```python
import argparse
import logging
import sys
import os
import time
from pathlib import Path
from typing import Iterator, List, Optional
import cv2
import csv
import numpy as np

# Ensure src path is available for imports
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from src.config import AppConfig
from src.video import VideoLoader, parse_roi, calculate_optical_flow_value
from src.ai import MediaPipePoseEstimator, SkeletonPoint
from src.logic import HandTurnDetector, get_rough_hand_status_from_mp, get_motion_status_from_mp, \
    RESET_MOTION_STATUS, READY_MOTION_STATUS, SPEAK_MOTION_STATUS, RAPID_MOTION_STATUS
from src.network import GrpcClient, encode_frame

# Generated Protobuf modules
try:
    from src import ksl_sentence_recognition_pb2 as pb2
except ImportError:
    import ksl_sentence_recognition_pb2 as pb2

# Configure root logger
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

# Constants from MFC
OPTICAL_FLOW_THRESH = 0.45
OPTICAL_FLOW_HOLD_FRAME = 3

def setup_logging(level: int):
    """Sets up logging for the application."""
    logging.getLogger().setLevel(level)
    # Other loggers might need to be adjusted here if specific libraries are too verbose
    logging.getLogger('grpc').setLevel(logging.WARNING) # Reduce grpc verbosity
    logging.getLogger('mediapipe').setLevel(logging.WARNING) # Reduce mediapipe verbosity
    logging.getLogger('protobuf').setLevel(logging.WARNING) # Reduce protobuf verbosity


def process_video_stream(
    config: AppConfig,
    video_loader: VideoLoader,
    pose_estimator: MediaPipePoseEstimator,
    grpc_client: GrpcClient
) -> None:
    """
    Main pipeline to process video, perform AI inference, apply logic, and stream via gRPC.
    """
    session_id = f"session_{Path(config.video_path).stem}_{os.getpid()}"
    frame_count = 0
    prev_skeleton: List[SkeletonPoint] = []
    current_motion_status: int = READY_MOTION_STATUS # Initial state
    right_hand_detector = HandTurnDetector()
    left_hand_detector = HandTurnDetector()

    # MFC KeyFrame Logic States
    prev_gray: Optional[np.ndarray] = None
    speak_start_flag = False
    frozen_flag = 0
    saved_keyframe_count = 0
    detected_keyframes: List[int] = []

    try:
        logger.info(f"Starting video processing for session: {session_id}")

        # Prepare output file if specified
        output_file = None
        if config.output_path:
            try:
                output_file = open(config.output_path, 'wb')
                logger.info(f"Saving packets to {config.output_path}")
            except OSError as e:
                logger.error(f"Failed to open output file: {e}")

        # Prepare debug CSV file if enabled
        debug_csv_file = None
        debug_csv_writer = None
        if config.debug_file:
            try:
                start_f = config.frame_range[0] if config.frame_range else 0
                end_f = config.frame_range[1] if config.frame_range else "end"
                # Construct filename: video_filename_start_end.csv
                video_name = config.video_path.name
                csv_filename = f"{video_name}_{start_f}_{end_f}.csv"

                debug_csv_file = open(csv_filename, 'w', newline='')
                debug_csv_writer = csv.writer(debug_csv_file)
                debug_csv_writer.writerow(['frame_index', 'avg_motion', 'Rdev', 'Ldev'])
                logger.info(f"Saving debug info to {csv_filename}")
            except OSError as e:
                logger.error(f"Failed to open debug CSV file: {e}")

        def frame_generator() -> Iterator[pb2.Frame]:
            nonlocal frame_count, prev_skeleton, current_motion_status, prev_gray, speak_start_flag, frozen_flag, saved_keyframe_count, detected_keyframes, debug_csv_writer # For updating outer scope var

            for frame_image_rgb in video_loader.get_frames():
                frame_count += 1

                # Check frame range if specified
                if config.frame_range:
                    start_frame, end_frame = config.frame_range
                    if frame_count < start_frame:
                        continue
                    if frame_count > end_frame:
                        logger.info(f"Reached end of specified frame range ({end_frame}). Stopping.")
