# 

# **NimbusAI**

# **Version: Plot-Salt-Smooth-Graph (PSS)**

# **Code Documentation**

## **Table of Contents**

1. Overview
2. Flask Application (app.py)
3. Java Processing Engine
4. JavaScript Client
5. Integration Points

## 

## 

## 

## 

## 

## 

## **Overview**

The Nimbus AI PSS system is built with a hybrid architecture:

* **Frontend**: HTML/CSS/JavaScript client application
* **Backend**: Flask-based Python server
* **Processing Engine**: Java-based PSS pipeline implementation
* **Data Storage**: File-based storage system for data and visualizations

This documentation provides a detailed breakdown of each component and method, intended for developers who need to maintain, extend, or troubleshoot the system.

## **Flask Application (app.py)**

### **analyze\_csv(file\_path)**

**Process**:

1. Reads the CSV file into a pandas DataFrame
2. Calculates basic statistics and metadata:
   * Number of rows and columns
   * Column names
   * Identification of numeric and categorical columns
   * Missing value counts per column
   * Statistical summary using pandas describe()
3. Returns the DataFrame, statistics, and any error message

**Notes**:

* The function uses pandas' data type detection to classify columns
* The statistical summary includes count, mean, std, min, 25%, 50%, 75%, and max for numeric columns
* Error handling catches any exceptions during reading or analysis

#### **get\_bot\_response(message, session\_data)**

python

def get\_bot\_response(message, session\_data):

"""Generate a response from the chatbot based on user message and session data"""

message\_lower = message.lower()

session\_id = session\_data.get('session\_id')

logger.info(f"Processing message for session {session\_id}: '{message}'")

if not session\_data.get('file\_processed'):

return {

'message': "Please upload a CSV file first so I can analyze it."

}

*# Default fallback*

return {

'message': "I'm not sure how to help with that. You can ask me to show plots, explain the pipeline, or type 'Help' to see all options."

}

**Purpose**: Processes user messages and generates appropriate responses based on the message content and session data.

**Parameters**:

* message (string): User's message text
* session\_data (dict): Dictionary containing session data, including processed files, visualizations, and statistics

**Returns**:

* response (dict): Response object containing message text and optional visualization or raw data

**Process**:

1. Converts message to lowercase for case-insensitive matching
2. Logs the incoming message with session ID
3. Verifies that a file has been processed for this session
4. Matches message against various patterns to determine the appropriate response:
   * Help requests
   * Explanation requests for salting or smoothing
   * Requests for various visualizations (initial, salted, smoothed, final)
   * Requests for raw or processed data
   * Requests for dataset explanation and statistics
5. Returns a response object with the appropriate content

**Notes**:

* Response objects can include 'message', 'visualization', and 'rawData' fields
* Visualization responses include the URL to the appropriate PNG file
* Raw data responses include a string representation of the data
* The function uses string matching rather than NLP for simplicity
* The actual visualization files are generated by the Java processing engine

### **Route Handlers**

#### **index()**

python

@app.route('/')

def index():

"""Render the main page"""

return render\_template('index.html')

**Purpose**: Serves the main application page.

**Returns**: Rendered HTML for the main page

**Notes**: Uses Flask's template rendering to serve the index.html template

#### **upload\_file()**

python

@app.route('/api/upload', methods=['POST'])

def upload\_file():

"""Handle file upload and processing"""

if 'file' not in request.files:

return jsonify({'error': 'No file part'}), 400

file = request.files['file']

if file.filename == '':

return jsonify({'error': 'No selected file'}), 400

if file and allowed\_file(file.filename):

*# Generate a new session ID*

session\_id = str(uuid.uuid4())

*# Store session ID in the user's Flask session*

session['session\_id'] = session\_id

logger.info(f"Created new session: {session\_id}")

filename = secure\_filename(file.filename)

file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], f"{session\_id}\_{filename}")

return jsonify({'error': 'File type not allowed. Please upload a CSV file.'}), 400

**Purpose**: Handles file uploads, creates sessions, and initiates the processing pipeline.

**Process**:

1. Validates that a file was included in the request
2. Checks that the file has a name
3. Verifies that the file is a CSV (using allowed\_file)
4. Generates a new UUID for the session
5. Stores the session ID in the Flask session
6. Saves the uploaded file with a session-specific filename
7. Analyzes the CSV file using analyze\_csv()
8. Processes the file with the Java engine using process\_csv\_with\_java()
9. Stores all session data in the sessions dictionary
10. Returns a success response with the initial visualization

**Returns**:

* On success: JSON response with confirmation message, visualization URL, and session ID
* On error: JSON response with error message and appropriate HTTP status code

**Notes**:

* This is the entry point for new data processing
* Each file upload creates a new session
* The session ID is stored both server-side (in the sessions dictionary) and client-side (in Flask's session)

#### **chat()**

python

@app.route('/api/chat', methods=['POST'])

def chat():

"""Process chat messages"""

data = request.json

message = data.get('message', '')

session\_id = data.get('session\_id')

if app.debug:

response['\_debug'] = {

'session\_exists': session\_id in sessions if session\_id else False,

'available\_sessions': list(sessions.keys()),

'timestamp': time.time()

}

return jsonify(response)

**Purpose**: Processes chat messages from the user and returns appropriate responses.

**Process**:

1. Extracts the message and session ID from the request
2. Logs the incoming request with session ID
3. If no session ID is provided, attempts to retrieve it from the Flask session
4. Validates that the session exists in the sessions dictionary
5. Gets a response using get\_bot\_response() if the session is valid
6. Returns a default message if no valid session is found
7. Always includes the session ID in the response
8. Adds debug information in development mode

**Returns**: JSON response with bot message, optional visualization URL, and session ID

**Notes**:

* This endpoint handles all user interactions after the initial file upload
* Session validation is critical for ensuring users see their own data
* Debug information helps with troubleshooting in development

#### **get\_visualization(session\_id, filename)**

python

@app.route('/visualizations/<session\_id>/<filename>')

def get\_visualization(session\_id, filename):

"""Serve visualization images"""

*# Check if this is a valid session*

if session\_id not in sessions:

logger.warning(f"Attempt to access invalid session: {session\_id}")

return 'Session not found', 404

file\_path = os.path.join(app.config['VISUALIZATIONS\_DIR'], session\_id, filename)

if os.path.exists(file\_path):

logger.info(f"Serving visualization: {file\_path}")

