Relazione Progetto: Dashboard per Analisi Parkinson

Studente: Mehdi Ouassou
Esame: Progettazione di App React
Data: 24 Febbraio 2026

---

1. Introduzione e Contesto del Progetto

Questo documento descrive la progettazione e lo sviluppo di "Parkinson Analysis", un'applicazione web full-stack per il comando remoto e la gestione di sessioni di videoregistrazione per pazienti affetti da malattia di Parkinson. Il sistema è stato concepito per un utilizzo in contesti clinici, fornendo agli operatori uno strumento intuitivo ed efficiente per un processo di acquisizione dati altrimenti complesso e soggetto a errori.

L'architettura si basa su una scheda NVIDIA Jetson Orin remota sulla quale sono installate due telecamere Intel RealSense per l'acquisizione video sincronizzata. La dashboard, un'applicazione web sviluppata in React e ottimizzata per laptop, permette di gestire l'intero ciclo di vita della sessione:

- **Registrazione:** Avviare, mettere in pausa, riprendere e interrompere le registrazioni video da entrambe le telecamere.
- **Monitoraggio Live:** Visualizzare i feed video in tempo reale (MJPEG) e monitorare lo stato del sistema.
- **Gestione Dati:** Eseguire la conversione post-registrazione dei dati grezzi (`.bag`) in formato video standard (`.mp4`).
- **Analisi:** Avviare pipeline di analisi basate su intelligenza artificiale (YOLOv8-Pose) per l'estrazione di keypoint scheletrici.

- **Annotazione:** Fornire strumenti per l'annotazione (tagging) frame-by-frame degli eventi motori.
- **File Management:** Gestire, analizzare la qualità, scaricare ed eliminare i file generati.

Sebbene il focus accademico del corso sia sul frontend React, la realizzazione di questo progetto ha richiesto un profondo lavoro di ingegneria full-stack, che è stato fondamentale per il successo dell'applicazione e che verrà descritto in dettaglio.

2. Progetto Full-Stack (Oltre React)

Il requisito iniziale descriveva l'integrazione di un frontend con una pipeline di applicativi pre-esistente. In realtà, non essendo disponibile un backend funzionante al momento dello sviluppo, l'intero sistema è stato sviluppato da zero.

Vista la natura dell'esame, si era inizialmente valutata la possibilità di utilizzare un'API mockata (es. JSON Server). Tuttavia, questa strada è stata scartata per motivi architetturali e di affidabilità: un mock non avrebbe mai potuto simulare le reali complessità hardware e i bug del sistema. Problematiche come i *frame drop* durante la scrittura su disco, la latenza reale dello streaming MJPEG, la gestione dell'offset temporale tra le telecamere e i tempi di attesa per la conversione video NVENC richiedevano un backend reale per poter testare efficacemente la resilienza del frontend. Sviluppare il backend è stato un passaggio obbligato per far emergere e risolvere i bug della UI, garantendo ai ricercatori del laboratorio un prodotto finale realmente funzionante e pronto all'uso clinico.

Questo approccio ha comportato l'affronto di numerose sfide tecniche che dimostrano una competenza estesa a tutto lo stack tecnologico.

2.1. Backend API con FastAPI

La parte fondamentale del sistema sulla Jetson Orin è un'API RESTful asincrona sviluppata in Python con il framework **FastAPI**. Questa scelta è stata dettata dalla necessità di alte prestazioni, gestione efficiente delle operazioni I/O (come lo streaming video) e una robusta validazione dei dati grazie all'integrazione con **Pydantic** (`models.py`).

L'API gestisce endpoint critici per: - **Controllo della registrazione** (`/recording/start` , `/stop` , `/pause` , `/resume`): Implementa una macchina a stati per gestire il ciclo di vita della registrazione. - **Streaming video** (`/camera/{camera_id}`): Fornisce un flusso MJPEG a FPS variabile (30 fps in idle, 10 fps durante la registrazione per risparmiare risorse) per il monitoraggio live. - **Gestione delle telecamere** (`/cameras/info` , `/restart` , `/swap`): Permette di ottenere informazioni sulle telecamere connesse, riavviarle in caso di problemi e scambiare l'assegnazione fisica (Frontale/Laterale). - **Pipeline asincrone** (`/conversion/start` , `/processing/start`): Utilizza le `BackgroundTasks` di FastAPI per avviare processi di lunga durata (conversione video e analisi AI) senza bloccare l'interfaccia utente.

