**Natural Language Processing Final Report**

**Name: Avantika**

**Reg No:21MID0011**

**1.J Component GitHub Repo**



**3. NLP Hands - On**

**2. J Component Report**

**4: Datasets** are not involved for this project

**5: Medium.com article**

YouTube Context analysis for regional language using API

*Submitted in partial fulfilment of the award of the degree of*

**M.Tech [Integrated]**

**Computer Science and Engineering**

**By:**

Rishpraveen.S (21MID0151) Avantika Girigoswami (21MID0011)

**Under the Guidance of Dr. Sharmila Bhanu.K**

****

# Table of Contents

1. Abstract
2. Introduction
3. Problem Statement
4. Modules
5. Packages/API Used
6. Methodology with Architecture
   1. Architecture Overview
   2. Data Flow Description
7. Results and Discussion
8. Constraints/Limitations
9. Challenges Faced and How They Were Fixed
10. Code Implementation
    1. manifest.json
    2. background.js (Service Worker)
    3. popup.html & popup.js (User Interface)
    4. content.js (Content Script)
    5. options.html & options.js (Configuration)
    6. rules.json (Network Request Rules)
11. References

# Abstract

With the exponential growth of online video watching, YouTube stands as a behemoth for information and entertainment. Nevertheless, within this content expanse come challenges in evaluating video content, analysing audience reception, and validating claims effectively. This project outlines building a Chrome Extension, the "YouTube Context Analyzer," that would make users have a greater contextual understanding of YouTube videos. Utilising the YouTube Data API and Google's Generative AI (Gemini), the extension retrieves video transcripts, analyses comment sections for sentiment, and provides on-demand fact-checking of highlighted text through a right-click context menu. By providing these capabilities in-browser, the extension hopes to better educate users, encourage critical viewing, and enable users to make more informed choices regarding the content they engage with.

# Introduction

Web video sites, and especially YouTube, are at the heart of today's information creation and consumption. As valuable as they are in terms of educational and entertainment content, however, users struggle to get a sense of the fundamental message of a video, the general tone of the community debate, or the truthfulness of claims made. Basic metadata is provided by default by YouTube features, but more involved contextual analysis takes considerable manual labour. This project fills this void by developing a browser extension that embeds sophisticated analysis tools into the YouTube watching experience. The "YouTube Context Analyzer" offers features to: Try to fetch video transcripts for rapid content scanning, Analyze sentiment distribution (positive, negative, neutral) in the comment section through AI, and Enable users to highlight text on the page or transcript and conduct an AI-driven fact-check. Using Google's YouTube Data API for data lookup and the Gemini API for advanced language understanding operations (sentiment analysis, fact-checking), the extension adds useful context not otherwise easily accessible, with the goal of a more informed and aware user experience.

# Problem Statement

YouTube viewers often have problems in effectively judging the content and context of videos:

* **Time-Wasting Content Evaluation:** Determining the most important information or points of a video takes viewing the entire video or fast-forwarding, which is inefficient. Having access to a transcript can make this greatly faster.
* **Knowing What's Being Said:** Reading manually through scores or thousands of comments to understand audience response is unfeasible. Users do not have a quick snapshot of the overall sentiment about a video.
* **Verifying Information:** Videos and comments can contain factual claims, misinformation, or opinions presented as facts. It takes verifying this information usually by navigating away from YouTube to search other sources, breaking the viewing experience.

There is a demand for an integrated tool that offers users easily accessible transcripts, summarizes sentiment of comments, and provides instant fact-checking functionality directly within the YouTube interface to solve these issues and encourage more critical consumption of online video content.

# Modules

The extension is built using a modular architecture:

1. **background.js (Service Worker):** The core processing unit.
   * Loads and manages API keys (YouTube, Gemini) from storage.
   * Listens for messages from the popup (popup.js) and context menu events.
   * Coordinates API calls to YouTube Data API (fetch comment threads, list caption tracks) and Google Gemini API (sentiment analysis,

fact-checking).

* + Implements logic for fetching transcripts (attempting page scraping first, then API fallback).
  + Processes data returned from APIs.
  + Sends status updates and results back to the popup.
  + Handles asynchronous operations using async/await.

1. **popup.html / popup.js (Popup UI):** The user interface displayed when the extension icon is clicked.
   * Provides buttons to trigger transcript fetching and comment analysis.
   * Displays status messages, processing indicators, and error notifications.
   * Renders the fetched transcript text.
   * Displays the results of the comment sentiment analysis (counts, sample comments).
   * Displays the results of fact-checking requests initiated via the context menu.
   * Listens for messages from the background script (background.js) containing results or status updates.
2. **options.html / options.js (Options Page):** (Code not provided but functionality implied)
   * Provides a user interface (opened in a separate tab) for users to input and save their YouTube Data API and Gemini API keys.
   * Uses chrome.storage.local (or sync) to securely store the API keys for use by background.js.

# content.js (Content Script):

* + Minimal script primarily used as a target for Chrome.scripting.executeScript from the background script.
  + Its main function in this implementation is to execute code within the context of the active YouTube tab to extract the current video ID from the page's URL (window.location.search).

# manifest.json (Configuration File):

* + Defines the extension's properties, version, name, and description.
  + Specifies required permissions (activeTab, scripting, storage, contextMenus, declarativeNetRequest).
  + Declares host permissions required to interact with YouTube and Google API endpoints.
  + Registers the background service worker (background.js).
  + Defines the popup UI (popup.html).
  + Registers the options page (options.html).
  + Sets up the context menu item ("Fact-Check Selection").
  + Declares the Declarative Net Request ruleset (rules.json).

# rules.json (Declarative Net Request Rules):

* + Contains rules to modify network requests. In this case, it includes a rule to potentially modify the Sec-Fetch-Site header for requests made from YouTube pages, which might assist in certain types of data fetching, although its specific impact requires detailed testing.

# Packages/API Used

* **Chrome Extension APIs:**
  + chrome.runtime: For messaging between scripts (sendMessage, onMessage) and managing the extension lifecycle (onInstalled).
  + chrome.storage: For storing API keys (local.get, local.set).
  + chrome.scripting: For executing scripts in the context of the active tab (executeScript).
  + chrome.tabs: For querying active tab information (query).
  + chrome.contextMenus: For creating and managing the right-click context menu item (create, onClicked).
  + chrome.declarativeNetRequest: For modifying network requests based on predefined rules.

# Web APIs:

* + fetch: For making HTTP requests to external APIs (YouTube, Gemini).
  + URLSearchParams: For parsing URL query parameters (to get video ID).
  + DOM APIs (document.querySelector, window.location): Used within executeScript to access page data.
  + JSON.parse, JSON.stringify: For handling JSON data from APIs.

# External APIs:

* + **YouTube Data API v3:**
    - commentThreads.list: To fetch top-level comments and replies for a video.
    - captions.list: To list available caption tracks for a video (used as a fallback for transcript fetching).

# Google Generative AI API (Gemini):

* + - models/gemini-1.5-flash-latest:generateContent: Used for comment sentiment analysis (balancing speed and cost).
    - models/gemini-1.5-pro-latest:generateContent: Used for fact-checking (leveraging stronger reasoning capabilities).

# Methodology with Architecture

* 1. **Architecture Overview**

The extension employs an event-driven architecture common in Chrome extensions, centred around a background service worker (background.js) that coordinates actions and communicates with the popup UI and external APIs.

**Initialisation:** On installation or startup, the background script loads API keys from chrome. storage and sets up the context menu.

# User Interaction (Popup):

* + - The user clicks the extension icon, opening popup.html.
    - popup.js sends a message (e.g., getTranscript, analyzeComments) to background.js containing the action and the active tab ID.

# User Interaction (Context Menu):

* + - The user selects text on the YouTube page.
    - The user right-clicks and selects "Fact-Check Selection".
    - The chrome.contextMenus.onClicked listener in background.js triggers, sending a factCheckSelection message internally with the selected text.

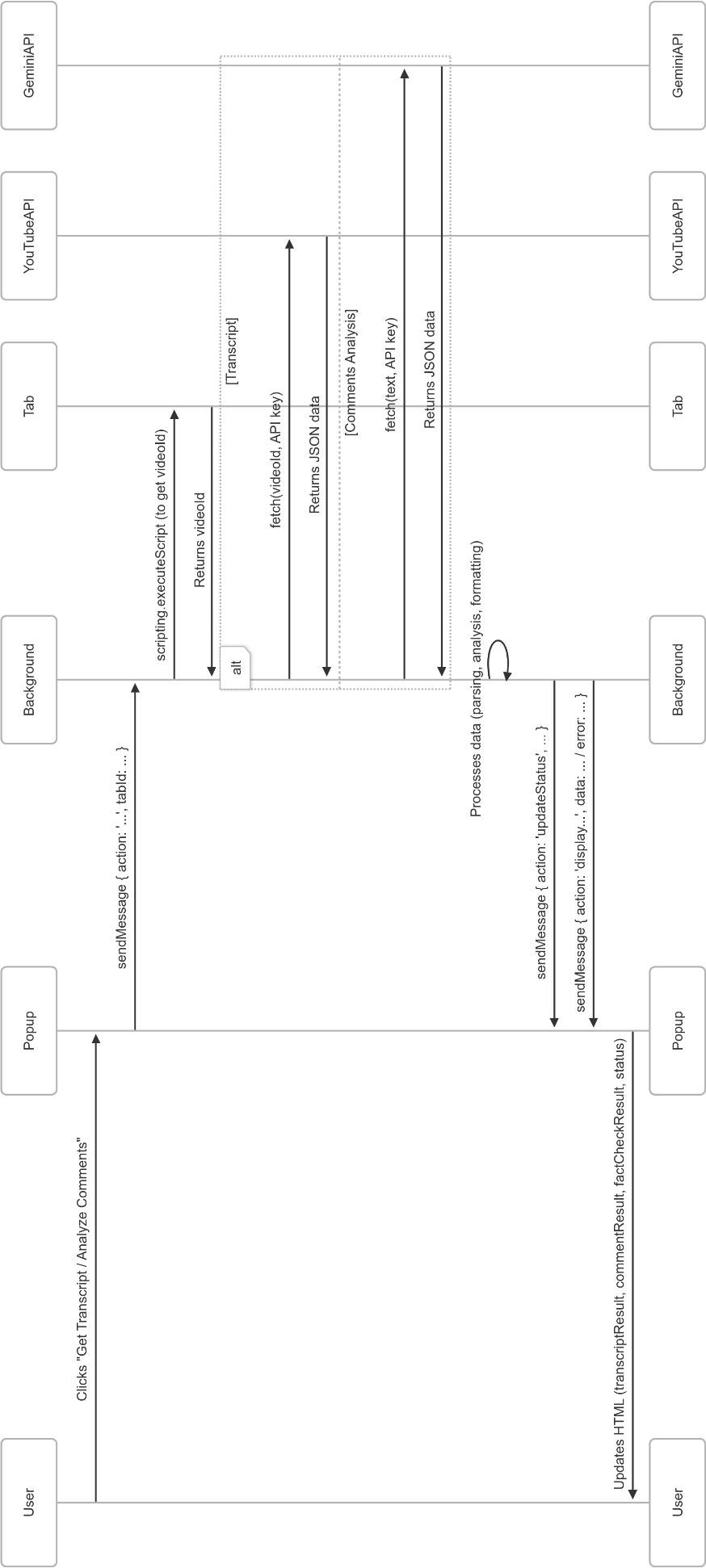
# Background Processing:

* + - background.js receives the message.
    - It sends status updates to the popup (updateStatus message).
    - If needed (for transcript/comments), it injects a script via Chrome. scripting. Execute the script in the active tab (content.js context) to retrieve the video ID.
    - **Transcript:** Attempts to fetch the watch page HTML, parse ytInitialPlayerResponse for caption track URLS, and fetch the transcript text. If parsing fails, it uses the YouTube Captions API to list available languages (but doesn't fetch the text).
    - **Comments:** Uses the YouTube CommentThreads API (handling pagination) to fetch comments. It then batches comments and sends them to the Gemini API (Flash model) for sentiment analysis, parsing the structured JSON response.
    - **Fact-Check:** Sends the selected text and a structured prompt to the Gemini API (Pro model), requesting a JSON response with verdict, confidence, explanation, and sources.
    - Handles potential errors during API calls or data processing.

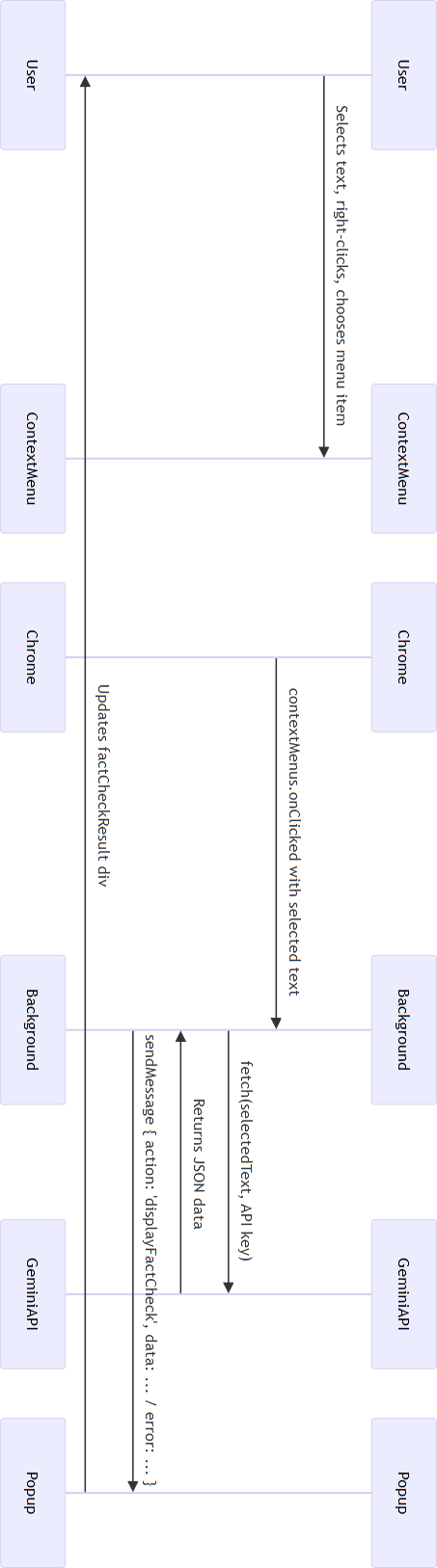
# Displaying Results:

* + - background.js sends the processed data or error messages back to popup.js using specific action messages (e.g., displayTranscript, displayCommentAnalysis, displayFactCheck).
    - popup.js receives these messages and updates the corresponding sections in popup.html to display the results to the user.