*# Set no-cache headers to prevent browser caching*

response = send\_file(file\_path, mimetype='image/png')

response.headers['Cache-Control'] = 'no-store, no-cache, must-revalidate, max-age=0'

response.headers['Pragma'] = 'no-cache'

response.headers['Expires'] = '0'

return response

else:

logger.warning(f"Visualization not found: {file\_path}")

return 'Visualization not found', 404

**Purpose**: Serves visualization image files for display in the web interface.

**Parameters**:

* session\_id (string): Session ID from the URL
* filename (string): Name of the visualization file to serve

**Process**:

1. Validates that the requested session exists
2. Constructs the file path for the requested visualization
3. Checks if the file exists
4. Serves the file with appropriate MIME type if found
5. Sets no-cache headers to prevent browser caching
6. Returns a 404 error if the file is not found

**Returns**:

* On success: The requested PNG file with appropriate headers
* On error: 404 error message

**Notes**:

* No-cache headers are important to prevent browsers from showing stale visualizations
* Session validation ensures users can only access visualizations from their own sessions

#### **download\_plot(session\_id, plot\_type)**

python

@app.route('/api/download/<session\_id>/<plot\_type>')

def download\_plot(session\_id, plot\_type):

"""Download a plot image"""

if session\_id not in sessions:

logger.warning(f"Attempt to download from invalid session: {session\_id}")

return 'Session not found', 404

return 'Visualization not found', 404

**Purpose**: Allows users to download visualization images.

**Parameters**:

* session\_id (string): Session ID from the URL
* plot\_type (string): Type of plot to download (initial, salted, smoothed, final)

**Process**:

1. Validates that the requested session exists
2. Checks if the session has visualizations
3. Based on the requested plot type, serves the appropriate file for download
4. Sets headers to trigger browser download instead of display

**Returns**:

* On success: The requested PNG file with download headers
* On error: 404 error message

**Notes**:

* The as\_attachment=True parameter causes browsers to download rather than display the file
* download\_name parameter sets the filename for the downloaded file
* Session validation ensures users can only download visualizations from their own sessions

#### **download\_processed\_data(session\_id)**

python

@app.route('/api/download/<session\_id>/processed\_data')

def download\_processed\_data(session\_id):

"""Download the processed data as CSV"""

if session\_id not in sessions:

logger.warning(f"Attempt to download data from invalid session: {session\_id}")

return 'Session not found', 404

if session\_id in sessions and sessions[session\_id].get('processed\_data') is not None:

*# Create a temporary file*

fd, temp\_path = tempfile.mkstemp(suffix='.csv')

try:

*# Write the processed data to the temp file*

sessions[session\_id]['processed\_data'].to\_csv(temp\_path, index=False)

return send\_file(

temp\_path,

as\_attachment=True,

download\_name=f"nimbus\_processed\_data\_{session\_id}.csv",

mimetype='text/csv'

)

finally:

*# Clean up the temp file (will be deleted after the response is sent)*

os.close(fd)

else:

return 'Processed data not found', 404

**Purpose**: Allows users to download their processed data as a CSV file.

**Parameters**:

* session\_id (string): Session ID from the URL

**Process**:

1. Validates that the requested session exists
2. Checks if the session has processed data
3. Creates a temporary file to store the CSV data
4. Writes the processed data to the temporary file
5. Serves the file for download
6. Ensures the temporary file is closed and will be deleted after the response is sent

**Returns**:

* On success: CSV file with the processed data
* On error: 404 error message

**Notes**:

* Uses pandas' to\_csv method to convert the DataFrame to CSV format
* Uses a temporary file to avoid storing duplicate data
* Ensures proper cleanup of temporary files
* Session validation ensures users can only download data from their own sessions

#### **get\_session()**

python

@app.route('/api/session', methods=['GET'])

def get\_session():

"""Get current session information"""

session\_id = session.get('session\_id')

if session\_id and session\_id in sessions:

logger.info(f"Session check: Active session {session\_id}")

return jsonify({

'active': True,

'session\_id': session\_id,

'file\_processed': sessions[session\_id].get('file\_processed', False)

})

else:

logger.info("Session check: No active session")

return jsonify({

'active': False,

'session\_id': None,

'file\_processed': False

})

**Purpose**: Provides information about the current user session.

**Process**:

1. Retrieves the session ID from the Flask session
2. Checks if the session ID exists and is valid
3. Returns session information including active status and processing status

**Returns**: JSON object with session information

**Notes**:

* Used by the frontend to check if there's an active session on page load
* Helps maintain session continuity between page refreshes
* Allows the frontend to determine if a file has already been processed

#### **clear\_session()**

python

@app.route('/api/clear\_session', methods=['POST'])

def clear\_session():

"""Clear the current session (useful for testing)"""

session\_id = session.get('session\_id')

if session\_id and session\_id in sessions:

*# Remove from sessions dictionary*

del sessions[session\_id]

*# Clear Flask session*

session.pop('session\_id', None)

logger.info(f"Cleared session: {session\_id}")

return jsonify({'success': True, 'message': f"Session {session\_id} cleared"})

return jsonify({'success': False, 'message': "No active session to clear"})

**Purpose**: Allows users to explicitly clear their current session.

**Process**:

1. Retrieves the session ID from the Flask session
2. If the session exists, removes it from the sessions dictionary
3. Clears the session ID from the Flask session
4. Logs the session clearing

**Returns**: JSON response indicating success or failure

**Notes**:

* Primarily useful for testing and debugging
* Allows users to "start fresh" without reloading the application
* Does not delete the uploaded files or generated visualizations from disk

### **Main Execution**

python

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Purpose**: Starts the Flask development server when the script is run directly.

**Notes**:

* Debug mode is enabled for development
* In production, a WSGI server like Gunicorn should be used instead

## **Java Processing Engine**

The Java processing engine implements the core Plot-Salt-Smooth-Graph pipeline. This section documents the key Java classes and methods.

