**Report: Trend Story Generation and Graph Building**

**Overview**

The system outlined consists of two primary components:

1. **Server-side application (server.py)** - This is a gRPC-based server that listens for requests and generates stories by processing trends. It also serves images related to the trends.
2. **Story Generation and Graph Building (StoryMaker.py)** - This file contains classes for generating trends' semantic graphs and producing story paths from them, enhancing the storytelling process with trend data.

### Full Execution Flow

1. **User Input**: Checkbox selections → generate\_btn.click() triggers get\_trend\_story().
2. **Request Creation**: TrendStoryRequest with selected tones/themes/styles/lang.
3. **gRPC Channel**: Client connects to f'{PC0\_IP}:{SERVER\_PORT}'.
4. **Remote Procedure Call**: stub.GetStory(request) sends request.
5. **Server Response**: Server returns story + image → decoded and shown in UI.
6. **Optional Dummy Section**: Static story titles and images are always displayed.

**1. server.py File**

The server.py file sets up a gRPC server to handle requests and generate responses, specifically designed for trend-based storytelling. Here’s a breakdown of the key sections:

* **Imports**: The necessary libraries like grpc, requests, json, StoryMaker, and others are imported to facilitate communication with external services, file handling, and graph processing.
* **SERVER\_PORT**: The server listens on port 50052, which can be adjusted if necessary.
* **TrendStoryServiceServicer Class**:
  + **GetStory Method**:
    - This method listens for incoming requests, processes trend data, and generates a response.
    - The trends are loaded from a trends.json file, which contains relevant trend data in a JSON format.
    - A test response is currently generated for the story by using a placeholder, but the logic can be expanded to call the StoryMaker class for story generation.
    - An image (clean\_trend\_graph.png) is read and returned as part of the response to the client.
* **gRPC Server Setup**:
  + The serve() function sets up the server, adds the service handler (TrendStoryServiceServicer), and listens for incoming requests. Once started, the server remains running and awaits client interactions.

**2. StoryMaker.py File**

The StoryMaker.py file provides functionality to process trends, create graphs, and generate stories. It utilizes libraries like networkx and matplotlib to build graphs and visualize relationships between trends.

* **Class 1: CleanTrendGraphBuilder**
  + **Purpose**: This class builds a graph representing the relationships between trends based on textual and metadata similarity.
  + **Graph Building**:
    - A TF-IDF vectorizer is used to compute text similarity between trends, generating a cosine similarity matrix.
    - Additional metadata like info\_gain, importance, and relevance are considered to enhance the connection strength between trends.
    - A graph is created using the networkx library, where trends are represented as nodes, and edges represent the relationships between them, weighted by similarity.
  + **Graph Plotting**:
    - A method for plotting the graph is provided, which uses matplotlib to visualize nodes and edges. The edges are colored based on the source type (e.g., YouTube, Google) to enhance the clarity of the relationships.
  + **Story Path Extraction**:
    - A method is provided to extract a logical sequence of trends that can form a story path. This path is extracted based on the graph's structure and weighted edges, prioritizing important trends.
* **Class 2: MessyTrendGraphBuilder**
  + **Purpose**: This class builds a more detailed semantic graph based on multiple factors such as keyword overlap, category match, date proximity, and sentiment similarity.
  + **Key Methods**:
    - **calculate\_connection\_weight()**: Combines multiple factors to calculate the weight of a connection between two trends.
    - **determine\_edge\_color()**: Colors edges based on the source of the connected trends (e.g., YouTube, Google).
    - **build\_graph()**: Builds the graph using the above methods to calculate connections between trends. If the connection weight exceeds a threshold, an edge is added to the graph.
  + **Graph Plotting**:
    - Similar to the CleanTrendGraphBuilder class, this class also provides functionality to visualize the graph. The edges are colored based on the source of the trends they connect.
* **General Functionality**:
  + **Trend Data Processing**: Both classes are designed to process trend data and build graphs that represent the relationships between trends based on various factors. These graphs are intended to be used for generating stories based on the trends' relationships.
  + **Story Path Extraction**: Both classes feature methods for extracting a logical "story path" through the graph, connecting trends that are related to each other based on their metadata and textual similarities.

