Building agentic workflows with LangGraph and Granite

Modern AI systems are evolving beyond simple prompt-response interactions. Today's AI agents can perform structured, multistep reasoning, decision-making and coordinate complex tasks autonomously. This emerging capability is known as an agentic workflow—a powerful shift in machine learning where agents operate through a series of logical steps to solve problems more effectively.

In this tutorial, we'll explore how to build such AI agentic workflows by using two key tools: LangGraph, a framework for constructing graph-based reasoning paths and IBM® Granite® models, a robust model that complements this structure. Each step in the workflow—called a "node"—is handled by an agent, typically powered by large language models. These agents move between states based on model outputs or conditional logic, forming a dynamic, decision-driven graph.

To bring these agentic workflows to life, we'll take a closer look at two essential components: LangGraph and the Granite model.

Understanding LangGraph

A framework for scalable, AI-driven workflows

LangGraph is a powerful framework designed to streamline the development of AI-driven workflows by representing AI models as stateful agents within a computational graph. It enables developers to build scalable, modular systems where each behavior or decision point is defined as a node in the graph.

With LangGraph, you can:

- Define each agent behavior as a distinct node
- Use algorithms or model outputs to determine the next step
- Pass state between nodes to preserve memory and context
- Visualize, debug and control the flow of reasoning with ease

Multi-agent systems and frameworks like LangGraph, when applied to generative AI (gen AI) tasks, typically structure task execution as either sequential or conditional workflows. Whether you are working with LangChain, IBM Granite models, OpenAI's GPT models or other artificial intelligence tools, LangGraph helps optimize your workflow for better scalability and performance.

Key components of LangGraph for automating complex workflows

LangGraph introduces a modern approach to orchestrating AI technologies by breaking down complex workflows into modular, intelligent components. Unlike traditional automation or Robotic process Automation (RPA), LangGraph enables dynamic, context-aware task execution by using real-time logic and memory. Here are the four key components that power this framework:

- Nodes: Nodes represent individual units of logic or action, such as calling an AI tool, querying data sources or performing a specific task. They are ideal for automating repetitive tasks within larger business processes.
- Edges: Edges define the flow between nodes, guiding how tasks are connected and executed. This structure supports flexible decision-making processes and allows workflows to adapt based on outcomes.
- Conditional edges (cyclical graphs): Cyclical graphs enable loops and conditional branching, allowing the system to revisit nodes based on logic or model outputs. This ability is crucial for handling iterative tasks and making informed decisions in dynamic environments.
- State (stateful graphs): The state acts as shared memory, preserving context and enabling the use of real-time data across nodes. This ability allows LangGraph to go beyond static flows and support adaptive, intelligent advancements in workflow automation.

Together, these components allow LangGraph to transform how organizations design and execute AI-driven workflows—bridging the gap between AI tools and real-world business processes.

Granite model—lightweight LLM for real-world problem solving

Granite-4.0-Tiny-Preview, developed by IBM Research®, is a lightweight yet capable open source language model designed to solve complex problems and practical natural language processing (NLP) tasks. While it's smaller than commercial models like GPT-4, Granite is fast, efficient and fully compatible with Hugging Face—making it a great choice for developers seeking operational efficiency without sacrificing performance.

Granite excels in:

- Intent classification–identifying user goals in chatbots or task-based systems
- Creative generation–producing summaries, dialog or short-form content
- Reasoning and summarization–ideal for workflows involving RAG or data analysis

In this tutorial, the Granite model plays a key role in various stages of the agentic workflow, supporting both problem solving and content generation. Its lightweight design makes it suitable for real-world applications where human intervention might be limited, and where scalable design patterns are essential for building robust AI solutions across diverse datasets and providers.

Use case

In this tutorial, we will build an agentic workflow that acts as a creative assistant for writing short animated screenplays.

Objective

Given a story idea from the user, the agent will:

- Identify the genre and tone suitable for the story
- Generate a brief plot outline
- Expand it into a key scene (for example, climax or turning point)

- Write dialog for that scene in screenplay format

This use case is designed to showcase both the reasoning and generative capabilities of a language model, structured through LangGraph's compositional workflow.

