Path to the **LLM** & **Generative AI**

Key Points

- **Model**
- **Generative AI**
- **LLM**
- **LangChain**

Model

What is model

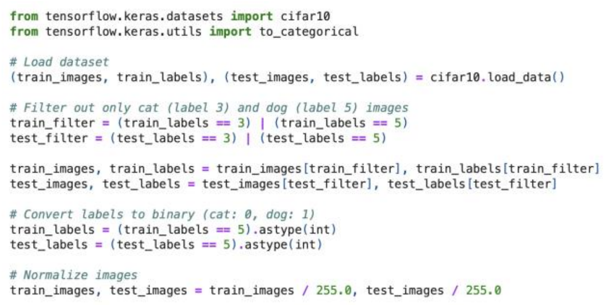
Imagine you want to create a program that can tell if a picture is of a cat or a dog.

Here's how you can use an ML/AI model for this:

1. **Data Collection**: Collect labeled images of cats and dogs.
2. **Data Preprocessing**: Prepare the images for training (resize, normalize, etc.).
3. **Model Training**: Use a neural network to learn from the images.
4. **Prediction**: Use the trained model to predict if a new image is a cat or dog.

Code Example

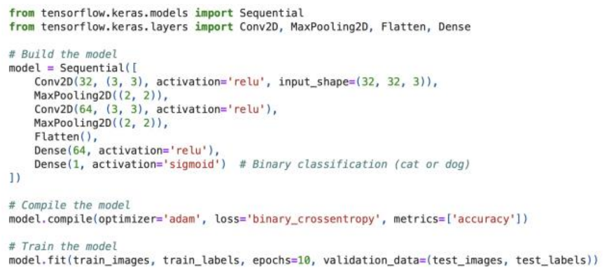
Let's see using Python and a popular ML library called TensorFlow/Keras.



```python
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load dataset
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

# Filter out only cat (label 3) and dog (label 5) images
train_filter = (train_labels == 3) | (train_labels == 5)
test_filter = (test_labels == 3) | (test_labels == 5)

train_images, train_labels = train_images[train_filter], train_labels[train_filter]
test_images, test_labels = test_images[test_filter], test_labels[test_filter]

# Convert labels to binary (cat: 0, dog: 1)
train_labels = (train_labels == 5).astype(int)
test_labels = (test_labels == 5).astype(int)

# Normalize images
train_images, test_images = train_images / 255.0, test_images / 255.0
```

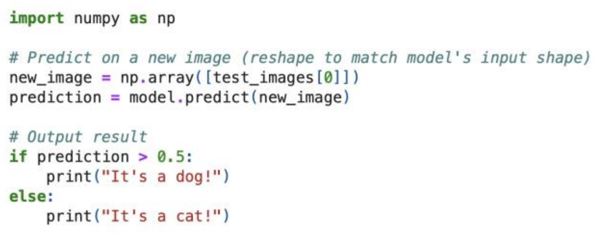
*1 - **Step 1: Data Collection***

We can use a dataset like CIFAR-10, which includes images of cats and dogs.



```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Build the model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')  # Binary classification (cat or dog)
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))
```

*2 - **Step 2: Build and Train the Model***



```python
import numpy as np

# Predict on a new image (reshape to match model's input shape)
new_image = np.array([test_images[0]])
prediction = model.predict(new_image)

# Output result
if prediction > 0.5:
    print("It's a dog!")
else:
    print("It's a cat!")
```

*3 - **Step 3: Make Predictions***

Generative AI

Generative AI refers to systems that can create new content, such as images, text, music, or even entire videos, that resemble existing data. These systems learn patterns from a large amount of data and then use this knowledge to generate similar but new data.

Example: Generating Images of Cats and Dogs

Imagine you want to create new, realistic images of cats and dogs. Generative AI can be used to achieve this.

How It Works

1. **Data Collection**: Gather a large dataset of images of cats and dogs.
2. **Training a Generative Model**: Use a type of neural network, like Generative Adversarial Networks (GANs), to learn the patterns in these images.
3. **Generating New Images**: Once trained, the model can create new images that look like the cats and dogs from the training set.

Code Example Using GANs (Generative Adversarial Networks)

Step 1: Data Collection

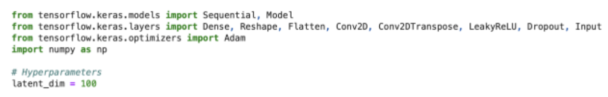
We can use the same CIFAR-10 dataset from before.

