# **SafeScan.Pro: An Advanced Phishing URL Detection System**

## **1. Introduction**

Welcome to the documentation for **SafeScan.Pro**, a robust and intelligent system designed to protect users from the ever-present threat of phishing attacks. In an era where malicious URLs are a primary vector for cybercrime, SafeScan.Pro provides a crucial layer of defense by analyzing URLs in real-time and leveraging both a custom-trained machine learning model and external threat intelligence to identify potential threats.

This document serves as a comprehensive guide for understanding, setting up, using, and extending the SafeScan.Pro project. It covers the core components, architectural design, installation procedures, and detailed explanations of its functionalities.

**Key Features:**

* **Machine Learning Powered:** Utilizes a Random Forest Classifier trained on a diverse dataset of legitimate and phishing URLs to identify suspicious patterns.
* **Real-time URL Analysis:** Extracts various features from submitted URLs on-the-fly to feed into the ML model.
* **External Threat Intelligence Integration:** Enhances detection capabilities by querying third-party security databases (specifically VirusTotal) for additional context and verification.
* **Intuitive User Interface:** A clean, responsive web interface built with React and Tailwind CSS for easy URL submission and clear presentation of analysis results.
* **Comprehensive Reporting:** Provides a detailed breakdown of the URL's risk score, classification, recommendation, and a deep dive into its extracted features.

**Target Audience:**

This documentation is intended for developers looking to understand the system's inner workings, users interested in how their URLs are analyzed, and anyone seeking to deploy or contribute to the project.

## **2. Project Overview**

SafeScan.Pro is structured as a full-stack web application, comprising a Python-based backend and a React-based frontend. The core idea is to provide a user-friendly tool that can quickly assess the safety of a given URL.

The process flow is as follows:

1. A user enters a URL into the web interface.
2. The frontend sends this URL to the Flask backend.
3. The Flask backend performs two main tasks:
   * It extracts a set of predefined features from the URL (e.g., length, presence of HTTPS, subdomain count). These features are then fed into the locally trained machine learning model.
   * Concurrently, it sends the URL to external threat intelligence APIs (currently VirusTotal) to gather additional security insights.
4. The backend compiles the results from both the ML model and external APIs.
5. This comprehensive analysis is sent back to the frontend.
6. The frontend displays the results in a structured and easy-to-understand dashboard, providing a risk score, classification, recommendation, feature breakdown, and external report summaries.

This hybrid approach ensures high accuracy by combining the pattern recognition strengths of machine learning with the vast, real-time threat databases of leading security vendors.

## **3. System Architecture**

SafeScan.Pro follows a client-server architecture, typical for web applications.

+----------------+ +-------------------+ +-----------------+  
| | | | | |  
| Frontend | | Flask Backend | | ML Model |  
| (React App) |------>| (Python API) |------>| (Random Forest) |  
| | | | | |  
+----------------+ +-------------------+ +-----------------+  
 ^ |  
 | | (API Calls)  
 | |  
 +------------------------+-----------------+  
 | |  
 | |  
 v v  
 +-----------------+ +-----------------+  
 | | | |  
 | VirusTotal API | | (Other APIs) |  
 | | | (Future Ext.) |  
 +-----------------+ +-----------------+

**Components:**

* **Frontend (React App):**
  + **Technology:** React.js, Tailwind CSS, Lucide Icons.
  + **Role:** Provides the interactive user interface for URL input, displays analysis results, and communicates with the backend API.
  + **Location:** frontend/src/App.js and frontend/public/index.html.
* **Backend (Flask API):**
  + **Technology:** Python, Flask, Pandas, Scikit-learn, Joblib.
  + **Role:**
    - Hosts the machine learning model.
    - Receives URLs from the frontend.
    - Performs feature extraction on the input URL.
    - Makes predictions using the loaded ML model.
    - Integrates with external threat intelligence APIs (e.g., VirusTotal).
    - Sends a comprehensive JSON response back to the frontend.
  + **Location:** backend/app.py.
* **Machine Learning Model:**
  + **Type:** Random Forest Classifier.
  + **Role:** Trained to identify patterns indicative of phishing or benign URLs based on extracted features.
  + **Training Script:** backend/model\_trainer.py.
  + **Saved Model:** backend/model/random\_forest\_model.pkl.
* **External Threat Intelligence (VirusTotal):**
  + **Service:** VirusTotal API.
  + **Role:** Provides a vast database of known malicious URLs and files, offering an additional layer of verification beyond the local ML model.
  + **Integration:** Handled within the Flask backend, but API keys are currently exposed in the frontend for simplicity (see security warning in section 8).