### **Main Class**

public class Main {

public static void main(String[] args) {

if (args.length < 3) {

System.err.println("Usage: java Main <input\_csv\_path> <output\_dir> <output\_csv\_path>");

System.exit(1);

}

String inputPath = args[0];

String outputDir = args[1];

String outputDataPath = args[2];

try {

*// Load the data*

List<DataPoint> data = loadData(inputPath);

*// Apply the PSS pipeline*

PSSProcessor processor = new PSSProcessor();

PSSResult result = processor.process(data, outputDir);

*// Save the processed data*

saveProcessedData(result, outputDataPath);

System.out.println("Processing complete. Visualizations saved to " + outputDir);

} catch (Exception e) {

System.err.println("Error processing data: " + e.getMessage());

e.printStackTrace();

System.exit(1);

}

}

}

**Purpose**: Entry point for the Java processing engine.

**Parameters**:

* Command-line arguments:
  + inputPath: Path to the input CSV file
  + outputDir: Directory for storing generated visualizations
  + outputDataPath: Path for the output CSV file with processed data

**Process**:

1. Validates command-line arguments
2. Loads data from the input CSV file
3. Creates a PSSProcessor instance
4. Processes the data using the PSS pipeline
5. Saves the processed data to the output CSV file
6. Returns success or error message

**Notes**:

* Called by the Flask application via subprocess
* Processing errors are reported through stderr
* Success messages are reported through stdout

### **DataPoint Class**

java

public class DataPoint {

private double x;

private double y;

public DataPoint(double x, double y) {

this.x = x;

this.y = y;

}

*// Getters and setters*

public double getX() { return x; }

public double getY() { return y; }

public void setX(double x) { this.x = x; }

public void setY(double y) { this.y = y; }

*// Copy method*

public DataPoint copy() {

return new DataPoint(this.x, this.y);

}

*// toString method for debugging*

@Override

public String toString() {

return "(" + x + ", " + y + ")";

}

}

**Purpose**: Represents a single data point with x and y coordinates.

**Methods**:

* Constructor: Creates a new data point with given x and y values
* Getters and setters: Access and modify x and y values
* copy(): Creates a new DataPoint with the same x and y values
* toString(): Returns a string representation for debugging

**Notes**:

* Used throughout the PSS pipeline to represent data points
* Simple data structure with no complex behavior

### **Smoother Class**

java

public class Smoother {

*// Constants for the Solter smoothing algorithm*

private static final int DEFAULT\_WINDOW\_SIZE = 5;

private static final double ALPHA = 0.3; *// Controls distance weight decay*

private static final double BETA = 0.7; *// Controls overall smoothing intensity*

*/\*\**

\* Apply the Solter smoothing algorithm to a list of data points.

\*

*\* @param data List of data points to smooth*

*\* @return List of smoothed data points*

*\*/*

public List<DataPoint> applySmoothing(List<DataPoint> data) {

return applySmoothing(data, DEFAULT\_WINDOW\_SIZE, ALPHA, BETA);

}

*// Other smoothing methods...*

}

**Purpose**: Implements the Solter smoothing algorithm and other smoothing methods.

**Methods**:

* applySmoothing(List<DataPoint> data): Applies smoothing with default parameters
* applySmoothing(List<DataPoint> data, int windowSize, double alpha, double beta): Applies smoothing with custom parameters

**Process**:

1. Creates a new list to store smoothed data points
2. Gets the number of data points
3. Handles the edge case where there are 0 or 1 data points (simply returns a copy)
4. Ensures the window size is valid (not larger than the dataset)
5. Calculates half the window size for later use

**Notes**:

* Creating a copy of the original data points ensures we don't modify the original data
* The window size validation prevents errors with small datasets

#### **applySmoothing - Main Smoothing Loop**

java

*// For each point in the dataset*

for (int i = 0; i < n; i++) {

DataPoint originalPoint = data.get(i);

double x = originalPoint.getX();

double originalY = originalPoint.getY();

double smoothedY;

*// For points at the edges, use smaller windows*

int start = Math.max(0, i - halfWindow);

int end = Math.min(n - 1, i + halfWindow);

*// Apply the Solter smoothing algorithm*

if (start == end) {

smoothedY = originalY; *// Only one point, no smoothing*

} else {

*// Continue with weighted average calculation...*

**Purpose**: Loops through each point in the dataset and prepares for smoothing.

**Process**:

1. For each data point in the dataset:
   * Retrieves the original point
   * Extracts the x and y values
   * Calculates the window boundaries (start and end indices)
   * Handles the edge case where the window contains only one point

**Notes**:

* Using Math.max and Math.min ensures window boundaries don't exceed the dataset limits
* This handles points at the beginning and end of the dataset gracefully
* The x-value is preserved in the smoothed data

#### **applySmoothing - Weight Calculation and Averaging**

java

*// Calculate weights and sum for weighted average*

double sum = 0;

double totalWeight = 0;

for (int j = start; j <= end; j++) {

double y = data.get(j).getY();

double distance = Math.abs(i - j);

double weight = Math.exp(-distance \* alpha) \* beta;

*// We increase the central point's weight*

if (j == i) {

weight \*= 1.5;

}

sum += y \* weight;

totalWeight += weight;

}

*// Calculate weighted average*

smoothedY = sum / totalWeight;

*// Create a new point with the smoothed value*

smoothedData.add(new DataPoint(x, smoothedY));

**Purpose**: Calculates the smoothed y-value using a weighted average.

**Process**:

1. Initializes variables for the weighted sum and total weight
2. For each point in the window:
   * Gets the y-value
   * Calculates the distance from the central point
   * Calculates the weight using an exponential decay function
   * Increases the weight of the central point
   * Adds the weighted y-value to the sum
   * Adds the weight to the total weight
3. Calculates the weighted average
4. Creates a new data point with the original x-value and the smoothed y-value

**Notes**:

* The exponential decay function (Math.exp(-distance \* alpha)) gives more weight to closer points
* The alpha parameter controls how quickly the weight decreases with distance
* The beta parameter scales all weights to control overall smoothing intensity
* Increasing the central point's weight helps preserve important features

### **Salter Class**

The Salter class implements various data salting methods for enhancing patterns. Let's examine it in parts:

#### **Class Definition and Constants**

java

public class Salter {

*// Random number generator for adding controlled variability*

private final Random random = new Random();

*// Constants for different salting methods*

private static final double DEFAULT\_TREND\_INTENSITY = 0.05;

private static final double DEFAULT\_CYCLE\_INTENSITY = 0.1;

private static final double DEFAULT\_RANDOM\_INTENSITY = 0.15;

*// Additional constants and parameters...*

**Purpose**: Defines the Salter class and its constants.

**Components**:

* Random number generator for controlled randomness
* Constants for default salting intensities for different data types
* Other configuration parameters

**Notes**:

* Using a class-level Random instance is more efficient than creating new ones
* Different salting intensities are used for different data characteristics

#### **applySalting Method - Main Implementation**

java

*/\*\**

\* Apply salting to a list of data points.

\* Salting adds controlled variability to enhance patterns.

