ChatGPT

# EUDR Due Diligence Statement (DDS) Generator Web App – Product Requirements Document

## Overview

The **EUDR DDS Generator** is a web application for creating Due Diligence Statements (DDS) in compliance with the EU Deforestation Regulation (EUDR). The application will guide operators or traders through a form to input required information (e.g. company details, product details, and origin geolocation) and will produce a standardized three-page DDS PDF report. This tool is intended to streamline compliance by ensuring all necessary data (such as HS codes, product characteristics, and geolocation of origin) are captured and formatted correctly [1] [2]. The app will be developed on Replit using **Node.js** (JavaScript) for both the server-side and front-end logic.

## Objectives and Scope

**Objectives:**
- Provide a user-friendly web-based form to collect all required DDS information in one place.
- Validate inputs (including verifying GeoJSON geometry data) to ensure accuracy and completeness of the DDS.
- Automatically generate a DDS PDF document (3 pages) that includes both dynamic user-provided content and static reference content (methodology and flowcharts).
- Enable users to download the generated DDS PDF for submission or record-keeping.
- Optionally maintain a list (dashboard) of previously generated DDS entries during a session for user reference.

**In Scope (Features):**
- Form fields for operator/trader information, product details (HS code, description, quantity, etc.), and production plot selection.
- GeoJSON polygon input (selection from list or file import) with validation and visual review.
- PDF generation module composing the final statement (with 3 specific pages).
- Embedding of static images (methodology diagram and flowcharts) into the PDF's page 2 and 3.
- Download functionality for the PDF.
- Front-end built on web technologies (HTML/CSS/JS) served by a Node.js backend on Replit.

**Out of Scope:**
- Actual submission to the EU's DDS registry or external APIs (the app is for generation only, not for filing the DDS online).
- Real-time risk analysis or map-based deforestation checks beyond embedding provided geo-information (the app assumes spatial analysis is done externally, and only handles input of results).
- Persistent database of DDS entries (the app may maintain a list in memory for the session, but long-term storage or multi-user account management is not included unless Replit's environment persistence is considered).

## User Flow

1. **Access the App:** The user (operator or trader) navigates to the web app URL on Replit. They are presented with a dashboard or home screen where they can create a new Due Diligence Statement.
2. **Start New Statement:** The user clicks "Create New DDS" (or similar). This opens the DDS input form.
3. **Enter Operator/Trader Info:** In the form's first section, the user inputs the operator or trader name, the registered address, and the place of activity (e.g. city and country where operations occur). These fields may be pre-filled from a previous session or template if available, or entered manually each time. Validation ensures none of these fields are left empty.
4. **Select Product HS Code:** The user selects the product's HS (Harmonized System) code from a dropdown list. The list can be pre-populated with relevant HS codes or allow searching by code or product name. Selecting an HS code may auto-fill the general product category (for user confirmation). *Example:* HS Code 1511 for palm oil. This step aligns with EUDR requirements to specify product type via HS code [1].
5. **Enter Product Details:** The user fills in specific product details: net mass (in kg), volume (in appropriate units, if applicable), the scientific name of the commodity (chosen from a dropdown of species names, to avoid mis-spelling), and the common name of the product. These fields ensure that both scientific and common nomenclature are captured. Quantity (net mass/volume) is required as per EUDR guidelines [1]. Form validations include numeric checks for mass and volume.
6. **Add Production Plot (Geolocation):** The user selects the origin production plot (the land area where the product was produced). There are two options:
7. **Select from Existing List:** A dropdown or list is provided containing pre-loaded **GeoJSON** polygons representing known production plots (from prior spatial analysis data in the system). These could be labeled by farm name or ID.
8. **Import GeoJSON:** If the desired plot is not in the list, the user can upload a GeoJSON file or paste GeoJSON text. The app will parse this input.
9. Upon selection or upload, the app validates that the GeoJSON is correctly formatted and contains at least one polygon geometry (with coordinates). If the GeoJSON is a FeatureCollection, it may allow selecting which polygon feature to use if multiple are present. Only polygon (or multi-polygon) geometries are acceptable (points/lines would be rejected). Validation errors (e.g., "No valid polygon found in GeoJSON") are shown for user correction. This supports the EUDR requirement for providing geolocation of origins, typically via coordinates or polygons [2].
10. **Review Geospatial Data:** Once a polygon is selected/imported, the user can review it on a mini map embedded in the form. The front-end uses a mapping library (for example, **Leaflet.js**) to display the polygon on an interactive map (with a basic OpenStreetMap layer or similar). This visual check lets the user confirm that the area is correct and within expected boundaries. If something is wrong (e.g., the polygon is in the wrong location or shape), the user can remove it and re-import or select a different one.
11. **Form Submission:** After all fields are filled and the geo-data is confirmed, the user clicks **"Generate DDS PDF"** (submission button). The app may prompt a final confirmation showing a summary of entered data. Once confirmed, the data is sent to the server (Node.js backend) for PDF generation.
12. **PDF Generation:** The backend takes the submitted form data and creates a DDS PDF document in the required format. (Details of PDF content structure are in the next section.) This process may take a moment as the PDF is being compiled with text and images.
13. **View/Download PDF:** When generation is complete, the user is presented with a link or button to download the PDF file to their device. The app might also display the PDF in an embedded PDF

viewer for a quick look, if feasible. The filename of the PDF could be auto-generated (e.g., `DDS_<OperatorName>_<Product>_<Date>.pdf`). The user saves the file to their local system.

