ChatGPT

Enhancing Fiber Defect Detection for Real-Time Video

Live Video Frame Acquisition

To enable continuous analysis, the system needs to capture frames from a live video source (e.g. a webcam or IP camera) in an ongoing loop. We can use **OpenCV** for this, as it provides a simple interface (`cv2.VideoCapture`) to grab frames in real time [1]. For example, one thread or loop will continuously read frames from the camera and pass them to the defect detection pipeline. This ensures new frames are constantly fed into the analysis with minimal delay. Key considerations for this module:

- **Video Source:** Support different sources (local camera index or network stream URL). OpenCV's `VideoCapture` can handle both (e.g. `VideoCapture(0)` for webcam, or an RTSP/HTTP stream for IP camera).
- **Frame Retrieval Loop:** Read frames in a loop and immediately process or enqueue them. Use a high capture rate and adjust the processing to keep up. If processing each frame is slow, the loop will naturally skip frames (it will process the next available frame after finishing the previous) to avoid accumulating latency [2].
- **Frame Format:** Convert frames to the format expected by the pipeline (e.g. ensure correct color space, resolution, etc.). The existing pipeline likely expects images in a certain orientation or size, so handle any necessary resizing or rotation from the camera feed.
- **Termination & Errors:** Provide a safe way to stop the loop (e.g. on user command or error) and release the camera resource. For instance, monitor for a specific keypress (if running locally with an OpenCV window) or a stop signal in the dashboard.

By leveraging OpenCV for video I/O, we achieve the continuous feed requirement. The capture loop should aim for **sub-second per iteration** to meet real-time needs, dropping or skipping frames if the analysis cannot keep up (this prevents increasing lag). In summary, this component sets up the live feed and ensures frames are ready for analysis as fast as possible.

Integration of Real-Time Frames with Detection Pipeline

Each incoming frame will be processed through the existing defect detection pipeline, reusing its modules (segmentation, detection, etc.) with minimal refactoring. The current pipeline already segments fiber end-face images into **core, cladding, and ferrule** regions and detects anomalies/defects in each region [3]. Our strategy is to **wrap these batch-processing steps into a real-time loop**:

1. **Reuse Segmentation (Stage 2)** – The `UnifiedSegmentationSystem` (from `separation.py`) can be initialized once and used to segment each frame into zones. In batch mode, the pipeline used a *multi-method consensus* of 11 algorithms to identify core/cladding boundaries [4]. For real-time, we might configure it to use a faster single method or a pre-trained model for segmentation to save time. If the camera and fiber are fixed, an initial segmentation on the first frame could be reused as

a mask for subsequent frames (assuming minimal movement), but generally we will segment each frame to get accurate region masks.

2. **Reuse Defect Detection (Stage 3)** – Initialize the `OmniFiberAnalyzer` once at startup with the chosen config (using `OmniConfig`). This avoids re-loading models or reference data for every frame. In the batch pipeline, they create one `OmniFiberAnalyzer` and call `analyze_end_face` for each image sequentially [5] [6] – we can adopt the same pattern for frames. Each frame (treated like an image) is analyzed to detect defects (e.g. scratches, contamination) in the segmented regions. The analyzer likely produces a list of detected defect regions with properties (coordinates, size, confidence, etc.). We will **skip the final data aggregation stage** (Stage 4) on each frame to avoid extra overhead – since we don't need to cluster across multiple images in real-time, each frame's result can be taken as-is. Any simple post-processing (like filtering very small defects or merging close-by ones) can be done on the fly.
3. **Avoid Disk I/O** – In the existing code, each stage writes outputs to disk (variations, segmented images, reports). For real-time streaming, we should operate in memory as much as possible. For example, instead of writing a defect report JSON and annotated image file for each frame, get the results in Python data structures directly. This may involve adding functions or modifying `analyze_end_face` to return data (defect list, etc.) and a visualization image array without saving to disk. If modifying the pipeline code is undesirable, a workaround is to write to a temporary location and read back, but that would add latency and is best avoided.
4. **Frame-by-Frame Variations** – The current *processing* stage creates multiple image variations for robustness (e.g. different lighting or filtering) [7]. Doing this for every video frame would be too slow. Instead, we can either skip this stage entirely (process only the original frame), or generate a smaller number of variations occasionally. A possible strategy is to apply one mild enhancement filter to each frame (if needed for defect visibility) rather than exhaustive augmentation. Keeping the pipeline modular, we might let the `reimagine_image` function run with a setting of `num_variations = 0 or 1` for real-time mode (meaning no heavy augmentation loops).
5. **Defect Data Extraction** – From the detection results of each frame, extract the needed info for display: defect bounding boxes or contours (pixel coordinates), the region each defect is in (we can determine this by checking which segmented zone the defect's centroid falls into, or by which segmented image produced it), and any severity or confidence score. The pipeline's detection output likely includes the defect's bbox and centroid [8] [9], and perhaps a classification of severity (e.g. "LOW", "HIGH" severity based on confidence [10]). We will use these to annotate the frame and to populate text/graphs on the dashboard.

