**🎯 Complete Code Explanation - Face Mask Detection System**

**📚 Table of Contents**

1. [Libraries & Imports](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#libraries--imports)
2. [UI Configuration & Styling](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#ui-configuration--styling)
3. [Core Detection Class](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#core-detection-class)
4. [Main Application Structure](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#main-application-structure)
5. [Image Detection Module](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#image-detection-module)
6. [Video Processing Module](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#video-processing-module)
7. [Live Camera Module](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#live-camera-module)
8. [Analytics Dashboard](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#analytics-dashboard)
9. [Performance Optimizations](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#performance-optimizations)
10. [Architecture Overview](https://claude.ai/chat/c5aa7b93-302a-4420-8251-21c59ae1c15a#architecture-overview)

**🔧 Libraries & Imports**

import streamlit as st # Web framework for creating the UI

import cv2 # OpenCV for computer vision operations

import numpy as np # Numerical operations on arrays/images

from ultralytics import YOLO # YOLOv11 model loading and inference

import tempfile # Temporary file handling for uploads

import os # Operating system interface

from datetime import datetime, timedelta # Time and date operations

import pandas as pd # Data manipulation (for future use)

import plotly.express as px # Interactive plotting

import plotly.graph\_objects as go # Advanced plotting

from PIL import Image # Python Imaging Library

import time # Time operations for FPS calculation

import threading # Multi-threading (for future use)

import queue # Queue operations (for future use)

**Why these libraries?**

* **Streamlit**: Creates professional web interface without HTML/CSS knowledge
* **OpenCV**: Industry standard for computer vision operations
* **Ultralytics**: Official YOLOv11 implementation - latest and fastest
* **Plotly**: Creates interactive, professional charts
* **PIL**: Better image handling than OpenCV for certain operations

**🎨 UI Configuration & Styling**

**1. Page Configuration**

st.set\_page\_config(

page\_title="AI Face Mask Detection System", # Browser tab title

page\_icon="😷", # Browser tab icon

layout="wide", # Use full screen width

initial\_sidebar\_state="expanded" # Sidebar open by default

)

**2. Custom CSS Styling**

st.markdown("""

<style>

.main-header {

font-size: 3rem;

font-weight: bold;

text-align: center;

background: linear-gradient(90deg, #FF6B6B, #4ECDC4); # Gradient text

-webkit-background-clip: text;

-webkit-text-fill-color: transparent;

margin-bottom: 2rem;

}

.stat-box {

background: linear-gradient(135deg, #667eea 0%, #764ba2 100%); # Blue gradient

padding: 1rem;

border-radius: 10px;

color: white;

text-align: center;

margin: 0.5rem 0;

}

.alert-box {

background: linear-gradient(135deg, #ff6b6b 0%, #ee5a24 100%); # Red gradient

padding: 1rem;

border-radius: 10px;

color: white;

text-align: center;

animation: pulse 2s infinite; # Pulsing animation for alerts

}

@keyframes pulse {

0% { opacity: 1; }

50% { opacity: 0.7; }

100% { opacity: 1; }

}

</style>

""", unsafe\_allow\_html=True)

**Purpose**: Creates a modern, professional look that stands out from basic Streamlit apps.

**🧠 Core Detection Class**

**Class Initialization**

class FaceMaskDetector:

def \_\_init\_\_(self, model\_path):

try:

self.model = YOLO(model\_path) # Load your trained YOLOv11 model

self.class\_names = self.model.names # Get class names (with\_mask, without\_mask, etc.)

self.detection\_history = [] # Store all detections

self.session\_stats = { # Track session statistics

'total\_detections': 0,

'with\_mask': 0,

'without\_mask': 0,

'incorrect\_mask': 0,

'start\_time': datetime.now()

}

except Exception as e:

st.error(f"Error loading model: {str(e)}")

