Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

AI Tools for Media Servers

Project Engineering

Year 4

Cillian Jennings

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

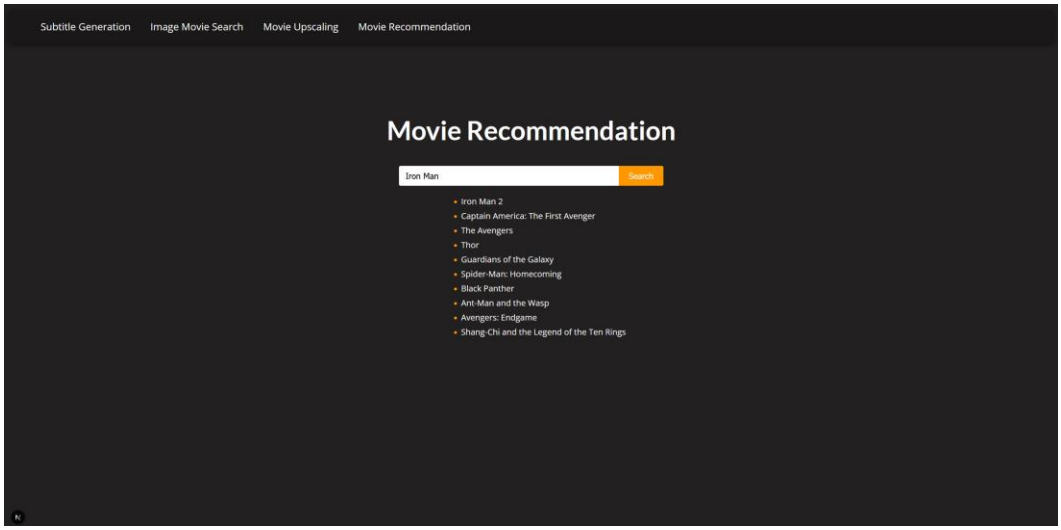

Figure 0.1 – Movie Recommendation Screen

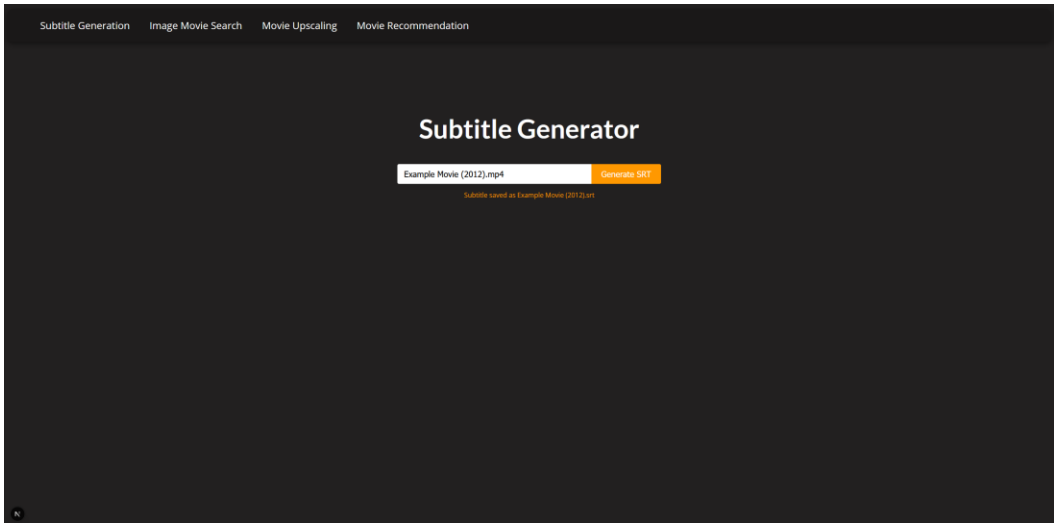

Figure 0.2 – Subtitle Generator Screen

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Acknowledgements

Thanks to my supervisor David Newell as well as the other Project Engineering coordinators Paul Lennon, Michelle Lynch, Niall O'Keefe and Brian O'Shea.

Table of Contents

1 Summary

For my final year project, I wanted to develop a website that would give the user useful tools to help manage their media server. It is a website that gives the user access to four useful tools to help manage/use their personal media server. The first feature is a media recommendation tool, where you input a movie, you recently watched, and it will give you recommendations on similar movies. The second feature is the subtitle generation tool which allows you to select a movie from your media library and it will generate a .srt subtitle file for it. The third feature is the image movie search tool, which allows you to upload a screenshot of a scene from a film and it will tell you the name of the film its from. The final feature is a movie upscaling tool where you select a movie from your media library, and it will upscale it, increasing the quality and resolution of the film.

For project management, Jira was used for tracking work throughout the duration of the project, along with OneNote which was used for logging of information like websites and such. Technologies such as React and Next.js were used for the frontend. Backend technologies include FastAPI, Uvicorn, and Pydantic. For machine learning/artificial intelligence the technologies used include: OpenAI's CLIP, OpenAI's Whisper, Meta's Llama model using llama.cpp, PyTorch, Meta's FAISS, and Real-ESRGAN. Other technologies include FFmpeg.

In completing this project, I have developed and improved my skills in project planning, management and documentation. I have improved software engineering skills and knowledge through working with these technologies and developing this project. I have increased my knowledge of how AI works and created a useful website thereby accomplishing my goals.