                        break

                if frame_count % 30 == 0:
                    logger.info(f"Processing frame {frame_count}")
                else:
                    logger.debug(f"Processing frame {frame_count}")

                # 1. Optical Flow Calculation
                curr_gray = cv2.cvtColor(frame_image_rgb, cv2.COLOR_RGB2GRAY)
                avg_motion = 0.0

                # AI Inference
                start_ai = time.time()
                current_skeleton = pose_estimator.process_frame(frame_image_rgb)
                logger.debug(f"[ETA_LOG] 4. AI 추론: {time.time() - start_ai:.6f} sec")

                # Calculate Rdev/Ldev for debug
                r_dev = 0.0
                l_dev = 0.0
                if config.debug_file and prev_skeleton and current_skeleton and len(prev_skeleton) > 16 and len(current_skeleton) > 16:
                     right_wrist_idx = 16
                     left_wrist_idx = 15
                     prev_R = np.array([prev_skeleton[right_wrist_idx].x, prev_skeleton[right_wrist_idx].y])
                     cur_R = np.array([current_skeleton[right_wrist_idx].x, current_skeleton[right_wrist_idx].y])
                     r_dev = float(np.linalg.norm(cur_R - prev_R))

                     prev_L = np.array([prev_skeleton[left_wrist_idx].x, prev_skeleton[left_wrist_idx].y])
                     cur_L = np.array([current_skeleton[left_wrist_idx].x, current_skeleton[left_wrist_idx].y])
                     l_dev = float(np.linalg.norm(cur_L - prev_L))

                if debug_csv_writer:
                    debug_csv_writer.writerow([frame_count, avg_motion, r_dev, l_dev])

            # Business Logic & State Management
            # Update motion status
            current_motion_status = get_motion_status_from_mp(
                current_motion_status,
                current_skeleton,
                prev_skeleton
            )

            # 2. Pre-condition Check: Hand Status
            # Must be != 0 (Ready/Down) to proceed with KeyFrame logic
            hand_status = get_rough_hand_status_from_mp(current_skeleton)

            is_turn_detected = False

            if hand_status != 0:
                # Update Optical Flow only when hand is detected (Match C++ behavior)
                if prev_gray is not None:
                    start_flow = time.time()
                    avg_motion = calculate_optical_flow_value(prev_gray, curr_gray)
                    logger.debug(f"[ETA_LOG] 3. Optical Flow 계산: {time.time() - start_flow:.6f} sec")
                prev_gray = curr_gray # Update reference frame only if hand status is valid

                # 3. Hand Turn Detection
                if len(current_skeleton) > 16: # Ensure wrist landmarks exist
                    right_wrist = (current_skeleton[16].x, current_skeleton[16].y)
                    left_wrist = (current_skeleton[15].x, current_skeleton[15].y)

                    DUMMY_DT = 1 / 30.0

                    if right_hand_detector.update(right_wrist, DUMMY_DT):
                        logger.debug(f"Frame {frame_count}: Right turn detected")
                        is_turn_detected = True
                    if left_hand_detector.update(left_wrist, DUMMY_DT):
                        logger.debug(f"Frame {frame_count}: Left turn detected")
                        is_turn_detected = True

            # Logic: Turn Detected -> Set speak_start_flag
            if is_turn_detected:
                speak_start_flag = True

            should_send_keyframe = False

            # 4. Final Send Condition: Turn happened + Motion Stabilized + Not Frozen + Not Currently Turning
            if speak_start_flag and avg_motion < OPTICAL_FLOW_THRESH and frozen_flag == 0 and hand_status != 0 and not is_turn_detected:
                should_send_keyframe = True
                # C++ does NOT reset speak_start_flag here. It persists until manually reset or session end.
                # speak_start_flag = False
                frozen_flag = OPTICAL_FLOW_HOLD_FRAME
                detected_keyframes.append(frame_count)

            # Decrement frozen flag
            if frozen_flag > 0:
                frozen_flag -= 1

            # prev_gray update moved inside hand_status check to match C++

            # Visualization (Pop window update)
            if not config.no_gui:
                try:
                    # Convert RGB (from VideoLoader) to BGR (for OpenCV Display)
                    vis_img = cv2.cvtColor(frame_image_rgb, cv2.COLOR_RGB2BGR)

                    if should_send_keyframe:
                        # Highlight Keyframe
                        cv2.putText(vis_img, "KEYFRAME DETECTED", (20, 50),
                                    cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 0, 255), 3)
                        cv2.rectangle(vis_img, (0, 0), (vis_img.shape[1]-1, vis_img.shape[0]-1), (0, 0, 255), 4)
                        logger.info(f"Frame {frame_count}: Sending KEYFRAME (AvgMotion: {avg_motion:.2f})")

                    cv2.imshow("KSL Client Stream", vis_img)
                    # Process GUI events (1ms delay)
                    if cv2.waitKey(1) & 0xFF == 27: # ESC to stop strictly if needed
                        pass
                except Exception as e:
                    logger.debug(f"Visualization error (headless?): {e}")

            # Encode to Protobuf Frame
            # Only send if it is a keyframe
            if should_send_keyframe:
                # Mode 1: Save Images Locally
                if config.save_keyframes_only == 1:
                    try:
                        kf_dir = Path("keyFrame")
                        kf_dir.mkdir(parents=True, exist_ok=True)
                        kf_filename = kf_dir / f"keyframe_{frame_count:06d}.jpg"
                        # Convert RGB to BGR for saving
                        cv2.imwrite(str(kf_filename), cv2.cvtColor(frame_image_rgb, cv2.COLOR_RGB2BGR))
                        logger.info(f"Saved keyframe to {kf_filename}")
                        saved_keyframe_count += 1
                    except Exception as e:
                        logger.error(f"Failed to save keyframe: {e}")

                # Mode 2: Encode only (useful for profiling or output file generation without IO overhead)
                elif config.save_keyframes_only == 2:
                    pass # Do nothing special, just proceed to encode

                start_encode = time.time()
                encoded_frame = encode_frame(
                    session_id=session_id,
                    index=frame_count,
                    flag=0, # KEYFRAME is START (0)
                    image=frame_image_rgb,
                    skeleton=current_skeleton
                )
                logger.debug(f"[ETA_LOG] 5. 데이터 인코딩: {time.time() - start_encode:.6f} sec")
                yield encoded_frame

            # Write to file if enabled (also only keyframes now)
            if output_file and should_send_keyframe:
                 serialized = encoded_frame.SerializeToString()
                 output_file.write(serialized)
                 output_file.flush()

            prev_skeleton = current_skeleton # Update for next frame

    # Stream frames
    # We need to handle the case where gRPC fails but we still want to generate frames for file output
    gen = frame_generator()

    use_offline_mode = False

    if config.save_keyframes_only > 0:
        logger.info(f"Keyframe Only Mode ({config.save_keyframes_only}) enabled. Skipping gRPC connection.")
        use_offline_mode = True
    else:
        try:
            # Explicitly check connection before passing generator to gRPC
            # This prevents the generator from getting stuck in "executing" state inside gRPC if connection fails immediately
            logger.info("Connecting to gRPC server...")
            grpc_client.connect()
            logger.info("Connected to gRPC server. Starting stream.")

            response_iterator = grpc_client.send_stream(gen)
            for response in response_iterator:
                if config.log_level > logging.CRITICAL:
                    print(f"Server response: {response.message}")
                else:
                    logger.info(f"Server response: {response.message}")

        except Exception as e:
                logger.warning(f"gRPC connection/streaming failed: {e}")
                logger.info("Switching to offline mode (saving to file if output path set)...")
                use_offline_mode = True

        if use_offline_mode:
            try:
                # Consume the rest of the generator to ensure processing continues
                for _ in gen:
                    pass
            except Exception as e_inner:
                logger.error(f"Error during offline processing: {e_inner}")

    except Exception as e:
        logger.exception(f"An unexpected error occurred during processing: {e}")
        sys.exit(1)
    finally:
        if 'debug_csv_file' in locals() and debug_csv_file:
            debug_csv_file.close()
            logger.info("Closed debug CSV file.")