How the workflow operates

Each of the following steps is implemented as a LangGraph node:

- **User input:** The user provides a high-level story idea to initiate the workflow.
- **Genre detection *(node_name - select_genre):*** An LLM analyzes the input to infer the appropriate genre and tone for the story.
- **Outline generation *(node_name - generate_outline):*** The LLM generates a short plot summary based on the selected genre.
- **Scene writing *(node_name - generate_scene)*:** The LLM writes a pivotal scene in prose, bringing the story to life.
- **Dialog writing *(node_name - write_dialogue)***: The LLM rewrites the scene as formatted screenplay dialog, suitable for production or further editing.

These nodes are connected sequentially into a LangGraph, and the model moves through them while carrying forward a mutable state dictionary.

This workflow strikes a balance between creative generation and structural planning. It demonstrates:

- LLM coordination through LangGraph
- Multistep storytelling with minimal manual intervention
- Creative automation in a domain where human imagination is essential

It also scales well, you can easily extend it by adding revision steps, multiple scene generators or even character-based branching.

Prerequisites

You need an IBM Cloud® account to create a watsonx.ai® project.

Steps

Step 1. Set up your environment

While you can choose from several tools, this tutorial walks you through how to set up an IBM account to use a Jupyter Notebook.

1. Log in to watsonx.ai by using your IBM Cloud account.
2. Create a watsonx.ai project. You can get your project ID from within your project. Click the **Manage** tab. Then, copy the project ID from the **Details** section of the **General** page. You need this ID for this tutorial.
3. Create a Jupyter Notebook.

This step opens a notebook environment where you can copy the code from this tutorial. Alternatively, you can download this notebook to your local system and upload it to your watsonx.ai project as an asset. To view more Granite tutorials, check out the IBM Granite Community. This tutorial is also available on GitHub

Step 2. Set up watsonx.ai Runtime service and API key

1. Create a watsonx.ai Runtime service instance (choose the Lite plan, which is a free instance).
2. Generate an application programming interface key (API Key).
3. Associate the watsonx.ai Runtime service to the project that you created in watsonx.ai.

Step 3. Installing the required libraries

This cell installs the core libraries required to use the IBM Granite model hosted on Hugging Face:

- transformers: This is the main library for loading and interacting with pretrained language models, includinggranite-4.0-tiny-preview .
- accelerate: Helps with efficient model loading and device placement, especially useful for using GPUs in a seamless manner.

The-q flag runs the installation quietly, suppressing verbose output for a cleaner notebook interface. These libraries are essential for downloading the model and handling inference efficiently in this tutorial.

Note: If you are running this tutorial in a virtual environment and do not have langgrapg preinstalled, use pip install langgraph to install it in your local environment.
!pip install -q transformers accelerate

Step 4. Importing the required libraries

This cell imports all the core libraries needed for building and running the agentic workflow:

AutoTokenizer andAutoModelForCausalLM fromtransformers : Used to load the Granite model and tokenize input prompts for generation.

torch : Provides GPU acceleration and tensor operations required for model inference.

time: Enables optional time tracking and performance monitoring.

StateGraph andEND fromlanggraph.graph : These are used to define and compile the agentic workflow graph.

IPython.display ,base64: Used to render the output neatly and enable optional download features for generated content in Jupyter Notebooks.

Together, these imports prepare the environment for model interaction, workflow structuring and output presentation.
Import libraries from transformers import AutoTokenizer, AutoModelForCausalLM import torch import time from langgraph.graph import StateGraph, END from IPython.display import display, HTML import base64

Step 5. Loading the Granite model and tokenizer

This cell loads IBM'sgranite-4.0-tiny-preview model and its corresponding tokenizer from Hugging Face:

model_id: Specifies the identifier for the pretrained model hosted on Hugging Face.

AutoTokenizer.from_pretrained(model_id): Loads the tokenizer associated with the Granite model. The tokenizer is responsible for converting human-readable text into input tokens for the model.

