AI Storyboarder - A Comprehensive Guide

Executive Summary

This document details the AI Storyboarder project, a web-based tool developed for the Hanze UAS CMGT AI Elective. The tool leverages OpenAI's GPT-3.5-turbo and DALL-E 3 APIs via a Flask backend and JavaScript frontend to automatically generate structured, multi-shot visual storyboards from user-provided text prompts, character descriptions, and style preferences. Key features include a Character Editor for visual consistency, interactive editing capabilities (regenerate, add, delete shots), and user controls for style and camera angles. This Toolbox covers the project's purpose, the underlying AI technologies (Transformers, Diffusion Models), system architecture, a detailed development journey including experimentation and iteration, setup instructions, code breakdown, real-world impact, limitations (cost, consistency, ethics, rate limits), and future enhancements. It aims to serve as a comprehensive guide for future students exploring AI applications in creative media.

1. Introduction

1.1 Project Overview: AI Storyboarder

This document serves as an in-depth guide designed for future Creative Media and Game Technology students at Hanze UAS. It details the process of understanding, recreating, and extending an innovative tool that leverages Artificial Intelligence (AI) to streamline the creation of visual storyboards. Whether you are new to AI or have prior experience, this guide provides the necessary technical insights, practical steps, and theoretical background.

What is an AI Storyboarder?

AI Storyboarder is an interactive web application designed to automatically generate visual storyboards from textual prompts. It integrates **OpenAI's GPT-3.5-turbo** for sophisticated text processing (generating structured scenes and shots, including descriptions, dialogue, emotions, camera angles, and shot types) and **DALL-E 3** for high-quality image generation based on these detailed prompts.

Key Features:

- **Structured Story Generation:** Transforms a simple story idea into a 3-scene storyboard, with each scene containing 2-3 distinct shots (frames).

- **Character Consistency:** Utilizes a dedicated "Character Editor" to define a main character (name, appearance, outfit, personality, motivation), ensuring this description is consistently applied across all generated scenes and shots.
- **Visual Style and Camera Control:** Allows users to select a global visual style (e.g., cinematic, realism, anime, comic) and a default camera angle.
- **Interactive Editing:** Enables users to regenerate entire scenes or individual shots, add new shots, and delete unwanted shots directly within the interface.
- **Download Functionality:** Provides options to download individual shot images.
- **User Experience:** Features a clean, responsive UI with dark mode support and loading indicators.

1.2 Purpose and Relevance

The Challenge: Traditional storyboarding is a crucial but often time-consuming and skill-intensive process in film, animation, and game development. It requires artistic ability and significant manual effort, creating bottlenecks in pre-production.

The Solution: AI Storyboarder addresses this challenge by automating the visualization process. It empowers creators, regardless of their drawing skills, to rapidly prototype narrative sequences, explore visual ideas, and communicate concepts effectively.

Benefits:

- **Efficiency:** Reduces storyboarding time from hours to minutes.
- **Accessibility:** Enables non-artists to create visual narratives.
- **Creativity Boost:** Provides AI-generated visuals that can spark new ideas.
- **Educational Value:** Serves as a practical example of applying advanced AI models (LLMs and diffusion models) to solve real-world creative problems, aligning perfectly with the goals of the AI Elective.

This project prepares students for an industry increasingly shaped by AI, demonstrating how these technologies can augment, rather than replace, creative workflows.

2. Understanding the Technology

This project integrates several key technologies. Understanding their roles, underlying principles, and why they were chosen is crucial for appreciating the system's design and limitations.

2.1 Core AI Technologies

1. GPT-3.5-turbo (Large Language Model - LLM):

- **Role:** Acts as the "scriptwriter" and "story structure engine." It takes the user's story idea and character description and breaks it down into a structured JSON format containing scenes and shots, complete with detailed descriptions, dialogue, emotions, camera angles, and shot types.
- **Underlying Technology (Transformers):** GPT-3.5-turbo is based on the Transformer architecture (Vaswani et al., 2017). Transformers use a mechanism called "self-attention" to weigh the importance of different words in the input sequence, allowing them to understand context, maintain coherence over long text passages, and generate highly relevant and structured output. This is critical for generating logical scene progressions and detailed shot descriptions.
- **Implementation Insight:** The system prompt engineering for GPT was iterated upon significantly (see Section 4) to force it to output reliable JSON and adhere to the scene → shots structure, including specific cinematic elements like camera angles and shot types.

2. DALL-E 3 (Text-to-Image Diffusion Model):

- **Role:** Acts as the "illustrator." It takes the detailed textual description of each *shot* (generated by GPT and refined in the backend) and creates a corresponding visual image.
- **Underlying Technology (Diffusion Models):** DALL-E 3 utilizes diffusion models (Ramesh et al., 2022). These models work by starting with random noise and gradually refining it, step-by-step, guided by the text prompt, until a coherent image matching the description is formed. They are trained on massive datasets of text-image pairs.