2.2. Integrazione Hardware e Sincronizzazione

Una delle sfide principali è stata l'interfacciamento con le due telecamere Intel RealSense in assenza di un cavo di sincronizzazione hardware.

- **Compilazione di `librealsense` :** È stato necessario compilare da sorgente la libreria `librealsense` e creare un ambiente Python pacchettizzato (tramite un tarball `jetson_orin_py38_env.tar.gz`) per il deployment sulla Jetson, gestendo dipendenze a basso livello.
- **Livello di Astrazione (`camera.py`):** È stato creato un layer di astrazione software per gestire le telecamere, supportando sia dispositivi reali sia la riproduzione da file `.bag` per lo sviluppo e il testing (`mock_bag` mode).
- **Sincronizzazione Software:** Per minimizzare il disallineamento temporale tra le due telecamere, è stato implementato un meccanismo di avvio della registrazione in due fasi basato su `threading.Barrier` :
    1. **Fase di Preparazione (parallela):** Entrambe le telecamere preparano la pipeline di registrazione contemporaneamente (operazione lenta).
    2. **Fase di Commit (sincronizzata):** Le telecamere attendono a una barriera e avviano la registrazione nell'istante esatto in cui l'ultima è pronta (operazione veloce). Questo approccio riduce l'offset di avvio a meno di 100ms, un valore accettabile per l'analisi clinica del movimento.

2.3. Pipeline di Dati e Analisi

L'intero flusso dati è stato progettato per garantire integrità e performance.

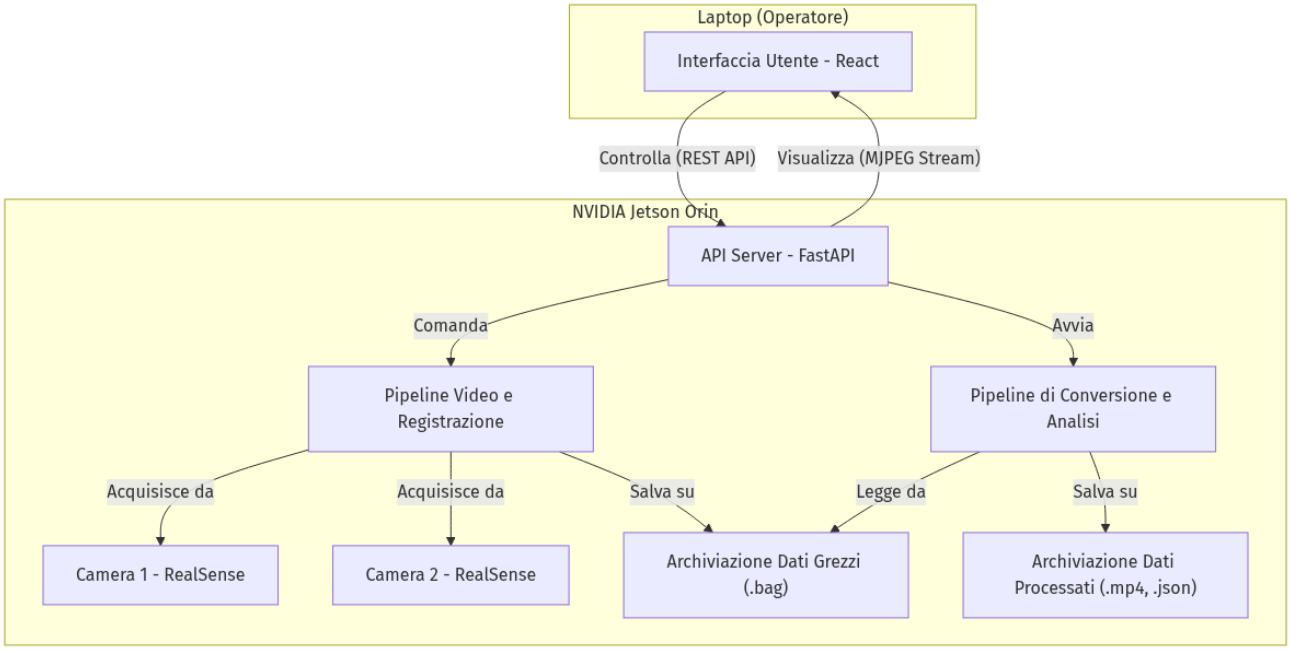
- **Registrazione Lossless (`writers.py`):** Durante la registrazione, i dati grezzi (RGB + profondità) vengono salvati in formato `.bag` direttamente dal SDK RealSense. Questo garantisce frame drops che si avvicinano sullo zero, un requisito fondamentale per l'analisi scientifica.
- **Conversione Post-Registrazione (`conversion.py`):** Dopo la sessione, l'utente può avviare un processo di conversione da `.bag` a `.mp4`. La pipeline seleziona in automatico l'encoder più performante disponibile (NVENC su Jetson/Desktop, altrimenti fallback su CPU) e gestisce la validazione dei file per evitare output corrotti.
- **Analisi con YOLOv8 (`processing.py`):** Una pipeline di analisi esegue l'inferenza con un modello `yolov8n-pose` per estrarre 17 keypoint scheletrici da ogni frame. Questa pipeline è ottimizzata per la Jetson tramite la generazione on-device di un motore **TensorRT**.