\*

*\* @param data List of data points to salt*

*\* @return List of salted data points*

*\*/*

public List<DataPoint> applySalting(List<DataPoint> data) {

*// Create a list for salted data*

List<DataPoint> saltedData = new ArrayList<>();

*// Analyze data characteristics*

DataCharacteristics characteristics = analyzeData(data);

*// Apply appropriate salting based on characteristics*

if (characteristics.hasTrend()) {

return applyTrendSalting(data, DEFAULT\_TREND\_INTENSITY);

} else if (characteristics.hasCycles()) {

return applyCycleSalting(data, DEFAULT\_CYCLE\_INTENSITY);

} else {

return applyRandomSalting(data, DEFAULT\_RANDOM\_INTENSITY);

}

}

**Purpose**: Main method that applies the appropriate salting technique based on data characteristics.

**Process**:

1. Creates a new list for salted data
2. Analyzes the data to determine its characteristics (trend, cycles, etc.)
3. Applies the appropriate salting method based on the characteristics:
   * Trend salting for data with trends
   * Cycle salting for data with cyclical patterns
   * Random salting for data with no clear patterns

**Notes**:

* The method delegates to specialized salting techniques
* Default intensities are used for each salting type
* Data characteristics determine the salting approach

#### **analyzeData Method**

java

*/\*\**

\* Analyze data to detect trends, cycles, and other characteristics.

\*

*\* @param data List of data points to analyze*

*\* @return DataCharacteristics object with analysis results*

*\*/*

private DataCharacteristics analyzeData(List<DataPoint> data) {

DataCharacteristics characteristics = new DataCharacteristics();

*// Detect trend*

double trendStrength = calculateTrendStrength(data);

characteristics.setTrendStrength(trendStrength);

*// Detect cycles*

double[] cyclicProperties = detectCycles(data);

characteristics.setCyclicStrength(cyclicProperties[0]);

characteristics.setCycleLength(cyclicProperties[1]);

*// Calculate noise level*

double noiseLevel = calculateNoiseLevel(data);

characteristics.setNoiseLevel(noiseLevel);

return characteristics;

}

**Purpose**: Analyzes the data to detect trends, cycles, and other characteristics.

**Process**:

1. Creates a new DataCharacteristics object
2. Calculates trend strength using linear regression
3. Detects cycles using autocorrelation or other techniques
4. Calculates noise level
5. Returns the characteristics object with all analysis results

**Notes**:

* This analysis guides the salting process
* Different characteristics lead to different salting approaches
* The DataCharacteristics class stores multiple properties of the data

#### **applyTrendSalting Method**

java

*/\*\**

\* Apply salting to data with trend characteristics.

\*

*\* @param data List of data points*

*\* @param intensity Intensity of the salting effect*

*\* @return List of salted data points*

*\*/*

private List<DataPoint> applyTrendSalting(List<DataPoint> data, double intensity) {

List<DataPoint> saltedData = new ArrayList<>();

int n = data.size();

*// Calculate trend parameters*

\*

\*

\*

return saltedData;

}

**Purpose**: Applies salting to data with trend characteristics.

**Process**:

1. Creates a new list for salted data
2. Calculates trend parameters (slope and intercept)
3. For each data point:
   * Calculates the trend component at this point
   * Adds controlled variability based on the deviation from the trend
   * Creates a new data point with the salted y-value

**Notes**:

* Using Gaussian randomness (random.nextGaussian()) provides more natural variation
* The intensity parameter controls the strength of the salting effect
* Salting is proportional to the deviation from the trend

#### **applyCycleSalting Method**

private List<DataPoint> applyCycleSalting(List<DataPoint> data, double intensity) {

List<DataPoint> saltedData = new ArrayList<>();

int n = data.size();

*// Get cycle properties from data analysis*

*// Apply cycle-based salting*

for (int i = 0; i < n; i++) {

DataPoint point = data.get(i);

double x = point.getX();

double y = point.getY();

*// Calculate phase within the cycle*

double phase = (x % cycleLength) / cycleLength;

*// Add controlled variability based on cycle phase*

double variability = intensity \* Math.sin(2 \* Math.PI \* phase) \* y \* 0.1;

*// Create salted point*

double saltedY = y + variability;

saltedData.add(new DataPoint(x, saltedY));

}

return saltedData;

}

**Purpose**: Applies salting to data with cyclic characteristics.

**Process**:

1. Creates a new list for salted data
2. Gets cycle properties from data analysis
3. For each data point:
   * Calculates the phase within the cycle
   * Adds controlled variability based on the cycle phase
   * Creates a new data point with the salted y-value

**Notes**:

* Using sine function adds periodic variation aligned with the detected cycle
* The intensity parameter controls the strength of the salting effect
* The cycleLength parameter determines the frequency of the added variation

#### **applyRandomSalting Method**

java

private List<DataPoint> applyRandomSalting(List<DataPoint> data, double intensity) {

List<DataPoint> saltedData = new ArrayList<>();

int n = data.size();

*// Apply random salting*

for (int i = 0; i < n; i++) {

DataPoint point = data.get(i);

double x = point.getX();

double y = point.getY();

*// Add controlled random variability*

double variability = range \* intensity \* random.nextGaussian() \* 0.1;

*// Create salted point*

double saltedY = y + variability;

saltedData.add(new DataPoint(x, saltedY));

}

return saltedData;

}

**Purpose**: Applies salting to data with no clear trend or cyclic patterns.

**Process**:

1. Creates a new list for salted data
2. Calculates the data range for scaling the variability
3. For each data point:
   * Adds controlled random variability scaled by the data range
   * Creates a new data point with the salted y-value

**Notes**:

* Using Gaussian randomness (random.nextGaussian()) provides more natural variation
* Scaling by the data range ensures appropriate variability relative to the data
* The intensity parameter controls the strength of the salting effect

### **PSSProcessor Class**

The PSSProcessor class orchestrates the entire PSS pipeline. Let's break it down:

#### **Class Definition and Fields**

java

public class PSSProcessor {

*// Component instances*

private final Plotter plotter;

private final Salter salter;

private final Smoother smoother;

private final GraphGenerator graphGenerator;

*/\*\**

\* Create a new PSS Processor with default components.