# Data Flow Description



*Fact-Check Flow:*

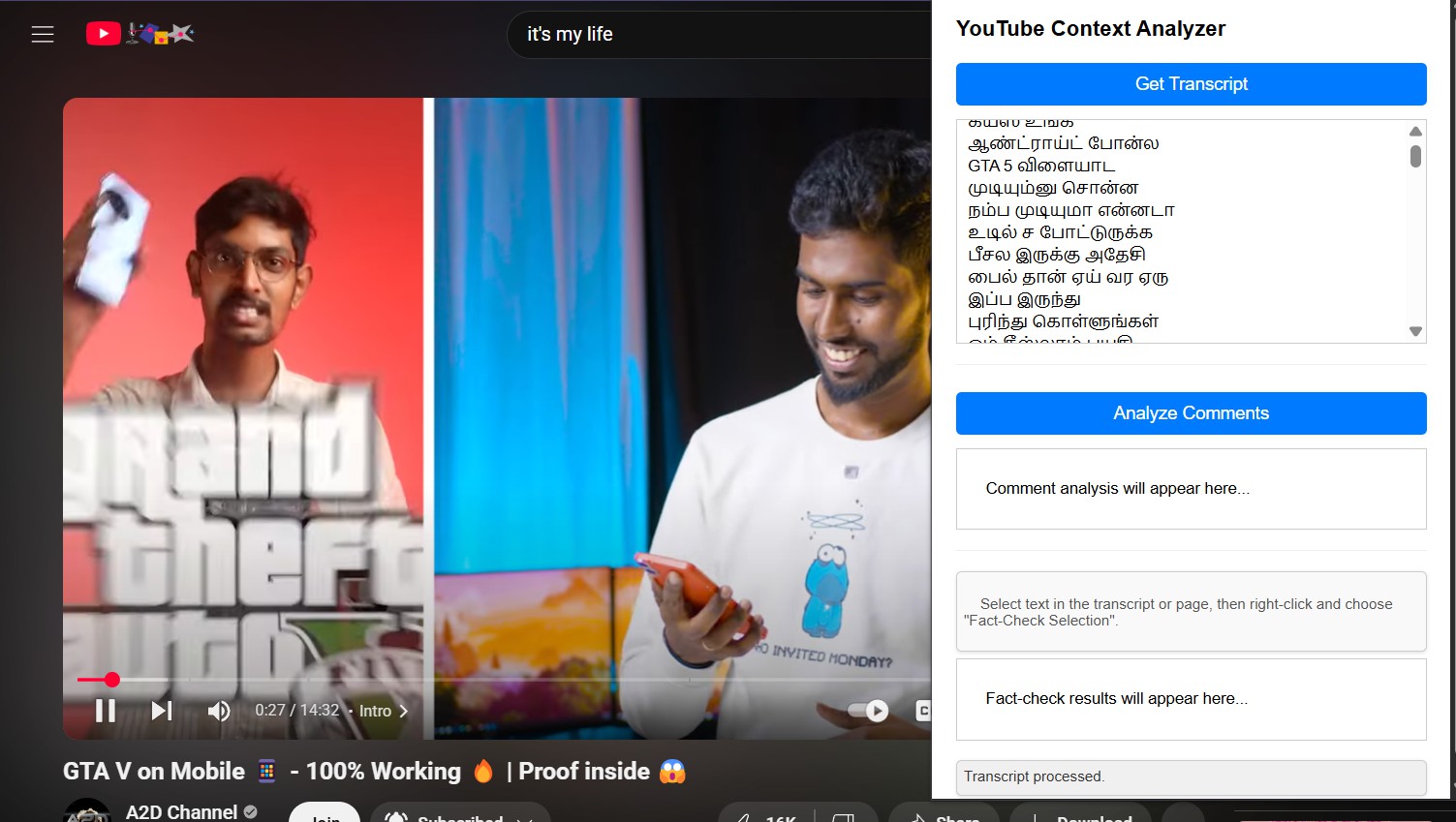
**

# Results and Discussion

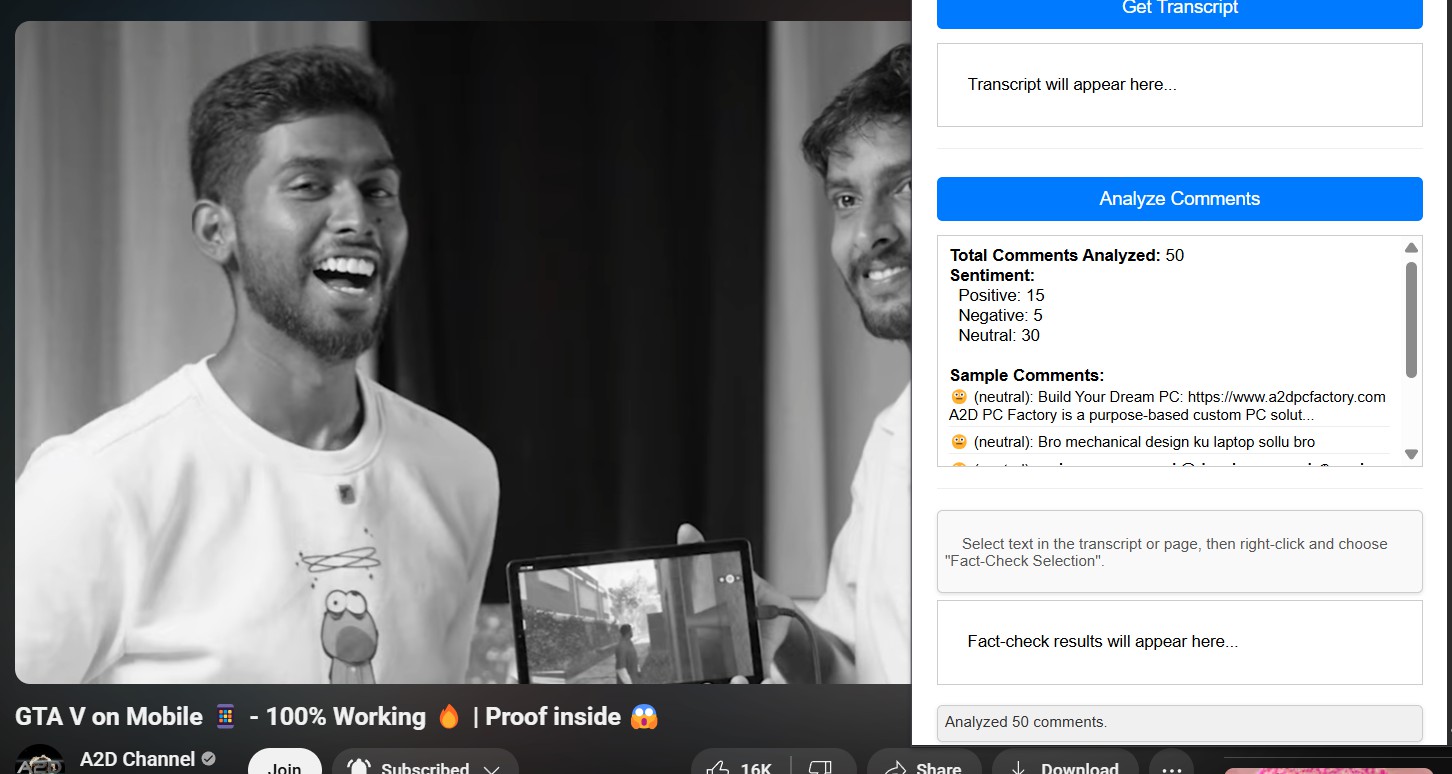
*(Note: As no specific result screenshots were provided, this section describes the expected outcomes based on the implemented code.)*

Upon successful execution, the extension provides the following results within its popup interface:

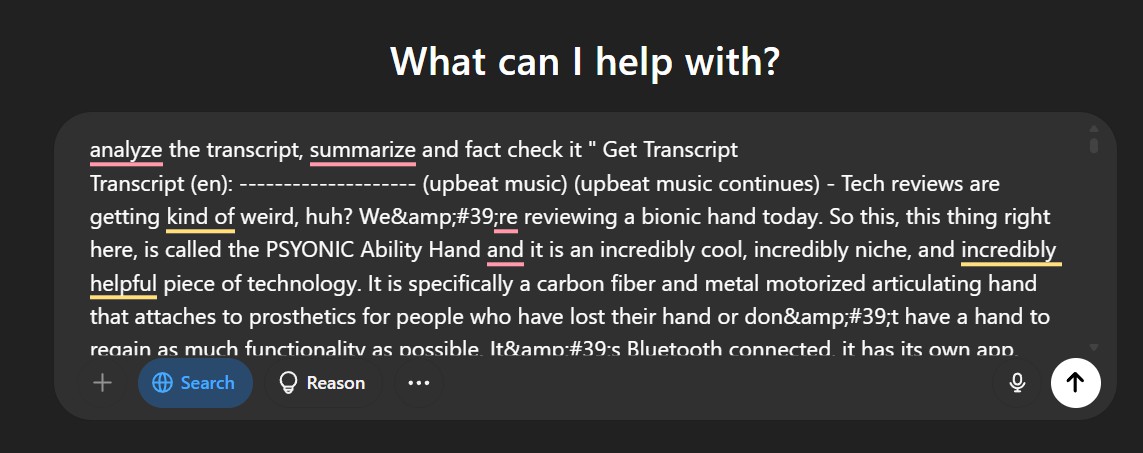
1. **Transcript Display:** The transcriptResult div is populated with the fetched transcript text. If fetched via page parsing, it includes the full text. If the fallback API method is used (due to parsing failure or lack of direct access), it displays a message listing the available caption languages found via the API, indicating that direct text download wasn't possible through that method. Errors during fetching are displayed clearly.

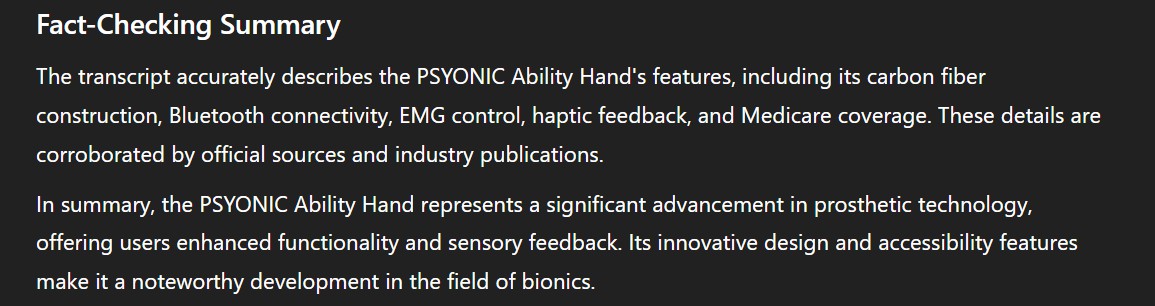


1. **Comment Analysis:** The commentResult div shows:
   * The total number of comments fetched and analyzed.
   * A breakdown of sentiment counts (Positive, Negative, Neutral).
   * A sample (e.g., first 5) of the analyzed comments, prefixed with a sentiment emoji and showing the beginning of the comment text. This gives a qualitative feel alongside the quantitative summary. Errors during API calls or analysis are reported.

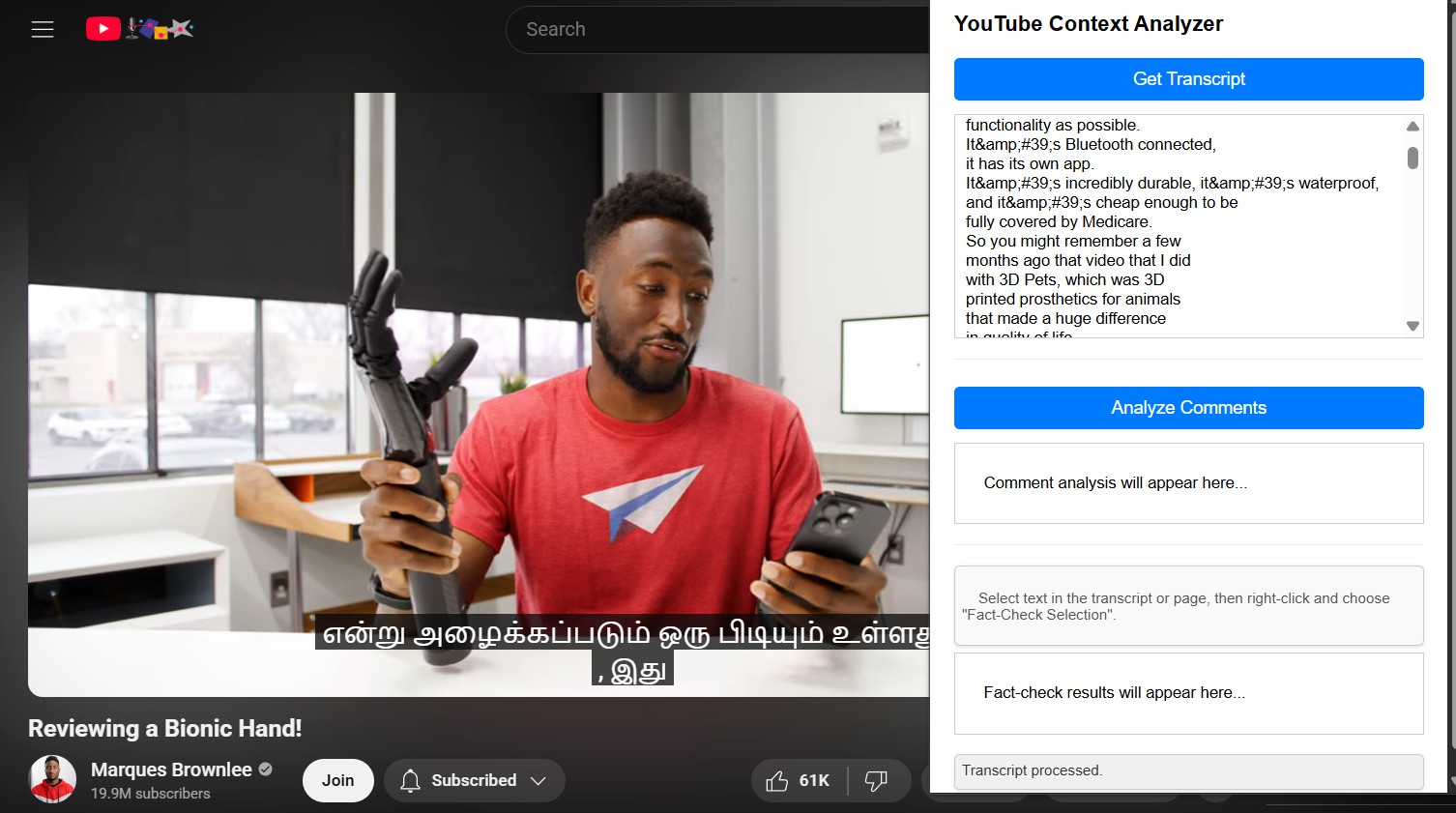


1. **Fact-Check Results:** When a fact-check is initiated via the context menu, the factCheckResult div displays the structured response from the Gemini API:
   * **Verdict:** (e.g., "True", "False", "Misleading", "Unverifiable").
   * **Confidence:** A percentage score indicating the AI's confidence in the verdict.
   * **Explanation:** A concise rationale for the verdict.
   * **Sources:** A list of potential source URLs or references provided by the AI (displayed as clickable links if possible). Errors are shown if the fact-check fails.