**Communication Flow:**

1. User inputs URL in React UI.
2. App.js sends POST request to http://127.0.0.1:5000/predict.
3. app.py receives request, extracts features, and runs ML prediction.
4. app.py also calls VirusTotal API with the URL.
5. app.py compiles ML prediction, risk score, extracted features, and VirusTotal report.
6. app.py sends a single JSON response back to App.js.
7. App.js updates its state and renders the detailed analysis on the UI.

## **4. Setup and Installation**

This section guides you through setting up and running the SafeScan.Pro project on your local machine.

### **4.1. Prerequisites**

Before you begin, ensure you have the following installed:

* **Python 3.8+:** Download from [python.org](https://www.python.org/downloads/).
* **Node.js (LTS version recommended, e.g., 18.x or 20.x):** Download from [nodejs.org](https://nodejs.org/en/download/). Node.js comes with npm (Node Package Manager).
  + **Note on Node.js 17+:** If you are using Node.js version 17 or higher, you might encounter an ERR\_OSSL\_EVP\_UNSUPPORTED error during frontend compilation. This is due to OpenSSL 3.0's stricter policies. The fix involves setting an environment variable, which will be covered in the Frontend Setup.
* **Git:** For cloning the project repository (optional, you can also download the zip). Download from [git-scm.com](https://git-scm.com/downloads).
* **Visual Studio Code (Recommended IDE):** Download from [code.visualstudio.com](https://code.visualstudio.com/).

### **4.2. Backend Setup**

The backend handles the machine learning model and API integrations.

1. Clone the Repository (or download):  
   Open your terminal or command prompt and navigate to your desired directory.  
   git clone <repository\_url> # Replace with your project's repo URL  
   cd SafeScan.Pro # Or whatever your project folder is named  
     
   If you downloaded a zip, extract it and navigate into the main project folder.
2. **Navigate to Backend Directory:**  
   cd backend
3. Create and Activate a Python Virtual Environment:  
   It's best practice to use a virtual environment to manage project dependencies.  
   python -m venv venv  
   * **On Windows:**  
     .\venv\Scripts\activate
   * **On macOS/Linux:**  
     source venv/bin/activate

You should see (venv) at the beginning of your terminal prompt, indicating the virtual environment is active.

1. Install Backend Dependencies:  
   Ensure you have a requirements.txt file in your backend directory with the following content:  
   flask  
   flask-cors  
   joblib  
   numpy  
   pandas  
   scikit-learn  
     
   Then, install them:  
   pip install -r requirements.txt
2. Place raw\_data.csv:  
   Ensure your machine learning training data (raw\_data.csv) is placed inside the backend/data/ directory. This file is crucial for training your model.
3. Train the Machine Learning Model:  
   This step generates the random\_forest\_model.pkl file, which your Flask app will load for predictions.  
   python model\_trainer.py  
   * **Verify Output:** Watch the terminal for output messages. You should see "Model saved successfully" at the end. This script will also print the Model classes: [...] which is useful for debugging if predictions are inverted.
4. Run the Flask Backend Server:  
   Keep this terminal open and running.  
   python app.py  
   * **Verify Output:** You should see messages indicating the Flask app is running, typically on http://127.0.0.1:5000.

### **4.3. Frontend Setup**

The frontend provides the user interface.

1. Open a NEW Terminal:  
   Do NOT close the terminal running your Flask backend. Open a separate terminal window.
2. **Navigate to Frontend Directory:**  
   cd frontend
3. **Install Frontend Dependencies:**  
   npm install  
   # If you prefer Yarn:  
   # yarn install  
     
   Also, ensure lucide-react is installed for the icons:  
   npm install lucide-react  
   # or  
   # yarn add lucide-react
4. Run the React Development Server (with OpenSSL fix if needed):  
   This command starts the React development server.
   * **For Windows (if using Node.js 17+ and encountering OpenSSL error):**  
     set NODE\_OPTIONS=--openssl-legacy-provider && npm start
   * **For macOS / Linux (if using Node.js 17+ and encountering OpenSSL error):**  
     NODE\_OPTIONS=--openssl-legacy-provider npm start
   * **For other cases (or if the above fix isn't needed):**  
     npm start  
     # or  
     # yarn start
   * **Verify Output:** This will compile the React application and should automatically open a new tab in your web browser, usually at http://localhost:3000/.