*\*/*

public PSSProcessor() {

this.plotter = new Plotter();

this.salter = new Salter();

this.smoother = new Smoother();

this.graphGenerator = new GraphGenerator();

}

**Purpose**: Defines the PSSProcessor class and initializes its components.

**Components**:

* Plotter: Handles the initial data visualization
* Salter: Implements data salting techniques
* Smoother: Implements smoothing algorithms
* GraphGenerator: Creates the final comparative visualization

**Notes**:

* Each component is responsible for a stage in the PSS pipeline
* Using separate classes follows the single responsibility principle

#### **process Method**

java

public PSSResult process(List<DataPoint> data, String outputDir) throws IOException {

*// Create result object*

PSSResult result = new PSSResult();

result.setOriginalData(data);

*// Stage 1: Plot*

File initialPlot = plotter.createPlot(data, new File(outputDir, "initial\_plot.png"), "Original Data");

result.setInitialPlotPath(initialPlot.getPath());

*// Stage 2: Salt*

List<DataPoint> saltedData = salter.applySalting(data);

File saltedPlot = plotter.createPlot(saltedData, new File(outputDir, "salted\_plot.png"), "Salted Data");

result.setSaltedData(saltedData);

result.setSaltedPlotPath(saltedPlot.getPath());

*// Stage 3: Smooth*

List<DataPoint> smoothedData = smoother.applySmoothing(saltedData);

File smoothedPlot = plotter.createPlot(smoothedData, new File(outputDir, "smoothed\_plot.png"), "Smoothed Data");

result.setSmoothedData(smoothedData);

result.setSmoothedPlotPath(smoothedPlot.getPath());

*// Stage 4: Graph*

List<List<DataPoint>> allSeries = new ArrayList<>();

allSeries.add(data);

allSeries.add(saltedData);

allSeries.add(smoothedData);

List<String> seriesNames = Arrays.asList("Original", "Salted", "Smoothed");

File finalGraph = graphGenerator.createMultiSeriesGraph(

allSeries, seriesNames, new File(outputDir, "final\_plot.png"), "PSS Pipeline Comparison");

result.setFinalGraphPath(finalGraph.getPath());

return result;

}

**Purpose**: Orchestrates the four stages of the PSS pipeline.

**Process**:

1. Creates a result object to store processed data and visualization paths
2. Stage 1 (Plot): Creates an initial plot of the original data
3. Stage 2 (Salt): Applies salting to the data and creates a plot of the salted data
4. Stage 3 (Smooth): Applies smoothing to the salted data and creates a plot of the smoothed data
5. Stage 4 (Graph): Creates a comparative visualization of all three data series

**Parameters**:

* data: The input data points to process
* outputDir: Directory for saving visualizations

**Returns**:

* PSSResult: Object containing all processed data and visualization paths

**Notes**:

* Each stage builds on the previous one
* All intermediate and final results are stored in the PSSResult object
* Visualizations are saved at each stage

## **JavaScript Client**

The JavaScript client handles the user interface and communication with the Flask server. Let's examine key components:

### **Document Ready Handler and Initialization**

javascript

$(document).ready(function() {

*// DOM Elements*

const chatMessages = $('#chat-messages');

const userInput = $('#user-input');

const sendButton = $('#send-button');

const uploadForm = $('#upload-form');

const fileUpload = $('#file-upload');

const uploadButton = $('#upload-button');

const vizContainer = $('#visualization-container');

const vizContent = $('#visualization-content');

const downloadBtn = $('#download-btn');

const rawDataContainer = $('#raw-data-container');

const rawDataContent = $('#raw-data-content');

const toggleRawBtn = $('#toggle-raw-data');

*// Variables*

let currentFile = null;

let dataAnalyzed = false;

let chatHistory = [];

let currentSessionId = null;

*// Initialize - Check for existing session*

checkSessionStatus();

**Purpose**: Sets up the JavaScript client when the page loads.

**Process**:

1. Defines references to DOM elements
2. Initializes variables for tracking state
3. Calls checkSessionStatus() to check for an existing session

**Notes**:

* Uses jQuery for DOM manipulation
* Keeps track of the current session ID
* Stores chat history for potential future features

### **checkSessionStatus Function**

javascript

*// Function to check if there's an active session*

function checkSessionStatus() {

$.ajax({

url: '/api/session',

type: 'GET',

success: function(response) {

if (response.active && response.session\_id) {

currentSessionId = response.session\_id;

dataAnalyzed = response.file\_processed;

console.log("Active session detected: " + currentSessionId);

*// If we have an active session, add a message to inform the user*

if (dataAnalyzed) {

addMessage("I detected you have data already processed. You can ask me to show you plots or explain the data.");

}

}

},

error: function(error) {

console.error("Error checking session status:", error);

}

});

}

**Purpose**: Checks if there's an active session when the page loads.

**Process**:

1. Makes an AJAX request to the /api/session endpoint
2. If an active session is found, updates the currentSessionId variable
3. Updates the dataAnalyzed flag based on the response
4. Adds a message to inform the user if data is already processed

**Notes**:

* This helps maintain session continuity between page refreshes
* Users can continue working with previously processed data

### **addMessage Function**

javascript

*// Add message to chat*

function addMessage(content, isUser = false) {

const messageType = isUser ? 'user' : 'bot';

const messageHtml = `

<div class="message ${messageType}">

<div class="message-content">

<p>${content}</p>

</div>

</div>

`;

chatMessages.append(messageHtml);

chatMessages.scrollTop(chatMessages[0].scrollHeight);

*// Store in chat history*

chatHistory.push({

content: content,

isUser: isUser

});

}

**Purpose**: Adds a message to the chat interface.

**Parameters**:

* content: The message text to display
* isUser: Boolean indicating if the message is from the user (true) or the bot (false)

**Process**:

1. Determines the message type (user or bot)
2. Creates HTML for the message with appropriate styling
3. Appends the message to the chat container
4. Scrolls the chat container to show the new message
5. Stores the message in the chat history array

**Notes**:

* Different styling is applied for user and bot messages
* Auto-scrolling ensures the latest message is visible
* Chat history is stored for potential future features

### **handleUserMessage Function**

javascript

*// Handle user message submission*

function handleUserMessage() {

const message = userInput.val().trim();

if (message === '') return;

*// Add user message to chat*

addMessage(message, true);

userInput.val('');

*// Process message and generate response*

processUserMessage(message);

}

**Purpose**: Handles user message submission from the input field.

**Process**:

1. Gets the message text from the input field
2. Checks if the message is empty (and returns if it is)
3. Adds the message to the chat display
4. Clears the input field
5. Calls processUserMessage() to handle the message

**Notes**:

* Triggered by clicking the send button or pressing Enter
* Prevents sending empty messages

### **processUserMessage Function - Part 1: Initial Checks**

javascript

*// Process user message and generate bot response*

function processUserMessage(message) {

*// Show typing indicator*

showTypingIndicator();