Step 2: Building the GAN

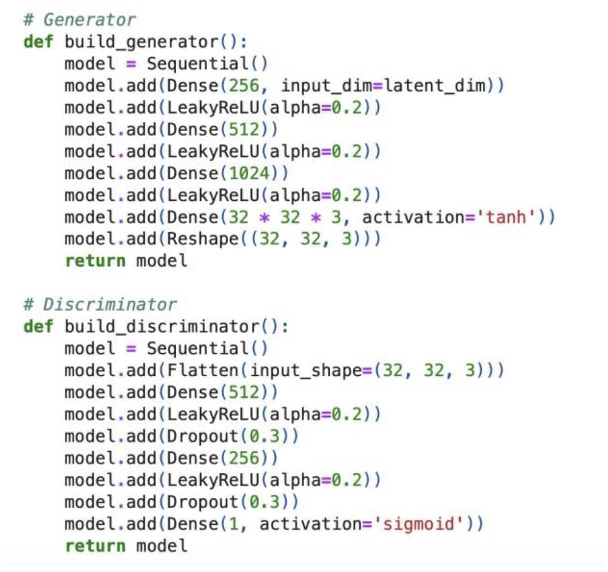
A GAN consists of two parts:

- **Generator**: Creates new images.

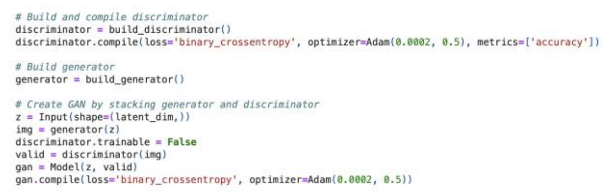
- **Discriminator**: Evaluates how realistic these images are.



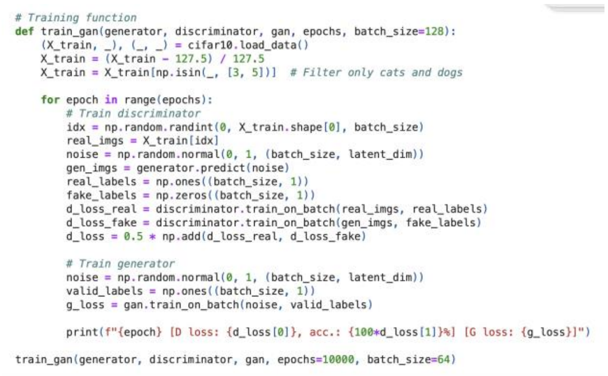
```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Reshape, Flatten, Conv2D, Conv2DTranspose, LeakyReLU, Dropout, Input
from tensorflow.keras.optimizers import Adam
import numpy as np

# Hyperparameters
latent_dim = 100
```



```
# Generator
def build_generator():
    model = Sequential()
    model.add(Dense(256, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(32 * 32 * 3, activation='tanh'))
    model.add(Reshape((32, 32, 3)))
    return model

# Discriminator
def build_discriminator():
    model = Sequential()
    model.add(Flatten(input_shape=(32, 32, 3)))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))
    return model
```



```
# Build and compile discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])

# Build generator
generator = build_generator()

# Create GAN by stacking generator and discriminator
z = Input(shape=(latent_dim,))
img = generator(z)
discriminator.trainable = False
valid = discriminator(img)
gan = Model(z, valid)
gan.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```



```
# Training function
def train_gan(generator, discriminator, gan, epochs, batch_size=128):
    (X_train, _), (_, _) = cifar10.load_data()
    X_train = (X_train - 127.5) / 127.5
    X_train = X_train[np.isin(_, [3, 5])]  # Filter only cats and dogs

    for epoch in range(epochs):
        # Train discriminator
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        real_imgs = X_train[idx]
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        gen_imgs = generator.predict(noise)
        real_labels = np.ones((batch_size, 1))
        fake_labels = np.zeros((batch_size, 1))
        d_loss_real = discriminator.train_on_batch(real_imgs, real_labels)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake_labels)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        valid_labels = np.ones((batch_size, 1))
        g_loss = gan.train_on_batch(noise, valid_labels)

