoff factor
    initial_delay = 10  # Initial delay in seconds

    for attempt in range(max_retries):
        try:
            results = run_evaluator.evaluate_run(root_run, example)
            return results
        except RateLimitError as e:
            if attempt < max_retries - 1:  # Don't delay on the last
attempt
                delay = initial_delay * (backoff_factor ** attempt)
                print(f"RateLimitError encountered. Retrying in
{delay} seconds...")
                time.sleep(delay)
            else:
                print("Max retries reached. Raising the
RateLimitError.")
                raise e  # Re-raise the exception if max retries are
reached

def run_engagement_evaluator(root_run: Run, example: Example) -> dict:
    evaluator = get_engagement_evaluator()
    run_evaluator = evaluator.as_run_evaluator()

    max_retries = 12
    backoff_factor = 2  # Exponential backoff factor
    initial_delay = 10  # Initial delay in seconds

    for attempt in range(max_retries):
        try:
            results = run_evaluator.evaluate_run(root_run, example)
            return results
        except RateLimitError as e:
            if attempt < max_retries - 1:  # Don't delay on the last
attempt
                delay = initial_delay * (backoff_factor ** attempt)
```

```
                print(f"RateLimitError encountered. Retrying in
{delay} seconds...")
                time.sleep(delay)
            else:
                print("Max retries reached. Raising the
RateLimitError.")
                raise e  # Re-raise the exception if max retries are
reached

def run_creativity_evaluator(root_run: Run, example: Example) -> dict:
    evaluator = get_creativity_evaluator()
    run_evaluator = evaluator.as_run_evaluator()

    max_retries = 12
    backoff_factor = 2  # Exponential backoff factor
    initial_delay = 10  # Initial delay in seconds

    for attempt in range(max_retries):
        try:
            results = run_evaluator.evaluate_run(root_run, example)
            return results
        except RateLimitError as e:
            if attempt < max_retries - 1:  # Don't delay on the last
attempt
                delay = initial_delay * (backoff_factor ** attempt)
                print(f"RateLimitError encountered. Retrying in
{delay} seconds...")
                time.sleep(delay)
            else:
                print("Max retries reached. Raising the
RateLimitError.")
                raise e  # Re-raise the exception if max retries are
reached

############################################################
#####   Evaluators Setup
```

```
def get_clarity_evaluator():
    criterion = {
        "conciseness": "Is this response concise, delivering the
necessary information in a clear and straightforward manner without
unnecessary elaboration? It should prioritize brevity while ensuring
that the answer remains complete and informative.",
        "coherence": "Is this response coherent, logically structured,
and easy to follow? The information provided should flow naturally,
with ideas and facts presented in a manner that makes sense as a
whole, ensuring that the user can easily understand the response."
    }

    eval_llm = ChatOpenAI(temperature=0.0, model="gpt-4-turbo")

    evaluator = LangChainStringEvaluator(
        "score_string",
        config={
        "criteria": criterion,
        "llm": eval_llm
        },
        prepare_data = lambda run, example: {
                "prediction": run.outputs["quest"],
                "input": example.inputs
            },
        )

    return evaluator

def get_engagement_evaluator():
    criterion = {
        "relevance": "Is this response relevant, directly addressing
the user's query without deviating into unrelated topics. It should
focus on providing information or solutions that are directly
applicable to the user's needs or context.",
```

```
        "engagement": "Is this response engaging, capturing the user's
interest and maintaining their attention throughout the response. It
should encourage further interaction or exploration."
    }

    eval_llm = ChatOpenAI(temperature=0.0, model="gpt-4-turbo")

    evaluator = LangChainStringEvaluator(
        "score_string",
        config={
        "criteria": criterion,
        "llm": eval_llm
        },
        prepare_data = lambda run, example: {
                "prediction": run.outputs["quest"],
                "input": example.inputs
            }
        )

    return evaluator

def get_creativity_evaluator():
    criterion = {
        "creativity": "Is this response creative, offering unique or
innovative solutions, ideas, or perspectives that demonstrate
originality and imagination. It should go beyond conventional or
expected responses, providing a fresh and interesting take on the
topic.",
        "narrative complexity" : "Is this response narratively
complex, incorporating multiple elements such as characters,
locations, and objectives in a way that creates a rich and engaging
story. It should involve various plot points, twists, and interactions
that enhance the overall narrative experience."
    }

    eval_llm = ChatOpenAI(temperature=0.0, model="gpt-4-turbo")
```

```
    evaluator = LangChainStringEvaluator(
        "score_string",
        config={
        "criteria": criterion,
        "llm": eval_llm
        },
        prepare_data = lambda run, example: {
                "prediction": run.outputs["quest"],
                "input": example.inputs
            }
        )

    return evaluator

if __name__ == "__main__":
    # Get the absolute path of the current script file
    script_path = os.path.abspath(__file__)