Modularity: We will create a function or class (e.g. `RealTimeAnalyzer`) that encapsulates this process for each frame. This function can call the existing pipeline functions in order: segmentation -> detection -> result parsing. By doing so, we **reuse the core logic** (e.g. the anomaly detection algorithms and segmentation methods) without rewriting them, satisfying the modular integration goal. The existing modules (`OmniFiberAnalyzer`, etc.) act as black boxes processing each frame. The pipeline's object-oriented design (e.g. using an analyzer object with a config) facilitates reuse across multiple frames. We ensure any initialization (like loading a knowledge base of typical fiber appearance) is done once upfront, and each frame analysis references that for anomaly detection. In real-time mode, the first frame could be used to build or load the reference model (if the detection algorithm uses a reference of a "clean" fiber) – subsequent frames then detect deviations as defects [11]. This way, the anomaly detection remains consistent across the stream.

By integrating at this level, we **avoid major refactoring** of the internal logic. We essentially wrap the pipeline's stages into a continuously running service: each loop iteration takes one frame through segmentation and detection and produces results immediately, without the batch file management.

Real-Time Dashboard Implementation

To display the live analysis results, we recommend building a lightweight dashboard application. This dashboard will show the video feed with overlays and textual defect information in real time. Several Python frameworks can facilitate this:

- **OpenCV + Flask (Web Server):** A classic approach is to use Flask to serve a webpage that contains the live video stream and data. OpenCV can capture frames and also draw overlay annotations (boxes, text) on each frame. We can then use a Flask route that continuously yields JPEG frames as a motion JPEG stream to the browser [12]. This technique keeps latency low by streaming frames as they are ready. PyImageSearch provides an example of combining OpenCV processing with Flask streaming: capture frames, apply an algorithm (e.g. detection), and stream the results to a webpage [13]. We would create an HTML template showing the video (via an `<img>` tag sourcing from the MJPEG route) and perhaps use a second route or AJAX calls to fetch defect stats (or embed the stats in the same stream overlay). Flask gives us flexibility to design custom HTML/JS for layout. This is a good option if we need a multi-client web dashboard or custom UI controls, and it can achieve real-time performance (dozens of FPS) if optimized.
- **Streamlit (Interactive Dashboard):** Streamlit allows quick creation of web apps in pure Python. By default, it's not geared for high-FPS video, but there are community components to help. For example, `streamlit-webrtc` can stream webcam feeds with low latency by using WebRTC under the hood [14]. We can use `webrtc_streamer` to capture and display video, and provide a callback that applies our defect detection to each frame in real-time [15]. Streamlit would let us easily add UI elements (buttons, checkboxes) for controlling the analysis if needed. The trade-off is that achieving true real-time (30 fps) may require the WebRTC component and some tuning, but it simplifies embedding charts or text alongside the video. It's an excellent rapid prototyping choice.
- **Gradio:** Gradio is another high-level library for ML demos with a web interface. Gradio 5+ has introduced streaming support and even a built-in WebRTC component for webcam feeds [16]. We could set up a Gradio interface with a live video input (webcam component) and have the output be the video frames with drawn defects, plus other outputs (labels or graphs). With `live=True` or using gradio-webrtc, the detection function would be called on each incoming frame and the result displayed immediately. Gradio is intuitive for showing model outputs and could achieve near real-time performance similar to Streamlit's approach.
- **Plotly Dash:** Dash is suited for building rich dashboards with charts and graphs. While it doesn't have native video streaming components, we can integrate OpenCV frames by encoding them (e.g. to base64 images) and updating an `html.Img` component in the layout on a timer. For instance, a Dash `dcc.Interval` could regularly trigger a callback that grabs the latest processed frame (with overlays) and updates the image source along with defect stats. Dash would make it easy to create live graphs (e.g. a bar chart of defect counts per region updating in real-time). However, achieving high frame rates might be tricky due to the overhead of callbacks and data transfer — it may be best for lower frame-rate monitoring or if rich analytics visuals are needed.
- **OpenCV GUI (Fallback):** In a scenario where a web UI is not required, simply using OpenCV's own window (`cv2.imshow`) to display the video with overlays is possible [1]. We could overlay defect annotations on the frame and show it in a window, while printing or logging defect coordinates in

the console. This would be the simplest “dashboard,” though not as feature-rich. Since the question leans towards a dashboard with organized info, this is likely a backup option for quick local testing rather than the final solution.

Recommendation: For a robust yet relatively simple solution, using **Flask with OpenCV** is a strong choice. It gives fine control over the streaming pipeline and UI layout, and OpenCV handles the video capture and drawing efficiently. This setup can achieve sub-second latency streaming frames to a browser [12]. Alternatively, if rapid development and an all-in-one solution are priorities, **Streamlit with** `streamlit-webrtc` is recommended – it will handle a lot of the streaming complexity for you (leveraging WebRTC to minimize latency) and allow easy addition of live charts or metrics. In either case, ensure that the video stream and defect data are synchronized and updated together on the dashboard.

Performance Considerations

Real-time processing introduces performance challenges, so careful optimizations are needed:

- **Frame Rate vs Processing Time:** If the detection pipeline cannot process at the full frame rate of the camera, we must balance speed and accuracy. One approach is to **sample frames** (e.g. analyze every nth frame) or allow frame skipping as mentioned above. The system should prioritize analyzing the latest frame available rather than queuing up a backlog. This prevents lag – intermediate frames are skipped if the system is still busy when new frames arrive, so the output stays timely [2]. In practice, if the camera is 30 FPS and the pipeline can do ~10 FPS, the dashboard will update ~10 times per second and simply not analyze every single frame.
- **Resolution and ROI:** Higher resolution images yield more accurate results but take longer to process. Consider capturing or resizing frames to a moderate resolution that still shows defects clearly (e.g. if the camera is 1080p but defects can be seen at 720p, downscale to reduce pixel count). Additionally, if the fiber end-face occupies only part of the frame, crop to that region to avoid wasting time on background pixels. Using the segmentation, one could mask out regions outside the ferrule to limit processing (some algorithms might already do this).
- **Hardware Acceleration:** Leverage hardware where possible. If the detection or segmentation uses any deep learning models or heavy computations, enable GPU acceleration. For example, using OpenCV’s DNN with CUDA or using an optimized runtime like **ONNX Runtime** can significantly speed up inference [17]. In a Gradio example, the YOLOv10 model was run with `onnxruntime-gpu` to achieve real-time performance, whereas a CPU would be much slower [17]. For our use-case, if any stage (like a neural network for defect classification or segmentation) can run on GPU, it will help maintain sub-second processing. Similarly, using vectorized numpy operations or C++ implementations for image processing (OpenCV is already optimized in C++) will improve throughput.
- **Parallelism:** Where possible, parallelize parts of the pipeline. One design is to have a separate thread for reading frames and another for processing so they work in parallel. For instance, a producer thread continuously captures the latest frame (and perhaps stores it in a variable), while a consumer thread picks up frames and runs the defect detection. This can utilize multi-core systems better – while one frame is being analyzed, the next frame can already be grabbed from the camera. However, be mindful of thread synchronization and data integrity (using thread-safe queues or similar). The OpenCV forum suggests that multi-threading can decouple capture rate from processing time if implemented properly [18]. If multi-threading is complex to implement, the

