HAUNT SCRIPT

CHEMBIAN RK-22BAI1431
SARVEESH KUMAR-22BAI1420
VISHWA-22BAI1466

Model Selection Report

Objective: Identify and evaluate appropriate pre-trained language models capable of generating coherent, immersive, and horror-themed stories based on structured prompts.

Models Considered:

1. **GPT-3.5 Turbo** (OpenAI): High fluency and versatility, strong context understanding.
2. **Claude 2** (Anthropic): Ethical safeguards, good at dialogue-based storytelling.
3. **Gemini 1.5 Flash** (Google): Fast, lightweight, optimized for streaming real-time outputs.

Final Choice: Gemini 1.5 Flash

Reasons for Selection:

- Designed for low-latency generation which is ideal for real-time applications.
- Seamless integration via the Google Generative AI SDK.
- Cost-effective for frequent, short-form generation compared to OpenAI models.
- Performed consistently well in horror tone adherence and creativity during pilot testing.

Prompt Engineering and API Usage

Prompt Structure Design: Prompting is key to effective interaction with any LLM. We created a dynamic prompt structure:

prompt = f"Write a horror story featuring the character '{character_name}' in the situation '{situation}' within {no_of_lines} lines."

Enhancements Introduced:

- Emphasized genre tone using keywords like "creepy", "eerie", "blood-curdling".
- Tried variants: e.g., "Make the ending unexpected" or "Use dark atmospheric visuals".
- Calibrated input length to prevent verbosity or truncation.

Chat Session Handling:

- Each story generation call was wrapped within `start_chat(history=[])` to ensure a fresh context.

- Error handling added to catch API downtime, retries, and timeout situations.

Function Breakdown:

```
def generate_horror_story(character_name, situation, no_of_lines):
    prompt = f"Write a horror story with the character name '{character_name}' and situation
'{situation}' in {no_of_lines} lines."
    chat_session = model.start_chat(history=[])
    response = chat_session.send_message(prompt)
    return response.text
```

Implementation and Integration

Code Structure:

- Core logic implemented in `horror.py`
- Frontend UI captures inputs and calls `generate_horror_story()`
- Gemini model initialized in backend using API key from secure config file

Example Input/Output:

- Input:
    - Character: “Elena”
    - Situation: “Trapped in an abandoned asylum on Halloween night”
    - Lines: 5
- Output:
  "Elena tiptoed past broken gurneys. A whisper echoed—her name. The hallway stretched unnaturally. Suddenly, her reflection blinked. Then smiled."

Observations:

- Vivid use of imagery without explicit instructions
- Coherent structure maintained even under 5-line limit
- Surprise ending aligns with horror twist conventions

Testing and Evaluation Protocol

Testing Metrics:

- Runtime per request (Goal: <5s)
- Output relevance to user input
- Repetition checks across multiple runs for same input
- Story uniqueness when modifying only one parameter (e.g., changing line count)

Stress Tests:

- Input edge cases: extremely long names or vague situations
- Rapid consecutive calls from multiple systems (simulate high traffic)
- Incorrect inputs (empty fields, symbols, etc.)

Refinements Based on Testing:

- Increased `max_output_tokens` to prevent cutoffs
- Adjusted `temperature` to 0.75 for balance between creativity and control
- Included handling for vague input like “haunted house” by expanding prompt internally

Conclusion: The model development phase focused on leveraging Gemini’s real-time capabilities with structured and optimized prompting. While no training was done from scratch, a significant part of our effort was in crafting, refining, and validating the way we interact with the model. Combined with error handling, stress testing, and creative tuning, this phase ensured that HauntScript could reliably generate unique and engaging horror micro-stories.