2.4. Deployment su NVIDIA Jetson

La distribuzione del software su un dispositivo embedded come la Jetson Orin ha richiesto la creazione di script di automazione (`deploy/setup_jetson.sh`) per: - Installare le dipendenze di sistema. - Scaricare e scompattare l'ambiente Python pre-compilato. - Generare il motore TensorRT ottimizzato per l'hardware specifico, un'operazione che richiede 10-15 minuti ma che garantisce un notevole aumento delle performance di inferenza.

3. Architettura del Sistema

L'architettura del sistema è chiaramente suddivisa tra il laptop dell'operatore, che ospita l'interfaccia utente (con la possibilità di hostare anche il frontend sulla Jetson), e la NVIDIA Jetson Orin, che gestisce tutta la logica di acquisizione e processamento.



Questa architettura client-server disaccoppia l'interfaccia utente dalla logica di processing, permettendo di avere un client leggero e reattivo e di concentrare le operazioni computazionalmente intensive sul dispositivo edge.

4. Analisi Approfondita del Frontend React

L'applicazione frontend è costruita con un approccio moderno per garantire performance, manutenibilità e un'eccellente esperienza utente in un contesto clinico.

Stack Tecnologico:

- **React 19:** Utilizzo delle ultime funzionalità della libreria, inclusi gli hook per la gestione dello stato e del ciclo di vita dei componenti.
- **Vite:** Build tool di nuova generazione che offre un'esperienza di sviluppo estremamente rapida grazie all'Hot Module Replacement (HMR) istantaneo.
- **TypeScript:** Garantisce un codice più robusto e auto-documentato grazie alla tipizzazione statica, riducendo gli errori a runtime.
- **React Router 7:** Gestisce la navigazione client-side tra le diverse sezioni dell'applicazione in modo pulito e dichiarativo.

- **Tailwind CSS:** Framework utility-first che ha permesso di costruire un'interfaccia utente custom, moderna e responsiva in modo rapido e consistente.

4.1. Struttura dei Componenti e Routing

Il progetto è organizzato in modo modulare, con una chiara separazione delle responsabilità:

- `src/pages`: Ogni file corrisponde a una delle sezioni principali dell'applicazione (`CameraFeeds.tsx`, `Conversion.tsx`, `Tagging.tsx`, `Processing.tsx`, `FileManager.tsx`).
- `src/components`: Contiene componenti riutilizzabili come `Layout.tsx` (che definisce la struttura comune con navigazione e footer), `Toast.tsx` (per le notifiche a comparsa) e `ErrorBoundary.tsx` per catturare errori JavaScript nell'albero dei componenti ed evitare il crash dell'intera applicazione.

Il routing, gestito in `App.tsx`, definisce la struttura di navigazione e associa ogni rotta al componente della pagina corrispondente, nidificandole all'interno del `Layout` principale.

```
// in src/App.tsx
const router = createBrowserRouter([
  {
    path: '/',
    element: <Layout />,
    children: [
      { index: true, element: <CameraFeeds /> },
      { path: 'conversion', element: <Conversion /> },
      // ... altre pagine
    ],
  },
]);

function App() {
  return (
    <ErrorBoundary>
      <ToastProvider>
        <RouterProvider router={router} />
      </ToastProvider>
    </ErrorBoundary>
  );
}
```

Questo setup garantisce che ogni pagina venga renderizzata all'interno di una struttura coerente (`Layout`), che contiene la barra di navigazione, il footer e il contesto per le notifiche (`ToastProvider`).

4.2. Pagina `CameraFeeds`: Gestione dello Stato in Tempo Reale

La pagina `CameraFeeds` gestisce lo stato della sessione di registrazione, sincronizzandosi in tempo reale con il dispositivo hardware remoto.