14. **DDS List/Dashboard:** After creation, the new DDS entry is added to an in-app list of statements (showing key info like statement ID, product, date). The main dashboard of the app will list all DDS statements created in the current session (and possibly allow downloading them again). For example, a table with columns: Statement ID/Name, Date, Product, and a download link. This corresponds to the "List of DDS created" functionality [3]. (Note: This list is not persisted long-term without a database; it exists for user convenience in the session.)
15. **Repeat or Edit:** The user can create additional statements by repeating the process. They may also choose to edit an existing statement's inputs if a mistake is noticed, then regenerate the PDF. (Edits would be done by preloading the form with the selected entry's data, allowing changes, and re-generating a PDF.)

Throughout the flow, the UI will provide guidance and validation messages to ensure the user enters correct information. For example, if a required field is missing or an invalid format is detected, the form will highlight it and prevent submission until fixed.

# Key Functional Components

## 1. Web Input Form Module (Front-End)

This component is the interactive form where users input all DDS data. It is composed of several sections:

- **Operator/Trader Information Section:** Fields for Operator/Trader Name, Address, and Place of Activity. These are simple text inputs. The UI might group these under a heading like "Operator Details". If the application is extended to support multiple operators, a search or selection could be provided, but initially it's manual input. All fields in this section are mandatory.

- **Product Details Section:** Fields and controls for product information:

- *HS Code:* A dropdown list of HS codes (with descriptions). This field is required. The list can be hardcoded for the relevant commodities or loaded from a static JSON. Auto-complete search can be enabled for usability due to the length of HS code lists.
- *Net Mass:* Numeric input (accepts only positive numbers). Possibly include a unit label (kg) next to it.
- *Volume:* Numeric input with unit (e.g., cubic meters or liters, depending on commodity) – this might not always apply; if not applicable, user can enter 0 or it can be optional depending on product.
- *Scientific Name:* A dropdown of possible scientific names corresponding to the product. For example, if palm oil is selected as product, the scientific name could be *Elaeis guineensis*. This ensures correct Latin names in the statement. If the product list is limited, this could be dynamically filtered based on HS code selection.
- *Common Name:* A text input for the common name of the product (e.g., "Palm Oil", "Robusta Coffee"). If HS code selection already implies a product, this might be pre-filled or suggested.

- All product fields are validated: mass and volume must be numbers (and within sensible ranges), and the names should not contain prohibited characters. These correspond to the "characteristics such as HS code, product description, and quantity" required for a DDS [1].

- **Production Plot Section:** Controls for selecting or uploading the GeoJSON polygon:

- *Plot Selection Dropdown:* Lists available polygons by name/ID. (This list could be populated from a known dataset of "Spatial Analysis" results provided to the app. Each entry internally has a GeoJSON geometry associated.)
- *GeoJSON Upload:* An upload button and/or text area for pasting JSON. The app accepts a `.geojson` file or a `.json` file following GeoJSON structure. When a file is uploaded, the front-end reads it (using FileReader API) and performs a quick client-side validation (checking for `"geometry": { "type": "Polygon" ... }` or MultiPolygon).
- *Import Confirmation:* After reading the GeoJSON, the app might display the name or some stats (e.g., area or bounding box) of the polygon. If it's valid, the user can proceed; if not, an error is shown (e.g., "Invalid GeoJSON format or no polygon geometry found").
- *Map Preview:* A small map widget that appears after selecting/importing. Using a library like **Leaflet**, the app will load the polygon onto the map. The user can pan/zoom to verify the location. This map is read-only (no drawing, unless we allow user to draw if no geojson; but out of scope since we rely on provided geojson).
- *Geometry Validation:* The system double-checks the GeoJSON on the server-side as well (using a Node.js library or custom logic) to prevent malformed data. It ensures the coordinates array is present and non-empty. If multiple features are present, the system uses the first or all (depending on design; likely one DDS per plot, so first polygon is used if multiple given, or we can require one at a time).

- Each selected/imported plot is linked to the DDS entry. The user can add **multiple plots** only if the regulation or use-case allows multiple origin areas in one statement (the EUDR allows multiple coordinates, but for simplicity of this tool, we assume one plot per DDS). If multiple were needed, the UI would allow adding multiple GeoJSON files and listing them; otherwise, just one.