- **Implementation Insight:** Achieving visual consistency (especially for characters) across multiple shots generated independently is a major challenge with current diffusion models like DALL-E 3, as they lack inherent "memory" between generations. This project mitigates this by injecting consistent character descriptions and style prompts into every DALL-E API call. Explicitly instructing against multi-panel layouts was also necessary to prevent DALL-E from generating entire storyboard grids instead of single frames (e.g., avoiding the word "frame" in prompts).

2.2 Supporting Technologies

- **Flask (Python Backend):** A lightweight micro-framework used to create the web server. It handles requests from the frontend, orchestrates the calls to the OpenAI API (both GPT and DALL-E), processes the AI responses, manages caching, and implements rate-limiting logic (time.sleep).
- **HTML/CSS/JavaScript (Frontend):** Standard web technologies build the user interface. HTML structures the content, CSS styles it (including dark mode and responsive layout), and JavaScript handles user interactions (button clicks, input gathering), makes asynchronous Workspace requests to the Flask backend, and dynamically renders the generated storyboard.
- **Threading (Python):** Initially used in the backend to make parallel API calls for image generation, later switched to sequential calls with delays (time.sleep(12)) to manage OpenAI rate limits effectively after encountering 429 errors.

2.3 Comparative Analysis of AI Models

The selection of GPT-3.5-turbo and DALL-E 3 was made after considering several alternatives available in the rapidly evolving AI landscape (as of early 2025).

- **Text Generation (LLM):**
    - **GPT-3.5-turbo (Chosen):** Offers a strong balance between performance, cost-effectiveness, and API availability/reliability. It proved capable of generating the required structured JSON output with careful prompting.
    - *Alternative: GPT-4/GPT-4-turbo:* Provides higher quality text generation and potentially better adherence to complex instructions (like JSON formatting) but comes at a significantly higher API cost, making it less suitable for a student prototype with frequent testing.

- *Alternative: Open Source LLMs (e.g., Llama 3, Mixtral):* Offer cost advantages (potentially free if self-hosted) but often require more complex setup, lack robust official APIs comparable to OpenAI's, and might yield less consistent results for structured generation tasks without fine-tuning.
- *Alternative: Other Commercial LLMs (e.g., Claude, Gemini):* Offer competitive performance but API availability, pricing models, and ease of integration were considered less optimal than OpenAI's ecosystem for this specific project scope at the time.

- **Image Generation:**
  - **DALL-E 3 (Chosen):** Provides high-quality image generation directly via the OpenAI API, integrating seamlessly with the GPT text generation component. Its ability to follow relatively complex prompts including style and character elements was deemed sufficient for the prototype.
  - *Alternative: Stable Diffusion:* A powerful open-source model. Offers potential for greater control (especially with tools like ControlNet for pose/composition) and can be run locally (free, requires GPU) or via APIs (Replicate, Hugging Face). However, setup is more complex, and achieving visual consistency requires significant prompt engineering or fine-tuning (e.g., LoRAs), increasing development time beyond the project scope.
  - *Alternative: Midjourney:* Known for exceptionally high artistic quality but crucially lacks an official API, making direct integration into an automated tool like this infeasible without resorting to unstable or Terms-of-Service-violating methods (like Discord bots).
  - *Alternative: DALL-E 2:* Lower cost than DALL-E 3 but significantly lower image quality and poorer prompt adherence.

Justification: The combination of GPT-3.5-turbo and DALL-E 3 via the unified OpenAI API provided the most pragmatic balance of performance, ease of integration, accessibility, and cost for developing this AI Storyboarder prototype within the course constraints.

2.4 Advanced Prompt Engineering Techniques

Achieving reliable and structured output from both GPT and DALL-E required specific prompt engineering strategies:

- **System Role Prompting (GPT):** Instructing GPT to act as a "professional storyboard writer" (role: system message) helps prime the model for the desired output style and domain knowledge.
- **Structured Output Enforcement (GPT):** The system prompt explicitly defines the required JSON structure ("scene_number", "shots", "frame_number") and includes instructions like "Return ONLY valid JSON. No explanations." This significantly improved the reliability of parsing the output. Initial attempts without strict formatting often resulted in conversational text mixed with JSON, causing backend errors.
- **Context Injection for Consistency (GPT and DALL-E):**
  - The user-defined character_description is dynamically inserted into the system prompt for GPT and into *every* individual shot prompt sent to DALL-E.
  - The selected style and camera_angle are also embedded in DALL-E prompts using the STYLE_PROMPTS dictionary and direct parameter passing.
  - The GPT prompt was refined to encourage narrative flow ("coherent story with a clear beginning, middle, and end").
- **Negative Prompting / Instruction Tuning (DALL-E):** Early experiments revealed DALL-E sometimes generated meta-images (pictures *of* storyboards) or multi-panel layouts when prompted with terms like "storyboard frame." Prompts were revised to use "cinematic image" or "single cinematic frame" and explicitly forbid unwanted elements ("No collage. No multi-panel. No text.").
- **Detail Enhancement (DALL-E):** Prompts for DALL-E were structured to include multiple facets (Style, Character, Camera, Scene Description, Composition instructions) to maximize visual detail and adherence.