        print(f"{epoch} [D loss: {d_loss[0]}, acc.: {100*d_loss[1]}%] [G loss: {g_loss}]")

train_gan(generator, discriminator, gan, epochs=10000, batch_size=64)
```

Summary

- **Generative AI**: Systems that create new data similar to what they were trained on.
- **GANs**: A type of generative model with a generator creating data and a discriminator evaluating it.
- **Training**: The generator improves by trying to fool the discriminator, and the discriminator improves by learning to distinguish real from generated data.

This process enables the creation of realistic new images of cats and dogs.

What is LLM?

LLM stands for **Large Language Model**. These are AI models designed to understand and generate human-like text based on vast amounts of data. They use deep learning techniques, particularly neural networks with many parameters, to learn patterns, grammar, context, and even some level of reasoning from the text they are trained on.

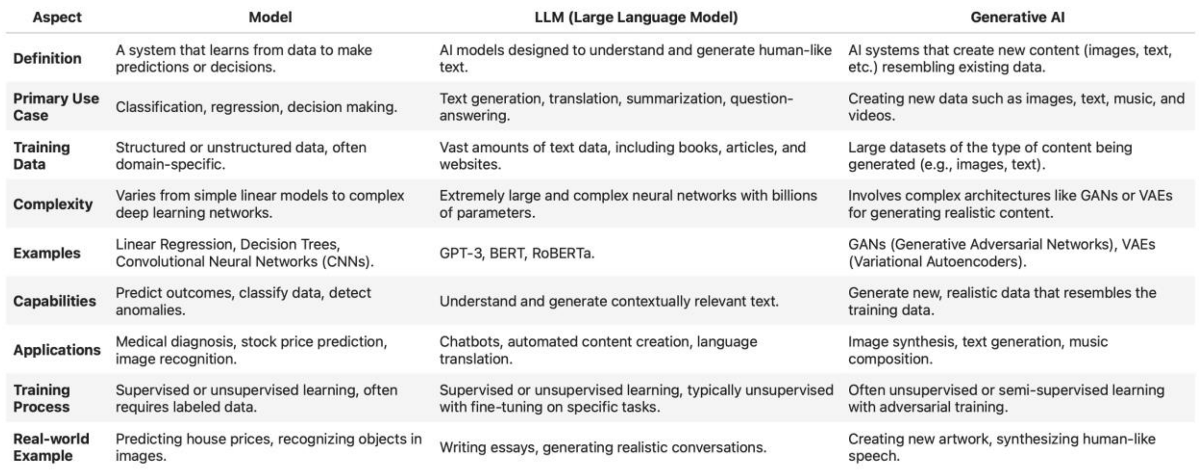
Key Features of LLMs

1. **Large-scale**: Trained on extensive datasets, often containing billions of words.
2. **Versatile**: Capable of performing various language-related tasks like translation, summarization, question-answering, and text generation.
3. **Context-aware**: Can understand and generate contextually relevant responses.

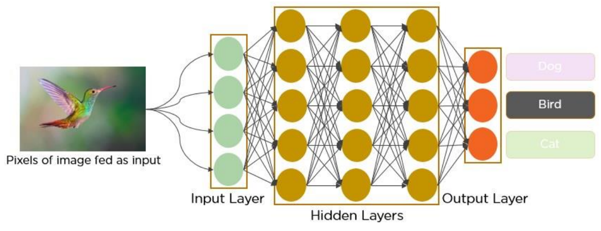
Why LLMs are Essential for Generative AI

1. **Natural Language Generation**: LLMs excel at creating human-like text, making them vital for applications like chatbots, automated content creation, and interactive AI.
2. **Learning Patterns**: They can learn intricate language patterns, enabling them to produce coherent and contextually accurate text.
3. **Adaptability**: LLMs can be fine-tuned for specific tasks, making them versatile tools for various applications within Generative AI.
4. **Enhancing Creativity**: They can assist in generating creative content such as stories, poems, and scripts, pushing the boundaries of automated content creation.

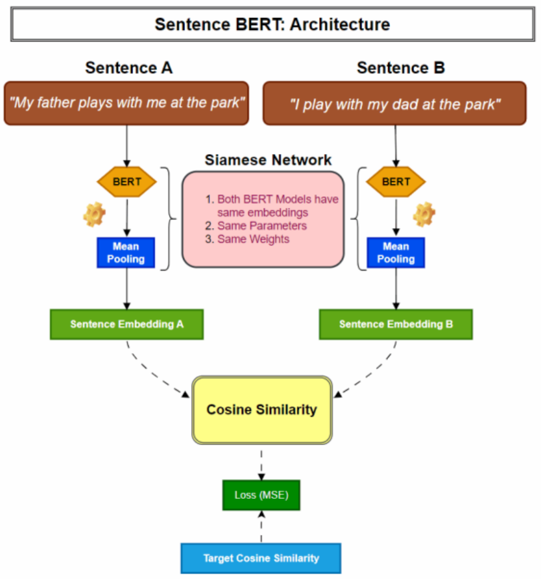
Model vs LLMs vs GenAI



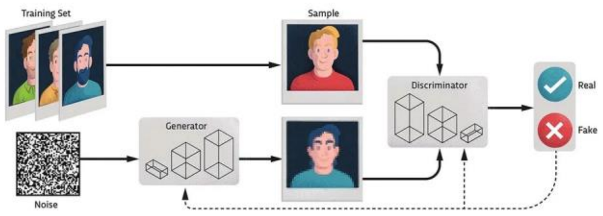
Aspect	Model	LLM (Large Language Model)	Generative AI
Definition	A system that learns from data to make predictions or decisions.	AI models designed to understand and generate human-like text.	AI systems that create new content (images, text, etc.) resembling existing data.
