**Team Members:**

* Finhas Demissie
* Selam Habtewolde
* Bealu Girma

# **Model Deployment**

## **1. Overview**

The deployment phase is the final step in making the machine learning model available for real-world applications. This process ensures that the landslide susceptibility prediction model can be accessed by users, allowing them to input geographic data and receive predictions about landslide risk levels. For this project, the model was deployed using Flask, allowing for web-based interaction where users can input the features and receive a prediction.

**Key Deployment Steps:**

1. Serialize the trained model for storage.
2. Serve the model using a Flask web application.
3. Set up API endpoints for predictions.
4. Implement security measures for data protection.
5. Monitor the model’s usage and performance.

## **2. Model Serialization**

**Process:**

Model serialization is the process of saving the trained model in a format that can be loaded and used for making predictions later. For this project, the trained Random Forest model was serialized using Python's **pickle** library.

**Steps:**

1. After training and validating the model, it was serialized to a file using pickle:

*import pickle*

*# Serialize model*

*with open('model.pkl', 'wb') as file:*

*pickle.dump(trained\_model, file)*

1. The **pkl** format was chosen because it is lightweight and efficient for storage.

**Considerations:**

* **Efficiency**: pickle is fast and supports a wide range of Python objects, making it ideal for saving complex machine learning models.
* **Compatibility**: It is important to ensure that the environment where the model is served has the same library versions as when the model was serialized.

## **3. Model Serving**

To serve the serialized model for predictions, the **Flask** framework was chosen due to its simplicity and flexibility in setting up web applications. Flask allows users to interact with the model via a web interface where they can input features such as Aspect, Elevation, and Precipitation, and receive predictions about landslide risk levels.

**Steps for Model Serving:**

1. **Load the serialized model** when the Flask app starts:

*with open('model.pkl', 'rb') as file:*

*model = pickle.load(file)*

1. **Create a web interface** where users can enter features and submit them to the model:

*@app.route('/predict', methods=['POST'])*

*def predict():*

*data = request.json*

*prediction = model.predict([data['features']])*

*return jsonify({'risk\_level': prediction})*

1. **Platform**: The model is hosted on an on-premises Flask server, but could also be deployed to cloud services such as **Heroku** or **AWS Lambda** for scalability

## **4. API Integration**

An API was developed to allow external systems to interact with the model and receive predictions programmatically.

### API Endpoints:

* **Endpoint**: /predict
  + **Method**: POST
  + **Input Format**: JSON object containing the feature values.
  + **Output Format**: JSON response with the predicted risk level.

## **5. Security Considerations**

To ensure the deployment is secure and prevents unauthorized access, the following measures were implemented:

1. **Authentication**: A simple token-based authentication system was added to restrict access to the API. Each request must include an authorization token.

*@app.route('/predict', methods=['POST'])*

*@token\_required # Decorator function to enforce token validation*

*def predict():*

*...*

1. **HTTPS Encryption:** To protect data in transit, the Flask server was configured to run behind a **reverse proxy** with SSL certificates, enabling HTTPS encryption.
2. **CORS Policy:** Configured Cross-Origin Resource Sharing (CORS) to ensure that only allowed domains can interact with the API.
3. **Input Validation:** Implemented strict validation of incoming data to prevent malicious inputs or attempts to crash the system

## **6. Monitoring and Logging**

Monitoring the performance of the deployed model is critical to ensuring it functions correctly in production. Flask was configured to log incoming requests and the model's predictions.

### Monitoring Approach:

1. **Metrics Tracked**:
   * **Number of API calls**: Logs every prediction request, including inputs and outputs.
   * **Response times**: Monitors how long each prediction takes.
   * **Prediction outcomes**: Logs predictions to track whether the model is biased toward any particular risk level.
2. **Error Handling**: Custom error handlers were implemented to log any unexpected behavior and prevent system crashes:

*@app.errorhandler(500)*

*def internal\_error(error):*

*app.logger.error(f"Server Error: {error}")*

*return "500 error"*

1. **Alerting Mechanisms**:
   * Integrated **Flask-Logging** to send alerts to the system administrator in case of repeated errors or anomalies in usage patterns.
   * Implemented a dashboard using **Grafana** (optional) to visualize API usage and model performance over time.

## **Conclusion**

The landslide susceptibility prediction model was successfully deployed using Flask. By following best practices for serialization, API development, security, and monitoring, the model is now capable of handling real-world prediction requests while ensuring robustness and security. This approach ensures that users can receive accurate and timely predictions about landslide risks based on their input data, making the system both accessible and reliable.