2 Poster

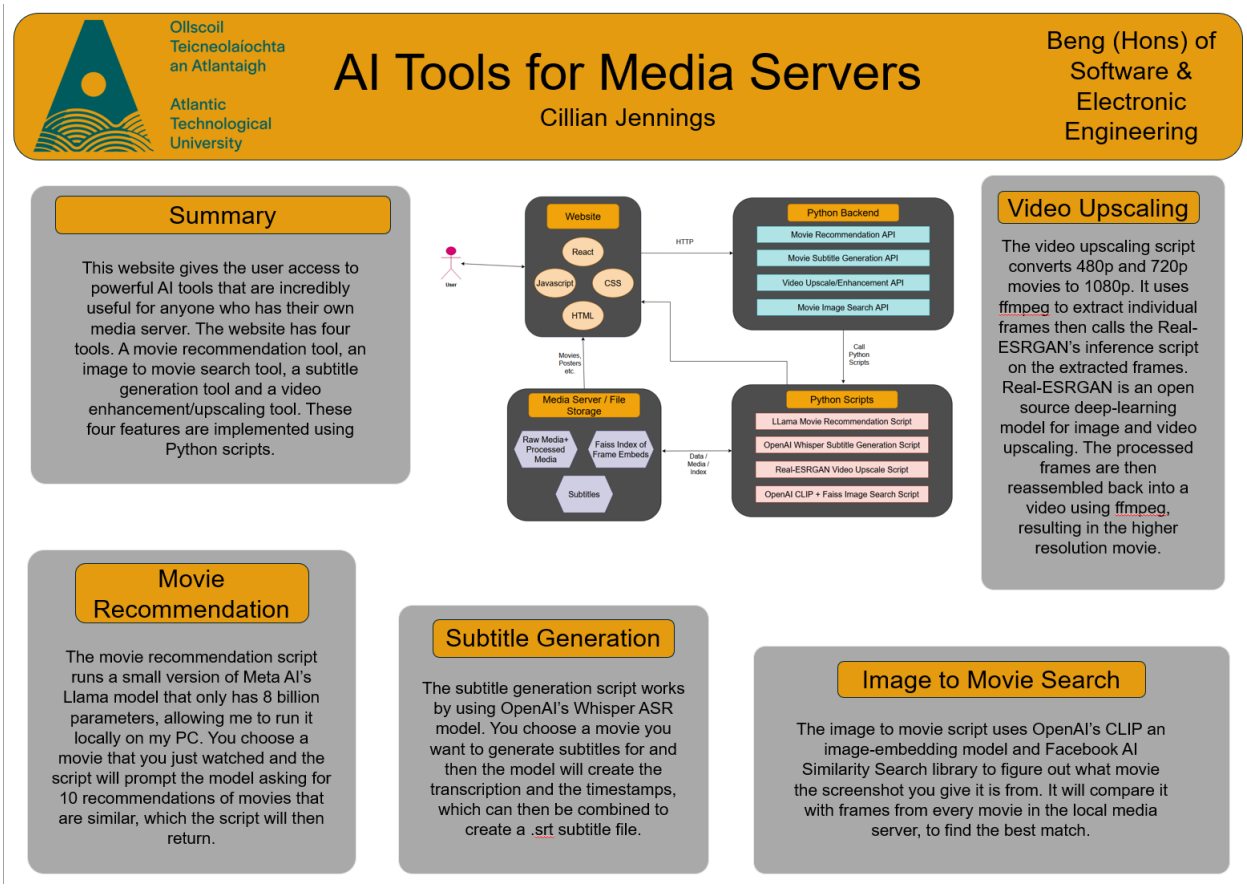

Figure 2.1 - Poster

3 Introduction

The objective of this project was to create a website that has useful media management tools to complement my media server while using and learning more about AI models. AI and machine learning are hot topics nowadays and with AI only getting more advanced by the day, it is clear that it will play a big role in the future, so I wanted to learn more about artificial intelligence. I have had a media server at home for the past 10 years and have been slowly upgrading it, from a raspberry pi 2 up to the 4 then a windows mini pc and then finally I built a pc with a bit more computing power and a GPU so I could try running so more demanding workloads on it. I decided I wanted to create a website that had useful tools for my media server.

The website has four tools. A video upscaling tool that improves the quality of my media. A movie image search tool that allows you to upload a screenshot of a scene from a movie and it will tell you the movie name. A subtitle generation tool where you can select the movie, and it will generate a subtitle .srt file for it. Finally, the movie recommendation tool where you enter a movie you just watched, and it will give you recommendations similar to that movie.

4 Tools and Technologies

4.1 React ***

React is a JavaScript library for building user interfaces. It lets you write independent components which can easily be combined to build pages. Instead of directly editing the DOM (the page) every time you want to change the way something looks, React uses a virtual DOM which it can quickly edit, and it will only make changes to the real DOM when everything is ready [1].

4.2 Next.js ***

Next.js is a React framework for building web applications with server-side rendering, static site generation, routing, and API routes all built in [2]. It has built-in support for React Server Components, Dynamic HTML Streaming, CSS support and more.

4.3 FastAPI ***

FastAPI is a modern, fast, web framework for building APIs with Python based on standard Python type hints [3]. It uses Pydantic for data validation and Starlette for the underlying ASGI server. FastAPI automatically generates interactive OpenAPI documentation and supports async endpoints for high performance.

4.4 Uvicorn

Uvicorn is an ASGI web server implementation for Python [4]. It runs asynchronous Python web applications (like those built with FastAPI) by handling event loops, HTTP parsing, and protocol implementation.

4.5 Pydantic

Pydantic is a data-validation library that uses Python type annotations to parse and validate incoming data [5].

4.6 OpenAI CLIP ***

OpenAIs CLIP (Contrastive Language-Image Pre-Training) is a model that learns a joint embedding space for images and text by training on pairs of images and their captions [6]. You can encode an image or a text prompt into a vector, then find matches via cosine similarity.

4.7 FAISS ***

FAISS is a library for efficient similarity search over large vector collections. HNSW (Hierarchical Navigable Small Worlds) is a graph-based approximate nearest neighbor algorithm [7]. It builds layers of increasingly sparse proximity graphs, allowing very fast searches by walking down from the top layer to approximate nearest neighbors.

4.8 OpenAI Whisper

Whisper is a general-purpose speech recognition model. It is trained on a large dataset of diverse audio and is also a multitasking model that can perform multilingual speech recognition, speech translation, and language identification [8].

4.9 Real-ESRGAN***

Real-ESRGAN is a GAN-based image super-resolution model designed to upscale low-resolution images by learning how to generate extra details [9]. It enhances the resolution of images, eliminating issues such as blurring and noise.

4.10 Llama.cpp***

Llama.cpp is a C/C++ implementation of Meta's LLaMA models [10]. It allows you to run models locally using a script. The main goal of llama.cpp is to enable LLM inference with minimal setup and good performance on a wide range of hardware. The Llama model I used was bartowski's 8 billion parameter instruct model [11].

4.11 PyTorch

PyTorch is a tensor and deep-learning library that provides imperative (eager) execution, dynamic graphs, and automatic differentiation [12]. PyTorch hooks into CUDA drivers to offload tensor operations to the GPU, accelerating large-matrix math common in neural nets.

4.12 FFmpeg***

FFmpeg is a command-line suite for handling multimedia: it can decode, encode, transcode, mux, demux, stream, filter, and play virtually any media format [13]. Ffprobe is a companion tool to inspect media files. It can extract metadata like codecs, bitrates, frame rates, resolutions, and stream layouts in machine-readable form.

5 Project Architecture

This is the architecture diagram for my project.

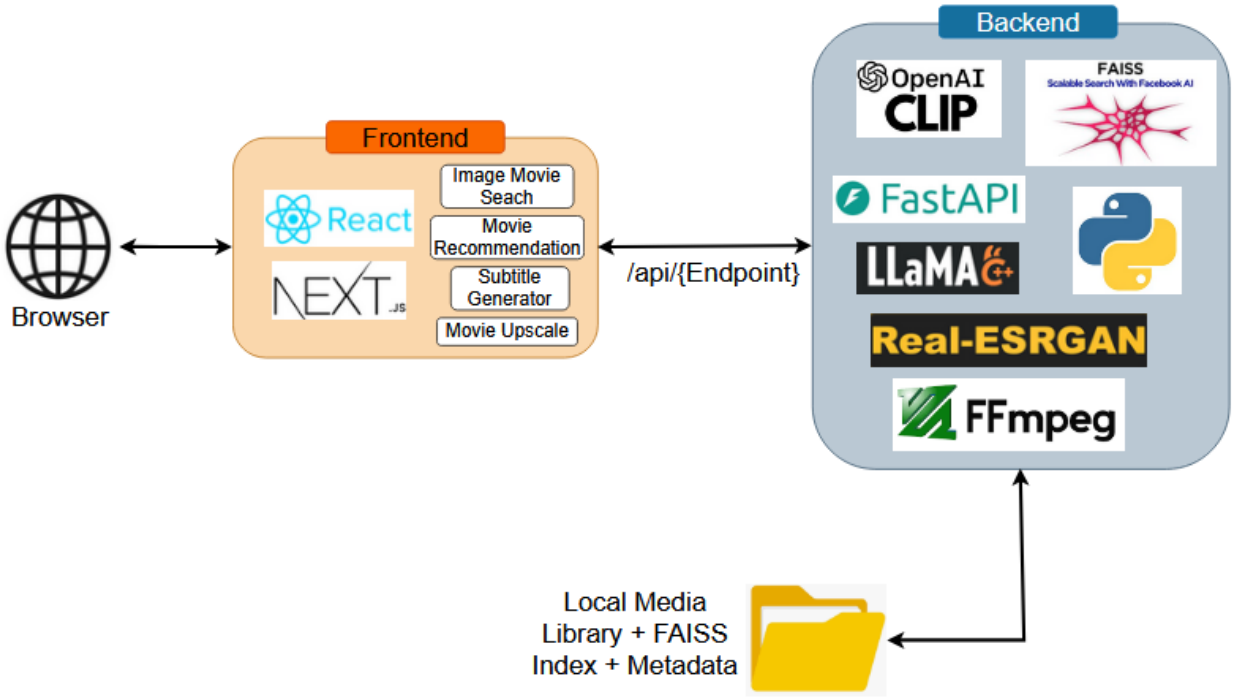

Figure 5.1 - Architecture Diagram

6 Project Plan

I used Jira as a project management tool. I created epics for each feature and marked it as done when completed.