    # Extract the directory containing the script file
    script_directory = os.path.dirname(script_path)

    # Change the working directory
    os.chdir(f"{script_directory}\\sample")

    run_evaluation_llama()
```

Fig 2.10: Evaluation Pipeline

1. Quest Generation Process:

The evaluate_quest_generation_llama function is responsible for generating quests based on user-provided inputs, such as prompts, objectives, locations, and characters. The quest generation process follows a three-step methodology:

1. **Initial Quest Generation**: The initial quest is produced using the quest_generator.generate_initial_quest function, which creates the foundational structure of the quest based on a prompt and game-specific elements.
2. **Objective Refinement**: The quest is further refined by adding specific objectives through the generate_quest_with_objectives function.
3. **Quest Formatting**: The final quest is formatted according to a predefined schema, ensuring that it adheres to structural requirements for further analysis and evaluation.

3. Evaluation Pipeline:

The run_evaluation_llama function orchestrates the evaluation of the generated quests using a predefined dataset ("ds-campbell-evaluation-50"). The evaluation is conducted using multiple evaluators, including metrics for clarity, engagement, and creativity. This process allows for the comparison of different models or iterations of quest generation, providing a systematic framework for assessing output quality.

4. Evaluator Wrappers:

The evaluation functions, such as run_clarity_evaluator, run_engagement_evaluator, and run_creativity_evaluator, serve as wrappers for specific evaluation metrics. These functions include mechanisms for handling rate limits from the external API by implementing retry logic with exponential backoff. In the event of a RateLimitError, the functions wait for progressively longer intervals before attempting to re-run the evaluation.

5. Evaluator Configuration:

Each evaluator is configured to assess specific qualitative metrics using LangChain's LLM evaluation capabilities. The three key evaluators are:

- **Clarity Evaluator**: Assesses the conciseness and coherence of the generated quests, ensuring they are logically structured and easy to follow.
- **Engagement Evaluator**: Evaluates whether the quest is engaging and relevant, maintaining the player's attention and addressing the intended prompt.
- **Creativity Evaluator**: Measures the creativity and narrative complexity of the quest, assessing how original and innovative the generated content is, as well as its integration of multiple narrative elements.

6. Execution and Evaluation:

The script concludes by setting the working directory and invoking the run_evaluation_llama function, which executes the quest generation and evaluation pipeline. This step triggers the generation of quests based on the input data and evaluates them according to the defined metrics, providing feedback on key aspects of the generated quests.

In conclusion, this code implements a robust, automated pipeline for the procedural generation and evaluation of quests, leveraging LLMs to produce content and evaluate its quality based on diverse qualitative metrics. By integrating evaluators focused on clarity, engagement, and creativity, the system enables a comprehensive analysis of quest generation performance across different models, ensuring the production of high-quality game content.

Artefacts

Campbell-Quest Python Package

The Campbell-Quest is the Python package that functions as the backend for the Procedurally Generated Pipeline. It employs LangChain and Ollama to establish a locally deployed agentic framework, facilitating the generation of procedural content within a controlled environment.

The **questGenerator** class and **questAgents** module integrates the language model to generate various aspects of game quests, including quest descriptions, objectives, and rewards. The code is organized into multiple key functions that interact with custom agents defined through the LangChain module.

Main Functions:

1. generate_initial_quest(quest_prompt, objective_info, location_info, character_info, model="llama3.1"):

- This function generates an initial version of a quest based on provided input information, such as the quest prompt, objectives, location, and character details.

- The `**quest_brainstormer.generate_quest**` function, part of the questAgents module, is responsible for generating the quest using the specified language model.

2. generate_quest_with_objectives(initial_generated_quest, location_info, character_info, model="llama3.1"):

- This function takes an initial quest, refines it, and defines specific objectives that the player will need to complete. It uses location and character information to ensure coherence.

- It relies on the `**quest_refiner.define_quest_objectives**` function to add clear objectives.

3. generate_quest_reward(initial_generated_quest, rewards, model="llama3.1"):

- This function assigns an appropriate reward to the quest based on a predefined list of possible rewards.

- It uses the `**quest_refiner.define_quest_reward**` to map a reward to the quest.

4. get_formatted_quest(quest, schema, model="llama3.1"):

- This function converts a generated quest into a structured format according to a provided schema.

- The `**quest_formatter.format_quest**` function is invoked to handle the formatting.

5. get_formatted_quest_with_rewards(quest, reward, schema, model="llama3.1"):

- This function generates a formatted quest with an added "rewards" field, ensuring the quest adheres to a specific schema while also including rewards.

- The `**quest_formatter.format_quest_with_rewards**` function is used here.

