1. HTML/CSS Markup

Create an HTML file named index.html and a CSS file named style.css. Add the following HTML code to the index.html file:

php

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Frontend App</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="container">
    <h1>My Frontend App</h1>
    <form id="data-form">
      <label for="interval">Interval (in seconds):</label>
      <input type="number" id="interval" name="interval" required min="1">
      <button type="submit">Start</button>
    </form>
    <div id="status"></div>
  </div>
  <script src="script.js"></script>
</body>
</html>
```

Add the following CSS code to the style.css file:

2. CSS

```css
.container {
   margin: auto;
   width: 80%;
   text-align: center;
}
h1 {
   margin-top: 50px;
}

form {
   margin-top: 30px;
}

label {
   margin-right: 10px;
}

input {
   width: 80px;
}

button {
   margin-left: 10px;
}

#status {
   margin-top: 30px;
   font-weight: bold;
}
```

JavaScript Code

Create a JavaScript file named script.js. Add the following code to the file:

3. javascript

```
const dataForm = document.getElementById('data-form');
const statusDiv = document.getElementById('status');

dataForm.addEventListener('submit', (event) => {
  event.preventDefault();
  const interval = parseInt(document.getElementById('interval').value);
  statusDiv.innerHTML = 'Collecting data...';
  setInterval(() => {
    fetch('/data')
      .then(response => response.json())
      .then(data => {
        statusDiv.innerHTML = 'Data collected!';
        console.log(data);
      })
      .catch(error => {
        statusDiv.innerHTML = 'Error collecting data';
        console.error(error);
      });
  }, interval * 1000);
});
```

Flask App

Create a Python file named app.py. Add the following code to the file:

4. kotlin

```
from flask import Flask, jsonify
```

```
import time

app = Flask(_name_)

@app.route('/data')
def get_data():
    # Collect data
    data = collect_data()
    # Return data as JSON response
    return jsonify(data)

def collect_data():
    # Code for collecting data goes here
    time.sleep(2) # Simulating data collection delay
    data = {'message': 'Data collected!'}
    return data

if _name_ == '_main_':
    app.run()
```

Running the App

Open a terminal and navigate to the directory where your files are stored. Run the following command to start the Flask app:

5. arduino

```
export FLASK_APP=app.py
flask run
```

Open a web browser and go to http://localhost:5000. Fill out the form and click the "Start" button. You should see a message that says "Collecting data..." and then "Data collected!" every n seconds (where n is the value you entered in the form). You can check the console log in the web browser to see the data that was collected.

Note: This is just a basic example and may need to be modified based on your specific requirements.

Here's some sample backend code for the proposed solution:

Data Ingestion Component:

1.Endpoint Security Providers

```
# Import necessary libraries
import requests
import json
import time

# Set API endpoint URL for endpoint security provider
api_url = "https://example.com/api"

# Define function to collect telemetry data from endpoint security provider
def collect_data():
    # Make API request to collect data
    response = requests.get(api_url)
    # Parse response JSON
    data = json.loads(response.text)
    # Return data
    return data

# Define function to continuously collect data at a set interval
def continuously_collect_data(interval):
    while True:
        # Collect data
        data = collect_data()
        # Store data in Apache Kafka
        # ...
        # Wait for interval seconds before collecting data again
        time.sleep(interval)
```

2.Apache kafka

```
# Import necessary libraries
from kafka import KafkaProducer

# Set up Kafka producer
producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

# Define function to publish data to Kafka
def publish_data(data):
    # Convert data to bytes
    data_bytes = json.dumps(data).encode('utf-8')
    # Publish data to Kafka
    producer.send('raw-data', value=data_bytes)
```

3.Kafka connect

```
# Install Kafka Connect connector plugin for endpoint security provider
# ...

# Configure Kafka Connect connector for endpoint security provider
connector_config = {
    'name': 'endpoint-security-provider',
    'config': {
        'connector.class': 'com.example.EndpointSecurityProviderConnector',
        'tasks.max': '1',
        'topics': 'raw-data',
        'api.url': 'https://example.com/api',
        # ...
    }
}
```

```
# Start Kafka Connect connector for endpoint security provider
# ...
```

4.Apache Cassandra

```
# Import necessary libraries
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider

# Set up Cassandra cluster
auth = PlainTextAuthProvider(username='user', password='password')
cluster = Cluster(['localhost'], auth_provider=auth)

# Set up Cassandra session
session = cluster.connect()

# Define function to store data in Cassandra
def store_data(data):
   # Prepare Cassandra query
   query = "INSERT INTO telemetry_data (id, data) VALUES (?, ?)"
   # Execute query with data
   session.execute(query, (data['id'], json.dumps(data)))
```

5.Data Pre-processing

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Set up Spark session
spark = SparkSession.builder.appName('data-preprocessing').getOrCreate()
```

```
# Define function to pre-process data
def preprocess_data(data):
    # Convert data to Spark DataFrame
    df = spark.createDataFrame([data])
    # Clean data
    df = df.fillna('')
    # Normalize data
    df = df.withColumn('normalized_col', lower(col('raw_col')))
    # Perform feature engineering
    df = df.withColumn('feature1', col('col1') + col('col2'))
    # Return pre-processed data as Python dictionary
    preprocessed_data = df.first().asDict()
    return preprocessed_data
```

6.Data Transformation

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Assume 'df' is the preprocessed DataFrame obtained from previous steps

# Define the numerical and categorical feature columns
numerical_cols = ['age', 'income']
categorical_cols = ['gender', 'marital_status', 'education']

# Apply standardization to numerical features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(df[numerical_cols])
df_scaled = pd.DataFrame(scaled_features, columns=numerical_cols)
```

```
# Apply one-hot encoding to categorical features
encoder = OneHotEncoder(handle_unknown='ignore')
encoded_features = encoder.fit_transform(df[categorical_cols])
df_encoded = pd.DataFrame(encoded_features.toarray(),
columns=encoder.get_feature_names(categorical_cols))

# Combine the scaled and encoded features
transformed_df = pd.concat([df_scaled, df_encoded], axis=1)

# View the transformed DataFrame
print(transformed_df.head())
```

7. Split the data into training and validation sets:

JavasCript

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
```

8. Scale the data using the MinMaxScaler:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
```

9.Train the neural network model:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

model = Sequential()
```

```
model.add(Dense(128, activation='relu', input_dim=X_train.shape[1]))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0001),
metrics=['accuracy'])


history = model.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val), batch_size=32,
epochs=50, verbose=1)
```

10. Evaluate the model on the test set:

```
X_test_scaled = scaler.transform(X_test)


test_loss, test_acc = model.evaluate(X_test_scaled, y_test)


print('Test accuracy:', test_acc)
```

11. Make predictions on new data:

```
new_data = [[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]]

new_data_scaled = scaler.transform(new_data)


prediction = model.predict(new_data_scaled)


print('Prediction:', prediction)
```

Note that you will need to import the necessary libraries at the beginning of your code to be able to use the functions and classes in the above code snippets.