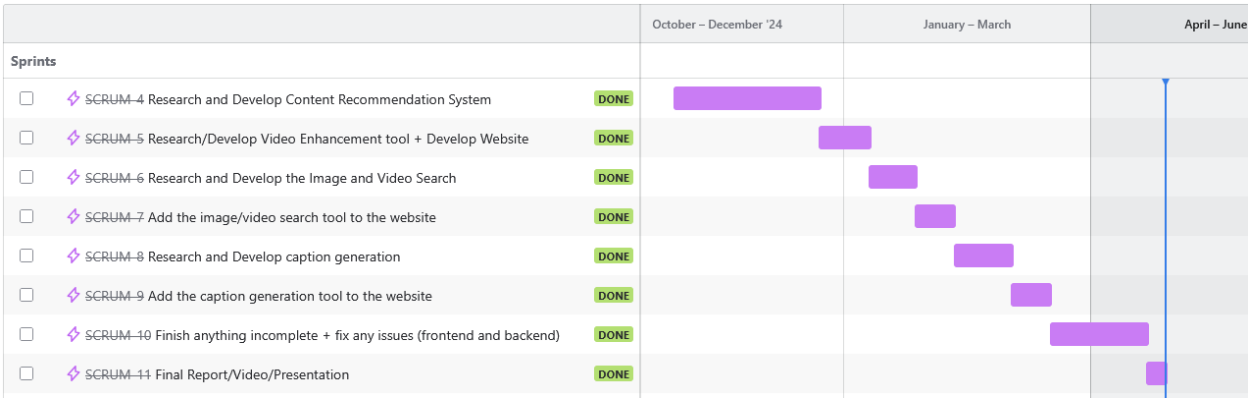

Figure 6.1 - Jira Project Roadmap

7 Frontend Code

This section will go into detail on the important code and software elements of the project. This section is specifically for the frontend code.

7.1 Main Navigation

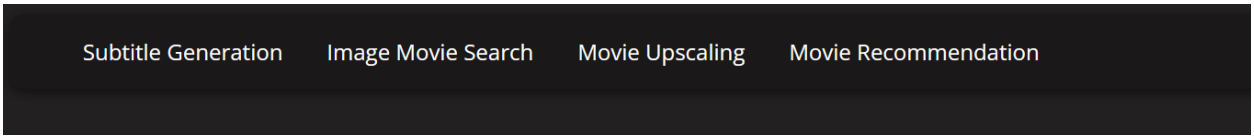

Figure 7.1.1 – Navigation Bar

This is the navigation bar that allows you to select which of the four tools you want to use. This uses Next.js link component to navigate between the four routes [14].

```
return (
  <header className={classes.header}>
    <nav>
      <ul>
        <li><Link href='/subtitle-generation'>Subtitle Generation</Link></li>
        <li><Link href='/image-search'>Image Movie Search</Link></li>
        <li><Link href='/movie-upscaling'>Movie Upscaling</Link></li>
        <li><Link href='/movie-recommend'>Movie Recommendation</Link></li>
      </ul>
    </nav>
  </header>
)
```

Figure 7.1.2 – Navigation Bar Code

7.2 Image Search Page

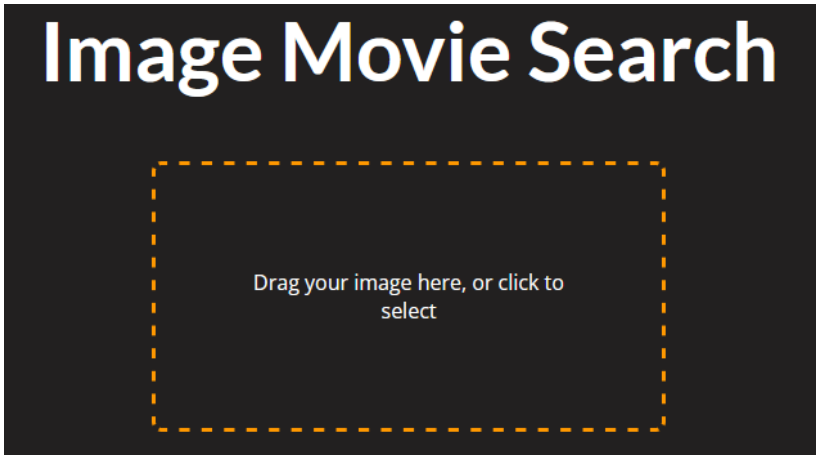

Figure 7.2.1 – Image Search

```
const [file, setFile] = useState(null)
const [result, setResult] = useState(null)
const [error, setError] = useState(null)
const [loading, setLoading] = useState(false)
```

Figure 7.2.2 – State Hooks

This code initialises 4 react useState hooks [15]. The file object the user uploaded, the result which is the JSON response from the /search-image endpoint, error which is any error message to display and a boolean flag called loading.

```
const uploadImage = async (file) => {
    setLoading(true)
    setError(null)
    setResult(null)

    const formData = new FormData()
    formData.append('file', file)

    const base = process.env.NEXT_PUBLIC_API_BASE_URL || 'http://localhost:8000/api';
    try {
        const res = await fetch(`${base}/search-image`, {
            method: 'POST',
            body: formData,
        })
        if (!res.ok) {
            const msg = await res.text()
            throw new Error(msg || 'Upload failed')
        }
        const json = await res.json()
        setResult(json)
    } catch (err) {
        console.error(err)
        setError('Sorry, could not process that image.')
    } finally {
        setLoading(false)
    }
}
```

Figure 7.2.3 – uploadImage Function

First it sets loading to true and clears any errors or results. Then it creates a FormData object and appends the file. Creates base variable containing the fetch URL. The environment variable is just a URL with the ip address allowing me to use the website from another PC on the network. Next it POSTs the form data directly to the FastAPI /search-image endpoint. If the response isn't 'ok' it'll print the error or reason why. On success, the JSON is stored in result. Finally, there is some error checking, network or server error will get that error message, and loading is set back to false.

```
const handleDrop = useCallback(
    (e) => {
        e.preventDefault()
        if (e.dataTransfer.files?.[0]) {
            const f = e.dataTransfer.files[0]
            setFile(f)
            uploadImage(f)
        }
    },
    []
)

const handleDragOver = useCallback((e) => {
    e.preventDefault()
}, [])
```

Figure 7.2.4 – handleDrop Function

The useCallback hook returns a memoized callback function [16], this prevents the component from re-rendering unless its props have changed. The preventDefault method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur [17]. The function first prevents the browser's default (which might open the image), grabs the first file, saves it and then calls uploadImage. HandleDragOver just calls preventDefault so the browser knows we are willing to accept a drop here.

```
const handleChange = useCallback(
    (e) => {
        if (e.target.files?.[0]) {
            const f = e.target.files[0]
            setFile(f)
            uploadImage(f)
        }
    },
    []
)
```