Primary Use Case	Classification, regression, decision making.	Text generation, translation, summarization, question-answering.	Creating new data such as images, text, music, and videos.
Training Data	Structured or unstructured data, often domain-specific.	Vast amounts of text data, including books, articles, and websites.	Large datasets of the type of content being generated (e.g., images, text).
Complexity	Varies from simple linear models to complex deep learning networks.	Extremely large and complex neural networks with billions of parameters.	Involves complex architectures like GANs or VAEs for generating realistic content.
Examples	Linear Regression, Decision Trees, Convolutional Neural Networks (CNNs).	GPT-3, BERT, RoBERTa.	GANs (Generative Adversarial Networks), VAEs (Variational Autoencoders).
Capabilities	Predict outcomes, classify data, detect anomalies.	Understand and generate contextually relevant text.	Generate new, realistic data that resembles the training data.
Applications	Medical diagnosis, stock price prediction, image recognition.	Chatbots, automated content creation, language translation.	Image synthesis, text generation, music composition.
Training Process	Supervised or unsupervised learning, often requires labeled data.	Supervised or unsupervised learning, typically unsupervised with fine-tuning on specific tasks.	Often unsupervised or semi-supervised learning with adversarial training.
Real-world Example	Predicting house prices, recognizing objects in images.	Writing essays, generating realistic conversations.	Creating new artwork, synthesizing human-like speech.



*4 - **CNN***



*5 - **BERT (LLM)***



*6 - **GANs (GenAI)***

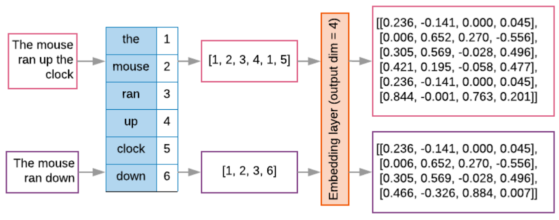
Token/Tokenisation/LLM

Tokens are the basic units of text that LLMs (Large Language Models) use to process and generate text. They can be words, subwords, characters, or even parts of words. Understanding how tokens work is crucial for grasping how LLMs handle language.

Tokenization is the process of converting a string of text into tokens. This is a fundamental step before feeding text into an LLM.

Steps in Tokenization

1. **Text Input**: The raw text that needs to be processed.
2. **Tokenization Process**: Breaking down the text into tokens.
3. **Tokens Output**: The resulting sequence of tokens.



7 - Token and Embeddings

Example

Consider the sentence: "Hello, how are you?"

- **Text Input**: "Hello, how are you?"
