AI-Assisted Development of Client-Side Face Detection

Report on AI Workflow and Security Implementation

February 5, 2026

Contents

1 Introduction

The objective of this project was to design and implement a browser-based face detection application with a focus on privacy and client-side execution. Unlike traditional computer vision systems that rely on backend processing, this application leverages modern web technologies—specifically JavaScript and WebAssembly (WASM)—to perform Machine Learning (ML) inference directly within the user's browser.

The development methodology for this project utilized a Generative AI assistant to accelerate the prototyping phase. The primary goal was to achieve a functional "One-Shot" solution—a working prototype generated from a single, highly optimized prompt, minimizing the need for iterative debugging. This report details the planning, prompt engineering strategy, security considerations, and deployment pipeline for the application.

2 Planning

The planning phase focused less on writing code manually and more on defining the architectural constraints for the AI assistant. The core requirements identified were:

- **Client-Side Execution:** No video data could be sent to a server.
- **Real-Time Performance:** The app must handle live video streams with minimal latency.
- **Visual Feedback:** Bounding boxes must be overlaid on detected faces.

2.1 Prompt Optimization

To maximize the probability of a successful "One-Shot" generation, a "Meta-Prompting" strategy was employed. Instead of immediately asking for the code, the AI was first tasked with refining the request itself. The initial raw request was: *"I have to do this project, could you generate an optimized prompt for an LLM like you? I want to build it in one shot."* This step was crucial because LLMs perform better when the prompt explicitly outlines constraints, libraries, and expected outputs. The AI generated a detailed technical prompt that specified the use of browser-based ML libraries and defined the testing criteria, ensuring the subsequent code generation would be contextually accurate. This iterative meta-prompting strategy is supported by recent research on **Optimization by PROmpting (OPRO)**, which demonstrates that LLMs can effectively optimize their own instructions to improve performance more than the human-written prompts alone [1].

Link to Original Chat: `https://gemini.google.com/share/2dc8f76950b5`

2.2 One-Shot Solution

Using the optimized prompt, the AI generated a complete, single-file solution stack utilizing **Google's MediaPipe Tasks Vision API**. The generated application adhered strictly to the "client-side only" requirement by loading the **BlazeFace (Short Range)** model via WebAssembly.

The solution architecture consisted of four distinct modules:

1. **Initialization Phase:** The application asynchronously loads the `FilesetResolver` and initializes the `FaceDetector` class. Notably, the AI configured the detector to use the `GPU` delegate, ensuring hardware acceleration for smooth performance.

2. **Video Pipeline:** Access to the webcam is managed via `navigator.mediaDevices.getUserMedia`, with a fallback to a specific resolution (1280x720) ideal for the BlazeFace model.

3. **Inference Loop:** A `requestAnimationFrame` loop creates a continuous stream. For every frame, the application calls `faceDetector.detectForVideo()`, calculates the inference latency, and updates a "Heads-Up Display" (HUD) overlay.

4. **Visualization:** The results are rendered onto an HTML5 Canvas. The AI implemented detailed drawing logic, including bounding boxes and six facial keypoints (eyes, nose, mouth, ears), applying a mirror transformation to align naturally with the user's movement.

3 Security Improvements

To address privacy concerns regarding biometric data, a specific consultation was conducted to ensure no video data leaves the client. The security strategy relies on a "Defense in Depth" approach. The prompt used was: *How can I ensure that any image is sent to a server? I would like to guarantee that the client doesn't connect to any new remote source.*

My idea was to stress the AI to find a way to guarantee that no data exfiltration occurs. The primary mechanism for preventing data exfiltration is a strict Content Security Policy (CSP). This functions as a whitelist, instructing the browser to block any network request to unauthorized domains.

3.1 Verification Procedures

To validate the "Local Only" claim, two key tests were performed:

- **The "Air Gap" Test:** The application was loaded, and the device was disconnected from the internet. The face detection continued to function at 60fps, proving that inference occurs entirely on the device's hardware.

- **Network Monitoring:** The browser's Developer Tools confirmed that network activity ceases completely after the initial download of the `.wasm` and `.tflite` assets.