Figure 7.2.5 – handleDrop Function

This function is pretty much the same as the previous one this is just for when the user clicks and picks a file manually rather than dragging the file in.

```
<main className={styles.main}>
    <h1 className={styles.title}>Image Movie Search</h1>

    <div
        className={styles.dropzone}
        onDrop={handleDrop}
        onDragOver={handleDragOver}
        onClick={() => document.getElementById('fileInput').click()}
    >
        {file ? (
            <p>Uploading: {file.name}…</p>
        ) : (
            <p>Drag your image here, or click to select</p>
        )}
        <input
            id="fileInput"
            type="file"
            accept="image/*"
            onChange={handleChange}
            className={styles.fileInput}
        />
    </div>
```

Figure 7.2.6 – Layout

Its all in a main container that is styled with CSS Modules. There is a div with the dropzone styles that handles the drag and drop, plus the click to upload file. If file is set it will show that message otherwise it will show the other message.

```
    {result && (
        <div className={styles.result}>
            <h2>Best match</h2>
            <p>
                <strong>Movie:</strong> {result.movie}
            </p>
            <p>
                <strong>Timestamp:</strong> {result.seconds}s
            </p>
            <p>
                <strong>Score:</strong> {result.score.toFixed(3)}
            </p>
        </div>
    )}
</main>
```

Figure 7.2.7 – Layout2

While loading is true the page shows that message, if it gets an error, it will print it. When result arrives, it displays a styled div that shows the best-match movie, the timestamp and the similarity score.

7.3 Movie Recommend Page

```
const [query, setQuery] = useState('')
const [recommendations, setRecommendations] = useState([])
const [error, setError] = useState(null)
```

Figure 7.3.1 – useState

3 useState hooks are initialised, query: the current text in the search input, recommendations: an array of movie titles and error: any error message to display

```
const handleSearch = async (e) => {
  e.preventDefault()
  setError(null)
  setRecommendations([])

  const base = process.env.NEXT_PUBLIC_API_BASE_URL || 'http://localhost:8000/api';
  try {
    const res = await fetch(`${base}/recommend`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ title: query }),
    })
    const { recommendations: recs } = await res.json()

    // if the API returned our sentinel message, treat as error
    if (recs.length === 1 && recs[0] === 'Error') {
      setError('Sorry, something went wrong – please try again.')
    } else {
      setRecommendations(recs)
    }
  } catch (err) {
    console.error(err)
    setError('Network error – please check your connection.')
  }
}
```

Figure 7.3.2 – handleSearch function

I won't go into much detail on repeated code that I have already explained how it works, but anyways we have the preventDefault method then it clears out any previous errors or recommendations. Then we have a POST request again this time to the /recommend endpoint

instead. It waits for the JSON response then extracts the recommendations. Then we have some error handling, if the backend sends the word 'Error' it will display that message. This unfortunately may happen kind of often, as sometimes the model outputs data that the script couldn't identify as a python list. Will go into more detail on this later. Don't worry though just press submit button again and you should get a proper response, worse case I have seen is it failed four times in a row, but most of the time its fine.

```
return (
  <>
    <main className={styles.main}>
      <h1 className={styles.title}>Movie Recommendation</h1>

      <form className={styles.searchForm} onSubmit={handleSearch}>
        <input
          type="text"
          placeholder="Search for movies…"
          value={query}
          onChange={(e) => setQuery(e.target.value)}
          className={styles.searchInput}
        />
        <button type="submit" className={styles.searchButton}>
          Search
        </button>
      </form>

      {/* show error if any */}
      {error && <p className={styles.error}>{error}</p>}

      {/* show list of recommendations */}
      {recommendations.length > 0 && (
        <ul className={styles.resultsList}>
          {recommendations.map((movie, idx) => (
            <li key={idx}>{movie}</li>
          ))}
        </ul>
      )}
    </main>
  </>
)
```

Figure 7.3.3 – Layout

There is main container with a CSS styles module again. Then we have the form container with again more css styling, inside it we have input which is just the search bar where you can input the movie you want. This is bound to query. When you press the button, it submits the form. If there is an error, it will display it. When it gets the recommendations, it displays them using .map.

7.4 Movie Upscaling Page

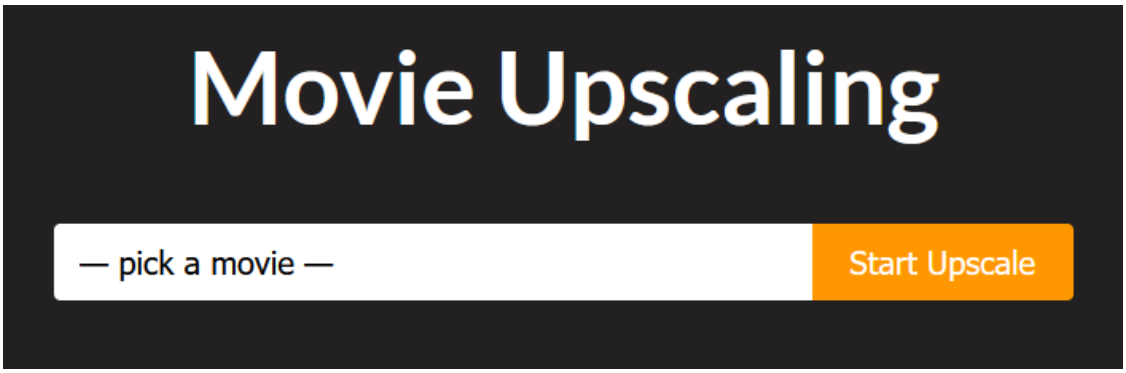

Figure 7.4.1 – Movie Upscaling Page

```
const [movies, setMovies] = useState([])
const [selected, setSelected] = useState("")
const [loading, setLoading] = useState(false)
const [message, setMessage] = useState(null)
const [error, setError] = useState(null)
```

Figure 7.4.2 – useState Hooks

We start with useState hooks again. Starting from the top we have an array of filenames which will be fetched from api/movies later, then we have the currently chosen movie from the dropdown menu, next is the flag that ill explain later, then we have the message that will be returned by the API, and finally any error messages.

```
useEffect(() => {
    fetch(`${base}/movies`)
        .then(r => r.json())
        .then(setMovies)
        .catch(() => setError("Could not load movie list"))
}, [])
```

Figure 7.4.3 – useEffect

The useEffect hook allows you to perform side effects in your components [18]. This hook calls GET /api/movies and then parses the JSON into movies hook. If it fails it sends that error message.

```
async function handleSubmit(e) {
    e.preventDefault()
    if (!selected) return
    setLoading(true); setError(null); setMessage(null)
    try {
        const res = await fetch(`${base}/upscale`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ movie: selected }),
        })
        if (!res.ok) throw new Error(await res.text())
        const { message } = await res.json()
        setMessage(message)
    } catch {
        setError("Upscaling failed")
    } finally {
        setLoading(false)
    }
}
```

