# Deployment Manifest - TbD V3

I understand you're still not seeing the content in the Canvas editor. That is frustrating. Since my attempts to open the Canvas are not working for you, I will provide the full content of the Deployment Manifest directly in this chat as a series of text blocks.

This contains everything you requested: the directory structure and every single code file, updated with all the V2 fixes.

## 📂 Directory Structure

Plaintext

```
tbd-v2/
├── deploy_v2.ps1          # Deployment Automation Script
├── Dockerfile             # Container Definition (with fix)
├── requirements.txt       # Python Dependencies
└── app/
    ├── __init__.py        # Empty
    ├── main.py            # Application Router (Lazy Loading fix)
    ├── schema.py          # Data Models
    └── services/
        ├── __init__.py    # Empty
        ├── dispatcher.py  # Service A Logic
        ├── worker.py      # Service B Logic
        ├── pipeline.py    # Core Vision Pipeline (V1 Logic)
        ├── segment.py     # Segmentation (Tuple fix)
        └── ocr.py         # OCR (V1 Logic)
```

---

## 1. Root Configuration Files

**`requirements.txt`**

Plaintext

```
# --- V1 Bridge Stack (Core Logic & API) ---
fastapi==0.104.1
uvicorn==0.24.0
pydantic==2.5.2
python-multipart
opencv-python-headless==4.8.1.78
pytesseract==0.3.10
scenedetect[opencv]==0.6.2

# --- V2 Cloud Native Stack (Infrastructure) ---
# Required for FR-03/FR-07 (Downloading video & Uploading JSON)
```

```
google-cloud-storage==2.14.0
# Required for FR-01/IR-02 (Asynchronous Task Queueing)
google-cloud-pubsub==2.19.0
```

### Dockerfile

```
Dockerfile
FROM python:3.9-slim

# Set working directory
WORKDIR /usr/src

# Ensure Python logs are streamed directly to the terminal (crucial for debugging)
ENV PYTHONUNBUFFERED=True

# 1. Install System Dependencies
# Added critical libraries for OpenCV headless support (libglib2.0-0, libsm6, etc.)
# This prevents "ImportError: libGL.so.1: cannot open shared object file"
RUN apt-get update && \
    apt-get install -y tesseract-ocr libtesseract-dev \
    libgl1 libglib2.0-0 libsm6 libxext6 libxrender1 && \
    rm -rf /var/lib/apt/lists/*

# 2. Copy and Install Python Dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 3. Copy Application Code
COPY ./app /usr/src/app

# 4. Define Environment
# PORT is injected by Cloud Run, defaulting to 8080 if missing
ENV PORT=8080

# 5. Start the Application
# Using 'sh -c' allows the shell to expand the ${PORT} variable correctly
CMD sh -c "uvicorn app.main:app --host 0.0.0.0 --port ${PORT}"
```