## **5. Backend Details**

The backend is the brain of SafeScan.Pro, handling data processing, machine learning, and external API communication.

### **5.1. Data Collection and Preprocessing (raw\_data.csv)**

The model is trained on a dataset of URLs, categorized as either benign or phishing. This data is typically stored in backend/data/raw\_data.csv. The quality and diversity of this dataset are crucial for the model's performance.

### **5.2. Feature Engineering (model\_trainer.py, app.py)**

The extract\_features(url) function is central to the project. It transforms a raw URL string into a set of numerical features that the machine learning model can understand. This function is identical in both model\_trainer.py (for training) and app.py (for real-time prediction) to ensure consistency.

**Key Features Extracted:**

* URLLength: Total length of the URL.
* URLLengthUnsafe: Binary (1 if URL length > 100, 0 otherwise). Long URLs can sometimes be a sign of obfuscation.
* DomainLength: Length of the domain part of the URL.
* IsHTTPS: Binary (1 if URL uses HTTPS, 0 for HTTP). HTTPS indicates a secure connection, but not necessarily a safe site.
* NoOfSubDomain: Number of subdomains in the URL. A high number might indicate suspicious activity.
* SubdomainUnsafe: Binary (1 if NoOfSubDomain > 3, 0 otherwise).
* NoOfLettersInURL: Count of alphabetic characters in the URL.
* NoOfDegitsInURL: Count of numeric digits in the URL.
* SpacialCharRatioInURL: Ratio of special characters (non-alphanumeric) to the total URL length.
* HasObfuscation: Binary (1 if common obfuscation techniques like @, multiple //, .., or % are present).
* TLDLegitimateProb: Binary (1 if the Top-Level Domain is common/legitimate, 0 otherwise).

### **5.3. Model Training (model\_trainer.py)**

The model\_trainer.py script is responsible for:

1. **Loading Data:** Reads raw\_data.csv.
2. **Feature Extraction:** Applies the extract\_features function to every URL in the dataset.
3. **Data Cleaning:** Handles missing values and ensures data types are correct.
4. **Data Splitting:** Divides the dataset into training and testing sets.
5. **Model Training:** Trains a RandomForestClassifier on the training data. Random Forests are ensemble models known for their robustness and ability to handle various data types.
6. **Model Evaluation:** Assesses the model's performance using metrics like Confusion Matrix, Classification Report, and Accuracy Score.
7. **Model Saving:** Persists the trained model to backend/model/random\_forest\_model.pkl using joblib for later use by app.py.

### **5.4. Flask API (app.py)**

app.py is the heart of the backend, providing the API endpoint for the frontend.

* **Model Loading:** On startup, it loads the random\_forest\_model.pkl file into memory. It also retrieves the exact feature names the model was trained on (model.feature\_names\_in\_) to ensure consistency during real-time prediction.
* **/predict Endpoint:**
  + Listens for POST requests containing a url in the JSON body.
  + Calls the extract\_features function on the submitted URL.
  + Creates a Pandas DataFrame from these features, ensuring the columns are in the exact order expected by the trained model.
  + Uses model.predict() to get the binary classification (0 or 1) and model.predict\_proba() to get the probability of each class.
  + **Crucially, it maps the numerical prediction to the correct 'benign' or 'phishing' string using model.classes\_ to avoid inverted results.**
  + Calculates a risk\_score based on the probability of the 'phishing' class.
  + Determines a classification (BENIGN, SUSPICIOUS, PHISHING) and recommendation based on the prediction and risk score.
  + **Integrates with VirusTotal API** (see Section 7).
  + Constructs a comprehensive JSON response containing:
    - analyzed\_url
    - risk\_score
    - classification
    - recommendation
    - feature\_details (human-readable extracted features)
    - ml\_summary (static text)
    - analyzed\_date
    - virustotalReport (from external API)
  + Returns this JSON response to the frontend.
* **CORS Enabled:** Flask-CORS is used to allow the React frontend (running on a different port) to make requests to the Flask backend.