AutoModelForCausalLM.from_pretrained(...): Loads the actual language model for causal (that is, generative) language modeling. This model can predict and generate text outputs based on the input.

Thetorch_dtype=torch.float32 argument explicitly sets the data type to float32, which is more memory efficient and broadly compatible—especially useful in GPU-constrained environments.

This step effectively initializes the Granite model as the **"reasoning engine"** behind our agentic workflow.
#Load Granite-4.0-Tiny-Preview model and tokenizer from Hugging Face model_id = "ibm-granite/granite-4.0-tiny-preview" tokenizer = AutoTokenizer.from_pretrained(model_id) model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float32)

Step 6. Utility function to generate text with Granite

This function,generate_with_granite , wraps the text generation process by using the Granite model. It accepts the prompt that is the input text to guide the model's response.

max_tokens : The maximum number of tokens to generate in the output (default: 200).

use_gpu : A flag indicating whether to run inference on the GPU (if available).

Key details:

device = torch.device(...): Dynamically selects GPU (cuda) if requested and available; otherwise, it defaults to CPU.

model.to(device): Loads the model onto the appropriate device just-in-time, helping conserve memory.

tokenizer(prompt, return_tensors="pt"): Converts the input string into token tensors for model processing.

model.generate(...): Starts text generation with sampling strategies including:

- do_sample=True: Enables randomness for more creative outputs.
- temperature=0.7 andtop_p=0.9: Control the diversity of generated text.

tokenizer.decode(...): Converts generated tokens back into readable text, removing any special tokens.

This function will be reused throughout the agentic workflow to invoke the Granite model at various decision or generation nodes.
def generate_with_granite(prompt: str, max_tokens: int = 200, use_gpu: bool = False) -> str: device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else "cpu") # Move model to device

```
only at generation time     model.to(device)     inputs = tokenizer(prompt,
return_tensors="pt").to(device)     outputs = model.generate(         **inputs,
        max_new_tokens=max_tokens,         do_sample=True,         temperature=0.7,         top_p=0.9,
        pad_token_id=tokenizer.eos_token_id     )     return tokenizer.decode(outputs[0],
skip_special_tokens=True).strip()
```

Step 7. Create node for selecting genre and tone *(node-1)*

This function,select_genre_node , defines the first node in our agentic screenplay generation workflow. It uses the Granite model to determine the genre and tone of the story based on the user's input.

Input:state , a dictionary that includes"user_input" (for example, "I want to write a whimsical fantasy story for children").

Prompt construction: The prompt asks the model to:

- Act as a creative assistant.
- Analyze the user's input.
- Recommend a genre and tone by using a specific output format:Genre: <genre> andTone: <tone>

Text generation: The prompt is passed to thegenerate_with_granite() utility function to generate a creative response.

Output parsing: A simple loop extracts the genre and tone from the model's response based on line prefixes ("Genre:" and"Tone:").

State update: The extractedgenre andtone values are inserted back into the state dictionary that will be passed to the next node.

This node acts as the creative classifier, enabling subsequent nodes to generate contextually aligned outlines, structures and scenes by using genre and tone as foundational parameters.

```
def select_genre_node(state: dict) -> dict:     prompt = f""" You are a creative assistant. The user wants
to write a short animated story. Based on the following input, suggest a suitable genre and tone for the
story. User Input: {state['user_input']} Respond in this format: Genre: <genre> Tone: <tone> """.strip()
   response = generate_with_granite(prompt)     # Basic parsing of output     genre, tone = None, None
   for line in response.splitlines():         if "Genre:" in line:             genre = line.split("Genre:")[1].strip()
      elif "Tone:" in line:             tone = line.split("Tone:")[1].strip()     # Update state     state["genre"] =
genre     state["tone"] = tone     return state
```

Step 8. Create node for generating the plot outline *(node - 2)*

Thegenerate_outline_node function defines the second node in the screenplay generation workflow. It builds upon the selected genre and tone to generate a concise plot outline for the story.

