**Technical Document - Sprint 1**

## 1. Overview

This document outlines the design and implementation of utilities aimed at integrating with the Gemini 1.5 Flash AI model. The utilities provide essential functionalities such as:

* Handling timeouts when calling the AI model.
* Validating input files, particularly image files.
* Error handling for invalid inputs, such as corrupted images or incorrect file paths.

This system is built in Python and leverages the **Pillow** library for image processing and **threading** for timeout management. It also uses environment variables for API key management and custom exceptions for error handling.

**2. Functionalities**

**2.1 Timeout Handling**

Timeout handling ensures that the call to the Gemini AI model does not exceed a predefined time limit (20 seconds in this case). This is important to prevent the system from hanging due to prolonged requests, such as waiting for a response from the API when there is network latency or an excessive server load.

**Function: timeout(timeout)**

* **Purpose**: A decorator that enforces a time limit on function execution. If the function exceeds the specified timeout, it raises a TimeoutException.
* **Parameters**:
  + timeout: Integer value specifying the maximum allowed execution time for the function (in seconds).
* **Usage**:
  + Wrap any function that interacts with the AI model or another service that might take too long to respond.

@timeout(timeout=20)

def generate\_content\_timeout(model, prompt, image=None):

# Generates content using the AI model with a timeout.

if image:

return model.generate\_content([prompt, image])

return model.generate\_content(prompt)

**Mechanism:**

* The original function (func) is executed in a separate thread using Python's **threading.Thread.**
* If the function completes within the given time, the result is returned.
* If the execution time exceeds the limit, a TimeoutException is raised and caught to inform the user.

def timeout(timeout):

def deco(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

res = [TimeoutException("It took too long, timeout error.")]

def new\_func():

try:

res[0] = func(\*args, \*\*kwargs)

except Exception as e:

res[0] = e

t = Thread(target=new\_func)

t.daemon = True

try:

t.start()

t.join(timeout)

except Exception as je:

print('Error starting thread')

raise je

ret = res[0]

if isinstance(ret, BaseException):

raise ret

return ret

return wrapper

return deco

**2.2 Image File Validation**

Ensuring that the provided image is valid is essential to prevent invalid data from being sent to the AI model. This prevents unnecessary API requests and improves the robustness of the system.

**Function: file\_does\_exist(file\_path)**

* **Purpose**: Checks whether the specified file path exists and confirms it is a file, not a directory.
* **Parameters**:
  + file\_path: String representing the path to the file.
* **Returns**: Boolean value (**True** if the file exists and is a valid file, **False** otherwise).

**Code:**

def file\_does\_exist(file\_path):

"""

Check if the file path exists and is a valid file.

"""

return os.path.exists(file\_path) and os.path.isfile(file\_path)

**Function: is\_valid\_image(file\_path)**

* **Purpose**: Verifies whether a file is a valid image by checking its extension and using the **Pillow** library to ensure the file is not corrupted.
* **Parameters**:
  + file\_path: String representing the path to the image file.
* **Returns**: Boolean value (**True** if the file is a valid image, **False** otherwise).

**Mechanism:**

* **Step 1**: The file extension is checked to match common image formats such as .png, .jpg, .jpeg, .gif, and .bmp.
* **Step 2**: The PIL.Image class attempts to open the file and verify its content. If the image is invalid or corrupted, an exception (IOError, SyntaxError) is raised.

**Code:**

def is\_valid\_image(file\_path):

"""

Check if the file is a valid image file.

"""

if not file\_path.lower().endswith((".png", ".jpg", ".jpeg", ".gif", ".bmp")):

return False

try:

with Image.open(file\_path) as img:

img.verify()

return True

except (IOError, SyntaxError):

return False

**2.3 Custom Exception for Timeout**

A custom exception, TimeoutException, is defined to handle cases where the timeout decorator detects that the function has taken too long to execute. This ensures the error is caught and handled appropriately in the system.

**Class: TimeoutException**

* **Purpose**: Custom exception to indicate that a function has timed out.

**Code:**

class TimeoutException(Exception):

"""Customized TimeoutException"""

Pass

**3. Error Handling**

The utility functions are equipped with robust error handling mechanisms:

1. **Invalid Argument Handling**:
   * The system checks whether the provided file is a valid image file using is\_valid\_image().
   * If the file path is invalid or the image is corrupted, an error message is displayed, and the function returns early, preventing further execution.
2. **Timeout Handling**:
   * The timeout() decorator handles cases where API requests take longer than expected by raising a TimeoutException. This prevents the system from becoming unresponsive during long waits.
3. **General Exceptions**:
   * General exceptions in the decorated function are caught and re-raised to provide detailed error messages to the user.

**4. Integration with the Gemini AI Model**

**5. Conclusion**