3.2 Refinements and Debugging

During the integration of the CSP, an initial implementation error was encountered. The first iteration of the policy included inline comments within the HTML `content` attribute, causing parsing errors. Furthermore, it was determined that the WebAssembly runtime requires specific permissions to compile binaries.

The prompt used to fix the CSP was: *copying and pasting the meta tag I'm receiving errors in console, I think that the comments inside are written in a wrong way.*

The policy was refined by removing comments and adding the `'unsafe-eval'` directive, which is required for JIT compilation in browser-based ML libraries.

4 Deployment Strategy

To ensure the application runs consistently across different environments I used a prompt asking for containerization options: *I want to be sure that this web-app works in different environments without distinction. How can I deploy it? Should I containerize it?.*

My Idea was to have a lightweight container that serves the static HTML file, because I don't want to get "on my machine it works" issues. The AI suggested me to use three main options:

- **Static Cloud:** Services like GitHub Pages or Netlify
- **Docker:** Using a lightweight web server like Nginx or Caddy
- **Simple Python/Node Server:** Using built-in HTTP servers for quick testing

I ended up choosing the Docker option, because it guarantees the same environment everywhere, but with a constraint: the HTTPS challenge, to ensure secure camera access.

4.1 The HTTPS Challenge in Isolated Environments

My prompt was: *Ok, I want to use Docker and to configure also the the HTTPS.*
A critical constraint identified during the planning phase is the browser's strict security model regarding hardware access. The API used for the camera, `navigator.mediaDevices.getUserMedia`, is restricted to **Secure Contexts**.

- **The Constraint:** Browsers strictly **block** camera access on insecure HTTP connections (e.g., `http://192.168.1.50`).
- **The Problem:** Standard Docker containers serve traffic over plain HTTP (port 80) by default.
- **The Solution:** The container must handle its own encryption.

4.2 Implementation: Standalone Secure Container

To create a truly portable solution that allows camera access immediately upon deployment (without requiring an external Reverse Proxy or a purchased domain), an automated "Self-Signed" strategy was implemented.

This approach involves two key components generated during the build process:

4.2.1 1. Automated Certificate Generation (Dockerfile)

Instead of manually mounting certificates, the `Dockerfile` was modified to install `OpenSSL` and programmatically generate a valid X.509 certificate during the image build. This ensures that every instance of the container has unique, valid cryptographic keys.

```
# Base image: Lightweight Nginx
FROM nginx:alpine

# 1. Install OpenSSL
RUN apk add --no-cache openssl

# 2. Create SSL directory
RUN mkdir -p /etc/nginx/ssl

# 3. Generate a Self-Signed Certificate (Valid 365 days)
RUN openssl req -x509 -nodes -days 365 \
    -newkey rsa:2048 \
    -keyout /etc/nginx/ssl/selfsigned.key \
    -out /etc/nginx/ssl/selfsigned.crt \
    -subj "/C=US/ST=State/L=City/O=Organization/CN=localhost"

# 4. Copy Configurations
```

```
COPY nginx.conf /etc/nginx/nginx.conf
COPY face_app.html /usr/share/nginx/html/index.html

# Expose HTTP and HTTPS ports
EXPOSE 80
EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]
```

Listing 1: Dockerfile with Automatic SSL Generation

4.2.2 2. Secure Server Configuration (Nginx)

A custom `nginx.conf` was injected to enforce security. It performs two critical functions:

1. **Force HTTPS:** Any request on Port 80 is strictly redirected to Port 443 (HTTPS).
2. **SSL Termination:** It uses the generated certificates to encrypt the traffic before it leaves the container.

```
server {
    listen 80;
    server_name localhost;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name localhost;

    ssl_certificate /etc/nginx/ssl/selfsigned.crt;
    ssl_certificate_key /etc/nginx/ssl/selfsigned.key;

    # Modern TLS Protocols
    ssl_protocols TLSv1.2 TLSv1.3;
}
```

Listing 2: Nginx Configuration for SSL Termination

4.3 Operational Verification

Upon running the container ('docker run -p 443:443 ...'), the application is accessible via `https://localhost`.