simpler route is as mentioned: let the capture loop naturally throttle to the processing speed (which inherently drops frames if needed).

- **Efficiency in Pipeline Stages:** Optimize each stage for speed. For segmentation, if using a simplified approach (e.g. one fast edge-detection or Hough transform to find the fiber core), tune its parameters to be efficient. For detection, if the algorithm allows adjusting sensitivity or iterations, choose a setting that is a good trade-off between speed and thoroughness. Also, disable any unnecessary logging or debugging visuals during real-time operation, as these can slow down the loop.
- **Memory Management:** Ensure that we don't accumulate large data in memory over time. Since we process frames in a loop, avoid storing every result indefinitely. For example, keep only recent defect stats if needed for trend, otherwise discard or overwrite old data. This prevents memory bloat in a long-running application (especially if images were being stored each frame – which we avoid here).
- **Testing Latency:** It's important to measure end-to-end latency (from camera capture to display update) and frame throughput during development. This can be done by timestamping frames or using simple counters. If latency is above the sub-second goal, identify bottlenecks (e.g. if segmentation consensus is too slow, reduce the number of algorithms or use a faster method). The goal is to **keep processing time per frame well under 1 second**, ideally in the range of 30-100ms for truly smooth video, but if that's not attainable with current methods, ~200-500ms per frame may be acceptable for near real-time feedback.

By addressing these considerations, the system can maintain a fluid live analysis. For instance, using a GPU-accelerated model for defect detection and focusing on the latest frame (rather than trying to process every single frame) will help achieve a responsive experience.

Modular Design and Code Reuse

To incorporate real-time support without major refactoring, we will structure the solution to **wrap around the existing modules**:

- **Non-Intrusive Additions:** Instead of rewriting the core pipeline functions, we add a new module or extend `app.py` with a "Real-Time Mode". This could be triggered by a command-line option or a new menu entry (e.g., "4. Live Camera Analysis") in the interactive menu. This mode would initialize the pipeline (load config, set up the analyzer and segmentation) once, then enter a loop capturing frames and processing them.
- **Reuse Existing Classes:** The design leverages the fact that classes like `UnifiedSegmentationSystem` and `OmniFiberAnalyzer` are self-contained. For example, we instantiate `OmniFiberAnalyzer` with the pipeline config (as done in batch mode) [5] . This means all thresholds and parameters (min defect area, confidence threshold, etc.) from config.json apply in real-time as well. We then feed frames to its analysis function. We might need to slightly adapt methods to accept raw images: if `analyze_end_face` insists on an image file path, we could write a small in-memory adapter (like writing the frame to a temporary JPEG in memory or disk, or better, modify `analyze_end_face` to accept a numpy array directly). Such changes are localized and do not change the overall pipeline structure.
- **Maintain Output Structure (if needed):** Even though we won't save images for every frame, we can still utilize the pipeline's output generation for debugging or occasional snapshots. For instance, the `visualize_comprehensive_results` in detection.py creates an analysis image with all overlay drawings [19] [20] . We can call this periodically (say every Nth frame or on demand) to save a

snapshot of the analysis to disk for record-keeping, while most frames are just shown live. The key is that **the pipeline's functions are used as is, just invoked in a loop** rather than once per user-selected image.