*// If no file has been uploaded yet and session is not active*

if (!currentFile && !currentSessionId && !message.toLowerCase().includes('help')) {

setTimeout(() => {

removeTypingIndicator();

addMessage("Please upload a CSV file first so I can analyze it. You can also ask for help if you need guidance.");

}, 1000);

return;

}

**Purpose**: First part of the function that processes user messages and prepares for sending to the server.

**Process**:

1. Shows a typing indicator to provide visual feedback
2. Checks if there's no active file or session
3. If no file/session is available and the message isn't asking for help, shows a prompt to upload a file
4. Returns early in this case to avoid unnecessary server requests

**Notes**:

* The typing indicator simulates the bot "thinking"
* The timeout creates a more natural conversational flow
* Special handling for 'help' allows users to get guidance even without a file

### **processUserMessage Function - Part 2: Sending the Request**

javascript

*// Prepare data to send to server*

const requestData = {

message: message,

hasFile: !!currentFile || !!currentSessionId, *// Either we have a file or an active session*

fileProcessed: dataAnalyzed

};

*// Always include session ID if we have one*

if (currentSessionId) {

requestData.session\_id = currentSessionId;

console.log("Sending request with session ID: " + currentSessionId);

}

*// Send message to server for processing*

$.ajax({

url: '/api/chat',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(requestData),

**Purpose**: Prepares and sends the message to the server.

**Process**:

1. Creates a requestData object with the message and state flags
2. Adds the session ID to the request if available
3. Sends an AJAX POST request to the /api/chat endpoint with the JSON data

**Notes**:

* The hasFile flag indicates whether a file has been uploaded or a session is active
* The fileProcessed flag indicates whether data analysis is complete
* Including the session ID is crucial for maintaining session continuity

### **processUserMessage Function - Part 3: Handling the Response**

javascript

success: function(response) {

removeTypingIndicator();

*// Update session ID if provided in response*

if (response.session\_id) {

currentSessionId = response.session\_id;

console.log("Session ID updated to: " + currentSessionId);

}

*// Mark data as analyzed if we got a successful response*

dataAnalyzed = true;

*// Add bot response to chat*

addMessage(response.message);

*// Update visualization if available*

if (response.visualization) {

updateVisualization(response.visualization);

}

*// Move upload form below visualization if needed*

$('.chat-input-container').before($('#file-upload-container'));

*// Update raw data if available*

if (response.rawData) {

updateRawData(response.rawData);

}

},

error: function(error) {

removeTypingIndicator();

console.error("Error processing message:", error);

addMessage("Sorry, I encountered an error processing your request. Please try again.");

}

**Purpose**: Handles the server response and updates the UI accordingly.

**Process**:

1. Removes the typing indicator
2. Updates the session ID if provided in the response
3. Marks data as analyzed
4. Adds the bot's response message to the chat
5. Updates the visualization if included in the response
6. Repositions the upload form for better UX
7. Updates the raw data display if included in the response
8. Handles errors by showing an error message

**Notes**:

* Session ID updating ensures consistency with the server
* The UI is updated based on the response content
* Error handling provides feedback to the user when things go wrong

### **updateVisualization Function**

javascript

*// Update visualization*

function updateVisualization(vizData) {

vizContainer.show();

*// If vizData is an image URL*

if (vizData.type === 'image') {

*// Add session ID and timestamp to prevent browser caching*

const timestamp = new Date().getTime();

const urlWithParams = `${vizData.url}?sid=${currentSessionId}&t=${timestamp}`;

vizContent.html(`<img src="${urlWithParams}" alt="Data Visualization">`);

downloadBtn.attr('data-src', vizData.url);

console.log("Updated visualization with URL: " + urlWithParams);

}

*// If vizData is HTML/SVG content*

else if (vizData.type === 'html') {

vizContent.html(vizData.content);

}

}

**Purpose**: Updates the visualization display based on server response.

**Parameters**:

* vizData: Object containing visualization data (type and URL/content)

**Process**:

1. Shows the visualization container
2. If the visualization is an image:
   * Adds session ID and timestamp parameters to prevent caching
   * Updates the HTML with an img tag using the URL
   * Sets the download button's data-src attribute
3. If the visualization is HTML/SVG content:
   * Directly sets the HTML content

**Notes**:

* Cache busting with timestamp and session ID is crucial for showing the correct visualizations
* Supports both image URLs and direct HTML/SVG content
* The download button is updated to point to the current visualization

### **updateRawData Function**

javascript

*// Update raw data display*

function updateRawData(data) {

*// Make sure the container is initially visible*

rawDataContainer.show();

let formattedData = '';

if (typeof data === 'string') {

formattedData = data;

} else if (Array.isArray(data)) {

formattedData = data.map(row => {

if (Array.isArray(row)) {

return row.join(', ');

} else {

return JSON.stringify(row);

}

}).join('\n');

} else {

formattedData = JSON.stringify(data, null, 2);

}

rawDataContent.html(`<pre>${formattedData}</pre>`);

*// Update the toggle button text*

toggleRawBtn.text('Hide Raw Data');

}

**Purpose**: Updates the raw data display based on server response.

**Parameters**:

* data: The raw data to display (string, array, or object)

**Process**:

1. Shows the raw data container
2. Formats the data based on its type:
   * Strings are used as-is
   * Arrays are formatted with line breaks
   * Objects are formatted as JSON with indentation
3. Updates the HTML with a pre-formatted version of the data
4. Updates the toggle button text

**Notes**:

* Handles different data types gracefully
* Uses pre tags for preserving formatting
* Updates the toggle button text to match the current state

## **JavaScript Client (Continued)**

### **File Upload Handling (Continued)**

#### **AJAX File Upload and Processing**

javascript

*// Upload file to server*

$.ajax({

url: '/api/upload',

type: 'POST',

data: formData,

processData: false,

contentType: false,

success: function(response) {

currentFile = file.name;

dataAnalyzed = true;

*// Save session ID from response*

if (response.session\_id) {

currentSessionId = response.session\_id;

console.log("New session created: " + currentSessionId);

}

*// Add success message*

addMessage(response.message);

*// Update visualization if available*

if (response.visualization) {

updateVisualization(response.visualization);

}

*// Update raw data if available*

if (response.rawData) {

updateRawData(response.rawData);

}

},

error: function(error) {

console.error("Error uploading file:", error);

addMessage("Sorry, I encountered an error analyzing your file. Please try again or try a different CSV file.");

}

});