## **6. Frontend Details**

The frontend is built with React.js, providing a dynamic and interactive user experience.

### **6.1. React Components (App.js)**

The entire user interface logic resides within frontend/src/App.js.

* **State Management:** useState hooks are used to manage the application's state, including the input URL, the analysis results, loading indicators, and error messages.
* **Conditional Rendering:** The UI dynamically switches between displaying static informational cards (when no URL has been scanned) and the detailed analysis results (after a successful scan).
* **Event Handling:** handleUrlChange updates the URL input, and handleSubmit triggers the API calls to the backend.

### **6.2. User Interface (UI)**

The UI is styled using **Tailwind CSS**, a utility-first CSS framework, which allows for rapid and consistent styling directly within the JSX.

* **Responsive Design:** Tailwind's responsive prefixes (sm:, md:, lg:) are used to ensure the layout adapts gracefully to different screen sizes.
* **Visual Elements:**
  + **Header:** Displays the project title "SafeScan.Pro" with a prominent shield icon.
  + **URL Input Section:** Contains the input field for URLs and a "Scan" button. It includes loading spinners and error message displays.
  + **Static Info Cards:** Before a scan, three cards highlight the project's key aspects: "Real-time Analysis," "ML-Powered," and "Threat Detection."
  + **Analysis Results Section:**
    - **Security Analysis Card:** Shows the analyzed URL, a dynamic risk score bar, classification (BENIGN, SUSPICIOUS, PHISHING), and a recommendation.
    - **Feature Analysis Card:** Breaks down the individual features extracted from the URL, providing insights into why a URL might be flagged.
  + **External Threat Intelligence Section:** (See Section 7) Displays summarized reports from VirusTotal.
  + **Footer:** Contains copyright information.

### **6.3. API Communication**

The frontend communicates with the backend via fetch API calls:

* **handleSubmit function:** Orchestrates the communication.
  + Sends a POST request to the Flask backend's /predict endpoint.
  + Parses the JSON response from the backend.
  + Updates the analysisResult state, triggering UI re-renders.
  + Also triggers calls to external threat intelligence APIs.

## **7. External Threat Intelligence (VirusTotal)**

While the local ML model is powerful, its knowledge is limited to the data it was trained on. To enhance detection capabilities and provide broader context, SafeScan.Pro integrates with external threat intelligence platforms.

### **7.1. VirusTotal API Integration**

VirusTotal is a widely recognized service that aggregates data from numerous antivirus engines, website scanners, and blacklisting services.

* **API Keys:**
  + Your VirusTotal API key (fa2edeb4ceb240ee6405593337e2974dbb22b1ae333406955b532a1eeb476d7f) is used to authenticate requests to the VirusTotal API.
  + **Security Warning:** For this project, the API key is placed directly in frontend/src/App.js. **This is highly insecure for production environments.** In a real-world application, all API calls to external services should be proxied through your backend (app.py) to keep API keys confidential and prevent client-side exposure.
* **fetchVirustotalReport(url) function:**
  1. **URL Submission:** Sends a POST request to VirusTotal's /urls endpoint to initiate an analysis of the provided URL.
  2. **Polling for Results:** After submitting, the API returns an analysisId. The function then waits for a fixed period (15 seconds in this implementation) and then fetches the analysis report using this analysisId. In a production system, this would be a more sophisticated polling mechanism with exponential backoff.
  3. **Report Summarization:** Extracts key statistics from the VirusTotal report, such as the number of malicious, suspicious, harmless, and undetected detections.
  4. **Error Handling:** Catches API-specific errors (e.g., invalid API key, analysis not found) and sets appropriate error messages.