**3. Integration of Files**

The server-side implementation (server.py) and the graph-building/story-generation logic (StoryMaker.py) are closely integrated:

* The server receives requests, which likely contain information about tones, themes, and styles.
* Based on this request, the server processes the trend data (from trends.json), creates a graph using one of the StoryMaker.py classes, and generates a story.
* The server can also send an image of the trend graph to the client to visually represent the relationships between the trends.

**Overall Workflow: From Client to Server**

1. **User selects options** (Tone, Theme, Style, Language) from the Gradio interface.
2. On clicking **"Generate via gRPC"**, the client:
   * Combines user selections into a request object.
   * Sends a TrendStoryRequest via gRPC to the server (running on PC0).
3. The **gRPC server** processes the request and returns:
   * A story as a string.
   * A generated image (trend graph) as raw bytes.
4. The **client decodes** the image bytes and displays both story and image in the Gradio interface.

**Client-Side Overview**

#### UI with Gradio

* Built using gr.Blocks() for a responsive layout.
* Includes:
  + CheckboxGroups for tones, themes, styles, language.
  + A Button to trigger story generation.
  + A Textbox and Image element to show server output.
* Additional feature: displays **dummy stories and images** in a scrollable HTML container for visual appeal.

#### 🎯 Core gRPC Function: get\_trend\_story

* **Input**: Four lists (tones, themes, styles, language).
* **Validation**: Ensures each category has at least one selected item.
* **Processing**:
  + Creates a gRPC channel to the IP and port of the server.
  + Constructs a TrendStoryRequest using comma-joined strings from selected lists.
  + Sends the request using the stub’s GetStory() method.
* **Response Handling**:
  + Receives response.response (text story).
  + Decodes response.image\_data (image in bytes) into a PIL image.
* **Output**: Returns story and image to Gradio.

#### Helper Function: create\_dummy\_image(title)

* Creates placeholder images with centered text for UI purposes.

### TrendStory\_pb2\_grpc.py: gRPC Client Stub

This file is auto-generated by grpcio-tools from your .proto file. It contains:

#### 🔹 TrendStoryServiceStub

* Used by the client to send requests.
* self.GetStory: Maps to rpc GetStory (TrendStoryRequest) returns (TrendStoryResponse) in .proto.

#### 🔹 TrendStoryServiceServicer

* Used on the server side (not invoked in client.py).
* Defines the function GetStory() which should be implemented in server.py.

### Assumptions & Requirements

* The gRPC server (server.py) must be **running on PC0**.
* Both machines (PC0 and PC1) must be on the **same local network**.
* Port 50052 must be **open** and **not blocked** by firewall.
* TrendStory\_pb2.py and TrendStory\_pb2\_grpc.py must be up-to-date with .proto.

**Testing**

To ensure the reliability and correct interaction between the gRPC client and server components of the TrendStory Generator system, unit tests were implemented using Python's unittest framework. These tests validate both the correctness of gRPC binding and the expected behavior of client-server communication.