**Purpose**: Sends the file to the server and handles the response.

**Process**:

1. Configures the AJAX request with the FormData object
2. Sets processData and contentType to false (required for file uploads)
3. On success:
   * Updates the current file name
   * Marks data as analyzed
   * Saves the session ID from the response
   * Adds the success message to the chat
   * Updates visualization if available
   * Updates raw data if available
4. On error:
   * Logs the error to the console
   * Shows an error message in the chat

**Notes**:

* The processData: false and contentType: false settings are required for proper file upload
* Session ID is stored for future requests
* Visualization and raw data are updated based on the server response

### **File Input Change Handler**

javascript

*// Handle file input change (update selected filename)*

fileUpload.on('change', function() {

const fileInput = fileUpload[0];

if (fileInput.files.length > 0) {

const fileName = fileInput.files[0].name;

$('.custom-file-upload').text(`Selected: ${fileName}`);

} else {

$('.custom-file-upload').html('<span class="upload-icon">📁</span> Choose CSV File');

}

});

**Purpose**: Updates the file input label when a file is selected.

**Process**:

1. Checks if a file has been selected
2. If a file is selected, updates the label to show the file name
3. If no file is selected, resets the label to the default text

**Notes**:

* Provides visual feedback about the selected file
* Improves user experience by confirming file selection

### **Raw Data Toggle**

javascript

*// Toggle raw data view*

toggleRawBtn.on('click', function() {

*// Get direct reference to the DOM element*

const rawContainer = document.getElementById('raw-data-container');

*// Toggle display directly on the DOM element*

if (rawContainer.style.display === 'none' || rawContainer.style.display === '') {

rawContainer.style.display = 'block';

this.textContent = 'Hide Raw Data';

} else {

rawContainer.style.display = 'none';

this.textContent = 'Show Raw Data';

}

});

**Purpose**: Toggles the visibility of the raw data container.

**Process**:

1. Gets a direct reference to the raw data container DOM element
2. Checks its current display state
3. If hidden, shows the container and updates the button text
4. If visible, hides the container and updates the button text

**Notes**:

* Uses direct DOM manipulation for reliable display toggling
* Updates button text to match the current state
* Improves user experience by allowing them to hide/show data as needed

### **Download Button Handler**

javascript

*// Handle download button click*

downloadBtn.on('click', function() {

const imgSrc = $(this).attr('data-src');

if (imgSrc) {

*// Add session ID to download URL to ensure correct image*

const downloadUrl = imgSrc.includes('?')

? imgSrc + `&sid=${currentSessionId}`

: imgSrc + `?sid=${currentSessionId}`;

const link = document.createElement('a');

link.href = downloadUrl;

link.download = 'nimbus\_visualization.png';

document.body.appendChild(link);

link.click();

document.body.removeChild(link);

} else {

addMessage("Sorry, there's no visualization available to download.");

}

});

**Purpose**: Handles the download button click for saving visualizations.

**Process**:

1. Gets the image source URL from the data-src attribute
2. Adds the session ID as a URL parameter for session-specific downloads
3. Creates a temporary download link
4. Sets the download attribute to trigger a file download
5. Programmatically clicks the link and then removes it
6. Shows an error message if no visualization is available

**Notes**:

* Adding the session ID ensures the correct image is downloaded
* Using the download attribute forces a download rather than opening the image
* The temporary link approach works in most modern browsers
* Error handling prevents user confusion if no visualization is available

### **Event Listeners Setup**

javascript

*// Event Listeners*

sendButton.on('click', handleUserMessage);

userInput.on('keypress', function(e) {

if (e.which === 13) {

handleUserMessage();

}

});

*// Initial greeting message is already in the HTML*

**Purpose**: Sets up event listeners for user interactions.

**Process**:

1. Adds a click event listener to the send button
2. Adds a keypress event listener to the input field (for Enter key)
3. Both event listeners call the handleUserMessage function

**Notes**:

* Supporting both button clicks and Enter key presses improves usability
* The initial greeting message is defined in the HTML rather than added via JavaScript

### **Typing Indicator Functions**

javascript

*// Show typing indicator*

function showTypingIndicator() {

const typingHtml = `

<div class="message bot typing-indicator">

<div class="message-content">

<p><span class="dot"></span><span class="dot"></span><span class="dot"></span></p>

</div>

</div>

`;

chatMessages.append(typingHtml);

chatMessages.scrollTop(chatMessages[0].scrollHeight);

}

*// Remove typing indicator*

function removeTypingIndicator() {

$('.typing-indicator').remove();

}

**Purpose**: Manages the typing indicator that shows when the bot is "thinking".

**Process**:

* showTypingIndicator:
  1. Creates HTML for the typing indicator with animated dots
  2. Appends it to the chat messages container
  3. Scrolls to make it visible
* removeTypingIndicator:
  1. Removes the typing indicator element from the DOM

**Notes**:

* The typing indicator provides visual feedback during processing
* The animated dots are styled with CSS animations
* Removing the indicator happens when the response is ready

### **checkRawData Function**

javascript

*// Function to check raw data*

function checkRawData() {

if (currentSessionId) {

$.ajax({

url: '/api/chat',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify({

message: "show raw data",

hasFile: true,

fileProcessed: true,

session\_id: currentSessionId

}),

success: function(response) {

console.log("Raw data response:", response);

*// Check if we got any raw data*

if (response.rawData) {

console.log("Raw data received. First 100 chars:",

response.rawData.substring(0, 100));

$('#raw-data-content').html(`<pre>${response.rawData}</pre>`);

$('#raw-data-container').css('display', 'block');

console.log("Raw data container should now be visible");

} else {

console.log("No raw data in response");

}

},

error: function(err) {

console.error("Error fetching raw data:", err);

}

});

} else {

console.log("No session ID available - can't fetch raw data");

}

}

**Purpose**: Utility function to fetch and display raw data from the server.

**Process**:

1. Checks if a session ID is available
2. If available, sends a request to the /api/chat endpoint with a "show raw data" message
3. On success, checks if raw data was received
4. If raw data is available, updates the raw data container and makes it visible
5. Logs detailed information for debugging

**Notes**:

* This function can be called programmatically to fetch raw data
* It's primarily used for debugging or when raw data display is needed
* Uses jQuery selectors rather than the stored DOM references for flexibility

## **Integration Points**

This section documents how the different components of the Nimbus AI PSS system interact.

