Project Overview & Goal

This project is a Global Web Footprint Analyzer, a sophisticated, single-page web application designed to visualize a website's global infrastructure. It's built on a serverless architecture using Google Cloud Platform and a modern JavaScript frontend.

The goal is to create a multi-faceted analysis toolkit. A user enters a URL, and the application provides several different interactive visualizations (maps, heatmaps, and charts) that answer key questions about the website's performance, structure, and loading process.

The Tech Stack:

Backend: A Python Google Cloud Function (2nd Gen) that acts as an orchestrator. It uses Selenium with a headless Chrome browser to perform a deep analysis of the target URL.

Database: Google Cloud Firestore is used as a real-time database to stream the analysis results from the backend to the frontend.

Frontend: A static web application built with plain HTML, CSS, and modern JavaScript (ES Modules). It's deployed on Firebase Hosting.

Visualization Libraries: The frontend uses Leaflet.js for mapping, Leaflet.heat for heatmaps, D3.js for advanced charts (Treemap, Radial Clock), and Chart.js for a waterfall chart.

Current Status: What is Working Properly

The backend is fully functional and has been the most stable part of the project.

Working Backend (main.py):

Orchestration: The Cloud Function (analyze-web-footprint) successfully receives a URL via an HTTP POST request.

Parallel Processing: It correctly launches two analysis tasks in parallel for speed:

Task A (Supply Chain & Journey): It uses Selenium with performance logging enabled to get the fully rendered HTML and a detailed network timeline. It reliably finds a large number of assets (e.g., 138+ assets for theguardian.com). It performs GeoIP lookups on all server IPs.

Task B (DNS Latency): It successfully measures the DNS lookup latency for the target domain against ~10 global public DNS resolvers.

Real-Time Data Streaming: As the backend gathers data, it correctly writes the results into a new document in the analyses collection in Firestore. This includes a detailed assets array (with location and timing info) and a dns_latency_results array. This part is working flawlessly.

The Core Problem: What is Currently Broken

The entire problem is now on the frontend. After the last major upgrade to the script.js file (v13), the application has become completely unresponsive.

The Main Bug:

The "Analyze" button is dead. When a user enters a URL and clicks the "Analyze" button, absolutely nothing happens.

The browser's developer console shows no errors on page load or on button click.

The browser's Network tab shows that no network request is being made to the backend Cloud Function.

My Diagnosis:
This is a critical JavaScript regression. The addEventListener on the "Analyze" button is likely failing silently, or the initializeAppLogic() function is not being called correctly. The problem occurs before the fetch() call is ever attempted. The HTML and CSS seem fine, so the bug is almost certainly located within the script.js file's initialization or event handling logic.

Desired Features & Functionality (What We Are Trying to Build)
The goal is to fix the bug and fully implement the six-view visualization dashboard. The frontend should subscribe to the Firestore document in real-time and render the correct visualization based on the selected tabs.

Main Tab 1: Loading Journey (This has 4 sub-views)

Sub-view 1: Classic View:

Map: Display a pin for the user's location and a pin for every unique server location found in the assets array. Draw a static line from the user to each server.

Side Panel: Display a list of all assets, grouped by their server location (city, country), exactly like the user's screenshot. When a user hovers over a location group in the panel, the corresponding line on the map should highlight in a bright color (e.g., blue).

Sub-view 2: Global Waterfall:

Map: Display pins for the user and all servers.

Side Panel: Use Chart.js to render a horizontal bar chart (a waterfall chart). Each bar represents an asset, positioned on a timeline based on its load_start_time. When a user hovers over a bar in the chart, the pin for that asset's server should be highlighted on the map.

Sub-view 3: Radial Clock:

This view should hide the map and side panel and display a single, full-width D3.js visualization. It should draw a circular "clock" representing the total load time, with colored arcs for each asset showing when it was downloading.

Sub-view 4: Content Treemap:

This view should also hide the map and side panel. It should use D3.js to render a treemap where the data is grouped by server location and then by asset. The size of each rectangle should represent the asset's file size or load duration.

Main Tab 2: Infrastructure Density

Map: Use the lat and lon from all assets to render a vibrant and clearly visible heatmap showing the "center of gravity" of the website's infrastructure.

Side Panel: Display a simple summary of the findings.

Main Tab 3: DNS Performance

Map: Use the known coordinates of the global DNS resolvers and the latency_ms data to render a heatmap showing which parts of the global DNS network are fastest from the backend's perspective. The heatmap should be vibrant (green for fast, red for slow).

Side Panel: Display a list of the DNS

Current Status: What is Working Properly
The backend is fully functional and has been the most stable part of the project.

Working Backend (main.py):

Orchestration: The Cloud Function (analyze-web-footprint) successfully receives a URL via an HTTP POST request.

Parallel Processing: It correctly launches two analysis tasks in parallel for speed:

Task A (Supply Chain & Journey): It uses Selenium with performance logging enabled to get the fully rendered HTML and a detailed network timeline. It reliably finds a large number of assets (e.g., 138+ assets for theguardian.com). It performs GeoIP lookups on all server IPs.

Task B (DNS Latency): It successfully measures the DNS lookup latency for the target domain against ~10 global public DNS resolvers.

Real-Time Data Streaming: As the backend gathers data, it correctly writes the results into a new document in the analyses collection in Firestore. This includes a detailed assets array (with location and timing info) and a dns_latency_results array. This part is working flawlessly.

Firestore Data Structure & Frontend Access
This section describes exactly what data the backend saves to Firestore and how the frontend is designed to access it.

Document Structure:
For each analysis, the backend creates a single new document in a top-level collection named analyses. The ID of this document is a unique UUID (e.g., a1b2c3d4-...) which is returned to the frontend when the analysis starts.

A typical document has the following top-level fields:

```
{
"status": "starting", // -> "running" -> "completed"
"target_url": "https://www.example.com",
"status_supply_chain": "found_138_assets", // -> "completed"
"status_dns_latency": "running", // -> "completed"
"assets": [ /* Array of asset objects / ],
```

"dns_latency_results": [/ Array of DNS result objects */]
}

1. The assets Array:
   This is the primary data for most visualizations. It is an array of objects, where each object represents a single asset found on the page. The backend adds new assets to this array one by one as they are discovered.

Example of a single asset object:

```
{
"url": "https://i.guim.co.uk/img/...",
"domain": "i.guim.co.uk",
"type": "Image/Media",
"ip": "151.101.157.111",
"lat": 13.0837,
"lon": 80.2702,
"city": "Chennai",
"country": "India",
"isp": "Fastly",
"load_start_time": 1696440005123.456
}
```

2. The dns_latency_results Array:
   This is the data for the DNS Performance heatmap. It is an array of objects, where each object represents the result from one global DNS server.

Example of a single DNS result object:

```
{
"resolver_name": "Cloudflare (USA)",
"latency_ms": 25.43
}
```