These techniques were developed iteratively based on observing AI outputs and debugging failures, demonstrating a practical application of prompt engineering principles.

3. System Architecture and Workflow

The AI Storyboarder operates as a client-server application with distinct frontend and backend components coordinating AI model interactions.

Workflow Steps:

1. **User Input:** The user enters a story idea, defines the main character using the Character Editor, selects a visual style, and sets a default camera angle via the index.html interface.
2. **Frontend Request:** script.js collects these inputs and sends an asynchronous Workspace request (POST) to the Flask backend endpoint /generate-storyboard.
3. **Backend Processing (GPT):** app.py receives the request. It constructs a detailed system prompt for GPT-3.5-turbo, including the user's inputs and instructions to generate a 3-scene, multi-shot storyboard structure in JSON format.
4. **Backend Processing (DALL-E):** The backend parses the JSON response from GPT. For each shot description, it constructs a specific prompt for DALL-E 3, incorporating the scene details, character description, style, and camera angle. It then calls the DALL-E API to generate an image for *each shot*, respecting rate limits with delays (time.sleep(12)).
5. **Backend Response:** Once all images are generated (or placeholders/errors are noted), the backend compiles the complete storyboard data (scenes with shots, including descriptions and image URLs) into a JSON array and sends it back to the frontend.
6. **Frontend Rendering:** script.js receives the storyboard data. The renderStoryboard function dynamically creates HTML elements (scene wrappers, shot cards with images, descriptions, buttons) and displays the complete storyboard in the #output div.
7. **User Interaction:** Buttons on each shot card allow the user to trigger further backend calls (/regenerate-shot, /add-shot) or perform client-side actions (Delete Shot, Download Image). The UI updates dynamically after each interaction.

3.2 Code Structure

This section details the role of each file in the project:

- **app.py (Python Backend):**
    - Uses the **Flask** framework to create the web server and API endpoints. CORS(app) allows the frontend (running on a different origin like file:///) to communicate with the backend server.
    - Integrates with the **OpenAI API** using the openai library.
    - Contains the core logic for **prompt engineering**:
        - generate_full_story(): Sends a system prompt to **GPT-3.5-turbo** to generate a structured JSON response containing scenes and shots, including details like title, setting, characters, action, emotion, dialogue, camera angle, and shot type. It also includes logic to parse the response and handle potential JSON errors.
        - generate_image(): Constructs detailed prompts for **DALL-E 3** for each shot, incorporating user-defined style, character description, camera angle, and scene context. It explicitly instructs against multi-panel layouts to ensure single-frame images.
    - Implements **in-memory caching** (CACHE) to store results of generate_image calls, reducing redundant API usage and costs for identical prompts.
    - Manages **API rate limits** by introducing a time.sleep(12) delay between DALL-E 3 calls, preventing 429 errors.
    - Uses **environment variables** (os.getenv("OPENAI_API_KEY")) to securely load the OpenAI API key.
    - Includes basic error handling for API calls and returns JSON responses to the frontend.
    - Defines **API endpoints** for main generation (/generate-storyboard) and interactive edits (/regenerate-scene, /regenerate-shot, /add-shot).
- **script.js (JavaScript Frontend Logic):**
    - Handles all client-side interactivity and communication with the backend.

- generateStoryboard(): Collects all user inputs from index.html (story idea, character details, style, camera angle) and sends them to the /generate-storyboard backend endpoint using the Fetch API.
- Manages the application state by storing the received storyboard data in the currentStoryboard JavaScript array.
- renderStoryboard(): Dynamically updates the HTML DOM to display the storyboard. It creates elements for scenes and shot cards, populating them with data (images, descriptions, etc.) from currentStoryboard. It also attaches event listeners to the interactive buttons.
- **Event Handlers for Buttons:** Functions associated with "Regenerate Scene," "Regenerate Shot," "Add Shot Below," "Delete Shot," and "Download Image" buttons. These functions either modify the currentStoryboard array directly (for delete) or make Workspace calls to the corresponding backend endpoints for regeneration or adding new shots, then call renderStoryboard to update the UI. Includes error handling and loading state management.
- downloadImage(): A utility function using the <a> tag download attribute trick to save images.