- **Submission & Review:** At the bottom of the form, a **"Generate PDF"** button triggers the form submission. Before submission, the front-end could display a summary of entered data (especially if the form is long, a quick review step improves accuracy). The form data is packaged into a JSON object and sent via HTTP POST to the Node.js server (if using a traditional form post, an Express endpoint will receive it; if using an SPA approach, an AJAX call will send the data).

- **Usability & Design:** The front-end will be built with a clean, user-friendly design. Using a CSS framework like **Bootstrap** (or Replit's default styling) can ensure responsive layout and consistent styling of form controls. Headings and field groupings will make the form easy to follow. Field placeholders and help text (e.g., examples of format) will guide users. Short validation messages will appear near fields as needed (using client-side JS for instant feedback).

- **Multi-Step Enhancement (if any):** If the form is very long, we may implement it as a multi-step wizard (e.g., Step 1: Operator Info, Step 2: Product Info, Step 3: Plot Info, Step 4: Confirmation) rather than one long page. Navigation buttons (Next/Back) would control the steps. This aligns with the steps outlined in the internal guidance [3]. However, for the initial version, a single-page form is simpler, with logical sections clearly delineated.

## 2. GeoJSON Validation & Mapping Module

On the back-end, a **GeoJSON validation component** will handle the uploaded or selected geospatial data:

- **Parsing & Validation:** Using a Node package (such as `@mapbox/geojsonhint` or `ajv` with a GeoJSON schema), the server will parse the GeoJSON string. It verifies that the JSON is well-formed, and that it conforms to GeoJSON structure (e.g., has `type: "Feature"` or `"FeatureCollection"`, with a valid `geometry`). It specifically checks that the geometry type is Polygon or MultiPolygon. If validation fails, the server responds with an error message, which the front-end will show to the user (preventing PDF generation until resolved).
- **Geometry Processing:** If needed, the server could compute basic info from the polygon (such as area in hectares or coordinates of centroid) to include in the DDS content (e.g., "Production Plot Area: X ha in Lat/Long bounds"). However, such calculations are optional. The main requirement is to link the polygon with the statement, not to do full GIS analysis.
- **Storage of GeoJSON:** The app will hold the GeoJSON data in memory (as part of the in-progress DDS entry) or save it to a temporary file if needed. The entire polygon data does not necessarily go into the PDF text (perhaps just a reference or an ID), but the existence of the polygon is crucial for compliance evidence. We might include a truncated form of the coordinates or a map snapshot in the PDF (discussed in the PDF section below).
- **Map Preview (Front-end):** As mentioned, the front-end uses a mapping library to display the polygon. This module ensures that the map component is initialized with the polygon data after upload/selection. The map script will likely convert the GeoJSON to a JavaScript object and add it to a Leaflet map layer. (Leaflet can directly take GeoJSON and handle display and bounding box zooming.)

This component ensures that only **valid geospatial data** is used and gives the user confidence that the correct plot is attached to the DDS.

## 3. PDF Generation Module (Node.js Backend)

Once the form data is validated, the **PDF Generation module** creates a **3-page PDF** document according to the specified format. The content of each page is as follows:

- **Page 1: Dynamic DDS Content** – This page will be generated based on the form inputs. It will typically include:
- **Title/Heading:** e.g., "Due Diligence Statement" and possibly a reference number or date.
- **Operator/Trader Info:** Name, address, place of activity of the operator/trader. Presented in a section labeled "Operator/Trader Details."
- **Product Information:** HS code and product description (common name) – possibly in one line or a small table; Quantity of product (net mass/volume); Scientific name of the commodity. These can be listed in a "Product Details" section.
- **Production Plot Info:** A description or identifier of the selected plot. If the plot has a name/ID from the list, use that (e.g., "Production Plot: Farm #123, Province X"). If it was imported, possibly include a truncated coordinate or a custom label ("Production Plot: [GeoJSON Import]"). Optionally, the page might include a small static map image of the polygon location. If a map image is desired, the system could leverage a map snapshot service or HTML canvas to draw the polygon on a map and include that image. However, this adds complexity; a simpler approach is to list coordinates or an official plot ID.

- **EUDR Statement Text:** Optionally, Page 1 can include some static text that states the operator's assertion of compliance (e.g., "The operator hereby declares that due diligence has been carried out for the above product in accordance with EU Regulation … and that the product is deforestation-free and legal."). This mimics the formal statement that might be required.

- **Layout:** Page 1 will be formatted in a clear, structured way. We may use a template with fixed positions or a simple flowing document. For example, it could be structured with labels and values in a two-column layout (label on left, value on right) for readability.