- **Tokenization Process**:Word-level tokenization: ["Hello", ",", "how", "are", "you", "?"]Subword-level tokenization (used by models like BERT): ["Hello", ",", "how", "are", "you", "?"]Byte Pair Encoding (BPE): ["Hell", "o", ",", "how", "are", "you", "?"]
- **Tokens Output**: Depends on the method used.

How LLMs Use Tokens

1. **Embedding**: Tokens are converted into vectors (embeddings) that represent their meanings in a high-dimensional space.
2. **Processing**: The LLM processes these token embeddings through multiple layers, capturing the context and relationships between tokens.
3. **Generation**: When generating text, the model predicts the next token based on the context provided by the previous tokens.

Here's an example using OpenAI's GPT-3, demonstrating how tokens work:

Step 1: Tokenization

- Input: "Hello, how are you?"
- Tokenization (example for GPT-3):

[15496, 11, 703, 389, 345]

Step 2: Embedding and Processing

The token IDs are converted into embeddings and processed through the model's neural network layers to understand context and meaning.

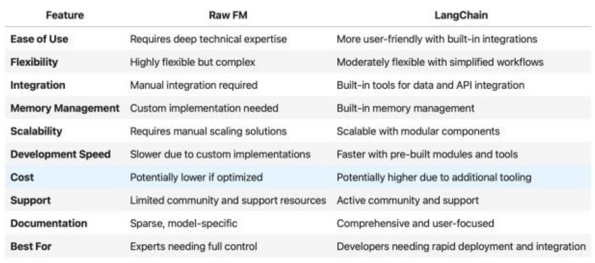
Step 3: Text Generation

The model predicts the next token based on the given context. For example, given the tokens [15496, 11, 703, 389, 345], it might predict the next token to be 345.

Summary

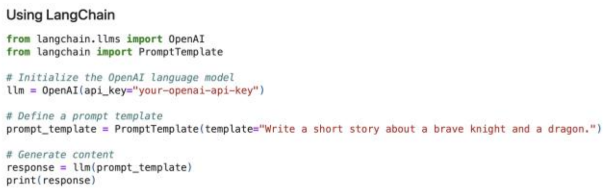
- **Tokens**: Basic units of text used by LLMs.
- **Tokenization**: Process of breaking down text into tokens.
- **Embedding**: Tokens are converted into vectors for processing.
- **Processing and Generation**: LLMs process these embeddings to understand context and generate text.

LangChain



Feature	Raw FM	LangChain
Ease of Use	Requires deep technical expertise	More user-friendly with built-in integrations
Flexibility	Highly flexible but complex	Moderately flexible with simplified workflows
Integration	Manual integration required	Built-in tools for data and API integration
Memory Management	Custom implementation needed	Built-in memory management
Scalability	Requires manual scaling solutions	Scalable with modular components
Development Speed	Slower due to custom implementations	Faster with pre-built modules and tools
Cost	Potentially lower if optimized	Potentially higher due to additional tooling
Support	Limited community and support resources	Active community and support
Documentation	Sparse, model-specific	Comprehensive and user-focused
Best For	Experts needing full control	Developers needing rapid deployment and integration

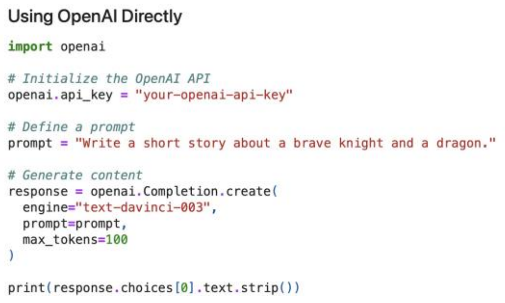
8 - FM API vs LangChain



Using LangChain