### deploy_v2.ps1

```
PowerShell
# ==================================================
# TbD V2 - Automated Deployment Script (PowerShell)
# ==================================================

# --- CONFIGURATION ---
# REPLACE WITH YOUR ACTUAL PROJECT ID
```

```
$PROJECT_ID = "tbd-v2"
$REGION = "us-central1"
$REPO_NAME = "tbd-repo"
$IMAGE_NAME = "tbd-v2-engine"
$TAG = "latest"

# Resource Names
$TOPIC_NAME = "tb-d-ingest-tasks"
$INPUT_BUCKET = "tbd-raw-video-$PROJECT_ID"
$OUTPUT_BUCKET = "tbd-results-$PROJECT_ID"
$DISPATCHER_SERVICE = "tbd-dispatcher"
$WORKER_SERVICE = "tbd-worker"

# Service Accounts
$DISPATCHER_SA = "tbd-dispatcher-sa"
$WORKER_SA = "tbd-worker-sa"

# ====================================================

Write-Host "--- STARTING DEPLOYMENT FOR PROJECT: $PROJECT_ID ---" -ForegroundColor Cyan
gcloud config set project $PROJECT_ID

# 1. Enable APIs
Write-Host "--- Enabling GCP APIs ---" -ForegroundColor Green
gcloud services enable artifactregistry.googleapis.com cloudbuild.googleapis.com run.googleapis.com
pubsub.googleapis.com storage.googleapis.com

# 2. Create Storage Buckets
Write-Host "--- Creating GCS Buckets ---" -ForegroundColor Green
gsutil mb -l $REGION "gs://$INPUT_BUCKET/"
if ($LASTEXITCODE -ne 0) { Write-Host "Bucket might already exist, continuing..." -ForegroundColor
Yellow }

gsutil mb -l $REGION "gs://$OUTPUT_BUCKET/"
if ($LASTEXITCODE -ne 0) { Write-Host "Bucket might already exist, continuing..." -ForegroundColor
Yellow }

# 3. Create Pub/Sub Topic
Write-Host "--- Creating Pub/Sub Topic ---" -ForegroundColor Green
gcloud pubsub topics create $TOPIC_NAME
if ($LASTEXITCODE -ne 0) { Write-Host "Topic might already exist, continuing..." -ForegroundColor
Yellow }

# 4. Create Artifact Registry & Build Image
Write-Host "--- Building & Pushing Container Image ---" -ForegroundColor Green
gcloud artifacts repositories create $REPO_NAME --repository-format=docker --location=$REGION
--description="TbD Engine Repository"
if ($LASTEXITCODE -ne 0) { Write-Host "Repo might already exist, continuing..." -ForegroundColor
Yellow }
```

```
# Submit build to Cloud Build
$IMAGE_URI = "$REGION-docker.pkg.dev/$PROJECT_ID/$REPO_NAME/$IMAGE_NAME`:$TAG"
gcloud builds submit --tag $IMAGE_URI .

# 5. Setup Service Accounts
Write-Host "--- Configuring IAM Service Accounts ---" -ForegroundColor Green
gcloud iam service-accounts create $DISPATCHER_SA --display-name "TbD Dispatcher"
if ($LASTEXITCODE -ne 0) { Write-Host "SA might already exist..." -ForegroundColor Yellow }
gcloud iam service-accounts create $WORKER_SA --display-name "TbD Worker"
if ($LASTEXITCODE -ne 0) { Write-Host "SA might already exist..." -ForegroundColor Yellow }

# Grant Dispatcher permission to Publish
gcloud pubsub topics add-iam-policy-binding $TOPIC_NAME
--member="serviceAccount:$DISPATCHER_SA@$PROJECT_ID.iam.gserviceaccount.com"
--role="roles/pubsub.publisher"

# Grant Worker permission to Read Input / Write Output
gsutil iam ch "serviceAccount:$WORKER_SA@$PROJECT_ID.iam.gserviceaccount.com:objectViewer"
"gs://$INPUT_BUCKET"
gsutil iam ch "serviceAccount:$WORKER_SA@$PROJECT_ID.iam.gserviceaccount.com:objectCreator"
"gs://$OUTPUT_BUCKET"

# 6. Deploy Dispatcher Service
# INCREASED MEMORY to 2Gi to prevent OOM on startup due to large library imports
Write-Host "--- Deploying DISPATCHER Service ---" -ForegroundColor Green
gcloud run deploy $DISPATCHER_SERVICE `
  --image $IMAGE_URI `
  --region $REGION `
  --service-account "$DISPATCHER_SA@$PROJECT_ID.iam.gserviceaccount.com" `
  --allow-unauthenticated `
  --memory 2Gi `
  --cpu 1 `
  --set-env-vars "SERVICE_TYPE=dispatcher,GCP_PROJECT_ID=$PROJECT_ID" `
  --tag v2-stable

# 7. Deploy Worker Service
Write-Host "--- Deploying WORKER Service ---" -ForegroundColor Green
gcloud run deploy $WORKER_SERVICE `
  --image $IMAGE_URI `
  --region $REGION `
  --service-account "$WORKER_SA@$PROJECT_ID.iam.gserviceaccount.com" `
  --no-allow-unauthenticated `
  --memory 2Gi `
  --cpu 2 `
  --timeout 3600 `
  --set-env-vars "SERVICE_TYPE=worker,GCP_PROJECT_ID=$PROJECT_ID" `
  --tag v2-stable
```

```
# 8. Create Pub/Sub Push Subscription
Write-Host "--- Linking Pub/Sub to Worker ---" -ForegroundColor Green

# Add delay to ensure service is fully registered
Write-Host "Waiting for service propagation..." -ForegroundColor Cyan
Start-Sleep -Seconds 10

$WORKER_URL = gcloud run services describe $WORKER_SERVICE --region $REGION --format
'value(status.url)'

if (-not $WORKER_URL) {
    Write-Error "Worker Service failed to deploy or URL retrieval failed. Cannot create subscription."
    exit 1
}
Write-Host "Worker URL retrieved: $WORKER_URL" -ForegroundColor Cyan

$SUBSCRIPTION_SA = "tbd-sub-invoker"
gcloud iam service-accounts create $SUBSCRIPTION_SA --display-name "PubSub Invoker"
if ($LASTEXITCODE -ne 0) { Write-Host "SA might already exist..." -ForegroundColor Yellow }

# Allow Subscription SA to invoke Worker
gcloud run services add-iam-policy-binding $WORKER_SERVICE --region $REGION
--member="serviceAccount:$SUBSCRIPTION_SA@$PROJECT_ID.iam.gserviceaccount.com"
--role="roles/run.invoker"

# Create Subscription
# Note: If subscription exists, we update it to ensure the endpoint is correct
gcloud pubsub subscriptions create tbd-worker-sub `
    --topic $TOPIC_NAME `
    --push-endpoint=$WORKER_URL `
    --push-auth-service-account="$SUBSCRIPTION_SA@$PROJECT_ID.iam.gserviceaccount.com" `
    --ack-deadline=600
if ($LASTEXITCODE -ne 0) {
    Write-Host "Subscription exists, updating endpoint..." -ForegroundColor Yellow
    gcloud pubsub subscriptions update tbd-worker-sub `
        --push-endpoint=$WORKER_URL `
        --push-auth-service-account="$SUBSCRIPTION_SA@$PROJECT_ID.iam.gserviceaccount.com"
}

Write-Host "================================================" -ForegroundColor
Cyan
Write-Host "DEPLOYMENT COMPLETE" -ForegroundColor Cyan
Write-Host "Dispatcher URL: $(gcloud run services describe $DISPATCHER_SERVICE --region
$REGION --format 'value(status.url)')"
Write-Host "Input Bucket:   gs://$INPUT_BUCKET"
Write-Host "Output Bucket:  gs://$OUTPUT_BUCKET"
Write-Host "================================================" -ForegroundColor
Cyan
```