- **Page 2: Static Methodology Diagram** – This page contains an image illustrating the **general methodology of EUDR compliance risk assessment**. This is a full-page static content page. We will embed the provided PNG diagram (filename starting `75bdb62d...png`) which outlines the steps/criteria for EUDR compliance verification (e.g., checking deforestation cut-off date, land legality, etc.). The image will be inserted to fit the page. A brief caption or title might be placed above the image (such as "EUDR Compliance Risk Assessment – Methodology Overview") to contextualize it, but the bulk is the image itself. The image ensures the DDS includes reference to how the risk assessment is done in general.

*Example of the static methodology diagram to be included on Page 2 of the PDF (depicting EUDR compliance verification steps). This diagram will be embedded as a full-page image in the DDS report.*

- **Page 3: Static Flowchart Diagrams** – This page will contain **two static flowcharts** side by side or one over the other (depending on their dimensions), drawn from the provided files "Draft_FlowChart LCC_English.pdf" and "FlowChart_AnalisaRisiko_Rev2_ENGVER.pdf." These flowcharts likely illustrate internal processes for Land Cover Change monitoring and Risk Analysis. We will extract these flowcharts (as images) and embed them into page 3. Tentatively, we can place one flowchart on the left and one on the right if they are half-page each, or one on top of the other if they are tall. If needed, page 3 can be formatted in landscape orientation to accommodate them side by side. Each flowchart might have a small title or caption for clarity (e.g., "Land Cover Change Monitoring Workflow" and "Risk Analysis & Mitigation Workflow").
- To embed the flowcharts, since they are originally PDFs, we will convert them to image format (PNG or JPEG) using a tool or library (this conversion can be done offline and the images included in the app's assets). The static images ensure that the content of those PDFs (flowchart visuals and any text in them) appear in the DDS.

- Page 3 thus serves as an **appendix page** with reference workflows supporting the due diligence process. It is entirely static content in terms of generation (no user input affects page 3, aside from the possibility of adding a reference note like "See company internal procedures below." if needed).

- **PDF Compilation:** We will use a Node.js PDF generation library to create the above pages and assemble them into one PDF. There are a few approaches to achieve this:
  **Option A – HTML Template + Puppeteer:** We can design HTML/CSS templates for each page (or a single HTML with page breaks) and use **Puppeteer** (Headless Chrome) to render it to PDF. Puppeteer excels at preserving complex layouts and can easily include images and styled text by leveraging web technologies [4] [5]. For example, we create an HTML file for page 1 with placeholders for form data, and separate HTML (or same with CSS page-breaks) for pages 2 and 3 containing the images. Puppeteer then prints this to a PDF, giving precise control over styling (fonts, positioning) via CSS. This method requires the Replit environment to handle the Puppeteer library and a headless

browser, which is feasible but somewhat heavy.

**Option B – PDF Libraries (pdf-lib or jsPDF):** We can use a pure JavaScript library like **pdf-lib** or **jsPDF** to programmatically generate the PDF. These libraries allow adding text, images, and shapes to PDF pages directly in code [5]. For instance, with pdf-lib we would create a new PDF document, add three pages, and for each page: draw text at coordinates for page 1 content, and embed images for pages 2 and 3. We have to manage coordinates and styling manually (e.g., calculating where to put each field's text on page 1). This gives more control in Node without needing a browser. JsPDF similarly could be used (often on the client side, but can run in Node with a canvas emulator).

**Option C – PDFMake (JSON template):** Another approach is to define the PDF content in a declarative JSON (using PDFMake). However, given the mix of text and full-page images, PDFMake or PDFKit would also work but offer no major advantage over pdf-lib for this case.

- We will likely choose **pdf-lib** for its ability to easily embed images and text. It supports embedding PNG/JPEG images and drawing them at specified sizes [6]. This is important for placing the provided static diagrams into the PDF with correct scaling.

- **Static Content Embedding:** Page 2 and 3 require embedding static images (PNG/JPEG) into the PDF. With the chosen PDF method, we will load these image files (the methodology PNG for page 2, and the two flowchart images for page 3) at runtime. For example, using pdf-lib, we can read the image file as bytes and use `pdfDoc.embedPng()` or `embedJpg()`, then `page.drawImage()` to place it covering the full page [6]. If using Puppeteer/HTML, we will simply reference the image files in `<img>` tags and ensure the CSS sizes them to fit the page. We must ensure the images have sufficient resolution for print; the provided files should be of adequate quality (if not, we might request higher-res versions). Each image will be scaled to fit within the page margins (likely full-bleed or near full-bleed on an A4 page).

- **Page Size and Format:** We will use standard A4 pages (210x297 mm) for the PDF. Page 2 and 3 images might be oriented as needed (if an image is landscape, we can rotate it or center it with whitespace). The PDF generation library will handle pagination and image placement.