### **7.2. Data Interpretation in UI**

The "External Threat Intelligence" section in the UI displays a summarized report from VirusTotal:

* **Loading Indicator:** Shows a spinner and "Scanning with VirusTotal..." message while the API call is in progress.
* **Error Messages:** Clearly displays error messages (e.g., "API key is disabled!", "URL not found in VirusTotal database.") if the API call fails or returns specific errors.
* **Summary Statistics:** Presents the counts for malicious, suspicious, undetected, and harmless detections.
* **Full Report Link:** Provides a direct link to the comprehensive report page on the VirusTotal website for deeper investigation.

## **8. How to Use SafeScan.Pro**

Using SafeScan.Pro is straightforward:

1. **Ensure Both Backend and Frontend are Running:**
   * Verify your Flask backend is running in one terminal (python app.py).
   * Verify your React frontend is running in another terminal (npm start).
2. **Open the Application:**
   * Navigate to http://localhost:3000/ in your web browser.
3. **Enter a URL:**
   * In the "URL Security Scanner" input field, type or paste the URL you wish to analyze.
   * **Examples:**
     + A legitimate URL: https://www.google.com
     + A potentially suspicious URL (for testing): http://example.com/login.php (if you have one from your dataset)
4. **Click "Scan":**
   * The application will display loading indicators.
5. **Interpret Results:**
   * **Security Analysis Card:**
     + **Risk Score:** A percentage indicating the likelihood of the URL being malicious. Higher means higher risk.
     + **Classification:** BENIGN, SUSPICIOUS, or PHISHING. This is the primary verdict from your ML model.
     + **Recommendation:** Actionable advice based on the classification.
   * **Feature Analysis Card:** Provides transparency into the extracted features that contributed to the ML model's decision. This helps understand *why* a URL was classified a certain way.
   * **External Threat Intelligence Card (VirusTotal):**
     + Shows a summary of detections from VirusTotal's vast database.
     + Look at "Malicious" and "Suspicious" counts. A non-zero count here is a strong indicator of danger.
     + Click "View Full Report" for detailed findings from various security vendors.

## **9. Troubleshooting Common Issues**

Here are some common issues you might encounter and how to resolve them:

* **"Failed to get prediction: Failed to fetch" / Blank Results:**
  + **Cause:** Frontend cannot connect to the backend, or backend is not responding with expected data.
  + **Fixes:**
    1. **Backend Not Running:** Ensure python app.py is actively running in its terminal.
    2. **Backend Errors:** Check the backend terminal for Python tracebacks or error messages after you click "Scan" in the frontend. Fix any Python errors in app.py.
    3. **Incorrect API\_BASE\_URL:** In frontend/src/App.js, ensure API\_BASE\_URL matches the exact address Flask is running on (e.g., http://127.0.0.1:5000).
    4. **Firewall:** Temporarily disable your operating system's firewall (e.g., Windows Defender Firewall) to check if it's blocking the connection. If it works, add an exception for Python/Flask.
    5. **Backend Response Structure:** If the Network tab shows 200 OK but results are blank, the backend might be sending malformed or unexpected JSON. Check the "Response" tab in your browser's Developer Tools (F12 -> Network tab -> click predict request -> Response tab) to see the exact JSON. Compare it with what App.js expects.
* **ERR\_OSSL\_EVP\_UNSUPPORTED during npm start:**
  + **Cause:** Incompatibility between newer Node.js versions (17+) and older dependencies using deprecated OpenSSL algorithms.
  + **Fix:** Run npm start with the NODE\_OPTIONS environment variable.
    - **Windows:** set NODE\_OPTIONS=--openssl-legacy-provider && npm start
    - **macOS/Linux:** NODE\_OPTIONS=--openssl-legacy-provider npm start
* **Incorrect ML Predictions (e.g., Google.com is Phishing):**
  + **Cause:** The mapping of the model's numerical output (0 or 1) to "Benign" or "Phishing" is inverted, or there's an issue with feature extraction consistency.
  + **Fix:**
    1. **Retrain Model:** Run python model\_trainer.py to ensure the model is freshly trained. Note the Model classes: [...] output.
    2. **Verify app.py Mapping:** In app.py, ensure the logic that converts model.predict() output to classification\_display correctly uses model.classes\_ to map the numerical index to the correct string label. The provided app.py code in the last update explicitly handles this.
    3. **Feature Consistency:** Double-check that the extract\_features function is *identical* in both model\_trainer.py and app.py. Any difference will lead to incorrect predictions.
* **External API Errors (e.g., "API key is disabled!", "Scan result not found"):**
  + **Cause:** Invalid API key, API rate limits, or the external service took too long to process.
  + **Fixes:**
    1. **API Keys:** Double-check that the URLSCAN\_API\_KEY and VIRUSTOTAL\_API\_KEY in frontend/src/App.js are exactly correct.
    2. **Rate Limits:** Free API tiers often have strict rate limits. If you scan too many URLs too quickly, you might hit these limits. Wait a few minutes and try again.
    3. **Polling Time:** The current 15-second wait might not always be enough for complex scans. In a production app, you'd implement a more robust polling loop.