Figure 7.4.4 – handleSubmit Function

Starts with preventDefault again, preventing default form submission then if no movie is selected it will return early. It then clears prior messages and errors and sets loading to true. It then sends POST to /upscale with the movie that was selected. If it errors, it will read the body text and throws to the catch statement. If it succeeds it displays the message received from the backend. It then puts loading back to false.

```
return (
    <main className={styles.main}>
        <h1 className={styles.title}>Movie Upscaling</h1>
        {error && <p className={styles.error}>{error}</p>}
        {message && <p className={styles.message}>{message}</p>}

        <form onSubmit={handleSubmit} className={styles.searchForm}>
            <select
                className={styles.searchInput}
                value={selected}
                onChange={e => setSelected(e.target.value)}
                required
            >
                <option value="">— pick a movie —</option>
                {movies.map(m => (
                    <option key={m} value={m}>{m}</option>
                ))}
            </select>
            <button
                type="submit"
                className={styles.searchButton}
                disabled={loading || !selected}
            >
                {loading ? "Upscaling…" : "Start Upscale"}
            </button>
        </form>
    </main>
)
```

Figure 7.4.5 – Layout

Again, everything uses CSS modules. If there is an error, it will be displayed on the page, if message is set then it will display it on the page. There is a form container again, inside which we have a select element that creates a drop-down list. It is disabled when loading is set to prevent changes mid-job, when no movie is selected is shows “pick a movie”. Just below that it maps movies into options elements, displaying all the movies you can choose. Then there is a submit button which you press when you have selected the movie. The button is disabled if loading is set or nothing is selected. Finally, it displays that message while waiting for a response from the API.

7.5 Subtitle Generation Page

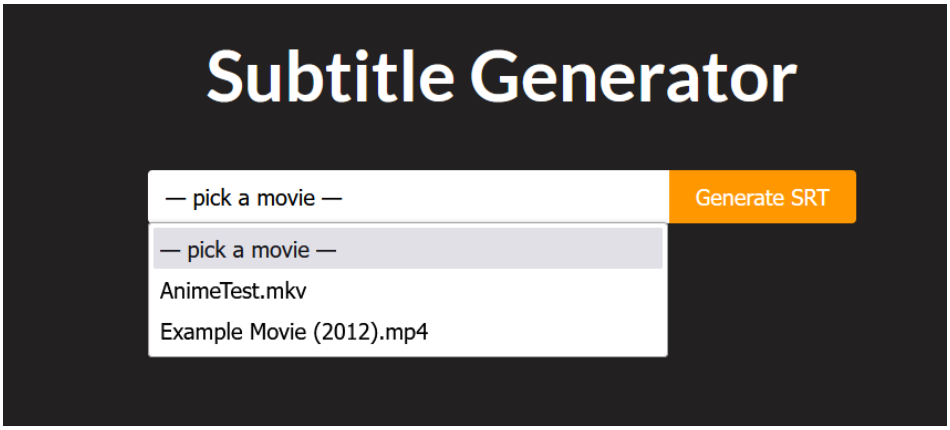

Figure 7.5.1 – Subtitle Generator Page

```
useEffect(() => {
  fetch(`${base}/movies`)
    .then(r => r.json())
    .then(setMovies)
    .catch(() => setError("Could not load movie list"))
}, [])
```

Figure 7.5.2 – useEffect

The subtitle generator page is very similar to the movie upscaling page. Here we again use useEffect. It will run once after the component first renders, calls GET api/movies parses the JSON into an array and stores it in movies, if it fails sets that error message.

```
async function handleSubmit(e) {
  e.preventDefault()
  if (!selected) return
  setLoading(true)
  setError("")
  setMessage("")

  try {
    const res = await fetch(`${base}/generate-subs`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ movie: selected }),
    })
    if (!res.ok) throw new Error(await res.text())
    const { message: msg } = await res.json()
    setMessage(msg)
  } catch {
    setError("Subtitle generation failed")
  } finally {
    setLoading(false)
  }
}
```

Figure 7.5.3 – handleSubmit function

Almost exactly the same as the handleSubmit function I explained before, the only differences being it POSTs to /generate-subs and the error message is different.

```
return (
  <main className={styles.main}>
    <h1 className={styles.title}>Subtitle Generator</h1>

    <form onSubmit={handleSubmit} className={styles.searchForm}>
      <select
        className={styles.searchInput}
        value={selected}
        onChange={e => setSelected(e.target.value)}
        disabled={loading}
        required
      >
        <option value="">— pick a movie —</option>
        {movies.map(m => (
          <option key={m} value={m}>{m}</option>
        ))}
      </select>
      <button
        type="submit"
        className={styles.searchButton}
        disabled={loading || !selected}
      >
        {loading ? "Generating…" : "Generate SRT"}
      </button>
    </form>

    {message && <p className={styles.message}>{message}</p>}
    {error   && <p className={styles.error}>{error}</p>}
  </main>
)
```

Figure 7.5.4 – Layout

This is almost exactly the same as the code explained in figure 7.4.5 with the only differences being the message displayed while loading and the title.

8 Backend Code

This section will go into detail on the important code and software elements of the project. This section is specifically for the backend code.

8.1 Recommendation Script

```
model_path = "/home/plex/Llama/models/Meta-Llama-3.1-8B-Instruct.Q4_K_M.gguf"
llm = Llama(
    model_path=model_path,
    n_ctx=4096,
    n_gpu_layers=32,
    n_threads=6,
    n_batch=512
)
```

Figure 8.1.1 – Model Initialisation

Loads the model once at startup using llamacpp [10]. The model I used was bartowski's llama 3.1 8b instruct model [11]. This is a much smaller version of the meta's llama 3.1 that only has 8 billion parameters enabling me to run it on my server. Specifically, the Q4_K_M version which I tested in LMStudio [19] to see which version best suited my hardware. LMStudio is an easy-to-use application that allows you to run different models locally on your machine. I used this to first test different models and how well they run to decide on which model is best for my use case. Anyways this code configures, the max amount of tokens in one prompt, how many layers to offload to the GPU, the number of CPU threads to allocate, and the number of input tokens to process at a time. I get these numbers as the default values from LMStudio when running the same model:

Figure 8.1.2 – LMStudio Default Values

```python
router = APIRouter()

class MovieRequest(BaseModel):
    title: str

class MovieResponse(BaseModel):
    recommendations: list[str]
```

Figure 8.1.2 – Router and Schemas

Creates an APIRouter which is a class used to group path operations, in this case to structure an app in multiple files [20]. Defines MovieRequest expecting a JSON and then defines MovieResponse which will serialise to JSON like {“recommendations”: [“M1”, “M2”, …]}.

```python
@router.post("/recommend", response_model=MovieResponse)
async def recommend_movie(req: MovieRequest):
    prompt = (
        f"Do NOT output any reasoning or steps. ONLY output the Python list. NOTHING else. List exactly ten movies similar to the movie: {req.title} as a python list. "
        f"They must be unique. Do NOT include the year of the movies or a description. If {req.title} has a sequel include it in the list."
    )
```

Figure 8.1.3 – Endpoint and Prompt

Set the endpoint to /recommend with router.post so /api/recommend calls this function, will explain the /api later. FastAPI parses the incoming JSON into req: MovieRequest. This is the prompt for the model, I have made quite a few revisions of this prompt to get it to give me the

result I want as much as possible. I was never able to get the model working perfectly 100 percent of the time, however.

```
resp = llm(prompt, max_tokens=128, temperature=0.7, top_p=0.9,
           repeat_penalty=1.1, frequency_penalty=0.5)

text = resp["choices"][0]["text"].strip()
print("Model text: " + text)
```