- **index.html (HTML Structure):**
  - Defines the main structure of the web page using standard HTML5 elements.
  - Includes input elements: <textarea> for story idea and character description, <select> for style and camera angle, <input> fields within the Character Editor section.
  - Contains buttons (<button>) linked to JavaScript functions via the onclick attribute (generateStoryboard, clearForm, generateCharacter, toggleDarkMode).
  - Defines the main container ( ) and the output area ( ) where the storyboard is rendered by script.js.
  - Includes inline <script> tags for simple utility functions: clearForm, toggleDarkMode, generateCharacter (which constructs the character description string).
  - Links the external script.js file. Contains inline <style> for CSS rules.

- **styles.css / Inline <style>:**
    - Defines visual styles using CSS selectors targeting HTML elements and classes.
    - Includes basic typography (font-family), background colors, text colors, margins, padding.
    - Styles for input elements (textarea, select, input), buttons (including :hover states).
    - Layout for scenes and shots: Uses display: flex and flex-wrap: wrap for .shots-container to arrange shot cards horizontally. Defines width: 30% for .scene in the initial layout attempts (later refined likely in inline styles or JS for the final scene > shots structure).
    - Styling for .shot-card (borders, padding, background).
    - **Dark Mode:** Implements theme switching using the .dark-mode class on the <body>. Overrides default colors for background, text, inputs, buttons, and cards when dark mode is active.
- **start.bat (Windows Script):**
    - A simple utility script for Windows users to easily start the Flask backend server.
    - Uses cd /d "%~dp0" to navigate to the script's directory.
    - Executes python app.py to run the Flask server.
    - Includes pause to keep the command prompt window open after execution, useful for seeing errors.

4. Development Journey: Iterations and Experimentation

The development of AI Storyboarder was an iterative process, driven by initial requirements, technical challenges, experimentation, and feedback incorporation. This section documents the key stages and learnings, addressing the assessment criteria regarding investigation, experimentation, and refinement.

Stage 1: Initial Concept and Proof of Concept (PoC)

- **Goal:** Validate the core idea: Can AI generate relevant images from a simple story prompt?
- **Implementation:** Basic Flask app linking directly to DALL-E 3. Simple HTML input, JS sends prompt, backend calls DALL-E, frontend displays 3 images.
- **Challenge:** Generated images were disconnected, random, and didn't form a narrative. DALL-E often misinterpreted "storyboard" or "frame" prompts, creating multi-panel images.
- **Experimentation (F1):** Tested various simple prompts. Compared "frame" vs "image" vs "shot" keywords in DALL-E prompts. Discovered "frame" keywords consistently led to undesirable meta-images of storyboards or multi-panel layouts. Established need for "single cinematic image" instruction.
- **Learning:** Direct text-to-image generation is insufficient for storyboarding. A structured approach involving text generation (scene breakdown) prior to image generation is necessary. Basic prompt variations are insufficient for narrative coherence.

Stage 2: Introducing GPT for Structure (Scene → Shots Architecture)

- **Goal:** Implement a proper storyboard structure (scenes containing multiple shots) and improve narrative coherence. Address initial feedback that the output wasn't a real storyboard.
- **Implementation:**
  - Introduced GPT-3.5-turbo (generate_full_story in app.py) to break the user prompt into 3 scenes, each containing 2-3 shots with details (description, camera, emotion, dialogue).
  - Instructed GPT to return structured JSON using specific system prompts.
  - Modified script.js (renderStoryboard) to display scenes and shots hierarchically based on the received JSON.
  - Updated DALL-E prompts to generate images per *shot*, not per scene.
- **Challenge:** GPT sometimes returned invalid JSON or fewer than 3 scenes, breaking the frontend. Visual consistency between shots was still poor. DALL-E rate limits (5 images/min) became a major bottleneck with 6+ image requests per storyboard.

- **Experimentation:** Refined GPT system prompts multiple times to enforce strict JSON output and the 3-scene structure. Implemented try...except blocks for JSON parsing in app.py. Experimented with parallel image generation using threading, observed 429 RateLimitError, then switched to sequential generation with time.sleep(12) to manage rate limits effectively.
- **Learning:** Structured output from LLMs requires robust prompting and error handling. API rate limits are a critical practical constraint requiring specific backend logic (sequential calls, delays). Visual consistency needs more than just structured text.

Stage 3: Enhancing Visual Consistency and User Control

- **Goal:** Address the major feedback point regarding inconsistent characters and styles. Give the user more control over the generation process.
- **Implementation:**
    - Added the "Character Editor" (index.html, inline JS generateCharacter) to allow users to define name, appearance, outfit, personality, and motivation.
    - Modified app.py to accept character_description and inject it into *every* GPT (for narrative context) and DALL-E (for visual appearance) prompt.
    - Added UI selectors (index.html) for style and default camera_angle, passing these parameters to the backend and into DALL-E prompts via the STYLE_PROMPTS dictionary.
- **Challenge:** While consistency improved significantly, DALL-E still sometimes varied character appearance slightly (e.g., minor changes in clothing details, facial features). Ensuring the combined prompt (scene + character + style + camera) remained effective and didn't confuse DALL-E required careful balancing.
- **Experimentation:** Tested the impact of detailed vs. brief character descriptions. Compared image results across the four styles (cinematic, realism, anime, comic) using the same story prompt to evaluate DALL-E's style adherence. Added basic in-memory CACHE to app.py to reduce redundant DALL-E calls during testing and save costs.
- **Learning:** Explicitly defining character and style in every prompt is the most effective way to achieve consistency with current models, though perfection is elusive. Caching API results is essential for efficient development and cost management.