- **Dynamic Content Formatting:** For page 1 text, the module will format the text cleanly. Likely using a combination of font sizing and styles (e.g., bold for labels). If using html+Puppeteer, we'll create a nicely formatted HTML (maybe a template using a templating engine like EJS to inject values). If using pdf-lib/jsPDF, we'll calculate positions for each line of text. Simpler is HTML template conversion. The text will be kept concise to fit one page. If some fields are long (e.g., address could be multiple lines), the template should accommodate wrapping or adjust font size.

- **Testing PDF Output:** We will test the PDF generation with sample data to ensure:

- All user inputs appear correctly on page 1 (no truncation or overlap, proper labels).
- Page 2 displays the methodology image clearly (readable text in the image, etc.).
- Page 3 shows both flowcharts clearly (might require adjusting their scale).

- The PDF opens correctly in common PDF readers and is not too large in file size (should be reasonably small, ideally under a few MB).

- **Performance Considerations:** Generating a PDF should be quick (a second or two). If Puppeteer is used, the first invocation might be slower due to launching headless Chrome, but subsequent ones are faster. On Replit, we must ensure enough memory/CPU for this. If performance is an issue, using pdf-lib (which is lighter) may be preferable.

### 4. Download & Storage Component

After PDF generation, the app must handle delivering the file to the user and managing any record of created statements:

- **File Download:** The server will send the generated PDF file back to the client. If using a normal form post, the response can be a PDF file with proper headers (`application/pdf`) forcing a download. If using an AJAX approach, we might send back a URL or binary data. A common solution is to save the PDF on the server (in a temp directory or memory) and then provide the client with a download link. Since Replit has a filesystem, we can save the PDF as a file (e.g., in a `tmp/` folder) and then the front-end can fetch that file via a known route (like `/download/statement123.pdf`). Alternatively, generate a Blob on the client (if using jsPDF client-side) and use the browser's download API. Given our approach is server-side generation, we'll likely do the former.
- The download button could simply be an `<a>` tag pointing to the file route, or the backend could immediately respond with the PDF data stream. We will ensure the user experience is smooth (perhaps showing a "Your document is ready, downloading..." message).

- **DDS List (Session Storage):** The application will keep track of generated DDS entries during the user's session. This can be in a simple in-memory array or an ephemeral JSON file. Each entry can store: an ID, timestamp, operator name, product name/HS code, and maybe the filename or path of the PDF. This allows the app to display the "List of DDS created" on the dashboard with basic info [3]. The user can click an entry to download the PDF again (the app either regenerates it from stored data or stores the PDF file itself). Given no database, we might rely on keeping the form data in memory and regenerate PDF on demand if the user wants to re-download, to avoid storing many binary files.

- Note: Because Replit instances may restart, this list is not persistent long-term. If persistence is required, integrating a lightweight database or using Replit's storage would be needed, but that's beyond initial scope.

- **Error Handling:** If PDF generation fails for some reason (e.g., corrupted image file, or an unhandled edge case), the user should receive an error message rather than a hang. The app will catch errors in the generation process and inform the user ("Failed to generate PDF, please try again or contact support."). Also, if the user tries to download an old link that's expired (file not on server), the app will handle that by regenerating or prompting a regeneration.

- **Security:** The download route will be simple and within the app; since this is not a multi-user system with private data, we do not need complex authentication. However, we ensure that arbitrary file paths cannot be requested (the app will only serve known generated PDFs to prevent any file exposure vulnerability).

## Data Structure & Models

**DDS Data Object:** The core data structure is the DDS form input and its associated geolocation. It can be represented in JSON as follows (keys and example values):

```
{
  "ddsId": "DDS-001",           // Unique ID for the statement (could be
generated, e.g., sequence or UUID)
  "operatorName": "ABC Trading Ltd.",
  "operatorAddress": "123 Palm Road, Jakarta, Indonesia",
  "placeOfActivity": "Jakarta, Indonesia",
  "productHSCode": "1511.90",   // e.g., HS code for palm oil
  "productCommonName": "Crude Palm Oil",
  "productScientificName": "Elaeis guineensis",
  "netMass": 25000,             // in kilograms
  "volume": 28.5,               // in cubic meters (example)
  "productionPlotId": "Plot-45",  // ID or name if selected from list (could be
null if custom import)
  "productionPlotGeoJSON": {   // The GeoJSON feature or geometry object
    "type": "Feature",
    "properties": { "name": "Plot-45" },
    "geometry": {
       "type": "Polygon",
       "coordinates": [ ... ]  // array of coordinates
    }
  },
  "createdDate": "2025-09-17T23:07:00Z"
}
```

- The `ddsId` can be generated when the user creates a new statement (for reference in the list and as a filename). If not needed explicitly, we might skip it.
- Fields directly correspond to form inputs. The `productionPlotGeoJSON` holds the actual geometry data (which could be a full feature with properties, or just the geometry; storing the whole feature allows retaining a name property if provided).
- We include `createdDate` for logging and to show on the list.