**What happens here?**

* Loads your trained YOLOv11 model
* Extracts class names automatically from the model
* Initializes tracking variables for analytics
* Handles errors gracefully if model loading fails

**Main Detection Method**

def detect\_image(self, image, confidence\_threshold=0.5):

try:

results = self.model(image, conf=confidence\_threshold) # Run YOLO inference

return self.process\_results(results[0], image) # Process and visualize results

except Exception as e:

st.error(f"Detection error: {str(e)}")

return image, []

**Process:**

1. Runs YOLOv11 inference on the input image
2. Filters detections by confidence threshold
3. Processes results and draws bounding boxes
4. Returns annotated image and detection data

**Results Processing**

def process\_results(self, result, image):

detections = []

annotated\_image = image.copy()

if result.boxes is not None:

for box in result.boxes:

# Extract detection information

x1, y1, x2, y2 = map(int, box.xyxy[0]) # Bounding box coordinates

confidence = float(box.conf[0]) # Confidence score

class\_id = int(box.cls[0]) # Class ID

class\_name = self.class\_names[class\_id] # Class name

# Color coding for different classes

colors = {

'with\_mask': (0, 255, 0), # Green for compliant

'without\_mask': (0, 0, 255), # Red for non-compliant

'mask\_weared\_incorrect': (0, 165, 255) # Orange for incorrect

}

color = colors.get(class\_name, (255, 255, 255))

# Draw bounding box

cv2.rectangle(annotated\_image, (x1, y1), (x2, y2), color, 2)

# Draw label with confidence

label = f"{class\_name}: {confidence:.2f}"

cv2.putText(annotated\_image, label, (x1, y1 - 5),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2)

**What this does:**

* Extracts bounding box coordinates from YOLO results
* Assigns colors based on detection class (green=good, red=bad)
* Draws rectangles around detected faces
* Adds text labels with confidence scores
* Updates session statistics

**🏗️ Main Application Structure**

**Session State Management**

if 'detector' not in st.session\_state:

st.session\_state.detector = None

if 'live\_detection' not in st.session\_state:

st.session\_state.live\_detection = False

**Purpose**: Streamlit runs the entire script on every interaction. Session state keeps variables persistent across runs.

**Sidebar Configuration**

with st.sidebar:

st.markdown("## 🔧 Configuration")

# Model upload

model\_file = st.file\_uploader(

"Upload YOLOv11 Model (.pt)",

type=['pt'],

help="Upload your trained YOLOv11 face mask detection model"

)

if model\_file:

# Save uploaded model temporarily

with tempfile.NamedTemporaryFile(delete=False, suffix='.pt') as tmp\_file:

tmp\_file.write(model\_file.read())

model\_path = tmp\_file.name

# Initialize detector

if st.session\_state.detector is None:

with st.spinner("Loading model..."):

st.session\_state.detector = FaceMaskDetector(model\_path)

st.success("✅ Model loaded successfully!")

**Key Features:**

* **File Upload**: Users upload their trained .pt model file
* **Temporary Storage**: Model is saved temporarily for processing
* **Loading Feedback**: Shows spinner while loading, success message when done
* **One-time Loading**: Model loads only once using session state

**Main Tabs Structure**

tab1, tab2, tab3, tab4 = st.tabs(["📸 Image Detection", "🎥 Video Detection", "📹 Live Camera", "📊 Analytics"])

**Organization**: Clean separation of different functionalities into tabs.

**📸 Image Detection Module**

with tab1:

st.markdown("### Upload Image for Detection")

uploaded\_image = st.file\_uploader(

"Choose an image...",

type=['jpg', 'jpeg', 'png'],

key="image\_uploader"

)

if uploaded\_image:

# Display original image

image = Image.open(uploaded\_image)

image\_np = np.array(image)

col1, col2 = st.columns(2)

with col1:

st.markdown("#### Original Image")

st.image(image, use\_column\_width=True)

with col2:

st.markdown("#### Detection Results")

with st.spinner("Detecting..."):

result\_image, detections = detector.detect\_image(image\_np, confidence\_threshold)

st.image(result\_image, use\_column\_width=True)