Use of Language Models and Prompting:

The code uses a language model through the `**ChatOllama**` class, which is part of the `**langchain_community.chat_models**` module. The model processes input and generates text-based outputs for quest generation and formatting tasks. These tasks include:

- System Prompts: The code defines system-level instructions that provide the AI with background context on the game world, setting, or task.

- User Prompts: These are more specific task-oriented instructions, guiding the AI on how to structure its output (e.g., "generate a quest with specific objectives").

For example, in `**generate_quest()**`, the AI is tasked with creating a quest based on input data, following both system-wide game design instructions and user-provided constraints (e.g., using only specified locations and characters).

Integration with LangChain:

The integration with LangChain involves the use of templates **(`SystemMessagePromptTemplate`, `HumanMessagePromptTemplate`)** that allow the system and user messages to be structured and passed into the language model. The outputs from the language model are then parsed using the `StrOutputParser` class to generate text-based responses.

Functions for Specific Quest Generation Aspects:

1. generate_quest()

- This function constructs a prompt where the system provides the AI with detailed instructions on generating a quest based on given objectives, locations, and characters.

- It formats the prompt using a system and user template and then passes the input to the language model (`ChatOllama`) to receive a quest description.

2. format_quest()

- This function generates a JSON representation of the quest based on a schema and uses a model to ensure the output adheres to the provided format.

3. format_quest_with_rewards()

- Similar to `format_quest()`, but it adds a rewards field to the JSON object of the quest.

4. define_quest_objectives()

- This function takes an existing quest and adds a detailed list of objectives. It works with character and location data to ensure that the objectives align with the quest context.

5. define_quest_reward()

- This function selects an appropriate reward for a quest from a list of possible rewards based on the quest's description and characteristics.

Dialogue Tree Generation:

Similarly, the **dialogueGenerator** class and **dialogueAgents** module is responsible for the procedural generation of Dialogue Trees. The central function, **`get_dialogues`**, initiates the process by extracting the required dialogues based on the input quest, characters, and other relevant contextual information. These dialogues are structured in a standardized JSON format, adhering to a predefined schema. The function then identifies the relevant NPCs involved in the interaction and associates each with specific dialogue cues, derived from the quest objectives. The system subsequently generates individual dialogue trees for each NPC, incorporating templates and logic pertinent to the narrative context of the game. The dialogue trees are subsequently refined and validated to ensure they maintain a logical flow and coherence with the quest’s storyline.

Each stage of the dialogue generation process is compartmentalized into distinct functions that handle specific tasks: generating context-sensitive dialogue, formatting the dialogue output into JSON format, and validating the logical consistency of the conversation flow. These processes are governed by strict adherence to game design principles, including conditions such as quest

progress, item possession, and narrative triggers, as well as in-game results, such as receiving new quests or acquiring items. The system ensures that the dialogue remains functional and integrated within the broader game mechanics, thus facilitating both narrative immersion and gameplay progression.

Enemy Generation:

The **enemyGenerator** class and **enemyAgents** module is responsible for the procedural generation of Enemies and NPCs. The primary function, **`get_enemies`**, orchestrates the entire workflow, generating detailed enemy data based on the quest context and ensuring adherence to a predefined schema.

The **`get_enemies`** function begins by extracting required enemy encounters from a quest using the **`enemy_refiner`** module, which identifies potential enemy interactions. These are then formatted into a structured JSON format, adhering to a specified schema. Each enemy is generated based on specific cues and contextual data from the quest, ensuring that the enemies are aligned with the narrative objectives. The enemy details are formatted and validated before being returned as a list of JSON objects, facilitating easy integration into the game's design and development pipeline.

Several auxiliary functions handle different aspects of the enemy generation process. For instance, **`generate_enemy`** utilizes AI to dynamically generate unique and contextually appropriate enemy names and attributes based on the quest and a template. Meanwhile, **`get_formatted_required_enemies`** and **`get_formatted_enemy`** ensure that the generated enemies adhere to the expected JSON schema, stripping any unnecessary text and outputting only valid, structured data. The **`get_required_enemies`** function analyzes a quest's objectives and identifies which encounters necessitate fighting enemies, ensuring that only relevant enemy interactions are processed.

Item Generation:

The **itemGenerator** class and **itemAgents** module is responsible for the procedural generation of quest relevant items. The workflow begins with the identification of necessary items for a given quest, facilitated by the **`item_refiner.get_required_items()`** function. These items are subsequently formatted into a valid JSON schema, ensuring all outputs conform to the requisite structure for integration into game development. Each identified item is then further processed

according to its type: action items, which are single-use, and equipment, which can be worn or wielded by players. Specific generation processes for each item type are employed to produce immersive descriptions and attributes, utilizing AI-based brainstorming and refinement.

The code employs several auxiliary functions tailored to different item types:

1. generate_action_item(): This function generates unique and compelling descriptions for action items, contextualized by the quest and item narrative. It ensures that the items are not only consistent with the game's storyline but also enhance the player's experience. Each item is formatted according to a predefined template.

2. generate_equipment(): This function creates descriptions for equipment items, such as armor and weapons, and includes the "allowedEquipLocation" field, which specifies where the item can be equipped (e.g., helmet, weapon, shield). The equipment is formatted based on a given template and integrated within the game’s context.

3. get_formatted_required_items() and **get_formatted_item():** These functions convert the generated items into valid JSON outputs, ensuring that the data is structured appropriately and adheres to the specified schema.

4. get_required_items(): This function evaluates the quest to determine the items the player must interact with. Each item is classified into categories (e.g., action items, equipment), and associated with the appropriate objective type (e.g., "pickup" or "destroy").

Each stage of the item generation process is automated, ensuring that the resulting items are both narratively cohesive and technically accurate for implementation in game development.

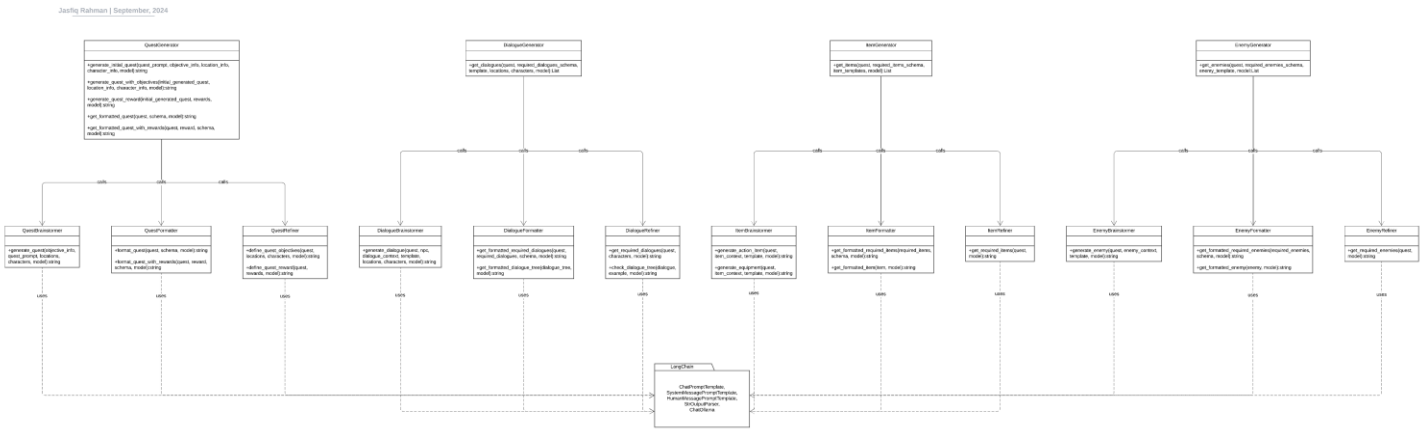

Fig 3.1: Python Package UML Diagram

Campbell-Quest Unity Tool

Quest Generation:

The tool serves to facilitate the creation and management of quests within the Unity editor. The primary component of this system is the **`QuestEditorWindow`**, which features two principal tabs: Context Editor and Quest Editor. The context editor is further subdivided into sections addressing quest prompts, objectives, locations, characters, and rewards, thereby streamlining the quest creation process. This user interface is designed to empower developers to generate quests efficiently and create associated assets with minimal friction.

The **`QuestGenerator`** class plays a pivotal role in the system by managing the creation of quest assets derived from JSON representations. It includes functionality to ascertain the existence of pre-existing assets within a specified directory, thereby ensuring that new assets are created only when necessary. Each quest comprises essential attributes such as name, description, goals, objectives, and rewards, which are encapsulated within **`QuestData`** and **`QuestMetadataFormat`** classes. This structural design allows for the customization of quests, ensuring that they align with the overarching requirements of the game.

Data serialization within this framework is executed via the Newtonsoft.Json library, which allows for the efficient conversion between JSON strings and object instances. This capability is integral to managing various quest data formats, enabling straightforward saving, loading, and manipulation within the Unity editor. Collectively, this quest editor system significantly enhances the Unity development process by providing a structured and user-friendly interface for the creation of engaging gameplay experiences, while simultaneously ensuring the efficient management of underlying data and assets.

The **`QuestProcessor`** class enhances the functionality of the quest editor system by providing methods for managing and generating quest data. This class is responsible for handling various components of a quest, including prompts, objectives, locations, characters, and rewards, which are represented as string fields. The user interface enables the population of these fields with sample data from text files, as well as the dynamic display of each component through separate methods. A scrolling text area is utilized for each category, allowing for easy input and editing.

A critical feature of the **`QuestProcessor`** is its ability to validate the completeness of the quest context, ensuring that all necessary fields are filled before generating a quest. Upon validation, the class constructs Python scripts that invoke a quest generation algorithm from the

`campbell_quest` Python module, ultimately producing a formatted quest. The generated quest data, including objectives and rewards, is presented in a structured format, facilitating the inclusion of customizable components. The class also employs the Newtonsoft.Json library to serialize and deserialize quest data, enabling seamless data management within the Unity editor.

Furthermore, the **`QuestProcessor`** class initializes reorderable lists for both objectives and rewards, allowing users to modify and organize these elements dynamically. It provides methods for displaying quest information, clearing quests, and creating or recreating quest assets based on the generated data. This comprehensive management of quest components not only streamlines the quest creation process but also ensures that quests are developed in accordance with predefined schemas, thereby enhancing the overall efficiency and user experience within the Unity environment. Together with the **`QuestEditorWindow`** and **`QuestGenerator`**, the **`QuestProcessor`** contributes to a cohesive and efficient system for quest management and generation in game development.