Figure 8.1.4 – Calling the LLM

This code calls the llm object like a function, passing generation parameters. I experimented with these parameters a lot to get the responses I wanted. Prompt is just the text prompt, max_tokens is the maximum number of tokens the model is allowed to generate in its response, temperature controls the randomness or creativity of the model, top_p enables nucleus sampling, basically this trims the “long tail” of unlikely tokens to keep responses reasonable. The repeat penalty is simply a penalty factor that discourages repeating the same exact token over and over. Frequency penalty is slightly different in that it applies a penalty based on how often a token has already appeared in generated text. Just to mention I originally didn’t have those last two penalties but the first few times I ran the model it would sometimes repeat the exact same movie repeatedly which is not ideal. The return value is a dict with a choices list, the first choice’s text is grabbed and stripped of whitespace. I also print it in the terminal for debugging purposes.

```
match = re.search(r"\[.*?\]", text, re.DOTALL)
if not match:

    return MovieResponse(recommendations=["Error"])

list_str = match.group(0)
```

Figure 8.1.5 – Regex Extraction

Uses regex to find the first python list in the raw text [21]. If there is no match it returns “Error”. Then sets list_str as the python list of movies from the response.

```
try:
    movies = ast.literal_eval(list_str)
    if not isinstance(movies, list):

        raise ValueError("Not a list")
except Exception as e:
    print("Parse error:", e)
    movies = ["Error"]

print("Final movies list:", movies)
return MovieResponse(recommendations=movies)
```

Figure 8.1.6 – Parsing

Ast.literal_eval safely evaluates only Python literals. Then it verifies the results is a list, otherwise error. Again, I print out movies in the terminal for debugging purposes allows you to see what or if something is wrong with the prompt. Finally, it returns the final list in movieresponse, where fastapi serialises it to JSON.

8.2 Main.py Script

This script is basically the main script that brings the other four scripts together using FastAPi routers. I just have to run this one script instead of all four of them.

```
app = FastAPI(title="All-in-One Media API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "http://192.168.50.46:3000"],
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Figure 8.2.1 – App initialise and CORS

Creates a FastAPI application. Then mounts CORS middleware which is part of FastAPI [22], this is so the website works both locally and for any other PC in the house that is connected over LAN. It allows all HTTP methods and headers to flow through.

```
from recommend_app import router as rec_router
from image_search_app import router as img_router
from subtitle_app import router as subs_router
from upscaling_app  import router as up_router
```

Figure 8.2.2 – Import Routers

```
app.include_router(rec_router,    prefix="/api")
app.include_router(img_router,    prefix="/api")
app.include_router(subs_router,   prefix="/api")
app.include_router(up_router,     prefix="/api")

app.mount("/content", StaticFiles(directory="content"), name="content")

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

Figure 8.2.3 – Mounting routers + Static file

Each API router that is imported in mounted so that its paths are under /api. The four routers are the four scripts. Any request to /content/<filename> will directly serve the file from the content folder using fastapi's staticfiles [23], this is the folder for all your media. This is how the frontend can link to generated .srt files or upscaled videos. When you run python main.py it starts uvicorn [4] which I explained in the technologies section. The host = 0.0.0.0 listens on all interfaces, port 8000 is the default api port and reload=true enables auto reload which is handy for development.

8.3 Image Search Script

```
INDEX_PATH  = "/home/plex/embeddings/faiss_hnsw.index"
IDMAP_PATH  = "/home/plex/embeddings/id2meta.json"
DEVICE      = "cuda" if torch.cuda.is_available() else "cpu"
```

Figure 8.3.1 – Config Consts

Here I set the path to the FAISS index file and the JSON mapping of index IDs file that has the metadata (movie name, timestamp, file path). This sets the device const to basically choose GPU if available otherwise CPU. GPU is much faster, so I made sure I have the dependencies and cudatoolkit versions installed.

```
index = faiss.read_index(INDEX_PATH)


with open(IDMAP_PATH, "r", encoding="utf-8") as f:
    id2meta = json.load(f)


model, preprocess = clip.load("ViT-B/32", DEVICE)
model.eval()
```

Figure 8.3.2 – Startup Init

First line loads the FAISS [7] (explained in technologies) index into memory so you can do fast nearest-neighbour lookups. Next it reads the ID to metadata map into a Python dict. Load CLIP [6] (explained in technologies) onto the device const which is the GPU, grabbing both the model and its preprocess function. Then switches CLIP into evaluation mode to disable any training-only behaviour.

```
class ImageResult(BaseModel):
    movie: str
    seconds: int
    frame_path: str
    score: float
```

Figure 8.3.3 – Response Schema

Define imageresult, a Pydantic model [5] that ensures the endpoint always returns the four fields with the right types.

```python
@router.post("/search-image", response_model=ImageResult)
async def search_image(file: UploadFile = File(...)):

    data = await file.read()
    try:
        img = Image.open(io.BytesIO(data)).convert("RGB")
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid image file")


    with torch.no_grad():
        token = preprocess(img).unsqueeze(0).to(DEVICE)
        vec = model.encode_image(token).cpu().numpy().astype("float32")
    vec /= np.linalg.norm(vec, axis=1, keepdims=True)
```

Figure 8.3.4 – Endpoint & Image Embedding

POST /search-image accepts a multipart form upload called file. The response is validated against Imageresult. It first reads the raw data from the uploaded file, then wraps it in a BytesIO buffer (from the io module of Python) and opens with Pillow forcing an RGB image. The image module from Pillow (a PIL fork) provides a class that is used to represent a PIL (Python Imaging Library – adds support for different file formats) image [24]. If Pillow can't parse it, return a 400 error code. Next step is to compute the image embedding with CLIP. It starts by preprocessing the image, basically resizes, crops, normalises and converts to a tensor. Unsqueeze adds a batch dimension (so the shape becomes [1,3,H,W]). Then it moves it to the device, again this should be the GPU. The next line runs the CLIP vision encoder to get a feature vector. It then pulls it back to the CPU, converts it to a NumPy float32 array and L2-normalises so that cosine similarity in FAISS corresponds to dot-product distances.

```
D, I = index.search(vec, 1)
idx, dist = int(I[0][0]), float(D[0][0])

meta = id2meta[idx]

if not meta:
    raise HTTPException(status_code=502, detail="No metadata for index")

return ImageResult(
    movie      = meta["movie"],
    seconds    = meta["seconds"],
    frame_path = meta["frame_path"],
    score      = dist
)
```

Figure 8.3.5 – Nearest neighbour and metadata

The first piece of code queries the FAISS index using nearest neighbour and finds the single closest vector in the index. “I” is the index ID and “D” is the similarity score. Next line uses idx to fetch the movie/frame info from the JSON map. If its missing return a 502 error. Then it populates the imageResult with the four variables.

8.4 Subtitle Generation Script

```
print("Loading Whisper base.en model…")
whisper_model = whisper.load_model("base.en")
print("Whisper loaded.")

router = APIRouter()

class SubtitleRequest(BaseModel):
    movie: str

class SubtitleResponse(BaseModel):
    filename: str
    message: str
```

Figure 8.4.1 – Model Init/Router/Pydantic Models

First it loads the base.en Whisper model into memory once so future requests don’t reload. Creates the FastAPI router. Creates two classes which inherit from Pydantics BaseModel [25].