L'implementazione si basa su una macchina a stati finiti (`'idle'`, `'recording'`, `'paused'`, ecc.) per un controllo predicibile dell'interfaccia utente. Un `useEffect` hook avvia un'operazione di *polling* (interrogazione periodica) verso l'endpoint `/recording/status` per sincronizzare lo stato del frontend con quello del backend, che funge da "source of truth". Questo pattern assicura che metriche come la durata della registrazione e i frame catturati siano costantemente aggiornate.

```
// in src/pages/CameraFeeds.tsx

// Lo stato della registrazione è definito in modo esplicito
type RecordingState = 'idle' | 'initializing' | 'warming_up' | 'recording' | 'paused' |
'stopping';
const [recordingState, setRecordingState] = useState<RecordingState>('idle');

// Questo useEffect si attiva quando lo stato della registrazione cambia
useEffect(() => {
    // Il polling è attivo solo durante la registrazione
    const shouldPoll =
      recordingState === 'warming_up' ||
      recordingState === 'recording' ||
      recordingState === 'paused';

    if (!shouldPoll) return;

    // Funzione che interroga il backend
    const poll = async () => {
      try {
        const res = await fetch(`${API_URL}/recording/status`, { signal:
pollController.signal });
        const data = await res.json();
        if (!mountedRef.current) return; // Evita aggiornamenti su componente smontato

        // Aggiorna le metriche live (durata, frame, etc.)
        setRecordingDuration(data.duration ?? null);
        setFrameCounts(data.frame_counts ?? {});
      } catch (err) { /* ... */ }
    };

    poll(); // Chiamata immediata
    const interval = setInterval(poll, 1000); // E poi ogni secondo

    // Funzione di pulizia per fermare il polling
    return () => clearInterval(interval);
}, [recordingState]);
```

4.3. Pagine `Conversion` e `Processing`: Gestione di Task Asincroni

Queste sezioni gestiscono operazioni di lunga durata (conversione video e analisi AI) che vengono eseguite sul backend.

L'approccio implementativo garantisce la persistenza dello stato delle operazioni anche in caso di ricaricamento della pagina. Quando un'operazione viene avviata, il suo `job_id` univoco viene salvato nel `localStorage` del browser. Al ricaricamento della pagina, un

`useEffect` hook verifica la presenza di un `job_id` salvato, interroga il backend per ottenere lo stato corrente del processo e, se necessario, riprende il polling dello stato. Questo previene la perdita di contesto per l'utente su operazioni che possono richiedere diversi minuti.

```
// in src/pages/Conversion.tsx

// All'avvio, controlla se c'era un lavoro in corso
useEffect(() => {
    const savedJobId = localStorage.getItem('conversion_job_id');
    if (!savedJobId) return;

    // Chiede al backend lo stato di quel lavoro
    (async () => {
      const res = await fetch(`${API_URL}/conversion/status/${savedJobId}`);
      const data = await res.json();
      if (data.success && data.job && ['pending',
'converting'].includes(data.job.status)) {
        setCurrentJob(data.job);
        setIsLoading(true);
        // Se il lavoro è ancora in corso, fa ripartire il polling
        startStatusPolling(savedJobId);
      } else {
        localStorage.removeItem('conversion_job_id');
      }
    })();
}, [batches]); // Si attiva dopo aver caricato la lista dei batch

// Salva o rimuove il job_id quando lo stato del lavoro cambia
useEffect(() => {
    if (currentJob && (currentJob.status === 'converting' || currentJob.status ===
'pending')) {
      localStorage.setItem('conversion_job_id', currentJob.job_id);
    } else if (currentJob) {
      localStorage.removeItem('conversion_job_id');
    }
}, [currentJob]);
```

4.4. Pagina `Tagging`: Interattività e Manipolazione Media

Questa pagina fornisce uno strumento di annotazione video per marcare eventi specifici con precisione a livello di singolo frame.