This structure is kept in memory (for the session). If multiple DDS are created, we maintain an array of such objects. The PDF generation will use this object to fill in fields. For instance, on page 1 template, `operatorName` maps to the Operator Name field, etc.

**GeoJSON Data Format:** The application expects the GeoJSON to follow standard format (as per RFC 7946). Example accepted structure for a single polygon:

```
{
  "type": "Feature",
```

```
  "properties": { },
  "geometry": {
    "type": "Polygon",
    "coordinates": [
       [ [lng1, lat1], [lng2, lat2], ..., [lng1, lat1] ]
    ]
  }
}
```

or a FeatureCollection:

```
{
  "type": "FeatureCollection",
  "features": [ { "type": "Feature", "geometry": { "type": "Polygon", ... } } ]
}
```

The app will handle either by extracting the first Feature. The coordinates are expected to be in longitude/latitude (WGS84) as commonly provided. We'll document that the user should provide WGS84 coordinates if importing.
Internally, after validation, the GeoJSON might be stored as a string or an object. We might not need to store the entire geometry for long if not using it beyond the PDF; but to be safe for regeneration, we will keep it in the `DDS data object`. If memory is a concern and geometry is large, we could store a reference or summary instead.

**Temporary File Data (for PDF):** If we save PDFs to disk for download, the file naming convention can incorporate the `ddsId` or operator name plus date, ensuring uniqueness. No other complex data models are needed since there's no relational data beyond one DDS entry.

## Technology Stack

**Platform:** Replit (Node.js environment). The app will be launched as a Node.js web server (likely using **Express** framework for routing). Replit will host the development and allow live deployment for testing. We assume Replit can handle needed Node libraries (including those for PDF generation and possibly headless Chrome if using Puppeteer).

**Front-End:**
- **HTML5/CSS3/JavaScript** for the user interface. The UI will mostly be server-rendered (with possibly some client-side enhancements using JS for map and validations). We can use templating (EJS/Pug) to render pages if needed, or serve a static HTML with linked JS.
- **CSS Framework:** *Bootstrap 5* (for layout, form styling, and responsiveness) or a minimal CSS file for a clean look. This is to ensure the form is nicely spaced and mobile-friendly if needed.
- **Map Library:** *Leaflet.js* for embedding the map (plus Leaflet CSS and possibly a tile provider like OpenStreetMap tiles). This will be included via CDN or npm. Leaflet is lightweight and requires no API key. It will display the polygon for review.

- **Client-side Validation:** Light use of vanilla JS or a library for form validation (HTML5 built-in validation can handle required fields and basic types; custom script for things like ensuring scientific name is chosen, etc.).

**Back-End:**
- **Node.js + Express:** Express will handle routes: e.g., GET for the form page, POST for form submission, GET for downloads. It will also serve static files (like images and client JS). Middlewares might include body-parser (to handle form JSON) and possibly multer if we allow file upload (though for GeoJSON text we might just read it via JS and send JSON, avoiding multi-part upload complexity).
- **PDF Generation Library:** As discussed, likely **pdf-lib** (an npm library) for programmatic PDF creation. This library runs in pure JS and works in Node without native dependencies. We will use it to add text and images. If we choose Puppeteer instead, then **Puppeteer** will be in the stack (npm `puppeteer` package), and we'll have an HTML template and a generation function. Both approaches are on the table; the final choice will weigh ease of implementing complex layout versus resource usage.
- **GeoJSON Validation:** Could use `geojson-validation` or `@mapbox/geojsonhint` npm packages. Alternatively, a quick custom check or using JSON Schema via `ajv`. These ensure the back-end doesn't proceed with bad geo data.
- **Other Node Libraries:** If needed, **multer** (for handling file uploads of GeoJSON), **node-fetch** or axios (not likely needed unless pulling some external data), and possibly **cors** if we foresee any cross-origin use (likely not since served from same domain).
- **PDF Asset Handling:** The static images (methodology PNG and flowchart images) might be placed in a `/public/assets/` folder. The PDF generation code will load them. If using pdf-lib, we'll use Node's fs or path to read the files. If using Puppeteer/HTML, we just reference their paths in an `<img>` tag (Express static middleware will serve those).

**Development & Collaboration:** Replit provides an IDE-like environment. Multiple files will be created: e.g., `index.js` for server code, `views/` for any templates, `public/` for static files (JS, CSS, images). We will maintain version control (Replit auto-saves, but we might use Git if needed externally).

**Testing Tools:** During development, we will test in the Replit webview and externally via the provided URL. We will utilize PDF readers to verify the output. If needed, we can use Replit's console for logs and debugging info for the PDF generation step.