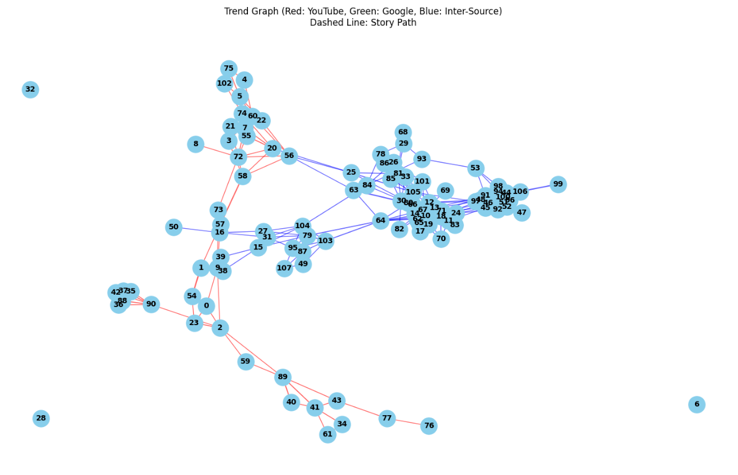
### ****Test Case Workflow Summary****

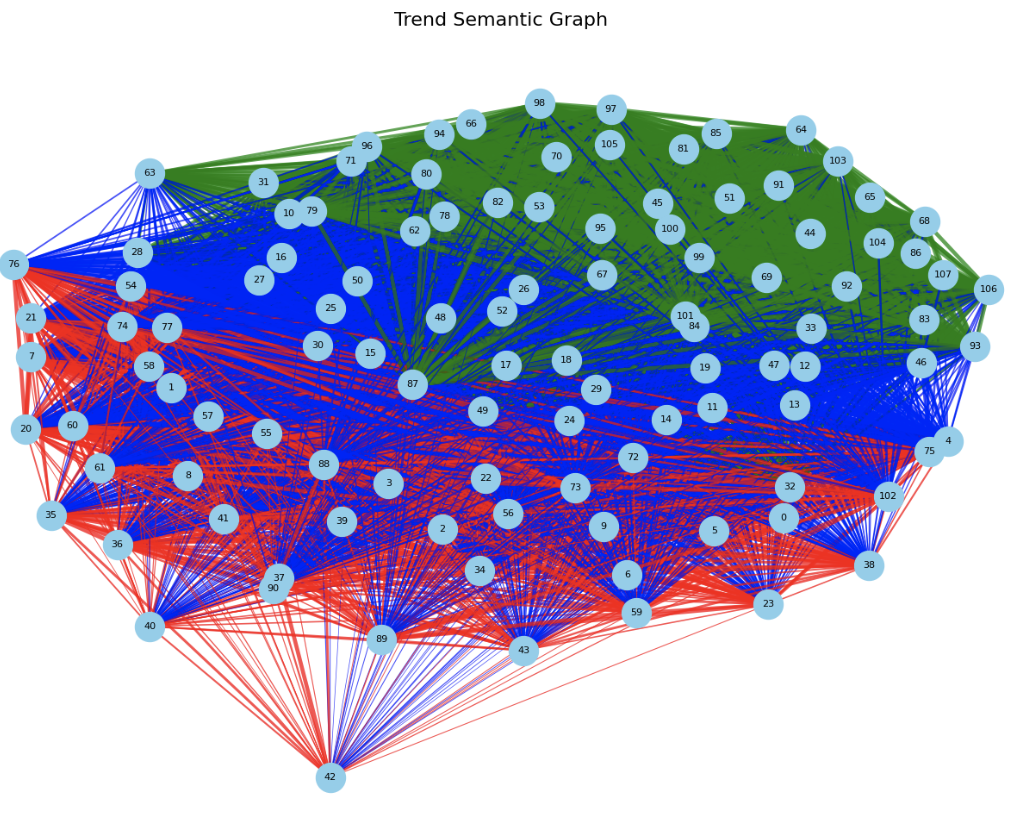
The following key aspects are tested:

| **Test Name** | **Purpose** |
| --- | --- |
| test\_stub\_creation | Verifies that the client stub was properly generated from the .proto file and contains the GetStory() method. |
| test\_servicer\_base\_class | Ensures the unimplemented server method correctly raises NotImplementedError and sets the gRPC error code. |
| test\_add\_servicer\_to\_server | Confirms that the gRPC server correctly binds the service implementation without errors. |
| test\_experimental\_get\_story\_method | Uses grpc.experimental (if available) to simulate a client call using TrendStoryService.GetStory static method and validates non-null response. |

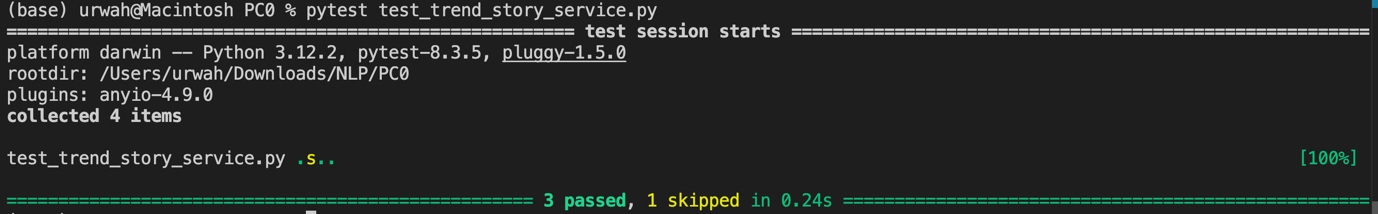
The test cases validate the robustness of the gRPC setup and ensure that both the auto-generated code and custom service bindings perform as expected. The fallback behavior in the base service class, stub correctness, and server-service integration were all verified. The optional experimental test also demonstrates compatibility with advanced gRPC usage when supported.