**Process Flow:**

1. User uploads an image file
2. Image is converted to numpy array (OpenCV format)
3. Two columns show original vs processed image
4. Detection runs with loading spinner
5. Results displayed with detection summary

**🎥 Video Processing Module**

def process\_video(detector, video\_path, confidence\_threshold):

cap = cv2.VideoCapture(video\_path) # Open video file

# Get video properties

fps = int(cap.get(cv2.CAP\_PROP\_FPS))

total\_frames = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))

duration = total\_frames / fps

# Progress tracking

progress\_bar = st.progress(0)

frame\_placeholder = st.empty()

frame\_count = 0

video\_detections = []

while cap.isOpened():

ret, frame = cap.read()

if not ret:

break

# Process every 5th frame for performance

if frame\_count % 5 == 0:

result\_frame, detections = detector.detect\_image(frame, confidence\_threshold)

# Display current frame

frame\_placeholder.image(result\_frame, channels="BGR", use\_column\_width=True)

# Update progress

progress = frame\_count / total\_frames

progress\_bar.progress(progress)

frame\_count += 1

cap.release()

**Key Optimizations:**

* **Frame Skipping**: Processes every 5th frame to maintain performance
* **Progress Tracking**: Shows real-time progress bar
* **Memory Management**: Releases video capture when done
* **Live Updates**: Shows processed frames in real-time

**📹 Live Camera Module**

def run\_live\_detection(detector, camera\_placeholder, stats\_placeholder, confidence\_threshold):

cap = cv2.VideoCapture(0) # Open default camera (index 0)

if not cap.isOpened():

st.error("❌ Could not open camera. Please check your camera connection.")

return

# Performance optimization

cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 640) # Set resolution

cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 480)

cap.set(cv2.CAP\_PROP\_FPS, 30) # Set frame rate

frame\_count = 0

fps\_counter = 0

start\_time = time.time()

while st.session\_state.live\_detection: # Continue while flag is True

ret, frame = cap.read()

if not ret:

break

# Process every 3rd frame for better performance

if frame\_count % 3 == 0:

result\_frame, detections = detector.detect\_image(frame, confidence\_threshold)

# Calculate FPS

fps\_counter += 1

elapsed\_time = time.time() - start\_time

if elapsed\_time >= 1.0:

fps = fps\_counter / elapsed\_time

fps\_counter = 0

start\_time = time.time()

# Add FPS counter to frame

cv2.putText(result\_frame, f"FPS: {fps:.1f}", (10, 30),

cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 255, 0), 2)

# Display frame

camera\_placeholder.image(result\_frame, channels="BGR", use\_column\_width=True)

**Performance Features:**

* **Optimized Resolution**: 640x480 for balance of quality and speed
* **Frame Skipping**: Every 3rd frame for smooth performance
* **FPS Calculation**: Real-time frame rate monitoring
* **FPS Display**: Shows performance directly on video
* **Controlled Loop**: Start/stop functionality via session state

**📊 Analytics Dashboard**

**Statistics Tracking**

def update\_stats(self, class\_name):

self.session\_stats['total\_detections'] += 1

if 'with\_mask' in class\_name.lower():

self.session\_stats['with\_mask'] += 1

elif 'without\_mask' in class\_name.lower():

self.session\_stats['without\_mask'] += 1

elif 'incorrect' in class\_name.lower():

self.session\_stats['incorrect\_mask'] += 1

def get\_compliance\_rate(self):

total = self.session\_stats['total\_detections']

if total == 0:

return 0

compliant = self.session\_stats['with\_mask']

return (compliant / total) \* 100

**Alert System**

if enable\_alerts and compliance\_rate < compliance\_threshold:

st.markdown(f"""