Stage 4: Adding Interactivity and Refinements

- **Goal:** Transform the prototype into a more usable and flexible tool, allowing users to refine the generated storyboard interactively, addressing the need for iteration capabilities mentioned in the course description.
- **Implementation:**
    - Added buttons ("Regenerate Scene", "Regenerate Shot", "Add Shot Below", "Delete Shot") to each scene/shot card in the UI (script.js renderStoryboard).
    - Implemented corresponding backend endpoints in app.py (/regenerate-scene, /regenerate-shot, /add-shot) that receive existing scene/shot data, call the relevant OpenAI API(s), and return the updated object.
    - "Delete Shot" functionality implemented client-side in script.js by modifying the currentStoryboard array and re-rendering, avoiding unnecessary backend calls.
    - Added "Download Image" button using client-side JavaScript (downloadImage function).
    - Implemented UI refinements: Dark Mode toggle, "Clear Form" button, loading indicator.
- **Challenge:** Managing the application state (currentStoryboard in JS) reliably and ensuring the UI updates correctly and efficiently after each asynchronous edit operation. Handling potential errors during regeneration API calls gracefully.
- **Experimentation:** Developed the renderStoryboard function to act as the single source of truth for UI updates, simplifying state management. Refined error handling in Workspace calls within the button onclick handlers to provide user feedback (e.g., showing an error message in the output div). Tested UI responsiveness during edits.
- **Learning:** Client-side state management coupled with targeted backend API calls is effective for building interactive AI tools. A central rendering function simplifies UI updates after state changes.

- **Fail Log. Testing Insights:** During development, I encountered multiple issues that required careful experimentation to resolve:
    - **Inconsistent Character Appearance:**
      Early in testing, DALL-E 3 would sometimes make the main character's appearance change between shots (e.g., clothing color or hairstyle shifting slightly). For example, a "blonde detective" might suddenly get brown hair in shot 3. To counter this, I systematically varied the length and detail of the character description and compared outputs.

    - **Multi-Panel and Collage Images:**
      Using terms like "frame" or "storyboard shot" often resulted in unwanted multi-panel layouts. I documented several failed outputs. The solution was to shift prompts toward "single cinematic image" and add negative phrases like "no collage, no text".

    - **API Rate Limits & Errors:**
      In one test, trying to generate all images in parallel (multi-threaded) produced frequent 429 RateLimitErrors, and images would simply not appear. Switching to sequential requests with a 12-second delay made generation stable.

    - **UI Confusion:**
      Test users (and I myself) often didn't realize the app was working during long generations. This led to the addition of a visible loading spinner and "Please wait…" message.

    - **Prompt Sensitivity:**
      When prompts were too vague (e.g., "A detective chases a thief"), DALL-E produced random, generic images. More detailed descriptions and specific styles produced much more reliable results.

These issues are reflected in the project's change history and led directly to the current design.

Stage 5: Final Polishing and Documentation

- **Goal:** Prepare the project for submission, ensuring all deliverables (Toolbox, Video, Code) meet course requirements and demonstrate a high level of competence.
- **Implementation:**
    - Finalized UI layout (horizontal shots container within scenes), styling (styles.css

and inline), and responsiveness.

    - Wrote the comprehensive Toolbox document (this document 🙂), detailing technology, architecture, the iterative process, challenges, learnings, and setup instructions.
    - Created the Showcase Video script, ensuring it covers functionality, design, relevance, interactivity, technology, experimentation, limitations, and impact within the 10-minute time limit.
    - Organized project files (app.py, script.js, index.html, styles.css, start.bat) and created README.md for repository documentation.
- **Challenge:** Clearly articulating the complex development process, technical decisions, AI limitations, and experimentation outcomes in the Toolbox and video script in a manner accessible to future students. Ensuring all specific points from instructor feedback were demonstrably addressed.
- **Learning:** Effective documentation and presentation, explicitly mapping project work to assessment criteria and feedback, are crucial for demonstrating understanding and achieving high grades in an academic setting.

This iterative process, involving continuous experimentation with AI prompts, systematically addressing technical limitations (like rate limits and consistency), incorporating user control features, and directly responding to instructor feedback, was essential in evolving the AI Storyboarder from a basic concept to a robust, functional, and well-documented prototype.

5. Setup and Installation Guide

This guide helps future students set up and run the AI Storyboarder project locally.