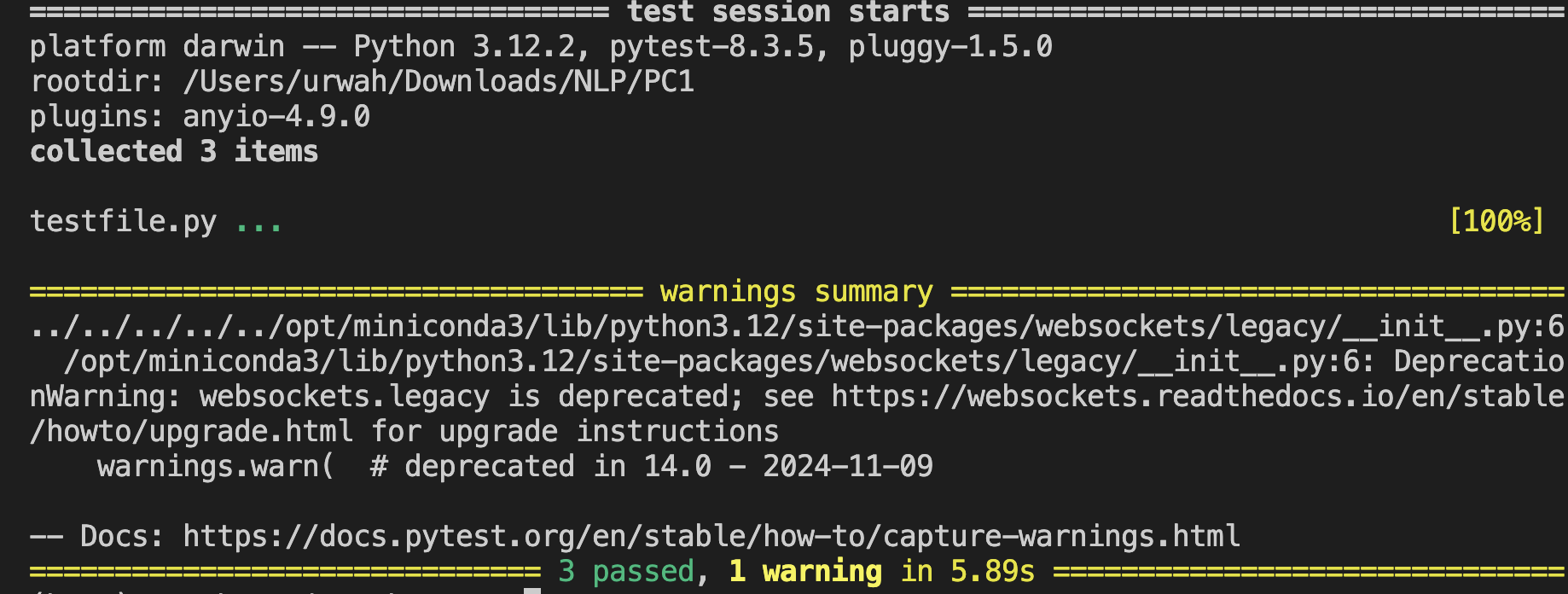
**Outputs**

****

****

**Test Cases Passing**

****

****

### ****Conclusion****

The TrendStory Generator system exemplifies an efficient client-server architecture using gRPC for real-time communication between a user-facing UI and a backend AI service. By integrating **Gradio** for the frontend and **gRPC** for RPC-based interaction, the system enables seamless input capture, rapid AI-driven story generation, and image visualization.

The modularity of the system—split across trendstory.proto, trendstory\_pb2.py, trendstory\_pb2\_grpc.py, client.py, and server.py—ensures high scalability, maintainability, and performance. The gRPC protocol significantly reduces latency and ensures type-safe message passing, while Gradio simplifies UI deployment for non-technical users.

This architecture can be easily extended for more complex storytelling features, such as multimedia support, feedback learning, and integration with trending data APIs, making it a solid foundation for future innovations in generative storytelling.