## **10. Future Enhancements**

SafeScan.Pro is a solid foundation for phishing detection. Here are some ideas for future enhancements:

* **Backend API Key Proxy:** Move all external API keys (VirusTotal, etc.) from the frontend to the Flask backend. The frontend would then call a backend endpoint (e.g., /api/virustotal-scan) which securely makes the call to the external API. This is a critical security improvement.
* **More External Threat Intelligence:** Integrate with other services like Google Safe Browsing API, PhishTank, or commercial threat intelligence feeds for even broader coverage.
* **Asynchronous External API Calls:** Implement a more robust asynchronous task queue (e.g., Celery with Redis) for external API calls, especially for services that require longer processing times, to avoid blocking the main Flask thread.
* **User Accounts and History:** Add user authentication and a database (e.g., SQLite, PostgreSQL) to store scan history for logged-in users.
* **Advanced UI/UX:**
  + Real-time progress updates for external API scans.
  + More interactive visualizations for feature analysis.
  + Dark mode/light mode toggle.
  + Copy-to-clipboard functionality for URLs.
* **Dockerization:** Containerize the frontend and backend using Docker for easier deployment and environment consistency.
* **Deployment:** Deploy the application to cloud platforms (e.g., Heroku, AWS, Google Cloud) for public access.
* **Model Retraining Automation:** Set up a pipeline to periodically retrain the ML model with new data to keep it up-to-date with evolving phishing techniques.

## **11. Conclusion**

SafeScan.Pro provides a practical and effective solution for combating phishing URLs by combining the power of machine learning with external threat intelligence. This documentation should equip you with the knowledge to understand, run, and further develop this project. By continually improving its detection capabilities and user experience, SafeScan.Pro can contribute significantly to a safer online environment.

## **Appendix: Feature Definitions**

Here's a detailed breakdown of the features extracted by the extract\_features function:

* **URLLength**: Integer. The total number of characters in the URL string.
* **URLLengthUnsafe**: Binary (0 or 1). 1 if URLLength is greater than 100 characters; 0 otherwise. Longer URLs can sometimes be used for obfuscation.
* **DomainLength**: Integer. The number of characters in the domain part of the URL (e.g., google.com from https://www.google.com/search).
* **IsHTTPS**: Binary (0 or 1). 1 if the URL scheme is https; 0 if it's http or another scheme.
* **NoOfSubDomain**: Integer. The count of subdomains in the URL. For www.example.com, this would be 2 (www and example).
* **SubdomainUnsafe**: Binary (0 or 1). 1 if NoOfSubDomain is greater than 3; 0 otherwise. Excessive subdomains can be a red flag.
* **NoOfLettersInURL**: Integer. The total count of alphabetic characters (a-z, A-Z) within the entire URL string.
* **NoOfDegitsInURL**: Integer. The total count of numeric digits (0-9) within the entire URL string.
* **SpacialCharRatioInURL**: Float. The ratio of non-alphanumeric characters (e.g., ?, =, &, -, \_, /, .) to the total length of the URL. Calculated as (count of special characters) / URLLength.
* **HasObfuscation**: Binary (0 or 1). 1 if the URL contains common obfuscation patterns such as @ symbol, multiple // (e.g., http://example.com//malicious), .. (directory traversal attempt), or % (URL-encoded characters); 0 otherwise.
* **TLDLegitimateProb**: Binary (0 or 1). 1 if the Top-Level Domain (e.g., .com, .org) is found in a predefined list of common/legitimate TLDs; 0 otherwise. This helps identify unusual or newly registered TLDs often used by phishers.