Not functional yet due to API limitations, but an alternate method is to use the transcript as a Chatgpt or other models prompt and verify the fact



A summary of the MKBHD’s youtube video



1. **Status Updates:** The status div provides real-time feedback on the extension's operations (e.g., "Fetching comments...", "Analyzing sentiment...", "Fact-check complete.", "Error: API Key invalid.").

# Discussion:

The add-on effectively brings several contextual analysis features into the YouTube environment. The transcript function facilitates rapid skimming of content. The comment analysis offers an invaluable, otherwise unobtainable, summary of community response. The fact-checking functionality provides an easy means of checking claims without exiting the page. The use of external APIs (YouTube, Gemini) makes the analysis strong but also adds dependencies and possible expense. The transcript retrieval strategy (page parse first) is a try to circumvent more rigid API requirements, but is less solid as a result than actual API approaches. That multiple Gemini models are used (Flash for comments, Pro for fact-checking) is a design choice that prioritizes

performance/cost over reasoning power. Overall, the tool supplies considerable value addition to those looking to delve deeper into context on YouTube.

# Constraints/Limitations

* **API Key Dependency:** Requires users to obtain and configure their own YouTube Data API and Gemini API keys via the options page. Functionality is blocked without valid keys.
* **API Costs & Quotas:** Use of YouTube Data API and Gemini API is subject to quotas and potential costs, which the user bears. Heavy usage might exceed free tiers.
* **Transcript Fetching Reliability:** The primary method of fetching transcripts by parsing page data (ytInitialPlayerResponse) is fragile and may break if YouTube changes its website structure. The API fallback only lists languages, not the text itself without potentially more complex authentication.
* **AI Accuracy and Bias:** The accuracy of sentiment analysis and fact-checking depends entirely on the Gemini models. These models can make errors, exhibit biases present in their training data, or struggle with nuanced/context-dependent language. Verdicts should be seen as informative aids, not definitive truths.
* **Limited Fact-Check Scope:** Fact-checking only operates on user-selected text snippets, not the entire video's audio/visual content. Context outside the selected text is missed.
* **Comment Analysis Scope:** Analysis is typically performed on a subset of comments (limited by maxResults and API pagination) for performance and cost reasons, not necessarily *all* comments on a very popular video.
* **Network Dependency:** Requires an active internet connection to communicate with Google APIs.
* **Browser Compatibility:** Designed as a Chrome Extension, likely compatible with other Chromium-based browsers (Edge, Brave) but not Firefox or Safari without modification.
* **Real-time Updates:** Analysis is triggered on demand (button click/context menu), not continuously monitoring in the background unless re-triggered.

# Challenges Faced and How They Were Fixed

1. **Dynamic Content Loading on YouTube:** YouTube heavily uses AJAX/SPA principles, meaning video changes or data updates don't always trigger a full page reload. Simple content scripts running only on page load would miss updates.
   * **Solution:** While not explicitly shown in the final code snippets for *this*

version, a common solution involves using MutationObserver in the

content script to detect relevant DOM changes (like the video player updating) and re-trigger data extraction. For this version, relying on user clicks in the popup for explicit re-analysis circumvents this for transcript/comments. The video ID is fetched dynamically on each request using scripting.executeScript.

1. **Extracting Video ID Reliably:** Getting the current video ID is crucial. Relying solely on the content script's initial load might fail if the user navigates within YouTube.
   * **Solution:** Implemented fetching the video ID *on demand* from the background script using Chrome.scripting.executeScript targeting the active tab. This ensures the ID corresponds to the currently viewed video when an action is triggered.
2. **Fetching Transcripts without OAuth:** Direct programmatic access to YouTube transcript text often requires OAuth 2.0 authentication, which is complex for a simple extension.
   * **Solution:** Employed a two-pronged approach:
     + Attempt to parse the ytInitialPlayerResponse object embedded in the YouTube watch page's HTML, which sometimes contains caption track URLs accessible without OAuth. This is less reliable but avoids complex authentication.
     + If parsing fails, fall back to using the YouTube Data API (captions.list) which only requires an API key but just lists available tracks, not the content. The result informs the user which languages are available.
3. **Structuring AI Prompts for Reliable Output:** Getting consistent and correctly formatted output (especially JSON) from Generative AI models like Gemini requires careful prompt engineering.
   * **Solution:** Developed specific prompts clearly outlining the task (sentiment analysis, fact-checking) and explicitly requesting the output *only* in a valid JSON format with predefined fields. Included instructions like "Respond ONLY with..." and used the responseMimeType: "application/json" parameter (where supported/effective) to further encourage correct formatting. Added try...catch blocks around JSON.parse to handle cases where the AI might still return malformed output.
4. **Handling API Errors and Rate Limits:** Calls to external APIs can fail due to invalid keys, network issues, quota limits, or server errors.
   * **Solution:** Implemented robust error handling using try...catch blocks around fetch calls. Checked the response.ok status and attempted to parse error messages from the API response body. Displayed informative error messages to the user via the popup's status area instead of crashing.
5. **Managing Asynchronous Operations:** Multiple steps involve asynchronous operations (fetching keys, getting video ID, API calls). Coordinating these correctly is essential.
   * **Solution:** Utilized async/await syntax extensively throughout the background.js script to manage asynchronous flows logically and avoid callback hell, making the code easier to read and maintain.

# Code Implementation

*(Includes the provided code snippets with explanations relevant to the final functionality)*

# manifest.json

{

"manifest\_version": 3,

"name": "YouTube Context Analyzer", "version": "0.2.0",

"description": "Extracts transcripts, analyzes comments, and fact-checks YouTube videos.",

"permissions": [ "activeTab", "scripting", "storage", "contextMenus",

"declarativeNetRequest"

],

"declarative\_net\_request": { "rule\_resources": [

{

"id": "ruleset\_1", "enabled": true, "path": "rules.json"

}

]

},

"host\_permissions": [ "\*://\*.youtube.com/\*", "https://\*.googleapis.com/\*"

],

"background": {

"service\_worker": "background.js"

},

"action": {

"default\_popup": "popup.html", "default\_icon": {

"16": "icons/icon16.png",

"32": "icons/icon32.png",

"48": "icons/icon48.png"

}

},

"options\_ui": {

"page": "options.html", "open\_in\_tab": true

},

"icons": {

"16": "icons/icon16.png",

"32": "icons/icon32.png",

"48": "icons/icon48.png",

"128": "icons/icon128.png"

}

}

* **manifest\_version: 3**: Specifies the current manifest version.
* **permissions**: Declares necessary permissions: activeTab (access current tab), scripting (inject scripts), storage (save API keys), contextMenus (add right-click menu), declarativeNetRequest (modify network requests).
* **declarative\_net\_request**: Links to rules.json for network request modification rules.
* **host\_permissions**: Grants access to YouTube pages and Google API endpoints.
* **background**: Registers background.js as the service worker.
* **action**: Defines the popup (popup.html) and icons.
* **options\_ui**: Specifies the options page (options.html) for API key configuration.

# background.js (Service Worker)

// --- Globals ---

let YOUTUBE\_API\_KEY = null; let GEMINI\_API\_KEY = null;

// --- Utility Functions ---

// Load API keys from storage async function loadAPIKeys() {

try {

const keys = await chrome.storage.local.get(['youtubeApiKey', 'geminiApiKey']);

YOUTUBE\_API\_KEY = keys.youtubeApiKey || null; GEMINI\_API\_KEY = keys.geminiApiKey || null;

console.log("API keys loaded",

YOUTUBE\_API\_KEY ? "YouTube: ✓" : "YouTube: ✗", GEMINI\_API\_KEY ? "Gemini: ✓" : "Gemini: ✗");

return { youtubeKey: YOUTUBE\_API\_KEY, geminiKey: GEMINI\_API\_KEY

};

} catch (err) {

console.error("Error loading API keys:", err); return { youtubeKey: null, geminiKey: null };

}

}

// Send status updates to popup

function updatePopupStatus(message, isError = false, isProcessing = undefined) {

const statusMessage = { action: "updateStatus", message: message, isError: isError };

if (isProcessing !== undefined) {

statusMessage.isProcessing = isProcessing;

}

chrome.runtime.sendMessage(statusMessage).catch(err => {/\* Popup likely closed \*/ });

}

// Send data results to popup

function sendDataToPopup(action, data, error = null) { const message = { action: action };

if (data) message.data = data;

if (error) message.error = error;

chrome.runtime.sendMessage(message).catch(err => {/\* Popup likely closed \*/ });

}

// Helper to extract video ID from tab async function getVideoIdFromTab(tabId) {

try {

const response = await chrome.scripting.executeScript({ target: { tabId: tabId },

function: () => { const params = new

URLSearchParams(window.location.search);

return params.get('v');

}

});

const videoId = response[0]?.result; if (!videoId) {

throw new Error("Could not extract video ID from URL.");

}

return videoId;

} catch (error) {

console.error("Error injecting script or extracting video ID:",

error);

}

}

throw new Error(`Failed to get video ID: ${error.message}`);