---

## 2. App Logic Files (`app/`)

### `app/__init__.py`

Python
```
# This file intentionally left empty to define the app package
```

### `app/main.py` (Lazy Loading Fix)

Python
```
import uvicorn
import os
from fastapi import FastAPI, HTTPException
from app.schema import TaskPayload

# --- Main Application Router ---
# The behavior of this app is determined by the SERVICE_TYPE env var in Cloud Run
SERVICE_TYPE = os.environ.get("SERVICE_TYPE", "dispatcher")

app = FastAPI(title=f"TbD Engine V2 - {SERVICE_TYPE.upper()} Service")

# --- LAZY LOADING IMPLEMENTATION ---
# We only import the specific service class we need for this instance.
# This prevents the 'Dispatcher' from crashing due to 'Worker' dependencies (like OpenCV).

if SERVICE_TYPE == "worker":
    # IR-03: The Worker service is triggered by Pub/Sub
    # Lazy Import: Worker logic is only loaded if we are running as a worker.
    try:
        from app.services.worker import WorkerService
        worker = WorkerService()
    except ImportError as e:
        print(f"CRITICAL: Failed to import WorkerService. Check dependencies. {e}")
        worker = None

    @app.post("/")
    async def pubsub_trigger(data: dict):
        if not worker:
            raise HTTPException(status_code=500, detail="Worker service failed to initialize.")

        try:
            # The Worker processes the message synchronously within this request.
            # Pub/Sub waits for the 200 OK response to ack the message.
            worker.process_pubsub_message(data)
            return {"status": "Processing initiated"}, 200
```

```
        except Exception as e:
            print(f"Worker processing failed: {e}")
            # Return 500 so Pub/Sub retries the message later
            raise HTTPException(status_code=500, detail=f"Worker failure: {e}")

elif SERVICE_TYPE == "dispatcher":
    # IR-01: The Dispatcher exposes the public HTTP endpoint
    # Lazy Import: Dispatcher logic is lightweight and loads quickly.
    from app.services.dispatcher import DispatcherService
    dispatcher = DispatcherService()

    @app.post("/submit", status_code=202)
    async def submit_video_task(payload: TaskPayload):
        """
        Receives the task payload (GCS URI) and queues it in Pub/Sub.
        Returns 202 Accepted immediately.
        """
        try:
            task_id = dispatcher.submit_task(payload)
            return {"status": "Task accepted and queued", "task_id": task_id}
        except Exception as e:
             raise HTTPException(status_code=500, detail=f"Dispatch failed: {e}")

else:
    @app.on_event("startup")
    async def startup_event():
        print("ERROR: SERVICE_TYPE environment variable not set or invalid.")
        print("Must be set to 'dispatcher' or 'worker'.")

if __name__ == "__main__":
    # Local dev entry point
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

**app/schema.py**