Input: The function receives the state dictionary containing:

- user_input: The original story idea.
- genre: Selected in the previous node.
- tone: Selected in the previous node.

Prompt construction: The model is instructed to:

- Act as a creative writing assistant.
- Consider the user's idea, genre and tone.
- Generate a brief plot outline (3–5 sentences) suitable for a short animated screenplay.

Text generation: The prompt is sent togenerate_with_granite() with a highertoken limit (max_tokens=250) to allow space for a multisentence outline.

State update: The generated plot outline is added to the state under the key"outline" , ready for use in the next phase of structure expansion.

This node translates abstract creative intent into a **narrative sketch**, providing a scaffold for the detailed three-act structure that follows. It ensures that downstream nodes work from a coherent, imaginative baseline.
def generate_outline_node(state: dict) -> dict: prompt = f""" You are a creative writing assistant helping to write a short animated screenplay. The user wants to write a story with the following details: Genre: {state.get('genre')} Tone: {state.get('tone')} Idea: {state.get('user_input')} Write a brief plot outline (3–5 sentences) for the story. """.strip() response = generate_with_granite(prompt, max_tokens=250) state["outline"] = response return state

Step 9. Create a node for generating a key scene from the outline *(node - 3)*

Thegenerate_scene_node function defines the third step in the workflow where the plot outline is transformed into a rich, narrative scene. This scene serves as a vivid dramatization of a turning point in the story, moving the idea from summary to storytelling.

Input: The node takes the state dictionary, which now includes:

- genre andtone
- Theplot outline from the previous node

Prompt construction: The model is instructed to:

- Act as a screenwriter
- Generate a turning point or climax scene by using the story'sgenre ,tone andoutline
- Write in prose format to preserve **readability** and **animation-friendly** description

Scene requirements:

- Vivid and descriptive (fitting for animation)
- Central to the emotional or narrative arc (for example, discovery, conflict or resolution)

Text generation:generate_with_granite is called withmax_tokens=300 , allowing enough space for immersive scene-building.

State update: The generated scene is added to the state dictionary under the"scene" key.

It introduces narrative immersion and visual storytelling to the workflow. Instead of merely summarizing the story, this node **brings it to life** with sensory and emotional detail—essential for scripting animated shorts.
def generate_scene_node(state: dict) -> dict: prompt = f""" You are a screenwriter. Based on the following plot outline, write a key scene from the story. Focus on a turning point or climax moment.

Make the scene vivid, descriptive, and suitable for an animated short film. Genre: {state.get('genre')} Tone: {state.get('tone')} Outline: {state.get('outline')} Write the scene in prose format (not screenplay format). """.strip() response = generate_with_granite(prompt, max_tokens=300) state["scene"] = response return state

Step 10. Create node for writing character dialog in screenplay format *(node - 4)*

Thewrite_dialogue_node function defines the fourth creative step in the storytelling workflow: converting a narrative scene into **formatted character dialog**. This step bridges the gap between prose and **screen-ready script writing**, giving characters a voice.

Input: The node expects statescene to contain a vivid story moment (typically a turning point or climax).

Prompt construction: The model is guided to:

- Act as a dialog writer
- Extract and adapt dialog from the scene
- Format the output in screenplay style by using:CHARACTER: Dialogue line

Guidelines for the dialog:

- Keep it short and expressive
- Ensure it’s appropriate for animation (visual, emotional, concise)
- Invent names if needed for clarity

Generation: Thegenerate_with_granite() call usesmax_tokens=300 , which balances expressiveness with brevity—ideal for short animated scripts.

State update: The dialog is saved into the state under thedialogue key for future steps (for example, display or editing).
def write_dialogue_node(state: dict) -> dict: prompt = f""" You are a dialogue writer for an animated screenplay. Below is a scene from the story: {state.get('scene')} Write the dialogue between the characters in screenplay format. Keep it short, expressive, and suitable for a short animated film. Use character names (you may invent them if needed), and format as: CHARACTER: Dialogue line CHARACTER: Dialogue line """.strip() response = generate_with_granite(prompt, max_tokens=300) state["dialogue"] = response return state

Step 11. Adding progress reporting to each node

This helper functionwith_progress is designed to wrap each workflow node with real-time progress indicators. It doesn't alter the logic of the original function—it simply tracks and prints execution status and timing to make long workflows more transparent.