- **Configurable and Extendable:** We keep the design modular such that switching components is easy. If in the future we develop a faster segmentation model (e.g. a neural network that finds core/cladding), we can swap that in for the current multi-method approach when in live mode. Similarly, the dashboard front-end is somewhat decoupled – it reads the processed frame and defect data provided by the back-end. This separation (capture & processing vs. display) means the core logic remains testable and usable headlessly (for example, one could run the real-time analyzer without a GUI to just log defects).
- **No Alert System:** As requested, we do not include any alert/notification mechanism in this design. That simplifies the integration – we focus purely on capturing data and displaying it live. (If needed later, one could hook in an alert on certain conditions, but it's outside our current scope.)

Overall, the enhancements are added **alongside** existing code, not tearing it apart. The batch pipeline can remain fully functional. We essentially add a new real-time pipeline path that calls many of the same functions under the hood. This modular approach ensures that improvements in the core defect detection (e.g. better anomaly detection algorithms or updated knowledge base) automatically benefit the live mode as well, since it's using the same code.

Dashboard UI Layout Ideas

For the dashboard's interface, clarity and responsiveness are key. Here we outline a possible layout and features for the real-time monitoring UI:

- **Main Video Panel:** The centerpiece of the dashboard is the live video feed from the fiber end-face. This panel displays the camera frames with defect **overlays** drawn on. Defect highlights could be bounding boxes or translucent masks over the defect regions. We can color-code the overlays by region (e.g. red boxes for core defects, blue for cladding, green for ferrule) or by severity (e.g. critical defects in red, minor in yellow). Each defect overlay might have a small label or number. For example, if a defect is detected, we could overlay a number near it corresponding to an entry in the defect list. The video panel should update in real-time, giving visual feedback as soon as a defect appears or disappears. It's also useful to draw the boundaries of the core and cladding on the video (e.g. a circle outline) for context, if the segmentation data is available.
- **Defect List/Info Sidebar:** Next to the video (or below it, depending on space), include a text panel or table listing details of currently detected defects in the frame. This could be a table with columns: **ID**, **Region**, **Coordinates** (e.g. centroid or bounding box), **Size** (in pixels), and **Severity/Confidence**. It might look like:

```
ID | Region    | Location (x,y) | Size (px) | Severity
1  | Core      | (120, 230)     | 50        | HIGH
2  | Cladding | (305, 280)     | 20        | LOW
```

  This list updates every frame (or every few frames) to reflect new detections. If a defect persists over multiple frames, the UI could keep it with the same ID for easier tracking (this would require some temporal tracking of defects, which could be a nice-to-have but not strictly required). Even without persistent IDs, updating the list each frame with the current defects is informative. The **region name**

for each defect is derived from the segmentation – since we know which zone the defect lies in, we display "Core/Cladding/Ferrule" accordingly. This textual info complements the visual overlay and is important for operators who might need exact data (e.g. to log a defect's position).