Dialogue Generation:

Similarly, the Dialogue generation pipeline comprises three primary components aimed at facilitating dialogue management. The **`DialogueEditorWindow`** class extends the Unity Editor's capabilities by providing a graphical interface for editing dialogue associated with quest assets. It employs the **`DialogueProcessor`** to generate and display dialogues based on the metadata of the selected quest. The editor window includes functionality for handling dialogue data, allowing users to view, modify, and create dialogue assets stored in a specified directory. A toolbar presents dialogue options tied to non-player characters (NPCs), and user interactions trigger the generation and saving of dialogue data.

The **`DialogueGenerator`** class serves as a utility for creating and managing dialogue assets derived from JSON data structures. It includes methods to check for the existence of dialogue assets, create new dialogue instances, and process dialogue choices based on player and NPC interactions. The generator employs various data structures to manage conditions, actions, and the arrangement of dialogue nodes within a visual interface.

Lastly, the **`DialogueData`** and **`ChoiceData`** classes define the structure of dialogue elements and player choices, encapsulating essential information such as NPC names, dialogue lines, and conditional choices. This structured approach to dialogue management enhances the organization, accessibility, and functionality of narrative elements within the Unity environment, thereby improving the overall game development process.

Npc Generation:

The core component of the Npc Generation pipeline, `**NpcEditorWindow**`, facilitates a graphical user interface for selecting quest and dialogue assets, enabling users to generate and manage NPCs based on the defined quests. The interface employs Unity's Editor GUI for user interaction, where it checks for the existence of the necessary assets and displays relevant information for editing and saving NPC-related data.

The `**NpcProcessor**` class orchestrates the generation of NPCs by processing quest metadata and utilizing the Python pipeline to generate enemy data formatted in JSON. It incorporates methods for generating enemies, clearing lists of generated enemies, and creating or recreating NPC assets based on the dialogue provided.

Complementing this functionality, the `**NpcGenerator**` class manages the creation of NPC prefabs and associates them with various components essential for gameplay, such as quest handling, dialogue interaction, and combat mechanics. This class ensures that each NPC is properly configured with necessary attributes, including their dialogue options and quest-related functionalities, enhancing the narrative depth and interactivity within the game.

Item Generation:

The core component of the Item Generation pipeline, `**ItemEditorWindow**`, enables users to select a quest asset and generate items derived from the corresponding quest data. This interface allows for the display, editing, and saving of item attributes.

The `**ItemGenerator**` class plays a pivotal role in creating item assets from JSON representations. It verifies the existence of item assets and orchestrates the creation of associated pickup objects and item data, thereby ensuring that the game environment remains consistent with the designed quest structure. The `**ItemProcessor**` class further complements this functionality by processing the item generation from formatted quest data, providing methods to display item information, clear generated items, and manage asset creation or recreation.

This extension offers an intuitive graphical user interface, utilizing Unity's Editor GUI to enable users to modify item attributes such as name, description, and type. The inclusion of asset

management features allows for efficient organization and storage of item assets within the designated directories.

Evaluation

Fig 4.1: Evaluation Results – Metric Scores

Results:

1. **Conciseness-Coherence Scores:**
    - **Llama3.1**: 8.82
    - **Gemma2**: 9.06
    - **Mistral-Nemo**: 8.90

Gemma2 has the highest score for conciseness and coherence, indicating it provides the most coherent and well-organized output. **Llama3.1** and **Mistral-Nemo** are close but slightly lower in this aspect.

2. **Creativity-Narrative Complexity Scores:**
    - **Llama3.1**: 3.96
    - **Gemma2**: 5.75
    - **Mistral-Nemo**: 6.98

Mistral-Nemo shows the highest creativity and narrative complexity, suggesting it generates the most intricate and imaginative narratives, followed by **Gemma2** and then **Llama3.1**.

3. **Relevance-Engagement Scores:**
    - **Llama3.1**: 8.73
    - **Gemma2**: 9.43
    - **Mistral-Nemo**: 9.80

Mistral-Nemo excels in relevance and engagement, making it the most engaging and relevant in its outputs, while **Gemma2** and **Llama3.1** are also effective but slightly less so.

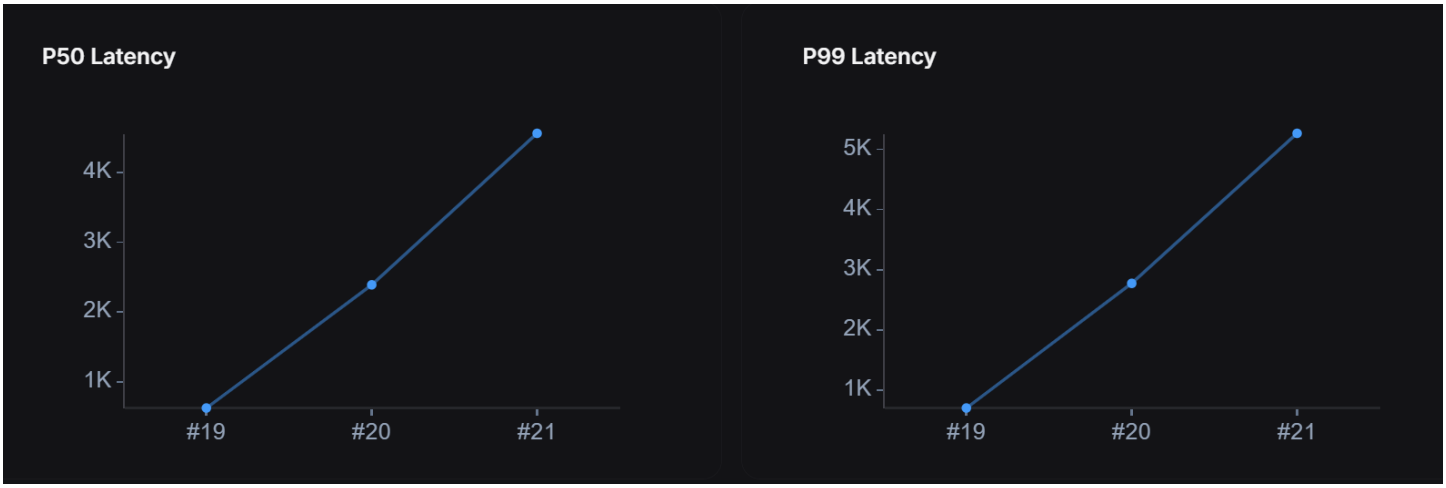

Fig 4.2: Evaluation Results – Latency

Latency:

- **Llama3.1**: P50: 622.66s, P99: 700.86s
- **Gemma2**: P50: 2388.48s, P99: 2772.30s
- **Mistral-Nemo**: P50: 4559.49s, P99: 5265.70s

Llama3.1 has the shortest latency, followed by **Gemma2** and then **Mistral-Nemo**, indicating that **Llama3.1** is the fastest in generating responses, with **Mistral-Nemo** being the slowest.

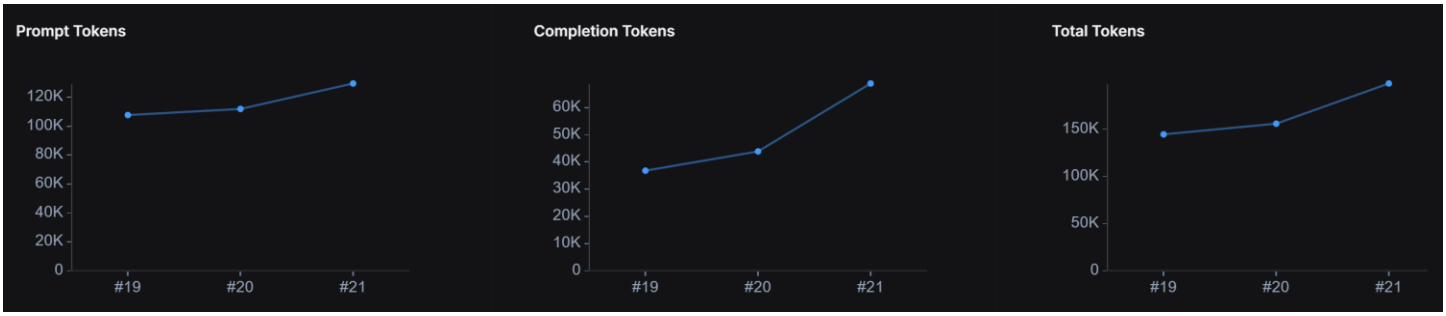

Fig 4.3: Evaluation Results – Token Usage

Token Usage:

- **Llama3.1**: Prompt Tokens: 107,648, Completion Tokens: 36,799, Total Tokens: 144,447