Function purpose: Wrap a node (for example, generate_scene_node) with a decorator that logs:

- Which step is running
- Step index and total count
- How long the step takes

Parameters:

- fn : The actual node function (for example, write_dialogue_node)
- label : A human-readable label for that function
- index : Step number in the sequence (for example, 2)
- total : Total number of steps in the workflow

Internal wrapper:

- Prints a starting message
- Records start time
- Calls the original function
- Prints a completion message with elapsed duration

Returns: A modified version of the function that adds **progress messages** but otherwise behaves identically.

As workflows grow, it becomes important to **track which step is being executed**, especially if some steps (like generation or editing) take longer or might cause issues such as memory overload. This progress wrapper ensures **transparency** and is helpful for **debugging** and **runtime diagnostics**.
Wrap with progress reporting def with_progress(fn, label, index, total): def wrapper(state): print(f"\n[{index}/{total}] Starting: {label}") start = time.time() result = fn(state) duration = time.time() - start print(f"[{index}/{total}] Completed: {label} in {duration:.2f} seconds") return result return wrapper

Step 12. Defining the LangGraph workflow

This cell defines the workflow logic for generating a short animated story by using LangGraph, a compositional graph-based programming framework designed for LLM workflows. Each step in the graph represents a creative task and they are executed in a specific sequence to produce the final screenplay.

Components of the workflow: StateGraph(dict) - Initializes the workflow with a dictionary-based state (that is, the working memory is a dict passed from node to node).

Node registration with progress tracking: Each step (genre selection, outline generation, scene writing, dialog writing) is added as a node with the with_progress() wrapper - graph.add_node("select_genre", with_progress(select_genre_node, "Select Genre", 1, 4))

This approach ensures that each node logs its runtime and progress when executed.

Workflow edges (node sequencing): The sequence of the creative pipeline is clearly defined: select_genre → generate_outline → generate_scene → write_dialogue

set_entry_point() andset_finish_point(): These define the start and end nodes of the workflow.

graph.compile(): Compiles the workflow into a runnable form (workflow) that can now be invoked with an initial state.

This structure enables a **modular**, **readable** and **debuggable** LLM workflow. Each stage in your creative process is isolated, can be profiled separately and can later be swapped or extended (for example, adding a “revise scene” step or “summarize output” node). Theworkflow is now ready to run with a prompt and will execute your creative pipeline step-by-step, showing live progress.

```
# Define LangGraph graph = StateGraph(dict) graph.add_node("select_genre",
with_progress(select_genre_node, "Select Genre", 1, 4)) graph.add_node("generate_outline",
with_progress(generate_outline_node, "Generate Outline", 2, 4)) graph.add_node("generate_scene",
with_progress(generate_scene_node, "Generate Scene", 3, 4)) graph.add_node("write_dialogue",
with_progress(write_dialogue_node, "Write Dialogue", 4, 4)) graph.set_entry_point("select_genre")
graph.add_edge("select_genre", "generate_outline") graph.add_edge("generate_outline",
"generate_scene") graph.add_edge("generate_scene", "write_dialogue")
graph.set_finish_point("write_dialogue") workflow = graph.compile()
```

Step 13. Running the LangGraph workflow and displaying the output

This final code cell is where you run the **complete creative workflow** and display the results of each stage of story generation.

initial_state: This command is the starting point of the workflow, where the user provides a creative idea or theme. The user_input serves as the seed for the story pipeline.

final_state: This command triggers the full LangGraph pipeline. The input is passed through each registered node (select_genre, generate_outline, generate_scene, and write_dialogue) in sequence.