The chosen tech stack (Node.js + Express + [pdf-lib/jsPDF/Puppeteer] + Leaflet) is fairly common and should cover the needs. Node.js is well-suited for PDF generation and file handling on the back-end [7], and the front-end stack ensures a smooth user experience.

## Front-End Design & UX Structure

The user interface will be simple, clean, and focused on guiding the user through the DDS creation. Key design/layout considerations:

- **Page Layout:** The main interface is essentially one page (single-page app feel) with sections for input and a section for output/list. We might divide the screen or sequence in a wizard format.
- If single-page: we use headings and maybe collapsible sections for "Operator Info", "Product Details", "Production Plot", and then a "Generate PDF" button at the end. The list of created statements (if any) can appear below or on a sidebar.

- If multi-step: each step shows only one section's fields at a time, with a progress indicator (e.g., Step 1 of 4). The final step shows a summary and generate button. After generation, the user is taken to a results page or back to dashboard.

- **Navigation:** A simple top header could say "EUDR DDS Generator". If a list of DDS is maintained, the left side or top might have a "Statements List" link or tab, and a "New Statement" button. This is minimal since the app's main function is just creating statements.

- **Forms and Controls:** We will label each input clearly. For example:

- Operator Name [text box]
- Address [textarea or text box]
- Place of Activity [text box]
- Product HS Code [dropdown]
- Net Mass (kg) [number box]
- Volume ($m^3$ or relevant) [number box]
- Scientific Name [dropdown]
- Common Name [text box]
- Production Plot [dropdown of existing + "Import" button]
- Upload GeoJSON [file input, visible when "Import" is chosen]
- Map Preview [a div that contains the map once a plot is selected]

We will use placeholder text to indicate formats (e.g., "e.g., 25,000" for Net Mass) and perhaps small info icons or tooltips for more explanation (e.g., what is an HS code?). For critical fields like HS Code and scientific name, if the user isn't sure, we might provide a link or hint (maybe a link to an external reference or a predefined list).

- **Static Images Display:** The methodology diagram and flowcharts are not directly shown in the form (they are just for the PDF). We might include a small thumbnail or an info section that tells the user, "Pages 2 and 3 of the PDF will include standard methodology reference diagrams." This way the user isn't surprised by additional pages. Possibly, we could show those diagrams on the web page in a help modal if they want to see them ahead of time.

- **Feedback:** After clicking "Generate PDF," the user should see some immediate feedback – e.g., a loading spinner or message "Generating your DDS document...". This prevents confusion if the process takes a couple of seconds. Once done, it can automatically trigger the download and show a success message "DDS PDF generated!". If using a list, the new entry appears there.

- **Accessibility:** Use proper HTML semantics (labels for inputs, alt text for images (though images are mostly in PDF only), high-contrast text). Ensure the form can be navigated via keyboard. This may not be a primary requirement but is good practice.

- **Example Data:** The UI might provide an example or template values when first loaded, to illustrate what to input (especially if a user is testing). For instance, maybe a "Use Sample Data" button to autofill the form for quick demo. This can be a nice-to-have feature.

- **Responsive Design:** The form should work on various screen sizes. Using Bootstrap grid or flexbox, the form can stack on narrow screens. The map preview might be hidden or shown as a static image on mobile to avoid performance issues. However, since typical users are likely on desktop when preparing documents, mobile optimization is secondary.

- **Localization:** Not in initial scope, but we will write text in clear English. If needed, labels could be easily changed for other languages in the future.

**Front-End Pages/Views:**
We foresee possibly two main views:
- **Main Form View** (`/new`): Contains the form to create a statement (as detailed).
- **Dashboard/List View** (`/` or `/list`): Shows the list of statements created with a button to create new.
This could be the same page or a separate route. Possibly, we integrate the list at the bottom of the form view, which might simplify navigation (the user scrolls down to see existing statements). For clarity, a separate dashboard page might be cleaner: user goes back to dashboard after generating a PDF. In that case, after generation, the app can redirect to dashboard where a success message is shown and the list is updated.

An example layout on a single page could be:

```
[ Header: "EUDR DDS Generator" ]

(New Statement Form)
-------------------
Operator Details:
 [Name _____]
 [Address _____]
 [Place of Activity _____]

Product Details:
 [HS Code v] [Net Mass __kg__] [Volume __m3__]
 [Scientific Name v] [Common Name _____]

Production Plot:
 [Select Plot v] [Or Upload GeoJSON __Choose file__]
 [Map Preview here once selected]

[ Generate PDF ]   (button)

-------------------
(List of Created Statements)
DDS #001 - Palm Oil - created Sep 17, 2025 - [Download PDF]
DDS #002 - Cocoa - created Sep 18, 2025 - [Download PDF]
...
```

This illustrates how the pieces come together on screen.