```
Python
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any

# --- NEW: V2 Task Payload Schema (TDD 3.2) ---
class TaskPayload(BaseModel):
    task_id: str = Field(..., description="UUID-V4 for the task")
    client_id: str = Field(..., description="Identifier for the client submitting the task")
    gcs_uri: str = Field(..., description="gs://bucket-name/video.mp4")
    output_bucket: str = Field(..., description="GCS bucket name for processed results")
    # Optional fields for future configuration
    config: Dict[str, Any] = Field(default_factory=dict, description="Processing configuration parameters")
```

```
# --- V1 Node Object (Retained for Core Logic) ---
class ActionNode(BaseModel):
    id: str = Field(..., description="Unique node identifier")
    type: str = Field("action", description="Fixed type for V1")
    timestamp_start: float = Field(..., description="Start time in seconds")
    timestamp_end: float = Field(..., description="End time in seconds")
    description: Optional[str] = Field(None, description="Human readable description")

    action_type: str = Field("click", description="Enum: click, type, drag, scroll")
    ui_element_text: str = Field(..., description="OCR result")
    ui_region: List[int] = Field(..., description="[x, y, w, h] bounding box")
    confidence: float = Field(..., description="OCR confidence score")

    next_node_id: Optional[str] = Field(None, description="Pointer to next node")


# --- V1 Root Object (Retained for Core Logic) ---
class Pathway(BaseModel):
    pathway_id: str = Field(..., description="UUID-v4")
    title: str = Field(..., description="SOP Title")
    author_id: str = Field(..., description="User ID")
    source_video: str = Field(..., description="Source Video URI")
    created_at: str = Field(..., description="ISO-8601 Timestamp")
    total_duration_sec: float = Field(..., description="Total duration")

    metadata: Dict[str, Any] = Field(default_factory=dict, description="Metadata")
    nodes: List[ActionNode] = Field(..., description="List of action nodes")
```

---

## 3. Service Logic Files (`app/services/`)

### `app/services/__init__.py`

Python
```
# This file intentionally left empty
```

### `app/services/dispatcher.py`

Python
```
import uuid
import os
import json
from google.cloud import pubsub_v1
from app.schema import TaskPayload
```

```
# The Pub/Sub topic name is a hard requirement (IR-02)
PUBSUB_TOPIC = "tb-d-ingest-tasks"

# Project ID is automatically available in the Cloud Run environment
PROJECT_ID = os.environ.get("GCP_PROJECT_ID", "local-dev-project")

class DispatcherService:
    def __init__(self):
        # Initialize Pub/Sub client
        # Note: In a local environment without GCP credentials, this might raise a warning
        # unless you have 'gcloud auth application-default login' set up.
        try:
            self.publisher = pubsub_v1.PublisherClient()
            self.topic_path = self.publisher.topic_path(PROJECT_ID, PUBSUB_TOPIC)
        except Exception as e:
            print(f"WARNING: Pub/Sub client failed to initialize (Expected if running locally without creds):
{e}")
            self.publisher = None

    def submit_task(self, payload: TaskPayload) -> str:
        """
        Receives an HTTP request payload, publishes it to Pub/Sub, and returns Task ID.
        """
        # Set a task ID if it wasn't provided (TDD 3.2)
        if not payload.task_id:
            payload.task_id = str(uuid.uuid4())

        print(f"Dispatching Task ID: {payload.task_id} for URI: {payload.gcs_uri}")

        if self.publisher:
            # The data sent to Pub/Sub must be a byte string
            data = json.dumps(payload.model_dump()).encode("utf-8")