Displaying results: The final state dictionary now contains keys populated by various nodes:

- "genre": The identified genre based on user input.
- "tone": The tonal quality of the story.
- "outline": a 3–5 sentence plot summary.
- "scene": A vivid turning-point scene written in prose.
- "dialogue": Character dialog formatted as a screenplay.

This section demonstrates how user intent is transformed into a complete miniscript through a step-wise, modular LLM workflow. It’s an end-to-end creative pipeline that is interactive, interpretable and customizable.

Note: The code will take approximately 15–17 mins to run if you are using GPU or TPU. It will take around 65–70 mins time on a local virtual environment to run and generate the output based on the infrastructure used to run the cede.

```
# Run workflow initial_state = {     "user_input": "I want to write a whimsical fantasy story for
children about a lost dragon finding its home." } final_state = workflow.invoke(initial_state) # Display
Results print("\n=== Final Output ===") print("Genre:", final_state.get("genre")) print("Tone:",
final_state.get("tone")) print("\nPlot Outline:\n", final_state.get("outline")) print("\nKey Scene:\n",
final_state.get("scene")) print("\nDialogue:\n", final_state.get("dialogue"))
```

Let's understand how the system transforms the user’s prompt—“I want to write a whimsical fantasy story for children about a lost dragon finding its home”—into a complete animated story. Each step builds on the previous one, guided by the workflow’s creative nodes and powered by the Granite model.

1. Genre and tone. The workflow begins by interpreting the user's original prompt: *I want to write a whimsical fantasy story for children about a lost dragon finding its home.* Based on this input, the select_genre_node correctly classifies the narrative as **whimsical fantasy** and identifies the appropriate enchanting and heartwarming tone. This outcome is accurate and contextually aligned, as the use of phrases like **“whimsical,” “for children”** and **“lost dragon finding its home”** clearly signals a

magical yet gentle storytelling style. The genre and tone act as foundational parameters that shape every subsequent generation step in the workflow.

2. Plot outline and character descriptions. In the next step, the model is asked to create a plot outline based on the identified genre, tone and the user's original idea. *The output not only includes a 3–5 sentence story summary but also includes bonus character descriptions*, likely due to prompt leakage or retained instruction formatting from earlier iterations.

The plot outline centers around a girl named Lily who discovers a wounded dragon and helps it return to the enchanted forest with guidance from an old herbalist. This storyline mirrors the user's intent exactly—focusing on a child-friendly magical journey with emotional undertones about healing, belonging and friendship. The character sketches of the dragon, Lily and the herbalist add depth, transforming a vague idea into a structured concept with defined roles, personalities and narrative responsibilities. This step ensures the story moves from abstract intention to a tangible structure suitable for screen adaptation.

3. Key scene. Given the full plot outline, thegenerate_scene_node writes a vivid scene in prose format that captures a turning point in the story—another element explicitly requested in the user's original intent to develop a short animation script.

The chosen moment is when Lily tends to the wounded dragon in the enchanted forest, establishing emotional rapport and mutual understanding between the characters. This moment is critical as it pivots the story toward the dragon's homecoming. The scene is rich in imagery and emotion, adhering to the "whimsical" and "heartwarming" constraints while also being visually expressive—perfectly suited for a short animated format.

The model's ability to maintain tone and genre consistency across stages demonstrates the value of LangGraph's state-passing workflow and the reasoning capabilities of the Granite model.

4. Dialog in screenplay format. Finally, thewrite_dialogue_node converts the prose scene into a structured dialog in screenplay format. The user prompt—focused on **creating a story for animation** —implicitly demands that the narrative is presented in a format ready for production or visualization. This node fulfills that goal by delivering well-formatted dialog between Lily, Drago (the dragon) and the old herbalist across two distinct scenes: the cottage and the enchanted forest. The dialog preserves emotional cues and character intent, and is formatted to match animation script writing conventions (for example, character names in uppercase, scene headers like[INT. LILY'S COTTAGE - DAY]) . The output remains short, expressive and emotionally resonant, which is essential for engaging young audiences in an animated setting.