// --- Core Logic Functions ---

// Attempt to fetch transcript text by parsing page data (avoids OAuth) async function fetchTranscriptFromPage(videoId, langPrefs = ['ta', 'en']) {

updatePopupStatus("Attempting to fetch transcript data from page...", false, true);

const watchUrl = [`https://www.youtube.com/watch?v=${](http://www.youtube.com/watch?v=%24)videoId}`;

try {

// Fetch the watch page HTML

const response = await fetch(watchUrl, { headers: {

// Try to mimic browser headers somewhat

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.0.0 Safari/537.36',

'Accept-Language': 'en-US,en;q=0.9'

}

});

if (!response.ok) {

throw new Error(`Failed to fetch watch page:

${response.statusText}`);

}

const html = await response.text();

// Find the player response data (often in a script tag)

// This regex is fragile and might need updating if YouTube changes structure

const playerResponseMatch =

html.match(/ytInitialPlayerResponse\s\*=\s\*({.+?})\s\*;/);

if (!playerResponseMatch || !playerResponseMatch[1]) {

console.log("Could not find ytInitialPlayerResponse in page

HTML.");

// Fallback to API list method

return fetchAvailableTranscriptLangsAPI(videoId);

}

const playerResponse = JSON.parse(playerResponseMatch[1]);

if (!playerResponse?.captions?.playerCaptionsTracklistRenderer)

{

console.log("No captions renderer found in player

response.");

return fetchAvailableTranscriptLangsAPI(videoId); //

Fallback

}

const tracks =

playerResponse.captions.playerCaptionsTracklistRenderer.captionTracks; if (!tracks || tracks.length === 0) {

console.log("No caption tracks found in player response."); return fetchAvailableTranscriptLangsAPI(videoId); //

Fallback

}

console.log("Available tracks found in page data:", tracks.map(t

=> ({ lang: t.languageCode, kind: t.kind })));

let bestTrackUrl = null; let foundLang = null;

// Find the best matching track based on preferences for (const lang of langPrefs) {

const track = tracks.find(t => t.languageCode === lang &&

!t.kind); // Prioritize non-ASR first if available if (track) {

bestTrackUrl = track.baseUrl; foundLang = lang;

break;

}

// Check ASR (auto-generated) if non-ASR not found for preferred lang

const asrTrack = tracks.find(t => t.languageCode === lang && t.kind === 'asr');

if (!bestTrackUrl && asrTrack) {

bestTrackUrl = asrTrack.baseUrl; foundLang = lang + " (auto)";

}

}

// If no preferred language found, take the first available

track

if (!bestTrackUrl && tracks.length > 0) {

bestTrackUrl = tracks[0].baseUrl;

foundLang = tracks[0].languageCode + (tracks[0].kind === 'asr' ? " (auto)" : "");

}

if (!bestTrackUrl) {

throw new Error("Could not find a suitable caption track URL in page data.");

}

updatePopupStatus(`Fetching transcript content for language:

${foundLang}...`);

// Fetch the actual transcript XML/timed text

const transcriptResponse = await fetch(bestTrackUrl); if (!transcriptResponse.ok) {

throw new Error(`Failed to fetch transcript content:

${transcriptResponse.statusText}`);

}

const transcriptXML = await transcriptResponse.text();

// Simple XML parsing to extract text (could be improved with DOMParser)

const lines = []; const textMatches =

transcriptXML.match(/<text.\*?>(.\*?)<\/text>/gs); if (textMatches) {

textMatches.forEach(match => {

let text = match.replace(/<[^>]+>/g, ''); // Remove tags text = text.replace(/&/g, '&').replace(/</g,

'<').replace(/>/g, '>').replace(/"/g, '"').replace(/'/g, "'"); // Decode HTML entities

lines.push(text.trim());

});

}

if (lines.length === 0) {

throw new Error("Transcript content fetched but no text

found after parsing.");

}

const fullTranscript = lines.join('\n');

sendDataToPopup("displayTranscript", `Transcript (${foundLang}):\n \n${fullTranscript}`);

return fullTranscript; // Success

} catch (error) {

console.error("Error fetching transcript from page:", error); updatePopupStatus(`Transcript fetch from page failed:

${error.message}. Trying API list fallback...`, true);

// Fallback to API list method on any error during page parsing return fetchAvailableTranscriptLangsAPI(videoId);

}

}

// Fallback: Use API key to list available languages (doesn't get text) async function fetchAvailableTranscriptLangsAPI(videoId) {

updatePopupStatus("Fetching available caption languages via API...");

const { youtubeKey } = await loadAPIKeys(); if (!youtubeKey) {

// Don't throw here, just send error to popup

const errorMsg = "YouTube API Key not set in options."; sendDataToPopup("displayTranscript", null, errorMsg);

updatePopupStatus(errorMsg, true, false); return null;

}

try {

const captionsListUrl =

[`https://www.googleapis.com/youtube/v3/captions?part=snippet&videoId=${](http://www.googleapis.com/youtube/v3/captions?part=snippet&videoId=%24)v ideoId}&key=${youtubeKey}`;

const response = await fetch(captionsListUrl);

if (!response.ok) {

const errorData = await response.json();

throw new Error(`API Error (${response.status}):

${errorData?.error?.message || response.statusText}`);

}

const captionsData = await response.json();

if (!captionsData.items || captionsData.items.length === 0) { throw new Error("No captions found via API for this

video.");

}

const languages = captionsData.items.map(item =>

`${item.snippet.language} (${item.snippet.trackKind})`);

const message = `Transcript text download requires OAuth or may be restricted.\nAvailable caption tracks found via API:\n-

${languages.join('\n- ')}`;

sendDataToPopup("displayTranscript", message); // Send info

message

return message; // Return the info string

} catch (error) {

console.error("Error fetching transcript list via API:", error); const errorMsg = `Transcript check via API failed:

${error.message}`;

// Don't throw here, just send the error message to the popup sendDataToPopup("displayTranscript", null, errorMsg);

updatePopupStatus(errorMsg, true, false); return null; // Indicate failure

}

}

async function fetchAndAnalyzeComments(videoId, maxResults = 50) { // Limit results initially

updatePopupStatus(`Fetching comments for video: ${videoId}...`, false, true);

const { youtubeKey, geminiKey } = await loadAPIKeys();

if (!youtubeKey) {

const errorMsg = "YouTube API Key not set in options.";

sendDataToPopup("displayCommentAnalysis", null, errorMsg);

updatePopupStatus(errorMsg, true, false); return; // Stop execution

}

if (!geminiKey) {

const errorMsg = "Gemini API Key not set in options.";

sendDataToPopup("displayCommentAnalysis", null, errorMsg); updatePopupStatus(errorMsg, true, false);

return; // Stop execution

}

let comments = [];

let nextPageToken = null; let fetchedCount = 0;

try {

// Fetch comments using YouTube Data API with pagination do {

let commentsUrl =

[`https://www.googleapis.com/youtube/v3/commentThreads?part=snippet&video](http://www.googleapis.com/youtube/v3/commentThreads?part=snippet&video) Id=${videoId}&maxResults=${Math.min(maxResults - fetchedCount,

100)}&key=${youtubeKey}&textFormat=plainText`; // Request plain text if (nextPageToken) {

commentsUrl += `&pageToken=${nextPageToken}`;

}

updatePopupStatus(`Fetching comment page (fetched

${fetchedCount}/${maxResults})...`);

const response = await fetch(commentsUrl); if (!response.ok) {

const errorData = await response.json();

// Specific check for quota error if (response.status === 403 &&

errorData?.error?.message?.includes('quota')) {

throw new Error(`YouTube API Quota Exceeded. Please check your usage limits in Google Cloud Console.`);

}

throw new Error(`YouTube API Error (${response.status}):

${errorData?.error?.message || response.statusText}`);

}

const commentsData = await response.json();

if (commentsData.items) {

const pageComments = commentsData.items

.map(item => ({ id: item.id, text:

item.snippet?.topLevelComment?.snippet?.textOriginal || '', // Use textOriginal

}))

.filter(comment => comment.text.trim().length > 0);

// Filter out empty comments

added

comments = comments.concat(pageComments);

fetchedCount += pageComments.length; // Use actual count

}

nextPageToken = commentsData.nextPageToken;

} while (nextPageToken && fetchedCount < maxResults); if (comments.length === 0) {

sendDataToPopup("displayCommentAnalysis", { totalAnalyzed: 0, totalFetched: fetchedCount, sentiment: { positive: 0, negative: 0, neutral: 0 }, sampleAnalyzedComments: [] });

updatePopupStatus(fetchedCount > 0 ? "Fetched comments, but none had text content." : "No comments found or fetched.", false, false);

return;

}

updatePopupStatus(`Analyzing ${comments.length} comments using Gemini...`);

// Prepare comments for Gemini batch analysis

const batchSize = 10; // Adjust batch size as needed const analyzedComments = [];

const sentimentCounts = { positive: 0, negative: 0, neutral: 0

};

for (let i = 0; i < comments.length; i += batchSize) { const batch = comments.slice(i, i + batchSize);

// Refined prompt with examples

const promptText = `Analyze the sentiment (positive, negative, or neutral) for each of the following YouTube comments. Respond ONLY with a valid JSON array where each element is an object containing the original 'id' and the 'sentiment' classification. Do not include any other text or markdown formatting.