            # Publish the message (IR-02)
            future = self.publisher.publish(self.topic_path, data)
            # We do NOT wait for the future.result(), ensuring a fast 202 Accepted return.
            print(f"Published to {self.topic_path}")
        else:
            print("MOCK PUBLISH: Pub/Sub not connected. Task would be queued here.")

        return payload.task_id


app/services/worker.py

Python
import os
import tempfile
import base64
```

```
import json
from urllib.parse import urlparse
from google.cloud import storage
from app.schema import TaskPayload, Pathway
from app.services.pipeline import build_pathway # Reuse of V1 core logic

# Use a standard temporary directory for local file processing (FR-03, NFR-04)
TEMP_DIR = tempfile.gettempdir()

class WorkerService:
    def __init__(self):
        # Initialize Storage Client
        # Note: If running locally without 'gcloud auth application-default login', this might fail.
        try:
            self.storage_client = storage.Client()
        except Exception as e:
            print(f"WARNING: Storage client failed to initialize: {e}")
            self.storage_client = None

    def _parse_gcs_uri(self, gcs_uri: str) -> tuple:
        """Parses a gs:// URI into bucket name and blob name."""
        parsed = urlparse(gcs_uri)
        if parsed.scheme != 'gs':
            raise ValueError(f"Invalid GCS URI scheme: {gcs_uri}")
        return parsed.netloc, parsed.path.lstrip('/')

    def _download_video(self, bucket_name: str, blob_name: str, local_path: str):
        """FR-03: Downloads the video from GCS to the local ephemeral disk."""
        if not self.storage_client:
            print(f"MOCK DOWNLOAD: Would download gs://{bucket_name}/{blob_name}")
            # Create a dummy file for local testing if GCS is down
            with open(local_path, 'wb') as f:
                f.write(b'mock_video_content')
            return

        bucket = self.storage_client.bucket(bucket_name)
        blob = bucket.blob(blob_name)

        print(f"Downloading gs://{bucket_name}/{blob_name} to {local_path}...")
        blob.download_to_filename(local_path)
        print("Download complete.")

    def _upload_pathway(self, pathway: Pathway, output_bucket_name: str, task_id: str) -> str:
        """FR-07: Uploads the final Pathway.json to the GCS results bucket."""
        # Define the output path in the GCS results bucket
        output_blob_name = f"{task_id}/pathway.json"
        gcs_uri = f"gs://{output_bucket_name}/{output_blob_name}"

        if not self.storage_client:
```

```
        print(f"MOCK UPLOAD: Would upload result to {gcs_uri}")
        return gcs_uri

    bucket = self.storage_client.bucket(output_bucket_name)
    blob = bucket.blob(output_blob_name)

    # Upload the JSON content
    json_data = pathway.model_dump_json(indent=2)
    blob.upload_from_string(json_data, content_type='application/json')

    print(f"Pathway uploaded successfully to: {gcs_uri}")
    return gcs_uri

def process_pubsub_message(self, pubsub_message_data: dict):
    """
    IR-03: Main entry point triggered by Pub/Sub push subscription.
    """
    # Decode the Pub/Sub data envelope
    if 'message' not in pubsub_message_data or 'data' not in pubsub_message_data['message']:
        # Fallback for local testing with direct JSON payload
        if 'task_id' in pubsub_message_data:
            payload_dict = pubsub_message_data
        else:
            raise ValueError("Invalid Pub/Sub message format.")
    else:
        message_data_bytes = base64.b64decode(pubsub_message_data['message']['data'])
        payload_dict = json.loads(message_data_bytes.decode('utf-8'))

    payload = TaskPayload.model_validate(payload_dict)

    task_id = payload.task_id
    gcs_uri = payload.gcs_uri

    print(f"--- Worker START: Task {task_id} ---")

    # 1. Prepare local file paths
    input_bucket, input_blob = self._parse_gcs_uri(gcs_uri)
    video_filename = os.path.basename(input_blob)
    local_video_path = os.path.join(TEMP_DIR, video_filename)

    # --- EXECUTION BLOCK ---
    try:
        # 2. Ingestion (FR-03)
        self._download_video(input_bucket, input_blob, local_video_path)