Each stage of the workflow translates the original prompt— *"a whimsical fantasy story for children about a lost dragon finding its home"* —into a structured, creative and expressive narrative output. From genre selection to dialog formatting, the system incrementally builds a coherent storytelling arc. The LangGraph framework ensures that transitions between tasks are logically connected, and the IBM Granite model enables context-sensitive text generation with a consistent tone. The result is a compact, screen-ready short animation story that emerges entirely from a single-line user input—demonstrating the practical power of agentic workflows in creative AI applications.

To make the storytelling experience even more engaging, here is a simple HTML-based visualization that beautifully formats the generated story elements—genre, tone, plot, scene and dialog. Plus, **with just one click, you can download the entire script** as a text file for future use or sharing. Let's bring the story to life on screen!

```
# Beautification and Downloadable option creation for user def
display_output_with_download(final_state):     genre = final_state.get("genre", "")     tone =
final_state.get("tone", "")     outline = final_state.get("outline", "")     scene = final_state.get("scene",
```

```
"")     dialogue = final_state.get("dialogue", "")     # Combine scene and dialogue into a full script
    script = f"""Genre: {genre} Tone: {tone} Outline: {outline} Scene: {scene} Dialogue: {dialogue}
"""     # Encode to base64     b64_script = base64.b64encode(script.encode()).decode()     # Create
downloadable HTML content     html = f"""     <h2>Genre & Tone</h2>     <p><strong>Genre:
</strong> {genre}</p>     <p><strong>Tone:</strong> {tone}</p>     <h2>Outline</h2>     <pre>
{outline}</pre>     <h2>Scene</h2>     <pre>{scene}</pre>     <h2>Dialogue</h2>     <pre>
{dialogue}</pre>     <a download="screenplay_script.txt" href="data:text/plain;base64,{b64_script}">
        <button style="margin-top: 20px; padding: 10px 20px; font-size: 16px;">Download Script as
.txt</button>     </a>     """     display(HTML(html)) # Run this after workflow.invoke()
display_output_with_download(final_state)
```

GPU usage and infrastructure notes

This tutorial uses the Granite-4.0-Tiny-Preview model for text generation. While it's one of the smaller models in the Granite family, it still requires a GPU-enabled environment to run efficiently—especially when executing multiple nodes in a LangGraph workflow.

Recommended setup:

- At least one NVIDIA V100 or A100 GPU
- 16 GB GPU memory or more for stable performance
- Python 3.8+ with torch, transformers and langgraph installed

Performance notes:

- You might encounter CUDA out-of-memory errors if running all nodes sequentially without clearing GPU memory.
- To minimize memory issues:
    - Move the model to GPU only during generation and back to CPU afterward.
    - Keep generation tokens modest (max_tokens=200–300).
    - Optionally remove or simplify nodes such as revision or detailed scene expansion.

If you're running this tutorial in a hosted notebook environment (for example, IBM watsonx.ai or Google Colab Pro), ensure that GPU is enabled in the runtime settings.

For low-resource environments, consider:

- Offloading model inference to Hugging Face Inference API.
- Reducing workflow complexity.
- Using CPU (with longer generation times).

Summary

In this tutorial, we built a modular, agentic storytelling workflow by using LangGraph and IBM's Granite-4.0-Tiny-Preview language model. Starting from a simple creative prompt, we constructed a step-by-step pipeline that classifies genre and tone, generates a plot outline, writes a key scene and finishes with screenplay-style dialog. Along the way, we demonstrated how to:

- Structure dynamic workflows by using LangGraph

- Integrate lightweight LLMs like Granite for creative reasoning
- Handle GPU memory constraints with ad hoc model loading
- Display and export story output in an easy to use format

This agentic framework is not only powerful for screen writing but can be extended to a wide range of creative or task-routing use cases. With just a few nodes, you've built a miniature writing assistant capable of turning a whimsical idea into a script-ready story.

Whether you're a developer, storyteller or researcher—this tutorial gives you a practical foundation to explore LLM-based workflow engineering in creative domains.

Ready to build your own agents? Let the creativity flow with IBM Granite models and IBM watsonx Orchestrate®.