- **Gemma2**: Prompt Tokens: 111,872, Completion Tokens: 43,872, Total Tokens: 155,744
- **Mistral-Nemo**: Prompt Tokens: 129,475, Completion Tokens: 68,840, Total Tokens: 198,315

Llama3.1 uses the fewest tokens overall, which might imply more efficient use of resources compared to **Gemma2** and **Mistral-Nemo**.

Observations:

- **Gemma2** provides the best balance between conciseness, coherence, and engagement but is slower and consumes more tokens than **Llama3.1**.
- **Mistral-Nemo** offers superior creativity and narrative complexity as well as the highest engagement but at the cost of higher latency and token usage.
- **Llama3.1** excels in performance efficiency with the fastest response times and lower token usage but is slightly lower in creativity and complexity compared to **Gemma2** and **Mistral-Nemo**.

Conclusions:

Llama3.1 is particularly well-suited for use with the procedurally generated quest system due to several key advantages:

- **Speed and Efficiency:** Llama3.1 has the shortest latency among the models tested, with a P50 latency of 622.66s and a P99 latency of 700.86s. This means it generates responses

faster than the other models, which is crucial for maintaining a smooth and responsive user experience.

- **Resource Efficiency:** It uses the fewest total tokens (144,447) compared to Gemma2 and Mistral-Nemo. This efficient use of tokens can lead to lower computational costs and faster processing, which is advantageous for applications with high token usage or budget constraints.
- **Conciseness and Coherence:** With a high conciseness-coherence score of 8.82, Llama3.1 provides clear and well-organized responses. This is essential for ensuring that the procedurally generated quests are easy to follow and understand, enhancing the overall quality of the user experience.
- **Good Balance of Relevance and Engagement:** Llama3.1 scores 8.73 in relevance and engagement, indicating that it generates content that is relevant and engaging, though not as high as Mistral-Nemo. Despite this, it still provides satisfactory engagement, which is important for maintaining user interest.

Llama 3.1's primary limitation lies in its relatively low creativity score. However, if processing speed is not a critical factor, it may be worthwhile to consider utilizing alternative large language models that offer enhanced creative capacities, thereby potentially yielding more innovative outputs.

Overall, Llama3.1 strikes a favourable balance between speed, efficiency, and quality. Its fast response times and lower token usage make it an ideal choice.

Ethics

Ethical Considerations

In the commercialization of this project, several ethical concerns arise. The use of Large Language Models (LLMs) to generate narrative content introduces the risk of perpetuating bias or inappropriate content embedded within the AI's training data. Since these models are trained on vast datasets, including text from a wide range of sources, the generated content may unknowingly incorporate harmful stereotypes or unintended biases. As developers, it is our responsibility to carefully review and moderate AI-generated narratives to ensure they align with ethical storytelling practices, avoiding harmful or offensive material.

Furthermore, the use of AI for narrative generation could raise questions about the impact on human creators. While the tool is designed to assist and augment human creativity, concerns may be raised about the potential devaluation of human labor in game design. Clear communication is necessary to emphasize that this tool complements, rather than replaces, the creative efforts of human developers.

Legal and Copyright Issues

In terms of legal considerations, the system's reliance on LLMs introduces potential copyright challenges. Since generative AI models are trained on datasets that include copyrighted works, there is a risk that AI-generated content may inadvertently replicate phrases, plot points, or other elements from copyrighted material. This can lead to disputes if the generated content is perceived as too similar to existing works.

To mitigate this, it is important to implement safeguards such as regular review processes to ensure that any generated narratives are original and not directly derived from copyrighted material. Additionally, if the system is commercialized, it may be necessary to consult legal experts to ensure compliance with copyright laws and to address potential liabilities in the use of AI-generated content.

Impact of Generative AI on the Project

Generative AI models used in this project may have been trained on copyrighted materials without explicit credit to the authors, which raises ethical concerns. The training data used by these models typically includes vast amounts of text from a variety of sources, often without explicit permission from the original authors. While the AI-generated content itself may not directly reproduce specific copyrighted works, the fact that the model's training process involved such material introduces a gray area in terms of intellectual property.

In this project, the risk is mitigated by using the tool primarily to generate supplementary content, such as side quests, rather than the main narrative. Nevertheless, we must remain vigilant in reviewing and editing AI-generated content to ensure it does not infringe on copyrighted works. Additionally, transparency regarding the use of generative AI and its limitations is crucial in communicating the nature of the content to users and stakeholders.

In conclusion, while this project presents exciting opportunities for enhancing narrative development in RPGs, it also requires careful ethical and legal considerations to ensure that the tool is used responsibly and that intellectual property rights are respected.