<div class="alert-box">

🚨 <strong>COMPLIANCE ALERT!</strong><br>

Current compliance rate ({compliance\_rate:.1f}%) is below threshold ({compliance\_threshold}%)

</div>

""", unsafe\_allow\_html=True)

**Interactive Charts**

# Pie chart for mask distribution

fig\_pie = go.Figure(data=[go.Pie(

labels=['With Mask', 'Without Mask', 'Incorrect Mask'],

values=[stats['with\_mask'], stats['without\_mask'], stats['incorrect\_mask']],

marker\_colors=['#00b894', '#e17055', '#fdcb6e'] # Green, Red, Yellow

)])

# Bar chart for compliance

fig\_bar = go.Figure([

go.Bar(x=['Compliant', 'Non-Compliant'],

y=[stats['with\_mask'], stats['without\_mask'] + stats['incorrect\_mask']],

marker\_color=['#00b894', '#e17055'])

])

**⚡ Performance Optimizations**

**1. Frame Rate Optimization**

# Process every 3rd frame for live detection

if frame\_count % 3 == 0:

result\_frame, detections = detector.detect\_image(frame, confidence\_threshold)

# Process every 5th frame for video

if frame\_count % 5 == 0:

result\_frame, detections = detector.detect\_image(frame, confidence\_threshold)

**Why?** Processing every frame is unnecessary and slows down the system. Skipping frames maintains smooth performance.

**2. Memory Management**

cap.release() # Always release camera/video resources

annotated\_image = image.copy() # Create copy to avoid modifying original

**3. Efficient Resolution**

cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 640) # Lower resolution = faster processing

cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 480)

**4. Confidence Threshold**

results = self.model(image, conf=confidence\_threshold) # Filter low-confidence detections

**🏛️ Architecture Overview**

**System Architecture:**

User Interface (Streamlit)

↓

Input Handler (Image/Video/Camera)

↓

FaceMaskDetector Class

↓

YOLOv11 Model (Your trained model)

↓

Results Processor (Bounding boxes, labels)

↓

Analytics Engine (Statistics, compliance)

↓

Visualization (Charts, alerts, displays)

**Data Flow:**

1. **Input**: User uploads model and selects input type
2. **Processing**: YOLOv11 processes frames/images
3. **Analysis**: Results are analyzed and statistics updated
4. **Visualization**: Results displayed with analytics
5. **Feedback**: Real-time updates and alerts

**Key Design Patterns:**

* **MVC Pattern**: Model (YOLO), View (Streamlit UI), Controller (Detection class)
* **Session Management**: Persistent state across Streamlit reruns
* **Error Handling**: Graceful failure with user feedback
* **Modular Design**: Separate functions for each feature

**🎯 Why This Code is Interview-Ready**

**Technical Skills Demonstrated:**

1. **Computer Vision**: YOLOv11, OpenCV, image processing
2. **Machine Learning**: Model inference, confidence thresholds
3. **Web Development**: Streamlit, responsive design, CSS
4. **Performance Optimization**: Frame skipping, memory management
5. **Data Visualization**: Interactive charts, real-time updates
6. **Software Engineering**: Clean code, error handling, modularity

**Advanced Features:**

1. **Real-time Processing**: Live camera with FPS optimization
2. **Multi-format Support**: Images, videos, live streams
3. **Analytics Dashboard**: Statistics, compliance monitoring
4. **Professional UI**: Modern design, responsive layout
5. **Alert Systems**: Compliance monitoring with visual alerts
6. **Performance Monitoring**: FPS counters, progress tracking

**Industry Best Practices:**

* **Error Handling**: Graceful failures with user feedback
* **Code Organization**: Clean class structure, modular functions
* **User Experience**: Loading spinners, progress bars, clear feedback
* **Performance**: Optimized for real-time operation
* **Scalability**: Easy to extend with new features

This code demonstrates **production-level** thinking and implementation that employers look for! 🚀