```
from langchain.llms import OpenAI
from langchain import PromptTemplate

# Initialize the OpenAI language model
llm = OpenAI(api_key="your-openai-api-key")

# Define a prompt template
prompt_template = PromptTemplate(template="Write a short story about a brave knight and a dragon.")

# Generate content
response = llm(prompt_template)
print(response)
```

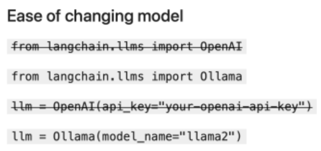
9 - Using LangChain



Using OpenAI Directly

```
import openai

# Initialize the OpenAI API
openai.api_key = "your-openai-api-key"

# Define a prompt
prompt = "Write a short story about a brave knight and a dragon."

# Generate content
response = openai.Completion.create(
  engine="text-davinci-003",
  prompt=prompt,
  max_tokens=100
)

print(response.choices[0].text.strip())
```

10 - Using OpenAI Directly



Ease of changing model

~~from langchain.llms import OpenAI~~

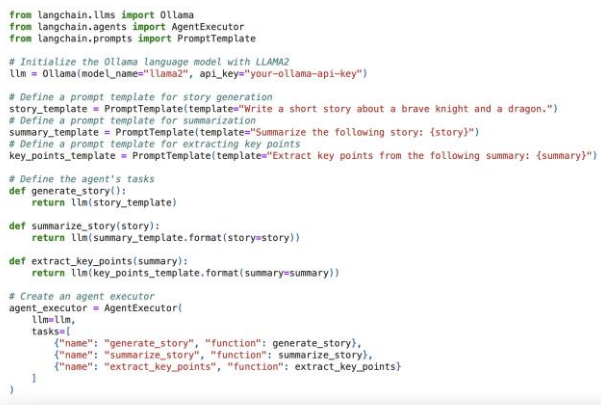
`from langchain.llms import Ollama`

~~llm = OpenAI(api_key="your-openai-api-key")~~

`llm = Ollama(model_name="llama2")`

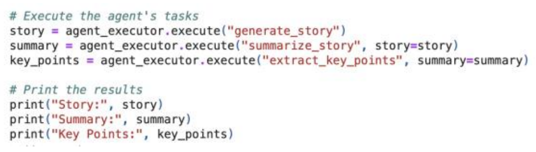
11 - Ease of LangChain

Example of using LangChain with an agent to perform a more complex task. In this example, the agent will generate a story, summarize it, and then extract key points.



```
from langchain.llms import Ollama
from langchain.agents import AgentExecutor
from langchain.prompts import PromptTemplate

# Initialize the Ollama language model with LLAMA2
llm = Ollama(model_name="llama2", api_key="your-ollama-api-key")

# Define a prompt template for story generation
story_template = PromptTemplate(template="Write a short story about a brave knight and a dragon.")
# Define a prompt template for summarization
summary_template = PromptTemplate(template="Summarize the following story: {story}")
# Define a prompt template for extracting key points
key_points_template = PromptTemplate(template="Extract key points from the following summary: {summary}")

# Define the agent's tasks
def generate_story():
    return llm(story_template)

def summarize_story(story):
    return llm(summary_template.format(story=story))

def extract_key_points(summary):
    return llm(key_points_template.format(summary=summary))

# Create an agent executor
agent_executor = AgentExecutor(
    llm=llm,
    tasks=[
        {"name": "generate_story", "function": generate_story},
        {"name": "summarize_story", "function": summarize_story},
        {"name": "extract_key_points", "function": extract_key_points}
    ]
)
```

12 - LangChain



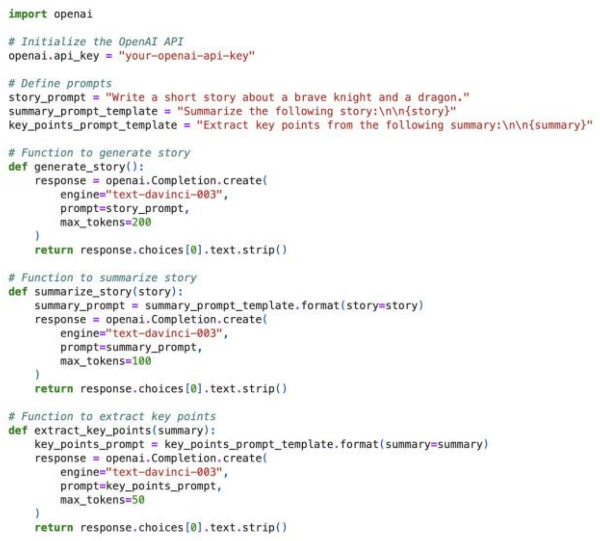
```
# Execute the agent's tasks
story = agent_executor.execute("generate_story")
summary = agent_executor.execute("summarize_story", story=story)
key_points = agent_executor.execute("extract_key_points", summary=summary)