Conclusion

This project set out to develop a Procedural Quest Generation system utilizing Large Language Models (LLMs) to improve quest design in role-playing games (RPGs), particularly for smaller development teams. Reflecting on the project, several key insights have emerged regarding its successes, challenges, and potential for future development.

One of the primary strengths of the system lies in its ability to generate coherent and narrative-rich side quests with minimal human intervention. This system provides substantial value to smaller game development teams by offering a scalable solution for generating engaging narrative content. The use of models such as Llama3.1, Gemma2, and Mistral-Nemo demonstrated significant flexibility. While Mistral-Nemo excelled in terms of creativity and narrative complexity, Llama3.1 offered superior performance in terms of processing speed and resource efficiency, which makes it particularly suited for real-time applications. This ability to balance creativity and efficiency allowed the system to meet several key objectives of the project, including the generation of immersive side quests that complement human-designed main storylines.

However, the project also faced several limitations. The most notable challenge was in achieving the same level of narrative depth and long-term coherence that human-designed quests typically offer. While the system performed well in generating short-term quests, more complex, multi-stage questlines required significant refinement. Additionally, while the creative potential of the LLMs was demonstrated, there remains a trade-off between creativity and computational efficiency, particularly when resources are constrained. Addressing this trade-off is a key area for improvement.

In terms of achieving the project's aims and objectives, the PQG system successfully implemented a locally deployed solution and integrated it into the Unity engine. The modular design of the system, along with its compatibility with various game engines, provides a strong foundation for future expansions. Furthermore, the evaluation process indicated that the system effectively produced quests that were coherent, engaging, and aligned with the intended narrative goals, thereby fulfilling the project's primary aim of reducing the burden on developers while maintaining high standards of narrative quality.

Future work should focus on improving the system's ability to generate more complex, multi-part questlines and developing mechanisms for creating overarching narrative structures. Additionally, efforts should be made to optimize the system's performance to enhance both

creativity and efficiency, particularly in resource-constrained environments. Further expansion could also include adapting the system for other narrative-driven game genres or enhancing its capacity for procedurally generating character development and dialogue systems.

In considering the potential commercialization of the project, several steps would be necessary. First, refining the user interface to ensure ease of use for non-technical game developers would be essential for broad adoption. Offering customizable LLM solutions tailored to specific game genres would further increase the tool's appeal. A cloud-based version, which reduces the need for local computational resources, could broaden the system's accessibility. Additionally, positioning the tool as a cost-effective solution for procedural quest generation, particularly for smaller studios, would open new commercial opportunities. Collaborating with established game development platforms to provide the tool as a plugin could also accelerate adoption and ease integration into existing development workflows.

In conclusion, this project has demonstrated that the use of LLMs for procedural quest generation offers substantial potential for reducing the workload on developers while maintaining high-quality narrative content. Although there are areas for improvement, particularly in terms of narrative complexity and performance optimization, the system has proven its value as a scalable solution for automating the creation of dynamic and engaging quests.

References

[1] Batman: Arkham Asylum (2009). [Video game]. Warner Bros. Interactive Entertainment.

[2] The Witcher 3: Wild Hunt (2015). [Video game]. CD Projekt RED.

[3] Minecraft (2011). [Video game]. Mojang Studios.

[4] Terraria (2011). [Video game]. Re-Logic.

[5] No Man's Sky (2016). [Video game]. Hello Games.

[6] de Lima, E.S., Feijó, B. and Furtado, A.L., 2022. Procedural generation of branching quests for games. Entertainment Computing, 43, p.100491. https://doi.org/10.1016/j.entcom.2022.100491

[7] Breault, V., Ouellet, S. and Davies, J., 2021. Let CONAN tell you a story: Procedural quest generation. Entertainment Computing, 38, p.100422. https://doi.org/10.1016/j.entcom.2021.100422

[8] Prins, V.L., Prins, J., Preuss, M. and Gómez-Maureira, M.A., 2023, April. Storyworld: Procedural quest generation rooted in variety & believability. In Proceedings of the 18th International Conference on the Foundations of Digital Games (pp. 1-4). https://dl.acm.org/doi/abs/10.1145/3582437.3587181

[9] Grey, J. and Bryson, J.J., 2011, April. Procedural quests: A focus for agent interaction in role-playing-games. In Proceedings of the AISB 2011 Symposium: AI & Games. https://researchportal.bath.ac.uk/en/publications/procedural-quests-a-focus-for-agent-interaction-in-role-playing-g

