Vertex ai pipeline:

- This Vertex AI pipeline takes an MP4 video from Google Cloud Storage (GCS), processes it through several stages to extract audio, transcribe the speech, generate subtitles, and overlay those subtitles back onto the original video. The final output is a video with embedded subtitles, stored back in GCS.
- The pipeline consists of four main components:

1. Extract MP3 audio from MP4 video
2. Transcribe audio to text using Google Cloud Speech-to-Text
3. Generate SRT subtitles from the transcription
4. Overlay the subtitles on the original video

- Each component is implemented as a Kube Flow Pipeline (KFP) component that runs in its own container, with the output of each component feeding into the next

Pipeline architecture:

The pipeline follows this data flow:

Input Video (MP4) → Extract Audio → MP3 Audio → Transcribe → JSON Transcript → Generate Subtitles →

Each component:

1.Takes input from GCS
2.Processes the data
3.Writes output back to GCS
4.Passes the GCS path to the next component

Component 1: Extract Audio from Video

This component extracts MP3 audio from an MP4 video file using FFmpeg.

```
@dsl.component(
base_image="python:3.9",
packages_to_install=["ffmpeg-python", "google-cloud-storage"],
)
def extract_audio_from_video(
input_video_gcs_path: str,
output_audio_gcs_path: str
) -> str:
```

@dsl.component: Decorator that defines this function as a Kubeflow Pipeline component
base_image: Specifies the Docker image to use (Python 3.9)
packages_to_install: Python packages to install in the container
Function parameters:
`input_video_gcs_path`
: GCS path to the input MP4 video
`output_audio_gcs_path`
: GCS path where the extracted MP3 will be stored
Return type: The function returns a string (the GCS path to the output audio)
code

```
# Install FFmpeg directly in the component
print("Installing FFmpeg...")
subprocess.run(["apt-get", "update", "-y"], check=True)
subprocess.run(["apt-get", "install", "-y", "ffmpeg"], check=True)
print("FFmpeg installed successfully")
```

This section installs FFmpeg directly in the container.We use `subprocess.run()` to execute shell commands
This approach is compatible with older versions of KFP that don't support the `install`parameter in the component decorator
python

```
# Create temporary directory for processing
with tempfile.TemporaryDirectory() as temp_dir:
# Download video from GCS
storage_client = storage.Client()
# Parse bucket and blob names
```

```
input_path = input_video_gcs_path.replace("gs://",
"")
bucket_name = input_path.split("/")[0]
blob_name = "/".join(input_path.split("/")[1:])
```

Creates a temporary directory for processing files
Initializes the GCS client
Parses the GCS path to extract bucket name and blob path

```
# Extract audio using FFmpeg
cmd = [
"ffmpeg", "-i", local_video_path,
"-vn", # No video
"-acodec", "mp3", # MP3 codec
"-ab", "192k", # Bitrate
"-ar", "44100", # Sample rate
"-y", # Overwrite output file
local_audio_path
]
try:
subprocess.run(cmd, check=True, capture_output=True)
print("Audio extraction completed successfully")
except subprocess.CalledProcessError as e:
print(f"FFmpeg error: {e.stderr.decode()}")
raise RuntimeError(f"Failed to extract audio: {e}")
```

Uses FFmpeg to extract audio from the video

- -vn: Disables video output
- `-acodec mp3`:Specifies MP3 as the output audio codec
- `-ab 192k`: Sets the audio bitrate to 192 kbps
- `-ar 44100`: Sets the audio sample rate to 44.1 kHz
- `-y: Automatically overwrites output file if it exists
- Error handling captures and reports FFmpeg errors

Component 2: Transcribe Audio to Text

This component transcribes the MP3 audio to text using Google Cloud Speech-to-Text API.

```
@dsl.component(
base_image="python:3.9",
packages_to_install=["google-cloud-speech", "google-cloud-storage"],
)
def transcribe_audio(
audio_gcs_path: str,
output_transcript_gcs_path: str,
language_code: str = "en-US"
) -> str:
```