5.1 Requirements

- **Python:** Version 3.8 or higher installed.
- **pip:** Python package installer (usually comes with Python).
- **Web Browser:** A modern browser like Chrome, Firefox, or Edge.
- **OpenAI API Key:** A valid API key from OpenAI with access to GPT-3.5-turbo and DALL-E 3 models. *Note: This typically requires a paid account!*
- **Code Editor:** Recommended: Visual Studio Code (VS Code).

5.2 Installation Steps

1. **Clone or Download the Project:**
    - If using Git: git clone https://github.com/arthaix/AI-Storyboarder
    - Otherwise, download the project files as a ZIP archive and extract them.
2. **Navigate to Project Directory:**
    - Open your terminal or command prompt (cmd).
    - Use the cd command to navigate into the project folder (e.g., cd AI-Storyboarder).
3. **Set Up Python Environment (Recommended):**
    - It's good practice to use a virtual environment:

    python -m venv venv

    Activate on Windows:
    .\venv\Scripts\activate

    Activate on macOS/Linux:
    source venv/bin/activate

4. **Install Dependencies:**
    - Install the required Python libraries:
      pip install flask flask-cors openai
5. **Set OpenAI API Key:**
    - The application reads the API key from an environment variable named OPENAI_API_KEY. You must set this variable before running the server.
    - **Windows (Command Prompt - Temporary for session):**
      set OPENAI_API_KEY=your_actual_api_key_here
    - **macOS/Linux (Terminal - Temporary for session):**
      export OPENAI_API_KEY=your_actual_api_key_here

- **Persistent Setup (Recommended):** Add OPENAI_API_KEY to your system's environment variables permanently through system settings.

5.3 Running the Application

1. **Start the Flask Backend Server:**
   - Make sure you are in the project directory in your terminal (and the virtual environment is activated, if used).
   - Run the Python script using the provided batch file (on Windows) or directly:
     # On Windows, double-click start.bat or run:
     start.bat
     # Or run directly on any OS:
     python app.py
   - You should see output indicating the server is running, typically on http://127.0.0.1:5000/. Keep this terminal window open.
2. **Open the Frontend:**
   - Navigate to the project folder in your file explorer.
   - Double-click the index.html file to open it in your default web browser.
3. **Use the Application:**
   - Enter a story idea, fill in the character details, select style/camera, and click "Generate Storyboard."
   - Interact with the generated storyboard using the editing buttons.

6. Code Breakdown

This section provides a more detailed explanation of each file's role and key code sections.

- **app.py (Python Backend):**
    - **Framework:** Uses **Flask** to handle web requests. CORS(app) allows the frontend (running on a different origin like file://) to communicate with the backend server.
    - **API Client:** Initializes the **OpenAI client** (openai.OpenAI(api_key=...)) using the API key loaded securely from environment variables (os.getenv).
    - **Prompt Templates:** Defines STYLE_PROMPTS dictionary for different visual styles.
    - **Core Logic - generate_full_story():**
        - Constructs a complex system prompt for **GPT-3.5-turbo** instructing it to act as a storyboard writer, generating a structured JSON (scenes array > shots array with detailed fields). Includes user's character description for context.
        - Calls the client.chat.completions.create method.
        - Parses the GPT response using json.loads(), with error handling.
    - **Core Logic - generate_image():**
        - Constructs the final prompt for **DALL-E 3**, combining style, character, camera, scene description, and explicit instructions for a single frame.
        - Implements **in-memory caching** (CACHE) based on prompt components to avoid re-generating identical images.
        - Includes time.sleep(12) to respect OpenAI's **rate limits** (5 images/minute).
        - Calls client.images.generate. Handles potential errors.
    - **API Endpoints:**
        - /generate-storyboard: Orchestrates the main workflow – calls generate_full_story, then iterates through each shot calling generate_image sequentially, and returns the final JSON.
        - /regenerate-scene: Re-generates images for all shots in a given scene.

        - /regenerate-shot: Re-generates image for a single shot.
        - /add-shot: Creates a new placeholder shot structure and generates its image.
    - **Server Start:** if __name__ == '__main__': app.run(debug=True) starts the Flask development server.
- **script.js (JavaScript Frontend Logic):**
    - **State Management:** Uses currentStoryboard = [] to hold the data for the displayed storyboard.
    - **Main Function - generateStoryboard():** Collects inputs, sends POST request to /generate-storyboard using Workspace, handles response, updates state, calls renderStoryboard. Includes loading/button disabling.
    - **Rendering - renderStoryboard():** Clears output, iterates through currentStoryboard, dynamically creates HTML for scenes and shot cards (using document.createElement), populates with data (title, description, image, etc.), attaches onclick listeners to buttons.
    - **Interaction Handlers:** Button clicks trigger Workspace calls to /regenerate-scene, /regenerate-shot, /add-shot (sending relevant data), or modify currentStoryboard directly (delete shot), followed by calling renderStoryboard to refresh UI.
    - **Utilities:** downloadImage(), plus simple inline functions in HTML for clearForm, toggleDarkMode, generateCharacter.
- **index.html (HTML Structure):**
    - Standard HTML5 structure. Links script.js. Includes inline <style>.
    - Main container (.container).
    - Input section: textarea for story, div.section for Character Editor (with input fields and generateCharacter button), select elements for camera and style.
    - Control buttons: "Generate Storyboard," "Clear," Dark Mode checkbox.
    - Output section: Loading div (#loading), Results title (#resultsTitle), main output container (#output) populated by script.js.
    - Footer.