### **Flask to Java Integration**

The Flask application integrates with the Java processing engine through the process\_csv\_with\_java function:

python

def process\_csv\_with\_java(file\_path, session\_id):

"""Process the CSV file with the Java components for plot, salt, smooth, and graph"""

output\_dir = os.path.join(app.config['VISUALIZATIONS\_DIR'], session\_id)

os.makedirs(output\_dir, exist\_ok=True)

output\_data = os.path.join(app.config['TEMP\_DIR'], f"{session\_id}\_processed\_data.csv")

*# Run the Java application*

cmd = [

app.config['JAVA\_BIN'],

'-cp',

os.pathsep.join([

os.path.join('build', 'lib', 'NimbusAI-PSS.jar'),

os.path.join('build', 'lib', 'jfreechart-1.5.3.jar'),

os.path.join('build', 'lib', 'jcommon-1.0.23.jar'),

]),

'Main',

file\_path,

output\_dir,

output\_data

]

*# Execute the Java application using subprocess*

result = subprocess.run(cmd, capture\_output=True, text=True)

**Integration Process**:

1. Flask prepares session-specific output directories
2. The Java classpath is constructed with all required JAR files
3. Command-line arguments are passed to the Java application:
   * Input CSV file path
   * Output directory for visualizations
   * Output CSV file path for processed data
4. The Java application is executed using Python's subprocess module
5. Standard output and error are captured for logging and error handling
6. The Java application generates visualizations and processed data
7. Flask reads the processed data and returns file paths to the client

**Key Considerations**:

* Each session has its own output directories to prevent conflicts
* Error handling captures both Java exceptions and process execution errors
* The Java application is responsible for:
  + Reading the input CSV
  + Running the PSS pipeline
  + Generating visualizations
  + Writing processed data to CSV
* Flask is responsible for:
  + Managing sessions
  + Handling HTTP requests
  + Coordinating processing
  + Serving files to the client

### **Flask to JavaScript Integration**

The Flask application and JavaScript client communicate through RESTful API endpoints:

#### **API Endpoint: /api/upload**

**Client to Server**:

* HTTP Method: POST
* Content Type: multipart/form-data
* Payload: CSV file

**Server to Client**:

* Content Type: application/json

Payload:  
 json  
{

"message": "Success message",

"visualization": {

"type": "image",

"url": "/visualizations/{session\_id}/final\_plot.png"

},

"session\_id": "{session\_id}"

* }

#### **API Endpoint: /api/chat**

**Client to Server**:

* HTTP Method: POST
* Content Type: application/json

Payload:  
 json  
{

"message": "User message",

"session\_id": "{session\_id}",

"hasFile": true,

"fileProcessed": true

* }

**Server to Client**:

* Content Type: application/json

Payload:  
 json  
{

"message": "Bot response",

"visualization": {

"type": "image",

"url": "/visualizations/{session\_id}/{visualization\_file}.png"

},

"rawData": "Optional data for display",

"session\_id": "{session\_id}"

* }

#### **API Endpoint: /api/session**

**Client to Server**:

* HTTP Method: GET

**Server to Client**:

* Content Type: application/json

Payload:  
 json  
{

"active": true,

"session\_id": "{session\_id}",

"file\_processed": true

* }

**Key Considerations**:

* Session ID is maintained across all requests for continuity
* Visualization URLs are relative and include the session ID
* Error responses include appropriate HTTP status codes and error messages
* The client is responsible for adding cache-busting parameters to visualization URLs
* The server includes appropriate headers to prevent caching of dynamic content

### **Java Classes Integration**

The Java components integrate through the PSSProcessor class:

java

public PSSResult process(List<DataPoint> data, String outputDir) throws IOException {

*// Create result object*

PSSResult result = new PSSResult();

result.setOriginalData(data);

*// Stage 1: Plot*

File initialPlot = plotter.createPlot(data, new File(outputDir, "initial\_plot.png"), "Original Data");

result.setInitialPlotPath(initialPlot.getPath());

*// Stage 2: Salt*

List<DataPoint> saltedData = salter.applySalting(data);

File saltedPlot = plotter.createPlot(saltedData, new File(outputDir, "salted\_plot.png"), "Salted Data");

result.setSaltedData(saltedData);

result.setSaltedPlotPath(saltedPlot.getPath());

*// Stage 3: Smooth*

List<DataPoint> smoothedData = smoother.applySmoothing(saltedData);

File smoothedPlot = plotter.createPlot(smoothedData, new File(outputDir, "smoothed\_plot.png"), "Smoothed Data");

result.setSmoothedData(smoothedData);

result.setSmoothedPlotPath(smoothedPlot.getPath());

*// Stage 4: Graph*

List<List<DataPoint>> allSeries = new ArrayList<>();

allSeries.add(data);

allSeries.add(saltedData);

allSeries.add(smoothedData);

List<String> seriesNames = Arrays.asList("Original", "Salted", "Smoothed");

File finalGraph = graphGenerator.createMultiSeriesGraph(

allSeries, seriesNames, new File(outputDir, "final\_plot.png"), "PSS Pipeline Comparison");

result.setFinalGraphPath(finalGraph.getPath());

return result;

}

**Integration Process**:

1. Each stage builds on the previous one in a sequential pipeline
2. The PSSProcessor orchestrates the workflow but delegates specific tasks to specialized classes:
   * Plotter: Creates individual visualizations for each stage
   * Salter: Applies salting to the original data
   * Smoother: Applies smoothing to the salted data
   * GraphGenerator: Creates the final comparative visualization
3. The PSSResult class collects all intermediate and final results
4. File paths are standardized for consistent access by the Flask application

**Key Considerations**:

* Each component has a single responsibility
* Data flows through the pipeline in a controlled manner
* Intermediate results are preserved for reference
* Standard file naming conventions simplify integration with Flask
* Exceptions are propagated to the Main class for proper error handling

## **Conclusion**

The Nimbus AI PSS system demonstrates a well-structured hybrid architecture:

1. **Separation of Concerns**:
   * Flask handles HTTP and session management
   * Java handles the core data processing pipeline
   * JavaScript handles the user interface and client-side logic
2. **Modularity**:
   * Each component can be developed and tested independently
   * The APIs between components are well-defined
   * Future enhancements can target specific components without affecting others
3. **Scalability**:
   * Session-based processing enables multi-user support
   * File-based storage simplifies deployment and scaling
   * Asynchronous processing prevents UI blocking
4. **Extensibility**:
   * New smoothing algorithms can be added to the Smoother class
   * New salting techniques can be added to the Salter class
   * New visualization types can be added to the Plotter and GraphGenerator classes