Subtitlerequest expects a JSON {“movie”: “movie.mp4”}. SubtitleResponse returns {“filename”: “Movie.srt”, “message”: “blabla”}.

```python
@router.get("/movies", response_model=list[str])
async def list_movies():
    
    try:
        files = os.listdir(CONTENT_DIR)
    except FileNotFoundError:
        raise HTTPException(500, detail=f"Content directory not found: {CONTENT_DIR}")
    videos = [f for f in files if f.lower().endswith((".mp4", ".mkv", ".avi", ".mov"))]
    return videos
```

Figure 8.4.2 – GET /movies

Reads the content folder (where your media is stored). Filters to common video extensions. Returns a list of filenames for the frontend dropdown. If there is an error it throws 500 error with a message.

```python
@router.post("/generate-subs", response_model=SubtitleResponse)
async def generate_subtitles(req: SubtitleRequest):
    video_path = os.path.join(CONTENT_DIR, req.movie)
    if not os.path.exists(video_path):
        raise HTTPException(404, detail=f"Video not found: {req.movie}")

    result = whisper_model.transcribe(video_path, fp16=False)
    segments = result.get("segments", [])
```

Figure 8.4.3 – POST /generate-subs

Code starts by accepting a SubtitleRequest. It then creates the full video path and gives a 404 error if missing. Next it calls the Whisper transcribe function on the video file. Fp16=false simply ensures compatibility on CPU or non fp16 GPUs. It then extracts the list of time-stamped text segments.

```python
def fmt(ts: float) -> str:
    h = int(ts//3600)
    m = int((ts%3600)//60)
    s = int(ts%60)
    ms = int((ts - int(ts)) * 1000)
    return f"{h:02d}:{m:02d}:{s:02d},{ms:03d}"

srt_name = os.path.splitext(req.movie)[0] + ".srt"
out_path = os.path.join(CONTENT_DIR, srt_name)
with open(out_path, "w", encoding="utf-8") as f:
    for i, seg in enumerate(segments, start=1):
        f.write(f"{i}\n")
        f.write(f"{fmt(seg['start'])} --> {fmt(seg['end'])}\n")
        f.write(seg["text"].strip() + "\n\n")

return SubtitleResponse(
    filename=srt_name,
    message=f"Subtitle saved as {srt_name}"
)
```

Figure 8.4.4 – Timestamp/SRT File Gen

The fmt function simply converts a float into HH:MM:SS,mmm for SRT compliance. It then determines the .srt filename by swapping the extension. Opens it for writing in utf-8. Then it iterates through segments writing: number(1,2, …), time span line, the transcript text and a blank line. Finally, it returns subtitleResponse with the filename and a confirmation message. Here is an example of the srt file produced:

```
00:00:00,000 --> 00:00:02,000
This is a level 7.

2
00:00:02,000 --> 00:00:06,000
As of right now, we are at war.

3
00:00:07,000 --> 00:00:09,000
What do we do?

4
00:00:30,500 --> 00:00:35,039
Put on the Chancellor's kingdom

5
00:00:40,000 --> 00:00:46,359
This is usually his chance for it
```

Figure 8.4.5 – Example .srt file

8.5 Move Upscaling Script

```
def run_command(cmd: str):
    print("", cmd)
    res = subprocess.run(cmd, shell=True)
    if res.returncode != 0:
        raise RuntimeError(f"Command failed: {cmd}")
```

Figure 8.5.1 – run_command function

This is a simple function that logs and executes a shell command and it raises an exception if the command exits non-zero.

```
@router.post("/upscale", response_model=UpscaleResponse)
async def upscale_movie(req: UpscaleRequest):
    inp = os.path.join(CONTENT_DIR, req.movie)
    if not os.path.isfile(inp):
        raise HTTPException(404, detail="Movie not found")

    name, ext = os.path.splitext(req.movie)
    temp_frames    = "frames"
    tmp_up_frames  = "upscaled_frames"
    temp_video     = "up_temp.mp4"
    audio_file     = "audio.aac"
    out_name       = f"{name}_upscaled{ext}"
    out_path       = os.path.join(CONTENT_DIR, out_name)
```

Figure 8.5.2 – POST endpoint and Path setup

POST /upscale receives the request model, builds the input path and gives 404 error if the file does not exist. Next it splits the movie into name and ext. Then defines temporary directories and filenames. Finally constructs the final output filename by appending _upscaled to the end of it.

```
for d in (temp_frames, tmp_up_frames):
    if os.path.isdir(d):
        shutil.rmtree(d)
for f in (temp_video, audio_file):
    if os.path.exists(f):
        os.remove(f)

os.makedirs(temp_frames, exist_ok=True)
os.makedirs(tmp_up_frames, exist_ok=True)
```

Figure 8.5.3 – Clean Directories

This code simply deletes the previous folders with the frames and upscaled frames and then creates them again. This is just to clear out all the frames from previous upscales.

```
try:

    run_command(
        f"ffmpeg -i {shlex.quote(inp)} -qscale:v 2 {temp_frames}/frame_%06d.png"
    )

    run_command(
        f"{PYTHON_EXEC} inference_realesrgan.py "
        f"-n RealESRGAN_x4plus_anime_6B "
        f"--model_path weights/RealESRGAN_x4plus_anime_6B.pth "
        f"-i {temp_frames} -o {tmp_up_frames}"
    )
```

Figure 8.5.4 – Frame extraction + inference

The first command uses FFmpeg to extract every frame in the movie that was selected to be upscaled and dumps them in the frames folder as high-quality PNGs. The next command calls the Real-ESRGAN inference_realesrgan.py script with their x4plus_anime_6B model in this case, where it will read from frames and write upscaled images to the upscaled_frames directory.

```
fps_cmd = (
    f"ffprobe -v 0 -select_streams v:0 "
    f"-show_entries stream=r_frame_rate -of csv=p=0 {shlex.quote(inp)}"
)
fps_proc = subprocess.run(fps_cmd, shell=True, capture_output=True, text=True)
num,den = fps_proc.stdout.strip().split('/')
fps = float(num)/float(den)

run_command(
    f"ffmpeg -framerate {fps} -i {tmp_up_frames}/frame_%06d_out.png "
    f"-c:v libx264 -pix_fmt yuv420p {temp_video}"
)
```

Figure 8.5.5 – FPS + Video assembly

The first command gets the original FPS by using FFprobe (part of FFmpeg) it then parses the num/den fraction into a float to get the fps. The next command uses FFmpeg to stich the upscaled frames back into a video at the original frame rate.

```
    run_command(f"ffmpeg -i {shlex.quote(inp)} -vn -acodec copy {audio_file}")
    run_command(f"ffmpeg -i {temp_video} -i {audio_file} -c:v copy -c:a copy {shlex.quote(out_path)}")
except Exception as e:
    raise HTTPException(500, detail=str(e))
finally:
    shutil.rmtree(temp_frames,     ignore_errors=True)
    shutil.rmtree(tmp_up_frames,    ignore_errors=True)
    for f in (temp_video, audio_file):
        if os.path.exists(f):
            os.remove(f)

return UpscaleResponse(
    filename=out_name,
    message=f"Upscaled video saved as {out_name}"
)
```

Figure 8.5.6 – Audio and Cleanup

The first two command handle the audio, it will first extract the original audio track into audio.acc then merges that audio with the upscaled video to produce the final result. Next, we have some error handling where any failure in the pipeline will raise a 500 error and message. It then removes any temporary directories and files, whether it succeeded or not. Finally, it returns the Pydantic response model containing the output filename and a message.

9 Ethics

I have put some thought into ethical considerations in my project and while I don't believe there is anything majorly unethical there are a few points I would like to make. The obvious point is that the website is designed to work with a person's own media library, with said media library potentially including copyrighted movies. If the library that this project is used on contains copyrighted material, then the image to movie search index would be built from copyrighted movie frames. Creating subtitles of a film are derivatives of the film itself which is also protected by copyright. The movie upscaling is the same in relation to derivatives and copyright.