    - Inline <script> for clearForm, toggleDarkMode, generateCharacter.
- **styles.css / Inline <style>:**
    - Defines visual styles: typography, colors, layout (flexbox for shots-container), borders, shadows.
    - Implements Dark Mode using .dark-mode class and CSS overrides for relevant elements (body, container, inputs, buttons, cards).
- **start.bat (Windows Script):**
    - Utility script for easy Flask server launch on Windows. Changes directory (cd), executes python app.py, and uses pause to keep the window open.

7. Real-World Impact, Limitations and Ethics

7.1 Value Proposition and Applications

- **Efficiency:** Dramatically reduces storyboarding time (minutes vs. hours).
- **Creativity Boost:** Offers AI-generated visual starting points and variations.
- **Accessibility:** Enables non-artists to visualize stories effectively.
- **Applications:** Film/Animation pre-production, Game Design narrative prototyping, Educational tool for AI and creative workflows, Content Creation brainstorming.

7.2 Performance Appraisal

Evaluating the performance of a generative AI tool like AI Storyboarder involves both qualitative and quantitative aspects:

- **Qualitative Assessment:**
    - *Narrative Coherence:* How well do the generated scenes and shots follow a logical story arc based on the user's prompt? (Assessed subjectively during testing).
    - *Visual Consistency:* How consistently are the character and style maintained across different shots and scenes? (Assessed subjectively; improved through character editor and prompt injection).
    - *Prompt Adherence (Image):* How accurately does the DALL-E 3 image reflect the detailed shot description (action, emotion, camera angle)? (Assessed subjectively).

    - *Usefulness:* Does the generated output provide a valuable starting point for a human storyboard artist or director? (Assessed based on project goals).

- **Quantitative Assessment:**
    - *Generation Time:* Measured the time taken from clicking "Generate" to displaying the full storyboard. Initial parallel implementation was faster but hit rate limits; sequential implementation with delays is slower (~1.5-2 minutes for 6 shots) but stable.
    - *API Success Rate:* Monitored backend logs for OpenAI API errors (rate limits, invalid requests, content policy flags). Implemented basic error handling and delays to improve stability.
    - *Cost Efficiency:* Tracked estimated API costs per storyboard generation. Implemented caching (CACHE in app.py) to significantly reduce costs during iterative testing or repeated identical requests.

While formal user testing with quantitative scoring wasn't conducted within the project scope, these appraisal methods were used iteratively during development to guide improvements, particularly in prompt engineering and system design to address coherence and performance issues.

7.3 Limitations and Challenges

- **Cost and Scalability (Financial):** OpenAI API usage incurs costs (~$0.12-$0.36+ per 6-shot storyboard). Scaling for heavy use requires budget or alternative models.
    - *Mitigation:* Implemented image caching. *Future:* Explore open-source models (Stable Diffusion).
- **Visual Consistency (Technical):** DALL-E 3 lacks perfect memory. Character appearance can still vary slightly between shots despite prompt engineering.
    - *Mitigation:* Detailed character descriptions injected into all prompts. *Future:* Explore techniques like LoRAs or visual consistency models if APIs become available.
- **AI Bias (Ethical):** Models can generate stereotypical depictions. Relates to course Module D discussions on algorithmic bias (e.g., Buolamwini).
    - *Mitigation:* Character Editor allows user definition. Prompts aim for neutrality. Critical review of output is necessary. *Future:* Bias detection tools, user flagging systems.

- **"Black Box" Problem (Ethical/Technical):** The exact reasoning behind GPT's scene structure or DALL-E's specific image generation is often opaque, making debugging and fine-tuning challenging. Relates to Module D discussions.
- **API Rate Limits (Technical):** OpenAI limits (5 DALL-E images/min) necessitated sequential generation with delays, impacting speed.
    - *Mitigation:* time.sleep(12) implemented in app.py.
- **Prompt Sensitivity:** Output quality is highly dependent on user input quality.

Although formal user testing was limited, I did share the prototype with several friends and received valuable feedback:

- “Sometimes the images don’t match my mental picture, especially the main character's face.”
- “It’s cool that I can regenerate a shot if I don’t like the image, but sometimes I want more control (like changing only the background or emotion).”
- “When the API was slow, I thought the app had frozen… the loading bar would really help.”