# Print the results
print("Story:", story)
print("Summary:", summary)
print("Key Points:", key_points)
```

13 - LangChain

In this example:

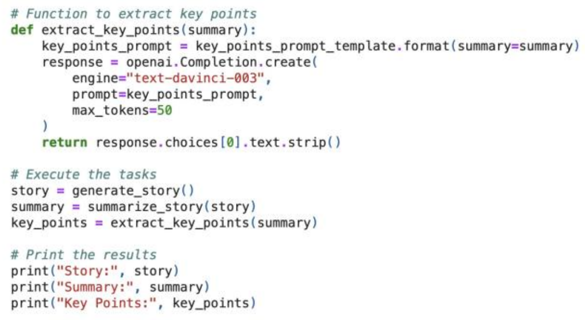
1. **Story Generation:** The agent generates a story about a brave knight and a dragon.
2. **Summarization:** The agent summarizes the generated story.

3. **Key Points Extraction:** The agent extracts key points from the summary.



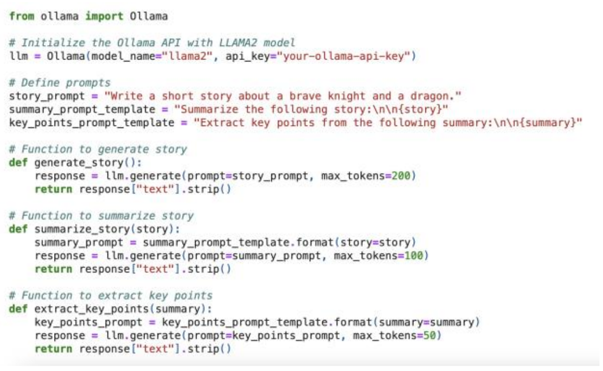
```python
import openai

# Initialize the OpenAI API
openai.api_key = "your-openai-api-key"

# Define prompts
story_prompt = "Write a short story about a brave knight and a dragon."
summary_prompt_template = "Summarize the following story:\n\n{story}"
key_points_prompt_template = "Extract key points from the following summary:\n\n{summary}"

# Function to generate story
def generate_story():
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=story_prompt,
        max_tokens=200
    )
    return response.choices[0].text.strip()

# Function to summarize story
def summarize_story(story):
    summary_prompt = summary_prompt_template.format(story=story)
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=summary_prompt,
        max_tokens=100
    )
    return response.choices[0].text.strip()

# Function to extract key points
def extract_key_points(summary):
    key_points_prompt = key_points_prompt_template.format(summary=summary)
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=key_points_prompt,
        max_tokens=50
    )
    return response.choices[0].text.strip()
```

14 - OpenAI



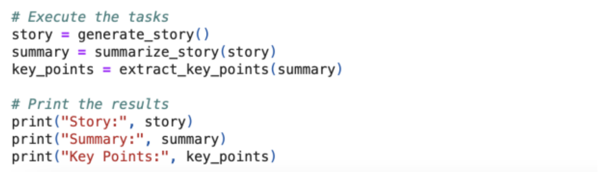
```python
# Function to extract key points
def extract_key_points(summary):
    key_points_prompt = key_points_prompt_template.format(summary=summary)
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=key_points_prompt,
        max_tokens=50
    )
    return response.choices[0].text.strip()

# Execute the tasks
story = generate_story()
summary = summarize_story(story)
key_points = extract_key_points(summary)

# Print the results
print("Story:", story)
print("Summary:", summary)
print("Key Points:", key_points)
```

15 - OpenAI



```python
from ollama import Ollama

# Initialize the Ollama API with LLAMA2 model
llm = Ollama(model_name="llama2", api_key="your-ollama-api-key")

# Define prompts
story_prompt = "Write a short story about a brave knight and a dragon."
summary_prompt_template = "Summarize the following story:\n\n{story}"
key_points_prompt_template = "Extract key points from the following summary:\n\n{summary}"

# Function to generate story
def generate_story():
    response = llm.generate(prompt=story_prompt, max_tokens=200)
    return response["text"].strip()

# Function to summarize story
def summarize_story(story):
    summary_prompt = summary_prompt_template.format(story=story)
    response = llm.generate(prompt=summary_prompt, max_tokens=100)
    return response["text"].strip()

# Function to extract key points
def extract_key_points(summary):
    key_points_prompt = key_points_prompt_template.format(summary=summary)
    response = llm.generate(prompt=key_points_prompt, max_tokens=50)
    return response["text"].strip()
```

16 - Ollama



```python
# Execute the tasks
story = generate_story()
summary = summarize_story(story)
key_points = extract_key_points(summary)

# Print the results
print("Story:", story)
print("Summary:", summary)
print("Key Points:", key_points)
```

17 - Ollama