Example Input Comments:

[{"id":"id\_1","text":"This video is amazing!"},{"id":"id\_2","text":"I

learned nothing."},{"id":"id\_3","text":"It was okay."}]

Example Output JSON: [{"id":"id\_1","sentiment":"positive"},{"id":"id\_2","sentiment":"negative "},{"id":"id\_3","sentiment":"neutral"}]

Actual Comments to Analyze:

${JSON.stringify(batch)}`;

const geminiApiUrl =

`https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flas h-latest:generateContent?key=${GEMINI\_API\_KEY}`; // Use Flash for

speed/cost

const geminiResponse = await fetch(geminiApiUrl, { method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ contents: [{ parts: [{ text: promptText }] }] })

});

if (!geminiResponse.ok) {

console.error(`Gemini API error

(${geminiResponse.status}) for comment batch ${Math.floor(i / batchSize)

+ 1}`);

const errorData = await geminiResponse.json().catch(()

=> ({})); // Try to parse error body

later

console.error("Gemini Error Details:", errorData);

// Skip this batch on error, maybe add placeholders

updatePopupStatus(`Warning: Gemini API error for batch

${Math.floor(i / batchSize) + 1}. Skipping analysis for this batch.`, true);

// Add neutrals for skipped batch to maintain counts? Or just skip adding. Let's skip.

continue; // Move to next batch

}

first

// \*\*\* START OF INTEGRATED DEBUGGING BLOCK \*\*\*

try {

const geminiResult = await geminiResponse.json();

// Check for explicit error structure in response body

if (geminiResult.error) {

console.error(`Gemini API returned an error object:

${geminiResult.error.message}`);

updatePopupStatus(`Warning: Gemini API returned error for batch ${Math.floor(i / batchSize) + 1}. Skipping analysis.`, true);

continue; // Skip this batch

}

const responseText =

geminiResult?.candidates?.[0]?.content?.parts?.[0]?.text || '';

// \*\*\* ADD THIS LOGGING \*\*\*

console.log("Raw Gemini Response Text for batch:", Math.floor(i / batchSize) + 1, "\n", responseText);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

backticks)

// Basic cleanup attempt for common issues (markdown

const cleanedText =

responseText.trim().replace(/^```json\s\*|```$/g, '');

// Try to parse the expected JSON array directly

const sentiments = JSON.parse(cleanedText); // Parse

cleaned text

// \*\*\* ADD THIS LOGGING \*\*\*

console.log("Parsed Sentiments for batch:", Math.floor(i

/ batchSize) + 1, "\n", sentiments);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (Array.isArray(sentiments)) {

sentiments.forEach(sentimentResult => {

const originalComment = batch.find(c => c.id ===

sentimentResult.id);

sentimentResult);

if (originalComment) {

// \*\*\* ADD THIS LOGGING \*\*\*

console.log("Processing sentimentResult:",

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const sentiment = (sentimentResult.sentiment

|| 'neutral').toLowerCase().trim(); // Lowercase and trim

analyzedComments.push({ ...originalComment,

sentiment: sentiment }); undefined) {

if (sentimentCounts[sentiment] !== sentimentCounts[sentiment]++;

} else {

console.warn(`Unexpected sentiment value

"${sentimentResult.sentiment}", defaulting to neutral.`); // Log unexpected values

sentimentCounts.neutral++;

}

} else {

console.warn("Could not find original comment for ID:", sentimentResult.id);

}

});

} else {

console.warn("Gemini sentiment response was not a valid JSON array:", cleanedText);

// Add default neutral for the whole batch if parsing failed at array level

batch.forEach(comment => {

analyzedComments.push({ ...comment, sentiment:

'neutral' });

sentimentCounts.neutral++;

});

}

} catch (parseError) {

console.error("Failed to parse Gemini sentiment response:", parseError, "Response text:",

geminiResult?.candidates?.[0]?.content?.parts?.[0]?.text);

// Fallback: Add default neutral for the whole batch on

parse error

'neutral' });

batch.forEach(comment => {

analyzedComments.push({ ...comment, sentiment:

sentimentCounts.neutral++;

});

updatePopupStatus(`Warning: Could not parse Gemini response for batch ${Math.floor(i / batchSize) + 1}.`, true);

}

// \*\*\* END OF INTEGRATED DEBUGGING BLOCK \*\*\*

} // End batch loop

const results = {

totalFetched: fetchedCount,

totalAnalyzed: analyzedComments.length, sentiment: sentimentCounts,

sampleAnalyzedComments: analyzedComments // Send all analyzed back for now

};

sendDataToPopup("displayCommentAnalysis", results);

updatePopupStatus(`Analyzed ${analyzedComments.length} comments.`, false, false);

return results;

} catch (error) {

console.error("Error fetching/analyzing comments:", error); const errorMsg = `Comment analysis failed: ${error.message}`; sendDataToPopup("displayCommentAnalysis", null, errorMsg);

updatePopupStatus(errorMsg, true, false);

// Don't re-throw, let the flow end gracefully after informing the user

}

}

async function performFactCheck(textToFactCheck) {

updatePopupStatus(`Fact-checking selected text...`, false, true); const { geminiKey } = await loadAPIKeys();

if (!geminiKey) {

const errorMsg = "Gemini API Key not set in options."; sendDataToPopup("displayFactCheck", null, errorMsg);

updatePopupStatus(errorMsg, true, false); return; // Stop execution

}

try {

const geminiApiUrl =

`https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-pro- latest:generateContent?key=${geminiKey}`; // Use Pro for better

reasoning

// Refined prompt requesting specific JSON structure and asking for JSON mime type

const prompt = { contents: [{

parts: [{

text: `Please act as a neutral fact-checker. Analyze the following claim: "${textToFactCheck}"

Respond ONLY with a single, valid JSON object containing the following fields:

* "verdict": A string classification ("True", "False", "Partially True", "Misleading", "Unverifiable", "Opinion").
* "confidence": A number between 0.0 (low confidence) and

1.0 (high confidence) in your verdict.

* "explanation": A concise string explaining your reasoning (1-2 sentences).
* "sources": An array of strings, listing URL(s) or credible references supporting your conclusion. If unverifiable or opinion, the array can be empty or contain a note like "No specific sources applicable".

Do not include any introductory text, concluding remarks, or markdown formatting like \`\`\`json ... \`\`\` around the JSON object.`

}]

}],

// Optional: Add safety settings if needed

//"safetySettings": [ ... ], "generationConfig": {

"responseMimeType": "application/json", // Explicitly

request JSON

"temperature": 0.2 // Lower temperature for more

factual, less creative response

}

};

const response = await fetch(geminiApiUrl, { method: 'POST',

headers: { 'Content-Type': 'application/json' }, body: JSON.stringify(prompt)

});

if (!response.ok) {

const errorData = await response.json().catch(() => ({}));

// Try to parse error, default to empty obj

console.error("Gemini Fact-Check Error Response Body:",

errorData);

throw new Error(`Gemini API Error (${response.status}):

${errorData?.error?.message || response.statusText}`);

}

// Since we requested JSON directly, parse it const geminiResult = await response.json();

// Extract content, checking for potential error structure first if (geminiResult.error) {

console.error(`Gemini API returned an error object during fact-check: ${geminiResult.error.message}`);

throw new Error(`Gemini fact-check failed:

${geminiResult.error.message}`);

}

let parsedResult;

// Check if the expected JSON payload is directly within the candidate's content part

if (geminiResult?.candidates?.[0]?.content?.parts?.[0]?.text) { try {

// Attempt to parse the text part, as sometimes the mime type might not be fully respected

// or the content is nested differently. const textContent =

geminiResult.candidates[0].content.parts[0].text;

parsedResult = JSON.parse(textContent);

} catch (e) {

console.warn("Failed to parse JSON from 'text' part, trying direct 'content' if available.");

// Fallback check: If the content itself is the JSON object (due to mime type request being fully respected)

if (typeof geminiResult?.candidates?.[0]?.content === 'object' && !Array.isArray(geminiResult.candidates[0].content.parts)) {

// This structure might vary based on API; adjust if

needed.

// The check `!Array.isArray(...)` attempts to

differentiate from the 'parts' structure.

parsedResult = geminiResult.candidates[0].content;

} else {

console.error("Could not find parsable JSON in Gemini fact-check response:", geminiResult);

throw new Error("Fact-check response from Gemini was not valid JSON or in expected format.");

}

}

} else if (typeof geminiResult?.candidates?.[0]?.content === 'object') {

// Handles cases where the mime type worked and the content object \*is\* the result

parsedResult = geminiResult.candidates[0].content; // Adjust based on actual API response structure if mime type works

} else {

console.error("Could not find parsable content in Gemini fact-check response:", geminiResult);

throw new Error("Could not find parsable content in Gemini fact-check response.");

}

// Validate expected fields (good practice) const formattedResult = {

verdict: parsedResult.verdict || "Unknown",

confidence: parsedResult.confidence !== undefined ? parsedResult.confidence : null,

explanation: parsedResult.explanation || "No explanation

provided.",

sources: Array.isArray(parsedResult.sources) ?

parsedResult.sources : ["No specific sources provided"] // Default if missing/empty

};

sendDataToPopup("displayFactCheck", formattedResult);

updatePopupStatus('Fact-check complete.', false, false); return formattedResult;

} catch (error) {

console.error("Error during fact-check:", error);

const errorMsg = `Fact-check failed: ${error.message}`; sendDataToPopup("displayFactCheck", null, errorMsg);

updatePopupStatus(errorMsg, true, false);

// Don't re-throw

}

}

// --- Event Listeners ---

// Listen for messages from popup or content script

chrome.runtime.onMessage.addListener((request, sender, sendResponse) =>

{

console.log("Background received message:", request);

let isAsync = false; // Flag to indicate if we need to return true

if (request.action === "getTranscript" && request.tabId) { isAsync = true; // We will handle this asynchronously (async () => {

try {

updatePopupStatus("Getting video details...", false,

true);

if needed

const videoId = await getVideoIdFromTab(request.tabId); console.log("Extracted Video ID:", videoId);

// Attempt fetch from page first, fallback to API list

await fetchTranscriptFromPage(videoId);

// Status/data sent within fetchTranscriptFromPage or

fetchAvailableTranscriptLangsAPI

} catch (error) {

// Errors handled and sent to popup within the functions or getVideoIdFromTab

console.error("Error in getTranscript flow:", error); const errorMsg = `Transcript Error: ${error.message}`; updatePopupStatus(errorMsg, true, false);

sendDataToPopup("displayTranscript", null, errorMsg); // Ensure error is displayed

}

})();

} else if (request.action === "analyzeComments" && request.tabId) { isAsync = true;

(async () => { try {

updatePopupStatus("Getting video details for comments...", false, true);

videoId);

const videoId = await getVideoIdFromTab(request.tabId); console.log("Extracted Video ID for comments:",

await fetchAndAnalyzeComments(videoId);