Note on User Experience: Because the certificate is self-signed rather than issued by a public Certificate Authority (CA), browsers will display a "Not Secure" or "Your connection is not private" warning. However, for the purpose of this project, this is acceptable behavior. The connection **is** encrypted, and the **Secure Context** requirement is technically satisfied, allowing the `getUserMedia` API to unlock the camera and the face detection to function on remote mobile devices connected to the same network.
Link to Original Chat: `https://gemini.google.com/share/2723ab111b63`

5 Development Platform and Tools

The project was developed using VS Code integrated with the GitHub Copilot agent. This environment was chosen due to my existing workflow familiarity and the access provided by the

GitHub Student Developer Pack. After evaluating various premium models available within Copilot, I selected Gemini 3.0 Pro, as it consistently demonstrated the highest level of capability for this specific use case. I would like to highlight that the prompt optimization strategy were performed using Gemini 3.0 with Reasoning, while the code generation was done using Gemini 3.0 Pro, because more advanced and suggested for coding tasks.

6 Cross-browser Testing

To ensure reliability across diverse environments, a compatibility test was conducted on four major browser engines. The application was deployed via the Docker container and accessed over the local network.

6.1 Test Results

Browser	**Engine**	**Status**
Chrome	Blink	**Pass**
Brave	Chromium	**Pass**
Firefox	Gecko	**Pass**
Safari	WebKit	**Pass**

Table 1: Cross-Browser Compatibility Matrix

7 Validation Results

7.1 Efficiency

With an optimized prompt the AI generated in a one shot prototype that correctly implemented complex client-side Machine Learning libraries (MediaPipe). The initial development time was reduced significantly. Using this strategy I was able to get a working prototype in few minutes, while without it I would have probably spent hours just trying to figure out how to load the model and how to structure the code.

7.2 CSP Security Refinements

The initial CSP generated by the AI was functional but contained inline comments that caused parsing errors. After refining the prompt to remove comments and include necessary directives for WebAssembly, the final CSP was successfully implemented, ensuring that the application only connects to authorized sources for scripts, styles, fonts, and model assets.

7.3 HTTPS Constraints and Media Access

A critical issue encountered during validation was the browser's strict enforcement of security policies regarding hardware access. Modern browsers (Chrome, Safari, Edge) block access to media devices ('navigator.mediaDevices.getUserMedia') on non-secure contexts (HTTP), unless the origin is strictly 'localhost'.

To resolve the HTTPS constraint while maintaining a portable, "runnable out-of-the-box" experience for evaluation, the Docker container was configured to handle its own encryption.

Implementation Detail: Unlike a production environment where certificates are managed by a Certificate Authority (CA), this project utilizes **ephemeral self-signed certificates**. The ‘Dockerfile‘ includes an ‘OpenSSL‘ command that generates a fresh 2048-bit RSA key and certificate pair dynamically during the image build process.

Security Acknowledgment: I acknowledge that while this approach secures the connection technically (enabling the ‘Secure Context‘), it triggers browser warnings because the certificate is not signed by a trusted root CA. In a production scenario, this would be replaced with:

1. **Trusted Certificates:** Using an automated solution like Let’s Encrypt or a commercially signed certificate.
2. **Secret Management:** Ensuring private keys are injected via secure environment variables or secret managers, rather than being generated or stored within the container file system.

The current ”generation-on-build” strategy was chosen specifically to ensure the examiner can run the project immediately without needing to manually generate or mount SSL credentials.

8 Conclusion

This project successfully demonstrated the efficacy of an AI-assisted workflow in modern web development. By utilizing a ”Meta-Prompting” strategy, the initial development time was reduced significantly, yielding a functional ”One-Shot” prototype that correctly implemented complex client-side Machine Learning libraries (MediaPipe).
The interesting thing is that the AI didn’t generated a complete a very secure CSP at first and usable on different environments, but with proper prompting it was able to refine it until it worked perfectly. Another interesting aspect is that the AI didn’t lose the context also after multiple prompts, as happens sometimes with other LLMs or less advanced models.
The final result is a zero-latency, privacy-first application that is easy to deploy and maintain, validating the use of Large Language Models as powerful architectural partners in software engineering.

References

[1] Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., & Zhou, D. (2023). *Large Language Models as Optimizers.* arXiv preprint arXiv:2309.03409. `https://arxiv.org/abs/2309.03409`