## Static Content Embedding Details

Because pages 2 and 3 of the DDS PDF are comprised of static content (images/diagrams), their integration requires some one-time setup:

- The **methodology diagram (PNG)** will be stored in the application (possibly in `public/images/methodology.png`). It will be included at its native resolution if possible. The PDF generation code must ensure this image scales to fill the page while maintaining aspect ratio. If using HTML template, a simple `<img src=".../methodology.png" style="width:100%; height:auto;">` inside a container that is the size of the page will work. If using pdf-lib, we will get image dimensions and use `page.drawImage()` covering the page (with slight margins if needed). This image is static and does not change with user input.

- The **flowchart PDFs** need conversion to images. We will do this ahead of time (manually converting PDF to PNG or JPEG for the two diagrams). Suppose we obtain `flowchart1.png` and `flowchart2.png`. These will also reside in the app's assets. On PDF page 3, we have two images to place: we can either shrink each to half-page size and place one at the top and one at the bottom, or if their aspect ratio is more landscape, place them vertically split. We will likely do one above the other for simplicity (each taking roughly half the page). The PDF generation script will embed both images onto page 3, one positioned at (marginLeft, topMargin) and the second at (marginLeft, midPageY). If using HTML, we can create a two-row layout on that page with each flowchart image in each row.

- **Quality Check:** Ensure the text within those flowchart images is legible in the PDF. That might require a minimum resolution. If the provided PDFs are high quality, converting to a 300 DPI image for A4 page will yield good results. We should test one in the PDF to make sure no blurriness. If issues, we might consider slicing each diagram onto its own page (making the PDF 4 pages instead of 3). But per requirements, it should be 3 pages total, so we aim to fit both on one page by scaling appropriately (and possibly orienting the page in landscape if both are wide charts).

- **Static vs. Dynamic Assembly:** Since pages 2 and 3 are always the same regardless of user input, we could even consider them as template pages appended to every PDF. The generation module might preload these as finished PDF pages and just attach them behind page 1. For example, using pdf-lib, we could load a pre-made PDF of those two pages and copy the pages in. However, given we have the images, generating them on the fly is straightforward. But the concept is to keep page 2 and 3 completely unchanged between different DDS outputs. This makes testing and maintenance easier — if the methodology diagram or flowcharts update in the future, we just swap the image files and all new PDFs will reflect that.

- **Citation/References on static pages:** We might include a small footer or note on these pages to reference their source or version (especially if these come from internal documents). For instance, "Methodology diagram source: KPN Compliance Dept." or a date. This is optional and depends on whether the stakeholders want that metadata in the DDS.

- **Memory considerations:** Embedding images in PDF uses memory; we will ensure to embed them once per document. If using html->pdf, not an issue. If using pdf-lib, once an image is embedded, we reuse the reference for efficiency (pdf-lib allows reusing an image object on multiple pages).

By clearly delineating how the static content is handled, we ensure that the generated DDS PDFs consistently include the required reference information (methodology and workflows) along with the dynamic user-specific content.

## Conclusion

This Product Requirements Document has outlined the full scope and design of the EUDR DDS Generator web application. The system will provide a comprehensive solution for creating due diligence statements with minimal user effort, leveraging form automation and PDF generation. Key functional components (form input, geojson validation, PDF assembly, download management) have been detailed along with the user flow and data structures. The Node.js-based tech stack (Express, PDF library, etc.) and a structured front-end will together ensure that the app is robust, user-friendly, and produces output that meets EUDR's requirements. By embedding the provided static methodology and flowchart content into the PDF, the tool not only captures user data but also delivers the contextual compliance information expected in such statements.

With this PRD as a guide, the development team can proceed to implementation confident that all major requirements and design decisions are documented and agreed upon. The end result will be a Replit-hosted web app that significantly simplifies the creation of Due Diligence Statements for EUDR compliance, saving time and reducing errors for operators and traders in scope.

**Sources:**

- European Commission Green Forum – *EUDR Information System* (fields required for DDS: product HS code, description, quantity; geolocation via GeoJSON) [1] [2]
- Nutrient Blog – *Node.js PDF Generation Libraries* (overview of PDF generation options like Puppeteer, pdf-lib, jsPDF) [5]

---


[1] [2] The Information System of the Deforestation Regulation - European Commission
https://green-forum.ec.europa.eu/nature-and-biodiversity/deforestation-regulation-implementation/information-system-deforestation-regulation_en

[3] EU Trace DDS Creation.pdf
file://file-KjGVKXHqnWH1aQpsWaahyK

[4] [5] [7] Node.js PDF generator: How to generate PDFs from HTML with Node.js | Nutrient
https://www.nutrient.io/blog/how-to-generate-pdf-from-html-with-nodejs/

[6] How to convert images to PDF in Node.js - Nutrient
https://www.nutrient.io/blog/how-to-convert-image-to-pdf-in-nodejs/