// Status/data sent within fetchAndAnalyzeComments

} catch (error) {

// Errors from getVideoIdFromTab or initial key checks

might land here

console.error("Error in analyzeComments flow:", error); const errorMsg = `Comments Error: ${error.message}`;

updatePopupStatus(errorMsg, true, false);

sendDataToPopup("displayCommentAnalysis", null,

errorMsg); // Ensure error is displayed

}

})();

} else if (request.action === "factCheckSelection" && request.text)

{

isAsync = true; (async () => {

try {

await performFactCheck(request.text);

// Status/data sent within performFactCheck

} catch (error) {

// Initial key check errors might land here

console.error("Error in factCheckSelection flow:",

error);

const errorMsg = `Fact-check Error: ${error.message}`; updatePopupStatus(errorMsg, true, false);

// No need to send error data again here,

performFactCheck does it if it runs

}

})();

}

// NOTE: No 'saveAPIKeys' handler needed here, options.js handles saving directly to storage.

return isAsync; // Return true if any async operation was started

});

// --- Context Menu Setup ---

chrome.runtime.onInstalled.addListener(() => { chrome.contextMenus.create({

id: "factCheckSelectedText",

title: "Fact-Check Selection",

contexts: ["selection"] // Show only when text is selected

});

console.log("Fact-Check context menu created.");

// Load keys once on startup/install just to check they exist loadAPIKeys().then(({ youtubeKey, geminiKey }) => {

if (!youtubeKey || !geminiKey) {

console.warn("One or more API keys are missing. Please set them in the extension options.");

// Maybe open options page automatically on first install?

// chrome.runtime.openOptionsPage();

}

});

});

// Listener for context menu clicks

chrome.contextMenus.onClicked.addListener((info, tab) => { if (info.menuItemId === "factCheckSelectedText" &&

info.selectionText) {

const selection = info.selectionText.trim();

if (selection.length > 500) { // Add a length limit for fact-checking

const errorMsg = "Selected text too long for fact-checking (max 500 chars).";

updatePopupStatus(errorMsg, true, false);

sendDataToPopup("displayFactCheck", null, errorMsg); return;

}

if (selection.length === 0) {

const errorMsg = "No text selected for fact-checking."; updatePopupStatus(errorMsg, true, false);

sendDataToPopup("displayFactCheck", null, errorMsg);

return;

}

console.log("Fact-check requested for:", selection);

updatePopupStatus("Starting fact-check...", false, true); // Update status immediately

// Send to our own background script handler chrome.runtime.sendMessage({

action: "factCheckSelection", text: selection

}).catch(err => console.error("Error sending fact-check message:", err));

}

});

console.log("Background service worker started/restarted.");

* **Globals:** YOUTUBE\_API\_KEY, GEMINI\_API\_KEY store keys loaded from storage.
* **loadAPIKeys**: Asynchronously retrieves API keys from chrome.storage.local.
* **updatePopupStatus, sendDataToPopup**: Utility functions for communicating with the popup UI.
* **getVideoIdFromTab**: Uses chrome.scripting.executeScript to run code on the page and extract the 'v' parameter (video ID) from the URL.
* **fetchTranscriptFromPage**: Attempts to fetch the transcript by parsing the ytInitialPlayerResponse from the video page's HTML. Includes logic to find appropriate language tracks. Crucially, it contains a fallback.
* **fetchAvailableTranscriptLangsAPI**: *Fallback function*. Uses the YouTube Data API (captions.list) to list available languages if page parsing fails. Note: This *lists* languages, doesn't fetch the actual transcript text via API in this implementation.
* **fetchAndAnalyzeComments**: Fetches comments using YouTube Data API (handles pagination). Batches comments and sends them to the Gemini API (Flash model) for sentiment analysis using a structured prompt. Parses the JSON response.
* **performFactCheck**: Called by the context menu listener. Sends selected text to the Gemini API (Pro model) with a prompt requesting a structured JSON output (verdict, confidence, explanation, sources). Parses the JSON response.

# Event Listeners:

* + chrome.runtime.onMessage: Handles messages from the popup (getTranscript, analyzeComments, factCheckSelection) and routes them to the appropriate async function.
  + chrome.runtime.onInstalled: Creates the context menu item on installation and performs an initial check for API keys.
  + chrome.contextMenus.onClicked: Listens for clicks on the context menu item and triggers the fact-checking process by sending a message to its own message listener.

# popup.html & popup.js (User Interface) popup.html

<!DOCTYPE html>

<html>

<head>

<title>YouTube Analyzer</title>

<link rel="stylesheet" href="popup.css">

<meta charset="UTF-8">

</head>

<body>

<h1>YouTube Context Analyzer</h1>

<div id="processing-indicator" style="display: none; color: orange;">Processing...</div>

<button id="getTranscriptBtn">Get Transcript</button>

<div id="transcriptResult" class="result-area"> Transcript will appear here...

</div>

<hr>

<button id="analyzeCommentsBtn">Analyze Comments</button>

<div id="commentResult" class="result-area"> Comment analysis will appear here...

</div>

<hr>

<div id="factCheckInfo" class="info-area">

Select text in the transcript or page, then right-click and choose "Fact-Check Selection".

</div>

<div id="factCheckResult" class="result-area"> Fact-check results will appear here...

</div>

<div id="status" class="status-area"></div>

<script src="popup.js"></script>

</body>

</html>

* Basic HTML structure with buttons for actions (getTranscriptBtn, analyzeCommentsBtn).
* Divs to display results (transcriptResult, commentResult, factCheckResult).
* A status area (status) and processing indicator.
* Links to popup.css and popup.js.

# popup.js

const transcriptBtn = document.getElementById('getTranscriptBtn'); const commentsBtn = document.getElementById('analyzeCommentsBtn');

const transcriptResultDiv = document.getElementById('transcriptResult'); const commentResultDiv = document.getElementById('commentResult');

const factCheckResultDiv = document.getElementById('factCheckResult'); const statusDiv = document.getElementById('status');

const processingIndicator =

document.getElementById('processing-indicator'); let isProcessing = false;

function setProcessing(processing) { isProcessing = processing;

processingIndicator.style.display = processing ? 'block' : 'none'; transcriptBtn.disabled = processing;

commentsBtn.disabled = processing;

}

function updateStatus(message, isError = false) { statusDiv.textContent = message;

statusDiv.style.color = isError ? 'red' : '#333'; // Use a default dark color for non-errors

console.log("Status:", message); if (isError) {

setProcessing(false); // Stop processing on error

}

}

function displayFormattedTranscript(transcriptText) {

// Simple text display for now

transcriptResultDiv.textContent = transcriptText;

}

function displayFormattedCommentAnalysis(analysisData) { let html = `<strong>Total Comments Analyzed:</strong>

${analysisData.totalAnalyzed}<br>`;

if (analysisData.totalFetched !== analysisData.totalAnalyzed) { html += `(Fetched ${analysisData.totalFetched}, analyzed

subset)<br>`;

}

html += `<strong>Sentiment:</strong><br>`;

html += ` Positive: ${analysisData.sentiment.positive}<br>`; html += ` Negative: ${analysisData.sentiment.negative}<br>`; html += ` Neutral: ${analysisData.sentiment.neutral}<br>`;

if (analysisData.sampleAnalyzedComments &&

analysisData.sampleAnalyzedComments.length > 0) {

html += `<br><strong>Sample Comments:</strong><br>`;

analysisData.sampleAnalyzedComments.slice(0, 5).forEach(c => {

// Show first 5

let sentimentEmoji = '😐';

if (c.sentiment === 'positive') sentimentEmoji = '😊';

else if (c.sentiment === 'negative') sentimentEmoji = '😞'; html += `<div style="font-size: 0.9em; margin-bottom: 3px;

border-bottom: 1px solid #eee; padding-bottom: 2px;">${sentimentEmoji} (${c.sentiment}): ${c.text.substring(0, 100)}${c.text.length > 100 ? '...' : ''}</div>`;

});

}

commentResultDiv.innerHTML = html;

}

function displayFormattedFactCheck(factCheckData) {

let html = `<strong>Verdict:</strong> ${factCheckData.verdict || 'Unknown'}<br>`;

if (factCheckData.confidence !== undefined && factCheckData.confidence !== null) {

html += `<strong>Confidence:</strong>

${(factCheckData.confidence \* 100).toFixed(0)}%<br>`;

}

if (factCheckData.explanation) {

html += `<strong>Explanation:</strong>

${factCheckData.explanation}<br>`;

}

if (factCheckData.sources && factCheckData.sources.length > 0 && factCheckData.sources[0] !== "No specific sources provided") {

html += `<strong>Sources:</strong><ul>`; factCheckData.sources.forEach(source => {

try {

// Try creating a clickable link

const url = new URL(source.startsWith('http') ? source :

`http://${source}`);

html += `<li><a href="${url.href}" target="\_blank">${url.hostname}</a></li>`;

} catch (\_) {

// If it's not a valid URL, just display the text html += `<li>${source}</li>`;

}

});

html += `</ul>`;

} else {

html += `<strong>Sources:</strong> Not provided or unable to extract.<br>`;

}

factCheckResultDiv.innerHTML = html;

}

// --- Button Listeners ---

transcriptBtn.addEventListener('click', () => { if (isProcessing) return;

setProcessing(true);

updateStatus('Requesting transcript...');

transcriptResultDiv.textContent = 'Processing...';

chrome.tabs.query({ active: true, currentWindow: true }, (tabs) => { if (tabs[0]?.id && tabs[0]?.url?.includes("youtube.com/watch"))

{

chrome.runtime.sendMessage({ action: "getTranscript", tabId:

tabs[0].id });

// Response handling is now done via the listener below

} else {

updateStatus("Not a YouTube video page or cannot access tab.", true);

transcriptResultDiv.textContent = "Please navigate to a YouTube video page.";

setProcessing(false);

}

});