This feedback reinforced the importance of the Character Editor, detailed prompt design, and visible UI states. In future work, more structured user testing and collection of suggestions would further refine usability.

7.4 Reflections on the Development Process

1. **AI as a Powerful but Imperfect Tool:** The project demonstrated AI's capability for rapid creative generation but also its limitations in consistency and contextual understanding, reinforcing the need for the "human in the loop".
2. **Iteration is Non-Negotiable:** Addressing AI challenges requires constant cycles of prompt testing, architectural adjustments (e.g., scene -> shots structure), and feature refinement based on observed results and feedback. The three levels of iteration mentioned in the course (approach, implementation, insight) were all necessary.

3. **System Design Over Simple API Calls:** Initial feedback correctly identified that merely connecting APIs wasn't enough. Building a system with structured prompting, state management, caching, and error handling was key to creating a valuable tool and meeting the course objectives.

8. Conclusion

The AI Storyboarder successfully demonstrates the potential of integrating large language models and image generation AI to automate and enhance the creative storyboarding process. It provides a functional, interactive prototype that addresses a real-world need in creative industries, embodying the principles of researching, applying, and experimenting with AI.

This Toolbox provides future students with a comprehensive understanding of the underlying technologies (Transformers, Diffusion Models), the system architecture (Flask, JS Frontend, OpenAI API), the iterative development process including experimentation and addressing feedback, and the practical steps to recreate or extend the project. While limitations regarding cost, consistency, and bias exist and were actively managed, the project serves as a valuable learning experience and a robust foundation for further exploration at the intersection of AI and creative technology.

Looking back, there are several areas I would improve or approach differently:

- **More Structured User Testing:** I would conduct and document formal user testing earlier in development, to catch usability pain points and see how non-developers interact with the tool.

- **Visual Evidence:** Including more screenshots of failures and intermediate results would strengthen the documentation and help future students.

- **Advanced Features:** If I had more time, I would prototype image inpainting or minor edits, more bias detection tools, and perhaps a detailed PDF export.

Still, the project taught me the value of iteration and that even sophisticated AI tools require careful prompt engineering, user-centric design, and honest review of their own limits.

9. References

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... and Polosukhin, I. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems (NIPS), 30*. https://arxiv.org/abs/1706.03762
2. Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical Text-Conditional Image Generation with CLIP Latents. *arXiv preprint arXiv:2204.06125*. https://arxiv.org/abs/2204.06125
3. OpenAI. (2025). *API Reference*. OpenAI Documentation. Accessed May 8, 2025, from https://platform.openai.com/docs/
4. Pallets Projects. (2025). *Flask Documentation*. Accessed May 8, 2025, from https://flask.palletsprojects.com/

10. Glossary of Terms

- **AI (Artificial Intelligence):** Field focused on creating systems that perform tasks requiring human-like intelligence.
- **API (Application Programming Interface):** Rules allowing software components to communicate (e.g., frontend to backend, backend to OpenAI).
- **Backend:** Server-side logic handling data, API calls, and processing (Python/Flask in this project).
- **Cache:** Temporary storage for frequently accessed data (used here for generated images to save API calls/cost).
- **Character Editor:** UI component for defining the main character's attributes.
- **DALL-E 3:** OpenAI's AI model for generating images from text descriptions.
- **Diffusion Model:** AI technique used by DALL-E 3, generating images by refining noise based on a prompt.
- **Fetch API:** JavaScript interface for making asynchronous network requests to the backend.
- **Flask:** A Python micro-framework for building web applications and APIs.
- **Frontend:** The user interface of the web application (HTML, CSS, JavaScript).

- **GPT-3.5-turbo:** OpenAI's Large Language Model used for text generation (scenes, shots, dialogue).
- **JSON (JavaScript Object Notation):** Lightweight data-interchange format used for communication between frontend, backend, and APIs.
- **LLM (Large Language Model):** AI model trained on vast amounts of text data, capable of understanding and generating human-like language (e.g., GPT).
- **Prompt:** Text instruction given to an AI model to guide its output.
- **Prompt Engineering:** The process of designing effective prompts to elicit desired responses from AI models.
- **Rate Limit:** Restriction imposed by API providers on the number of requests allowed within a specific time period.
- **Scene:** A major unit in a storyboard, typically containing multiple shots, representing a specific location or part of the narrative.
- **Shot (Frame):** A single visual representation within a scene, defined by composition, camera angle, and action.
- **Storyboard:** A sequence of visual frames outlining the narrative progression of a film, animation, or game.
- **Transformer:** AI architecture (used by GPT) excelling at processing sequential data like text, using self-attention mechanisms.
- **Token (OpenAI):** Unit of text used for processing and billing by OpenAI models.