Similar component definition as before
`language_code`:parameter has a default value of "en-US"

```
# Configure the speech recognition request
audio = speech.RecognitionAudio(uri=audio_gcs_path)
config = speech.RecognitionConfig(
encoding=speech.RecognitionConfig.AudioEncoding.MP3,
sample_rate_hertz=44100,
language_code=language_code,
enable_word_time_offsets=True,
enable_automatic_punctuation=True,
model="video" # Use the video model for better accuracy with video content
)
```

Creates a Speech-to-Text recognition request
Specifies MP3 encoding and 44.1 kHz sample rate
`enable_word_time_offsets=True`: Requests word-level timing information
`enable_automatic_punctuation=True`: Adds punctuation to the transcript
`model="video"`: Uses the video model, which is optimized for video content

```
# Handle different response formats (timedelta vs seconds/nanos)
if hasattr(word_info.start_time, 'seconds') and hasattr(word_info.start_time, 'nanos'):
# Old format with seconds and nanos
start_seconds = f"{word_info.start_time.seconds}.{word_info.start_time.nanos//1000000:03d}"
end_seconds = f"{word_info.end_time.seconds}.{word_info.end_time.nanos//1000000:03d}"
else:
# New format with timedelta
start_seconds = str(word_info.start_time.total_seconds())
end_seconds = str(word_info.end_time.total_seconds())
```

This code handles different response formats from the Speech-to-Text API
Older versions return objects with `seconds` and `nanos` attributes,Newer versions return `datetime.timedelta` objects
This ensures compatibility with different API versions

Component 3: Generate Subtitles from Transcription

This component converts the transcription JSON to SRT subtitle format.

```
@dsl.component(
base_image="python:3.9",
packages_to_install=["google-cloud-storage"],
)
def generate_subtitles(
transcript_gcs_path: str,
output_subtitles_gcs_path: str,
max_chars_per_line: int = 42,
max_lines_per_subtitle: int = 2
) -> str:
```

Component definition with parameters for controlling subtitle formatting
`max_chars_per_line`: Maximum characters per subtitle line (default: 42)
`max_lines_per_subtitle`: Maximum lines per subtitle (default: 2)

```
def format_time(time_str):
"""Convert seconds to SRT time format (HH:MM:SS,mmm)"""
seconds = float(time_str)
hours = int(seconds // 3600)
minutes = int((seconds % 3600) // 60)
seconds = seconds % 60
return f"{hours:02d}:{minutes:02d}:{seconds:06.3f}".replace(".", ",")
```

Function to convert seconds to SRT time format
SRT format requires times in HH:MM:SS,mmm format
Note the comma instead of period for milliseconds

```
# Group words into subtitles
srt_content = ""
```

```
subtitle_count = 1
current_subtitle = []
current_line = []
current_line_chars = 0
current_subtitle_lines = 0
start_time = None
```

Initializes variables for building subtitles.Tracks the current line and subtitle being built andKeeps count of characters and lines to enforce formatting limits

```
if current_line_chars + len(word_text) + 1 > max_chars_per_line:
# Line is full, add it to current subtitle
current_subtitle.append(" ".join(current_line))
current_line = [word_text]
current_line_chars = len(word_text)
current_subtitle_lines += 1
else:
# Add word to current line
current_line.append(word_text)
current_line_chars += len(word_text) + 1
```

Logic for building lines of text within character limits.When a line is full, it's added to the current subtitle.A new line is started with the current word

```
if current_subtitle_lines >= max_lines_per_subtitle or is_last_word:
# Add the current line if it's not empty
if current_line:
current_subtitle.append(" ".join(current_line))
# Create the subtitle entry
if current_subtitle:
srt_content += f"{subtitle_count}\n"
srt_content += f"{format_time(start_time)} --> {format_time(word_end)}\n"
srt_content += "\n".join(current_subtitle) + "\n\n"
subtitle_count += 1
```

Logic for finalizing a subtitle when it reaches the maximum number of lines.Creates the SRT format: index, time range, and text.Resets variables for the next subtitle