});

commentsBtn.addEventListener('click', () => { if (isProcessing) return;

setProcessing(true);

updateStatus('Requesting comment analysis...'); commentResultDiv.textContent = 'Processing...';

chrome.tabs.query({ active: true, currentWindow: true }, (tabs) => { if (tabs[0]?.id && tabs[0]?.url?.includes("youtube.com/watch"))

{

chrome.runtime.sendMessage({ action: "analyzeComments",

tabId: tabs[0].id });

// Response handling is now done via the listener below

} else {

updateStatus("Not a YouTube video page or cannot access tab.", true);

commentResultDiv.textContent = "Please navigate to a YouTube video page.";

setProcessing(false);

}

});

});

// --- Listener for results/status from background script ---

chrome.runtime.onMessage.addListener((request, sender, sendResponse) =>

{

console.log("Popup received message:", request);

if (request.action === "updateStatus") {

updateStatus(request.message, request.isError || false); if (request.isProcessing !== undefined) {

setProcessing(request.isProcessing);

}

} else if (request.action === "displayTranscript") {

setProcessing(false); // Transcript process finished if (request.data) {

displayFormattedTranscript(request.data); updateStatus('Transcript processed.');

} else if (request.error) {

transcriptResultDiv.textContent = `Error: ${request.error}`; updateStatus(`Error fetching transcript: ${request.error}`,

true);

}

} else if (request.action === "displayCommentAnalysis") { setProcessing(false); // Comment analysis finished

if (request.data) {

displayFormattedCommentAnalysis(request.data); updateStatus('Comment analysis complete.');

} else if (request.error) {

commentResultDiv.textContent = `Error: ${request.error}`; updateStatus(`Error analyzing comments: ${request.error}`,

true);

}

} else if (request.action === "displayFactCheck") {

setProcessing(false); // Fact check finished (assuming it was triggered)

if (request.data) {

displayFormattedFactCheck(request.data); updateStatus('Fact-check complete.');

} else if (request.error) {

factCheckResultDiv.textContent = `Error: ${request.error}`; updateStatus(`Fact-check error: ${request.error}`, true);

}

}

// No need to return true or call sendResponse here, background sends separate messages

});

// Initial status

updateStatus("Ready. Ensure API keys are set in options."); setProcessing(false)

* Gets references to HTML elements.
* **setProcessing**: Manages the visual state (disables buttons, shows indicator) during background operations.
* **updateStatus**: Updates the status message div.
* **displayFormatted... functions**: Format the data received from the background script into user-friendly HTML for display in the result divs.
* **Button Event Listeners**: Add listeners to the buttons. When clicked, they check if processing is ongoing, update the status, ensure the active tab is a YouTube video page, and then send the corresponding action message (getTranscript or analyzeComments) to the background script.
* **chrome.runtime.onMessage Listener**: Listens for messages from the background script (updateStatus, displayTranscript, displayCommentAnalysis, displayFactCheck). When a message is received, it calls the appropriate function to update the UI (status, results divs) and manages the processing state.

# content.js (Content Script)

console.log("YouTube Context Analyzer: Content script loaded (or injected).");

// This script primarily exists to be \*injected\* by the background script

// using chrome.scripting.executeScript() to get page-specific data like the video ID.

// It doesn't need much code itself initially, as the functions to execute

// are passed directly in the executeScript call from background.js.

// We \*could\* add listeners here if the background script needed to push

// UI changes \*to\* the page (e.g., highlight text), but let's keep it simple for now.

// Example of how it might listen for commands from background later: chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {

if (request.action === "highlightText") {

console.log("TODO: Highlight text on page:", request.textToHighlight);

// Add DOM manipulation logic here sendResponse({ success: true });

}

});

IGNORE\_WHEN\_COPYING\_START

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487). JavaScript IGNORE\_WHEN\_COPYING\_END

* This script is intentionally minimal in this version.
* Its primary role is to provide an execution context *within the YouTube page* for scripts injected by background.js (specifically, the function within getVideoIdFromTab that reads window.location.search).
* The example message listener shows how it *could* be used for more complex interactions (like highlighting text on the page based on commands from the background), but this is not implemented in the current core functionality.

# options.html & options.js (Configuration)

<!DOCTYPE html>

<html>

<head>

<title>YouTube Context Analyzer Settings</title>

<link rel="stylesheet" href="options.css">

<meta charset="UTF-8">

</head>

<body>

<h1>API Key Settings</h1>

<p>

Enter your API keys below. These are required for the extension to fetch data and perform analysis.

Keys are stored locally in your browser's storage.

</p>

<div class="form-group">

<label for="youtubeApiKey">YouTube Data API v3 Key:</label>

<input type="password" id="youtubeApiKey" name="youtubeApiKey" size="50" placeholder="Enter your YouTube Data API Key">

<p class="help-text">

Needed for comments and listing caption tracks. Get from

<a

href="https://console.cloud.google.com/apis/library/youtube.googleapis.c om" target="\_blank" rel="noopener noreferrer">Google Cloud Console</a>

(Enable YouTube Data API v3). Make sure to restrict its usage.

</p>

</div>

<div class="form-group">

<label for="geminiApiKey">Google AI Gemini API Key:</label>

<input type="password" id="geminiApiKey" name="geminiApiKey" size="50" placeholder="Enter your Gemini API Key">

<p class="help-text">

Needed for fact-checking and sentiment analysis. Get from

<a href="https://aistudio.google.com/app/apikey" target="\_blank" rel="noopener noreferrer">Google AI Studio</a>.

</p>

</div>

<button id="saveBtn">Save Keys</button>

<div id="status"></div>

<script src="options.js"></script>

</body>

</html>

* **options.html**: Would contain HTML form elements (input fields) for the user to enter their YouTube Data API Key and Gemini API Key. It would also include a "Save" button.
* **options.js**: Would contain JavaScript to:
  + Load any previously saved API keys from chrome.storage.local when the options page is opened and populate the input fields.
  + Add an event listener to the "Save" button. When clicked, it would read the values from the input fields and save them to chrome.storage.local using chrome.storage.local.set({ youtubeApiKey: '...', geminiApiKey: '...' }).
  + Provide visual feedback to the user (e.g., "Keys saved!").

const youtubeApiKeyInput = document.getElementById('youtubeApiKey'); const geminiApiKeyInput = document.getElementById('geminiApiKey'); const saveBtn = document.getElementById('saveBtn');

const statusDiv = document.getElementById('status');

// Saves options to chrome.storage.local function saveOptions() {

const youtubeKey = youtubeApiKeyInput.value.trim(); // Trim whitespace const geminiKey = geminiApiKeyInput.value.trim(); // Trim whitespace

// Basic validation (check if fields are empty) if (!youtubeKey || !geminiKey) {

statusDiv.textContent = 'Error: Both API Key fields are required.'; statusDiv.style.color = 'red';

// Clear status message after a few seconds setTimeout(() => {

statusDiv.textContent = '';

}, 4000);

return; // Stop saving if validation fails

}

chrome.storage.local.set(

{ youtubeApiKey: youtubeKey, geminiApiKey: geminiKey }, () => {

// Check for runtime errors during saving if (chrome.runtime.lastError) {

console.error("Error saving keys:", chrome.runtime.lastError); statusDiv.textContent = `Error saving:

${chrome.runtime.lastError.message}`; statusDiv.style.color = 'red';

} else {

console.log("API keys saved.");

statusDiv.textContent = 'API Keys saved successfully!'; statusDiv.style.color = 'green';

}

// Clear status message after a few seconds setTimeout(() => {

statusDiv.textContent = '';

}, 3000);

}

);

}

// Restores input field state using the preferences stored in chrome.storage.

function restoreOptions() {

// Retrieve the keys from storage, using empty strings as defaults if not found

chrome.storage.local.get(

{ youtubeApiKey: '', geminiApiKey: '' }, (items) => {

if (chrome.runtime.lastError) {

console.error("Error loading keys:", chrome.runtime.lastError);

statusDiv.textContent = `Error loading saved keys:

${chrome.runtime.lastError.message}`;

statusDiv.style.color = 'red';

} else {

youtubeApiKeyInput.value = items.youtubeApiKey || ''; // Ensure it's not null/undefined

geminiApiKeyInput.value = items.geminiApiKey || ''; // Ensure it's not null/undefined

console.log("API keys loaded into options page.");

}

}

);

}

// Add event listeners when the DOM content has loaded

document.addEventListener('DOMContentLoaded', restoreOptions); saveBtn.addEventListener('click', saveOptions);

# rules.json (Network Request Rules)

[

{

"id" : 1,

"priority": 1,

"action" : {

"type" : "modifyHeaders", "requestHeaders": [

{ "header": "Sec-Fetch-Site", "operation": "set", "value": "same-origin" }

]

},

"condition" : {

"urlFilter" : "\*://\*.youtube.com/\*", "resourceTypes" : ["xmlhttprequest"]

}

}

]

IGNORE\_WHEN\_COPYING\_START

content\_copy download Use code [with caution](https://support.google.com/legal/answer/13505487). Json

IGNORE\_WHEN\_COPYING\_END

* This file defines rules for the declarativeNetRequest API.
* **id, priority**: Rule identification and execution order.
* **action**: Specifies modifying request headers (modifyHeaders). It targets the Sec-Fetch-Site header and sets its value to same-origin. This *might* influence how YouTube servers or intermediate proxies handle certain background requests (xmlhttprequest), potentially making some data fetching easier or more reliable in specific scenarios, but its exact effect can be complex and depend on YouTube's infrastructure.
* **condition**: Applies this rule only to xmlhttprequest resource types originating from YouTube domains (\*://\*.youtube.com/\*).

# References

* Chrome Developer Docs: <https://developer.chrome.com/docs/extensions/>
* YouTube Data API V3 Overview: <https://developers.google.com/youtube/v3/getting-started>
* Google AI for Developers (Gemini API): <https://ai.google.dev/>
* MDN Web Docs (Fetch API, DOM, etc.): <https://developer.mozilla.org/>