One other area of consideration is that the project requires a computer with decent hardware that can use quite a bit of energy. This would increase the users carbon footprint which is not ideal nowadays with global warming being a major concern. Finally, this project uses a language model which could be biased towards or hallucinate movies which is another ethical consideration.

10 Conclusion

In conclusion, I believe I achieved many of the goals set at the start of the project, from creating a useful website that is helpful for anybody who has their own media server to learning all about how AI and machine learning works. I improved my software engineering skills, like python scripting and web page development.

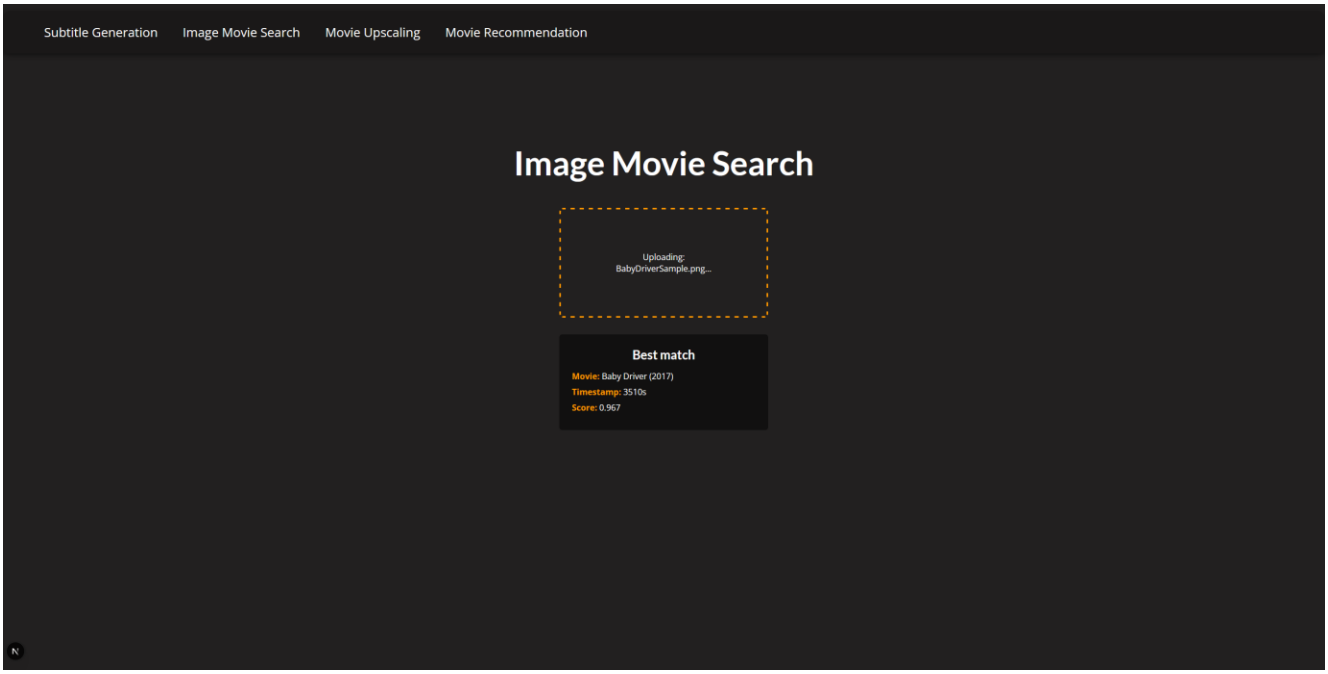

Figure 9.1 – Image Movie Search Screen

If I were to continue development of this project, there are definitely some changes I would make. I experimented around with a few different models to run locally, and I settled on one that did the job, but I would definitely like to experiment with more models and make some adjustments so that the model works better. I would have also liked to add a login feature, where you could login and it would save some information like previously searched images or previous recommendations. There are a whole bunch of areas that could be improved or developed further. I hope and expect everything I have learned on this project to be useful in my life and career.

11 References

[1] “React,” [Online]. Available: https://react.dev/. [Accessed 28 04 2025].

[2] “Nextjs,” [Online]. Available: https://nextjs.org/. [Accessed 28 04 2025].

[3] “FastAPI,” [Online]. Available: https://fastapi.tiangolo.com/. [Accessed 28 04 2025].

[4] “Uvicorn,” [Online]. Available: https://www.uvicorn.org/. [Accessed 28 04 2025].

[5] “Pydantic,” [Online]. Available: https://docs.pydantic.dev/latest/. [Accessed 28 04 2025].

[6] OpenAI, “CLIP,” [Online]. Available: https://github.com/openai/CLIP. [Accessed 28 04 2025].

[7] FacebookResearch, “FAISS,” [Online]. Available: https://github.com/facebookresearch/faiss. [Accessed 28 04 2025].

[8] OpenAI, “Whisper,” [Online]. Available: https://github.com/openai/whisper. [Accessed 28 04 2025].

[9] Xinntao, “Real-ESRGAN,” [Online]. Available: https://github.com/xinntao/Real-ESRGAN. [Accessed 28 04 2025].

[10] ggml-org, “Llama.cpp,” [Online]. Available: https://github.com/ggml-org/llama.cpp. [Accessed 28 04 2025].

[11] Bartowski, “Llama 3.1 8B,” [Online]. Available: https://huggingface.co/bartowski/Meta-Llama-3.1-8B-Instruct-GGUF. [Accessed 28 04 2025].

[12] “PyTorch,” [Online]. Available: https://github.com/pytorch/pytorch. [Accessed 28 04 2025].

[13] “FFmpeg,” [Online]. Available: https://github.com/FFmpeg/FFmpeg. [Accessed 28 04 2025].

[14] “nexjs-link,” [Online]. Available: https://nextjs.org/docs/pages/api-reference/components/link. [Accessed 28 04 2025].

[15] w3schools, “ReactUseState,” [Online]. Available: https://www.w3schools.com/react/react_usestate.asp. [Accessed 28 04 2025].

[16] w3schools, “useCallbackHook,” [Online]. Available: https://www.w3schools.com/react/react_usecallback.asp. [Accessed 28 04 2025].

[17] w3schools, “preventDefault,” [Online]. Available: https://www.w3schools.com/jsref/event_preventdefault.asp. [Accessed 28 04 2025].

[18] w3schools, “useEffectHook,” [Online]. Available: https://www.w3schools.com/react/react_useeffect.asp. [Accessed 28 04 2025].

[19] “lmstudio,” [Online]. Available: https://lmstudio.ai/. [Accessed 28 04 2025].

[20] fastapi, “apirouter,” [Online]. Available: https://fastapi.tiangolo.com/reference/apirouter/. [Accessed 28 04 2025].

[21] “Regex,” [Online]. Available: https://www.w3schools.com/python/python_regex.asp. [Accessed 28 04 2025].

[22] FastAPI, “CORS,” [Online]. Available: https://fastapi.tiangolo.com/tutorial/cors/. [Accessed 28 04 2025].

[23] fastapi, “static-files,” [Online]. Available: https://fastapi.tiangolo.com/tutorial/static-files/. [Accessed 28 04 2025].

[24] pillow, “image module,” [Online]. Available: https://pillow.readthedocs.io/en/stable/reference/Image.html. [Accessed 28 04 2025].

[25] Pydantic, “base_model,” [Online]. Available:
https://docs.pydantic.dev/latest/api/base_model/. [Accessed 28 04 2025].