        if not config.no_gui:
            cv2.destroyAllWindows()
        if 'output_file' in locals() and output_file:
            output_file.close()
            logger.info("Closed output file.")
        grpc_client.close()

        if config.log_level > logging.CRITICAL:
             print(f"Detected Keyframes Count: {len(detected_keyframes)}, Indices: {detected_keyframes}")
        else:
            logger.info(f"Finished processing for session: {session_id}. Total frames: {frame_count}")
            logger.info(f"Detected Keyframes Count: {len(detected_keyframes)}, Indices: {detected_keyframes}")
            if config.save_keyframes_only:
                logger.info(f"Total extracted keyframes: {saved_keyframe_count}")


def main():
    parser = argparse.ArgumentParser(
        description="Process video for KSL detection and stream to gRPC server."
    )
    parser.add_argument("video_path", type=Path,
                        help="Path to the input video file (e.g., movieTitle.mp4)")
    parser.add_argument("roi", type=str,
                        help="Region of Interest in format 'x,y,w,h' (e.g., '100,100,320,240')")
    parser.add_argument("--server", type=str, default="localhost:50051",
                        help="gRPC server address (default: localhost:50051)")
    parser.add_argument("--model-path", type=Path,
                        default=Path("./protos/pose_landmarker_full.task"),
                        help="Path to the MediaPipe Pose Landmarker model (.task file)")
    parser.add_argument("--output", type=Path,
                        help="Optional: Path to save the raw protobuf packet bytes stream (.bin file)")
    parser.add_argument("--frame", type=int, nargs=2, metavar=('START', 'END'),
                        help="Optional: Process only a specific range of frames (e.g. 100 500)")
    parser.add_argument("--keyframeOnly", type=int, choices=[0, 1, 2], default=0,
                        help="0: Normal gRPC stream. 1: Save keyframe images locally (no gRPC). 2: Encode to Protobuf only (no gRPC, no local image save unless output set).")
    parser.add_argument("--log-level", type=str, default="INFO",
                        choices=["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL", "NONE"],
                        help="Set the logging level (default: INFO)")
    parser.add_argument("--no-gui", action="store_true", help="Disable video visualization (OpenCV imshow).")
    parser.add_argument("--debugFile", action="store_true", help="Save debug information (motion values) to a CSV file.")

    args = parser.parse_args()

    # Setup logging based on argument
    if args.log_level.upper() == "NONE":
        log_level_val = logging.CRITICAL + 1
    else:
        log_level_val = getattr(logging, args.log_level.upper())
    setup_logging(log_level_val)

    try:
        # Parse ROI
        parsed_roi = parse_roi(args.roi)

        # Create AppConfig
        config = AppConfig(
            video_path=args.video_path,
            roi=parsed_roi,
            output_path=args.output,
            frame_range=tuple(args.frame) if args.frame else None,
            save_keyframes_only=bool(args.keyframeOnly),
            log_level=log_level_val,
            no_gui=args.no_gui,
            debug_file=args.debugFile
        )

        # Initialize components
        # Note: MediaPipe model path handling might need adjustment if default path is relative
        # Ensure model path exists
        if not args.model_path.exists():
             logger.warning(f"Model path {args.model_path} does not exist. Trying absolute path or check config.")
             # You might want to raise an error here depending on strictness

        video_loader = VideoLoader(config)
        pose_estimator = MediaPipePoseEstimator(str(args.model_path)) # Model path as string
        grpc_client = GrpcClient(args.server)

        # Run processing pipeline
        process_video_stream(config, video_loader, pose_estimator, grpc_client)

    except ValueError as e:
        logger.error(f"Configuration Error: {e}")
        sys.exit(1)
    except FileNotFoundError as e:
        logger.error(f"File Error: {e}")
        sys.exit(1)
    except ConnectionRefusedError as e:
        logger.error(f"Network Error: {e}")
        sys.exit(1)
    except Exception as e:
        logger.exception(f"An unexpected error occurred: {e}")
        sys.exit(1)


if __name__ == "__main__":
    main()
```