        # 3. Core Processing (FR-04, FR-05, FR-06)
        # NOTE: In a real GCS run, this needs a real video.
        # If mocking, build_pathway might fail if the file is 0 bytes.
        pathway: Pathway = build_pathway(local_video_path)
```

```
        # 4. Output and Persistence (FR-07)
        output_uri = self._upload_pathway(pathway, payload.output_bucket, task_id)

        # 5. Logging (IR-04)
        print(f"Worker SUCCESS: Task {task_id} completed. Output URI: {output_uri}")

    except Exception as e:
        print(f"Worker FAILED: Task {task_id} encountered an error: {e}")
        # Re-raise exception so Cloud Run returns 500 and Pub/Sub retries
        raise e

    finally:
        # 6. Disk Cleanup (NFR-04)
        if os.path.exists(local_video_path):
            os.remove(local_video_path)
            print(f"Cleaned up local file: {local_video_path}")

    print(f"--- Worker END: Task {task_id} ---")
```

**app/services/pipeline.py**

```
Python
# app/services/pipeline.py

import os
import uuid
import time
from typing import List, Tuple
import cv2

# CORRECTED IMPORTS: Use relative imports
from ..schema import Pathway, ActionNode            # ..schema means go up one directory (app/) then find schema
from .segment import detect_action_segments         # .segment means find segment.py in the current directory (services/)
from .ocr import get_key_frame, run_ocr             # .ocr means find ocr.py in the current directory (services/)


# Helper function to generate a simplistic description for the node
def generate_description(ocr_text: str, index: int) -> str:
    """Creates a default description based on the found text."""
    if ocr_text:
        return f"User interacted with element '{ocr_text}'"
    return f"User performed an unidentified action (Node {index + 1})"

def build_pathway(video_path: str) -> Pathway:
```

```
"""
Implements the full Screen-to-Schema pipeline (TDD Section 1.1)
to generate the Pathway.json object (FR-12).
"""
print(f"Starting processing for video: {os.path.basename(video_path)}")

# 1. Start timer and get video duration
start_time = time.time()
cap = cv2.VideoCapture(video_path)
# Check if the video is opened before accessing properties
if not cap.isOpened():
    raise IOError(f"Could not open video file for duration check: {video_path}")

total_duration_sec = cap.get(cv2.CAP_PROP_FRAME_COUNT) / cap.get(cv2.CAP_PROP_FPS)
cap.release()

# --- Step 2: Event Segmentation (The "When") ---
segments = detect_action_segments(video_path)
print(f"Found {len(segments)} action segments.")

nodes: List[ActionNode] = []

# --- Step 3 & 4: Visual Extraction & Node Generation ---
for i, (start_time_sec, end_time_sec) in enumerate(segments):
    node_id = f"node_{i + 1}"

    try:
        # 3.1 Extract the Key Frame (FR-07)
        frame, _ = get_key_frame(video_path, start_time_sec, end_time_sec)

        # 3.2 Run OCR on the frame (FR-08)
        ocr_results = run_ocr(frame)

        # TDD Heuristic for MVP V1: Take the highest confidence OCR result.
        best_result = max(ocr_results, key=lambda x: x['confidence'], default=None)

        # Default values if no good OCR result is found
        ui_text = best_result['text'] if best_result else "UNKNOWN_TEXT"
        ui_region = best_result['ui_region'] if best_result else [0, 0, 0, 0]
        confidence = best_result['confidence'] if best_result else 0.0

        # 4.1 Create Node object (FR-10)
        node = ActionNode(
            id=node_id,
            timestamp_start=start_time_sec,
            timestamp_end=end_time_sec,
            description=generate_description(ui_text, i),
            action_type="click", # Default action type for V1 PoC
            ui_element_text=ui_text,
```

```
            ui_region=ui_region,
            confidence=confidence,
            next_node_id=f"node_{i + 2}" if i + 1 < len(segments) else None # 4.2 Link nodes sequentially
        )
        nodes.append(node)

    except Exception as e:
        print(f"Skipping node {node_id} due to processing error: {e}")
        continue

# --- Step 5: Final Pathway Object Assembly ---
pathway = Pathway(
    pathway_id=str(uuid.uuid4()),
    title=f"Automated Pathway: {os.path.basename(video_path)}",
    author_id="tbd-v1-engine",
    source_video=os.path.basename(video_path),
    created_at=time.strftime('%Y-%m-%dT%H:%M:%S%z'),
    total_duration_sec=total_duration_sec,
    nodes=nodes,
    metadata={}
)