- **Statistics Panel:** A small section of the dashboard can show aggregate stats, updating in real-time. For example:
- **Defect Count per Region:** e.g. "Core: 2 defects, Cladding: 1, Ferrule: 0" for the current frame. This gives a quick overview of where most issues are.
- **Severity Summary:** e.g. a count of how many defects are Critical/High/etc. or simply highlight if any high-severity defect is present ("Critical defect detected in CORE region!"). Since we are not doing alarm handling, this is just for display – maybe color-code the text (red text if any critical defect exists).
- **Frame/Processing Rate:** (Optional) show the current processing FPS or latency, to reassure that the system is within real-time bounds (useful for development or if the user is interested in performance). These stats could be displayed as text or simple gauges. For instance, a bar graph could dynamically show counts per region. In a Dash app, this could be a live-updating bar chart; in Streamlit or Flask, just updating text is simpler but effective. Since the focus is defect detection, keeping the stats simple and relevant is best.
- **Layout Structure:** Visually, one could arrange the dashboard with the video feed on the left and the defect list/stats on the right. If using a web framework with a grid layout (CSS or a component library), allocate a larger area for the video. The sidebar could be a scrollable panel if many defects are listed. Another layout option is video on top, and underneath it a row with two columns: left for defect list, right for summary stats. The choice may depend on the display monitor aspect ratio and personal preference. The key is to make sure the video feed is prominent and the text info is adjacent for quick correlation.
- **Styling and UX:** Use clear labels and possibly highlight critical information. For example, if the system finds a "FAIL" condition (if there's a quality score or pass/fail criteria from the pipeline [21]), that could be displayed boldly. Even though we're not alerting, a visual indicator (like the text "PASS" or "FAIL" with a color background) based on the current frame's defects could be useful. Ensure the overlay drawings on the video are semi-transparent or outlined so as not to entirely obscure the fiber image – the operator should still see the fiber surface beneath the markings. Also, provide a legend or color explanation if multiple colors are used for different regions or severities.
- **Interactivity (Optional):** In a richer dashboard, one could allow the user to pause the stream, or capture a snapshot. Another idea is a toggle to switch between viewing the raw video and the annotated video, for comparison. While not strictly required, these features can improve the user experience during inspection tasks.

By implementing this layout, the user gets a comprehensive real-time view: the live video with highlighted defects draws attention immediately to any problem areas, and the side information provides exact details and counts. This mirrors how an inspector might work – first notice a highlighted spot on the fiber image, then check the description ("Defect in cladding region, small size, low severity") to decide how to act. The design thus meets the requirements of showing overlays, coordinates, region names, and stats in an organized, **dashboard-friendly** manner.

Overall, this plan upgrades the fiber optic defect detection pipeline from an offline batch tool to an interactive real-time system. It retains the powerful existing analysis capabilities [4] while introducing streaming and live visualization. With the recommended frameworks and optimizations in place, the end result will be a near real-time dashboard where fiber end-face defects are detected and displayed on the fly, aiding prompt inspection and decision-making.

Sources:

1. Fiber Optic Defect Detection Pipeline – README (Pipeline overview and features) [7] [4]
2. Thiago Alves, *Realtime Webcam Processing Using Streamlit and OpenCV* (OpenCV video capture and streaming) [1] [14]
3. Adrian Rosebrock, *Stream video to web browser with Flask and OpenCV* (Flask + OpenCV streaming approach) [13] [12]
4. Gradio Documentation – *Real Time Object Detection from Webcam* (WebRTC streaming for low latency) [17] [16]
5. OpenCV Forum – *Real-time frame processing discussion* (frame skipping and threading insights) [2]
6. Detection Pipeline Code (OmniFiberAnalyzer usage in loop) [5] [6]

---

[1] [14] [15] Realtime Webcam Processing Using Streamlit and OpenCV | thiagoalves.ai
https://thiagoalves.ai/building-webcam-streaming-applications-with-streamlit-and-opencv/

[2] [18] OpenCV. How do you catch a real time frame from OpenCV Video Capture - OpenCV
https://forum.opencv.org/t/opencv-how-do-you-catch-a-real-time-frame-from-opencv-video-capture/14880

[3] [4] [7] readme.md
file://file-HYRQ1vn8E1Xet87iQTPBGV

[5] [6] [21] app.py
file://file-Vvbk4MNjx28FywVNtFC7GK

[8] [9] [10] [11] [19] [20] detection.py
file://file-XA2QoJ4CmBoxwBcJhhpEne

[12] [13] OpenCV - Stream video to web browser/HTML page - PyImageSearch
https://pyimagesearch.com/2019/09/02/opencv-stream-video-to-web-browser-html-page/

[16] [17] Object Detection From Webcam With Webrtc
https://www.gradio.app/guides/object-detection-from-webcam-with-webrtc