Component 4: Overlay Subtitles on Video

This component overlays the SRT subtitles on the original video using FFmpeg.

```
@dsl.component(
base_image="python:3.9",
packages_to_install=["ffmpeg-python", "google-cloud-storage"],
)
def overlay_subtitles_on_video(
input_video_gcs_path: str,
subtitles_gcs_path: str,
output_video_gcs_path: str,
font_size: int = 24,
font_color: str = "white"
```

Component definition with parameters for subtitle appearance
`font_size`: Size of the subtitle text (default: 24),`font_color`: Color of the subtitle text (default: white

```
cmd = [
"ffmpeg", "-i", local_video_path,
"-vf", f"subtitles={local_subtitles_path}:force_style='FontSize={font_size},PrimaryColour=&H
"-c:a", "copy", # Copy audio stream
"-y", # Overwrite output file
local_output_video_path
]
```

FFmpeg command to overlay subtitles on the video
`-vf subtitles=...`: Video filter that adds subtitles
`force_style`: Applies custom styling to the subtitles
`-c:a copy`: Copies the audio stream without re-encoding
`-y`: Automatically overwrites output file if it exists

Pipeline Definition

The pipeline definition connects all components together.

```
@dsl.pipeline(
name="video-processing-pipeline",
description="A pipeline that processes MP4 videos, extracts audio, generates transcriptions,
)
def video_processing_pipeline(
input_video_gcs_path: str,
output_bucket: str,
language_code: str = "en-US"
) -> NamedTuple('Outputs', [('output_video', str)]):
```

`@dsl.pipeline`**: Decorator that defines this function as a pipeline**
Pipeline parameters:
`input_video_gcs_path`**: GCS path to the input video**
`output_bucket`**: GCS bucket for storing outputs**
`language_code`**: Language code for transcription**
Return type: A NamedTuple with an 'output_video' field

```
# Define output paths
video_basename = "video" # Using fixed names to avoid string manipulation in pipeline
output_audio_gcs_path = f"gs://{output_bucket}/output/audio/{video_basename}.mp3"
output_transcript_gcs_path = f"gs://{output_bucket}/output/transcripts/{video_basename}.json"
output_subtitles_gcs_path = f"gs://{output_bucket}/output/subtitles/{video_basename}.srt"
output_video_gcs_path = f"gs://{output_bucket}/output/videos/{video_basename}_with_subtitles.mp4
```

Defines fixed output paths for all intermediate and final outputs.Uses a fixed basename to avoid string manipulation in the pipeline
This is important because Kubeflow Pipelines doesn't allow string operations on pipelineparameters

```
# Step 1: Extract audio from video
extract_task = extract_audio_from_video(
input_video_gcs_path=input_video_gcs_path,
output_audio_gcs_path=output_audio_gcs_path
)
```

Calls the first component with the input video path
The component returns the GCS path to the extracted audio

```
# Step 2: Transcribe audio to text
transcribe_task = transcribe_audio(
audio_gcs_path=extract_task.output,
output_transcript_gcs_path=output_transcript_gcs_path,
language_code=language_code
)
```

Calls the second component with the output from the first component
`extract_task.output`**refers to the return value of the extract_audio_from_video component**

```
# Return the final output with proper naming
return NamedTuple('Outputs', [('output_video', str)])(overlay_task.output)
```

Returns the output of the final component as a NamedTuple.This format is required by Kubeflow Pipelines for proper output handling

Compilation and Execution

The pipeline is compiled to a JSON file and then executed.

```
# Compile script
if __name__ == "__main__":
from kfp.v2 import compiler
# Compile the pipeline
compiler.Compiler().compile(
pipeline_func=video_processing_pipeline,
package_path="fixed_video_processing_pipeline.json"
)
```

Compiles the pipeline to a JSON file when the script is run directly and The JSON file can be uploaded to Vertex AI Pipelines for execution

- Now go to the vertex ai pipeline and create a run and upload the json file
- And give the filename,buckename.
- Click submit then it will create a pipeline.
- All the documents and videos are stored in the bucket i.e google cloud storage.