# NFR-01: Check performance target (processing time < 2x video duration)
elapsed_time = time.time() - start_time
print(f"Processing finished in {elapsed_time:.2f}s. Target: <{total_duration_sec * 2:.2f}s.")

return pathway
```

**`app/services/segment.py` *(Fixed Tuple Issue)***

```
Python
# app/services/segment.py

import cv2
from scenedetect import VideoManager
from scenedetect import SceneManager
from scenedetect.detectors import ContentDetector
from typing import List, Tuple

# TDD specifies a default threshold of 30% pixel change
DEFAULT_THRESHOLD = 30.0

def detect_action_segments(video_path: str, threshold: float = DEFAULT_THRESHOLD) ->
List[Tuple[float, float]]:
    """
    Step 2: Event Segmentation (The 'When')
    Analyzes the video for scene changes to identify distinct Action Events (FR-04).
    """
```

```
    video_manager = VideoManager([video_path])
    scene_manager = SceneManager()

    # Use ContentDetector for visual differencing (FR-05)
    scene_manager.add_detector(ContentDetector(threshold=threshold))

    try:
        # Step 1: Ingest & Pre-process
        video_manager.start()

        # Analyze video frames
        scene_manager.detect_scenes(frame_source=video_manager)

        # Get a list of scenes.
        # Scenedetect 0.6+ returns a list of tuples: [(start_time, end_time), ...]
        scene_list = scene_manager.get_scene_list(video_manager.get_base_timecode())

        segments = []

        # Convert to standard (start_float, end_float) list
        for scene in scene_list:
            # Check if the item is a tuple or an object and extract time accordingly
            if isinstance(scene, tuple):
                start, end = scene
                # In newer versions, these are FrameTimecode objects which have get_seconds()
                start_time = start.get_seconds()
                end_time = end.get_seconds()
            else:
                # Fallback for older API versions if they return single objects
                print(f"Warning: Unexpected scene format: {type(scene)}")
                continue

            segments.append((start_time, end_time))

        return segments

    finally:
        video_manager.release()
```

**app/services/ocr.py**

```
Python
# app/services/ocr.py

import cv2
import pytesseract
from typing import List, Tuple, Dict, Any
```

```
def get_key_frame(video_path: str, start_time: float, end_time: float) -> Tuple[cv2.typing.MatLike, int]:
    """
    FR-07: Extracts the middle frame (Key Frame) of a stable state segment.
    """
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        raise IOError(f"Could not open video file: {video_path}")

    fps = cap.get(cv2.CAP_PROP_FPS)

    # Calculate the timestamp for the middle frame
    middle_time = (start_time + end_time) / 2

    # Calculate the target frame number
    target_frame_num = int(middle_time * fps)

    # Set the video position to the target frame
    cap.set(cv2.CAP_PROP_POS_FRAMES, target_frame_num)

    ret, frame = cap.read()
    cap.release()

    if ret:
        return frame, target_frame_num
    else:
        raise ValueError(f"Could not read frame at time {middle_time:.2f}s")


def run_ocr(frame: cv2.typing.MatLike) -> List[Dict[str, Any]]:
    """
    FR-08: Performs OCR on an image frame using Tesseract.
    """
    # Use pytesseract to get detailed box and confidence data (FR-11 prep)
    # The 'output_type=pytesseract.Output.DICT' provides structured data
    ocr_data = pytesseract.image_to_data(frame, output_type=pytesseract.Output.DICT)

    results = []

    # Iterate through detected text blocks
    for i in range(len(ocr_data['text'])):
        text = ocr_data['text'][i].strip()
        confidence = float(ocr_data['conf'][i]) / 100.0 # Convert 0-100 to 0.0-1.0

        # Only include results that are meaningful text and have high confidence
        if text and confidence > 0.5:
            # Define UI region as [x, y, w, h] bounding box (TDD Section 2.2)
            ui_region = [
                int(ocr_data['left'][i]),
                int(ocr_data['top'][i]),
```

```
            int(ocr_data['width'][i]),
            int(ocr_data['height'][i])
        ]

        results.append({
            "text": text,
            "confidence": confidence,
            "ui_region": ui_region
        })

return results
```