[10] Al-Nassar, S., Schaap, A., Zwart, M.V.D., Preuss, M. and Gómez-Maureira, M.A., 2023, April. QuestVille: procedural quest generation using NLP models. In Proceedings of the 18th International Conference on the Foundations of Digital Games (pp. 1-4). https://dl.acm.org/doi/abs/10.1145/3582437.3587188

[11] Ashby, T., Webb, B.K., Knapp, G., Searle, J. and Fulda, N., 2023, April. Personalized quest and dialogue generation in role-playing games: A knowledge graph-and language model-based approach. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (pp. 1-20). https://dl.acm.org/doi/full/10.1145/3544548.3581441

[12] Rimworld (2016). [Video game]. Ludeon Studios.

[13] Middle-earth: Shadow of Mordor (2014). [Video game]. Monolith Productions. Warner Bros. Interactive Entertainment.

[14] Game Maker's Toolkit. (2021, January 28). How the Nemesis System Creates Stories [Video]. YouTube. Retrieved from http://www.youtube.com/watch?v=Lm_AzK27mZY

[15] Parosu, I., Hage, E. and Magnusson, S., 2022. The Nemesis System: How games create stories. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1708404&dswid=-6690

[16] Wildermyth (2021). [Video game]. Worldwalker Games. Foxglove Games.

[17] Austin, N. (2021, July 15). Procgen in Wildermyth: Storytelling [Video]. YouTube. Presented at EPC2021. Retrieved from https://www.youtube.com/watch?v=A5BGDbLFRrE

[18] Meta Llama 3.1 (n.d.). [Software]. Meta. Retrieved from https://llama.meta.com/llama3.1/

[19] Gemma 2 (n.d.). [Software]. Google Cloud Vertex AI. Google LLC. Retrieved from https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/gemma2

[20] Mistral-Nemo (n.d.). A 12B Open-Source AI Model for Global, Multilingual Applications. [Software]. Mistral.ai. Retrieved from https://mistral.ai/news/mistral-nemo/

[21] Ollama (n.d.). A Platform for Running Large Language Models. [Software]. Ollama. Retrieved from https://ollama.com/

[22] LangChain (n.d.). A Framework for Building Applications with Large Language Models. [Software]. LangChain. Retrieved from https://www.langchain.com/

[23] LangSmith (n.d.). A Platform for Building and Managing LLM Applications. LangChain. [Software]. Retrieved from https://www.langchain.com/langsmith

[24] GPT-4 Turbo (n.d.). OpenAI API. [Software]. OpenAI. Retrieved from https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

[25] Unity (n.d.). Game Development Platform. [Software]. Unity. Retrieved from https://unity.com/

[26] Unity Python Scripting (n.d.). Unity Manual: Python Scripting. [Documentation]. Unity. Retrieved from https://docs.unity3d.com/Packages/com.unity.scripting.python@7.0/manual/index.html

[27] Unity Editor Scripting (n.d.). Unity Manual: Scripting the Editor. [Documentation]. Unity. Retrieved from https://docs.unity3d.com/ScriptReference/Editor.html

[28] GameDev.tv (n.d.). Online Game Development Courses & Tutorials. Retrieved from https://www.gamedev.tv/

[29] Unity Dialogue & Quests: Intermediate C# Game Coding (n.d.). [Course]. Udemy. Retrieved from https://www.udemy.com/course/unity-dialogue-quests/

[30] Visual Studio (n.d.). [Software]. Microsoft. Retrieved from https://visualstudio.microsoft.com/

[31] Visual Studio Code (n.d.). [Software]. Microsoft. Retrieved from https://code.visualstudio.com/

[32] Github (n.d.). Version control for Git. [Software]. Retrieved from https://github.com/

[33] ChatGPT (n.d.). [Software]. OpenAI. Retrieved from https://chatgpt.com/

[34] Game Maker's Toolkit (n.d.). YouTube channel. Retrieved from https://www.youtube.com/channel/UCqJ-Xo29CKyLTjn6z2XwYAw

[35] Design Doc (n.d.). YouTube channel. Retrieved from https://www.youtube.com/channel/UCNOVwMpD-5A1xzcQGbIHNeA

[36] Extra Credits (n.d.). YouTube channel. Retrieved from https://www.youtube.com/extracredits

[37] Adam Millard - The Architect of Games (n.d.). YouTube channel. Retrieved from https://www.youtube.com/@ArchitectofGames

[38] LangChain- Develop LLM powered applications with LangChain (n.d.). [Course]. Udemy. Retrieved from https://www.udemy.com/course/langchain/

[39] ScriptableObject (n.d.). [Class documentation]. Unity Manual. Retrieved from https://docs.unity3d.com/Manual/class-ScriptableObject.html

[40] Resources (n.d.). [Class documentation]. Unity Scripting Reference. Retrieved from https://docs.unity3d.com/ScriptReference/Resources.html

[41] AssetDatabase (n.d.). [Class documentation]. Unity Scripting Reference. Retrieved from https://docs.unity3d.com/ScriptReference/AssetDatabase.html

[42] MonoBehaviour (n.d.). [Class documentation]. Unity Scripting Reference. Retrieved from https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Resources Used

Assets

https://assetstore.unity.com/packages/3d/animations/rpg-character-mecanim-animation-pack-free-65284

https://assetstore.unity.com/packages/vfx/shaders/free-skybox-extended-shader-107400

https://assetstore.unity.com/packages/2d/gui/icons/food-icons-pack-70018

https://assetstore.unity.com/packages/3d/props/food/low-poly-fruit-pickups-98135

https://assetstore.unity.com/packages/2d/gui/icons/pixel-cursors-109256

https://assetstore.unity.com/packages/3d/environments/fantasy/polygon-fantasy-kingdom-low-poly-3d-art-by-synty-164532

https://assetstore.unity.com/packages/vfx/particles/simple-fx-cartoon-particles-67834

https://assetstore.unity.com/packages/2d/gui/icons/universal-bronze-icon-pack-120654

Udemy Course Series for Quest Framework

https://www.udemy.com/course/unityrpg/

https://www.udemy.com/course/unityinventory/

https://www.udemy.com/course/unity-dialogue-quests/

Software Resources

ChatGPT for editing and proofreading. Accessed: https://chatgpt.com/

Evaluation Data https://chatgpt.com/share/a0d3a713-7940-4460-8675-68daeda1aa72

DocuWriter.ai for documentation assistance. Accessed: https://www.docuwriter.ai/

PlayHT for voice generation (for Demo Video). Accessed: https://play.ht/