La pagina è costruita attorno a un elemento `<video>` HTML5, il cui stato di riproduzione è controllato da React. Gli hook `useRef` consentono l'accesso diretto ai metodi dell'elemento DOM (es. `play()`, `pause()`, `currentTime`), mentre lo stato di React (`useState`) viene utilizzato per tracciare il tempo di riproduzione e pilotare l'interfaccia. L'interfaccia permette di registrare le annotazioni (logs) in un array, che viene poi utilizzato per il salvataggio dei dati.

```
// in src/pages/Tagging.tsx

const videoRef = useRef<HTMLVideoElement>(null);
const [currentTime, setCurrentTime] = useState(0);
const [videoFps, setVideoFps] = useState(30);
const [actionLogs, setActionLogs] = useState<ActionLog[]>([]);

// Calcola il frame corrente basandosi sul tempo di riproduzione
const currentFrame = Math.floor(currentTime * videoFps);

// Funzione per aggiungere un'annotazione
const addActionLog = (direction: number) => {
    // Aggiunge una nuova riga alla tabella dei log
    setActionLogs(prev => [
        ...prev,
        { id: logIdCounter, frame: currentFrame, direction, action: DIRECTIONS[direction]
}
    ]);
    setHasUnsavedChanges(true); // Marca che ci sono modifiche non salvate
};

// Funzione per navigare frame-by-frame
const skipFrames = (frameCount: number) => {
    if (!videoRef.current) return;
    // Calcola il nuovo tempo e si assicura che non vada fuori dai limiti del video
    const newTime = Math.max(0, Math.min(duration, videoRef.current.currentTime +
frameCount / videoFps));
    videoRef.current.currentTime = newTime; // Imposta direttamente il tempo sul player
    setCurrentTime(newTime); // Aggiorna lo stato di React
};
```

4.5. Pagina `FileManager`: Gestione Dati e Download Progressivi

Questa pagina agisce come un file browser per i dati generati sul server, offrendo funzionalità di download, eliminazione e analisi della qualità.

Una funzionalità tecnica di rilievo è la gestione dei download di file di grandi dimensioni. Viene utilizzata l'API `fetch` con `ReadableStream` per scaricare i file in modo progressivo. Questo permette di monitorare l'avanzamento del download (byte ricevuti vs. totali) e di aggiornare una barra di progresso nell'interfaccia utente, migliorando significativamente l'esperienza utente rispetto a un download opaco.

```
// in src/pages/FileManager.tsx

const [downloadProgress, setDownloadProgress] = useState<Record<string,
DownloadProgress>>({});

const downloadFile = async (type: 'video' | 'bag', filename: string) => {
    // ...
    const res = await fetch(url, { signal: controller.signal });
    const total = parseInt(res.headers.get('content-length') || '0', 10);
    const reader = res.body!.getReader();
    let received = 0;

    // Legge i dati in arrivo in blocchi (chunks)
    while (true) {
        const { done, value } = await reader.read();
        if (done) break;
        received += value.length;

        // Aggiorna lo stato per il rendering della barra di progresso
        setDownloadProgress(prev => ({
            ...prev,
            [filename]: { received, total, percentage: (received / total) * 100, /* ...
*/ }
        }));
    }

    // A fine download, crea un URL locale (Blob URL) e lo usa per triggerare il
salvataggio nel browser
    const blob = new Blob(chunks);
    const downloadUrl = window.URL.createObjectURL(blob);
    const a = document.createElement('a');
    a.href = downloadUrl;
    a.download = filename;
    a.click();
    // ...
};
```

5. Conclusioni

Il progetto rappresenta una soluzione software complessa e completa, che va ben oltre la semplice implementazione di un'interfaccia utente. L'utilizzo di React, TypeScript e di un'architettura a componenti moderna ha permesso di creare un frontend robusto, manutenibile e user-friendly, perfettamente adatto a un contesto clinico. Allo stesso tempo, il lavoro svolto sull'intero stack tecnologico, dall'interfacciamento hardware a basso livello con le telecamere RealSense, alla progettazione di un'API backend performante, fino al deployment ottimizzato su un dispositivo edge come la Jetson Orin, dimostra una visione architetturale completa e la capacità di affrontare e risolvere problematiche di ingegneria del software complesse. Il risultato è un'applicazione pronta per l'uso, che risolve un problema reale con una soluzione tecnologicamente avanzata.

L'esperienza di sviluppo presso il Parco Tecnologico è stata estremamente formativa e gratificante. La possibilità di lavorare su un progetto full-stack, gestendo hardware avanzato come la NVIDIA Jetson AGX Orin e due telecamere Intel RealSense, ha offerto una sfida stimolante e completa. Il riscontro positivo del tutor sul risultato finale è motivo di grande soddisfazione. Si spera che questo lavoro possa servire come solida base per futuri progetti in ambito medico; proprio in quest'ottica è stata data priorità a una documentazione chiara e alla creazione di artefatti di deployment riutilizzabili, come il tarball dell'ambiente, per facilitare la comprensione e l'adozione del sistema da parte di futuri sviluppatori.