Deployment Solution Architecture: System Overview

Major Components

1) FastAPI Backend
    - Purpose: Acts as the backend service (hosted on Render) to process requests. It validates the input, runs the ML model inference (using a pre-trained autoencoder for dimensionality reduction and an XGBoost classifier for predictions), and returns JSON responses.
    - Inputs: JSON payloads (typically via POST requests) containing user input features.
    - Outputs: JSON responses with prediction results ("prediction": 0} or {"prediction": 1}).

2) Streamlit Frontend
    - Purpose: Provides an interactive user interface (hosted on Streamlit Community Cloud) where users enter feature values, view predictions, and interact with data visualizations.
    - Inputs: User inputs via forms.
    - Outputs: Display predictions and visual feedback based on responses from the FastAPI backend.

3) Machine Learning Model

    - Components:
        - ☐ An autoencoder for feature reduction (built using TensorFlow/Keras).
        - ☐ An XGBoost classifier that makes the final prediction.
    - Lifecycle: The model will be trained locally and saved as artifacts (autoencoder.keras, xgb_model.pkl for XGBoost model, feature_names.pkl for required features, scaler.pkl for scalar function, etc.).
    - Over time, the model will be retrained with new data if needed.

4) Data Storage

    - Inputs for Training: Training and test datasets containing benign and pathogenic variants will be saved as CSV files and stored in an allocated project GitHub repository.
    - Artifacts: Pre-trained model files (autoencoder, scaler, XGBoost model), and feature metadata (feature names, columns that will be dropped due to low variance) will be stored in an allocated GitHub repository.
    - Usage: These files will be loaded by the FastAPI backend during startup for inference.

Data Flow Between Components

1) User Input and Request Submission
    - Users will fill in the required fields in the Streamlit app.
    - The Streamlit frontend will then construct a JSON payload and send a POST request to the FastAPI backend (using the public URL that will be provided by Render).

2) Backend Processing

    - FastAPI will receive the request, validate the input using Pydantic models, scale and process the data through the autoencoder to reduce dimensionality, and then pass the reduced data to the XGBoost classifier for prediction.
    - The backend will return the prediction as a JSON response.

3) UI Update

- The Streamlit app will receive the JSON response and update the UI accordingly displaying the prediction 0 for Benign and 1 for Pathogenic.

Model Lifecycle

☐ Initial Training

The model (autoencoder and XGBoost) will be trained using the saved datasets. Once trained, the model artifacts (feature metadata and scaler) will be stored in the GitHub repository.

☐ Retraining Frequency

Depending on the availability of new labeled data, retraining might occur at fixed intervals (monthly or quarterly) or triggered by data updates.

☐ Data Required

New labeled datasets that include updated features and correct classification (benign or pathogenic) will be required for retraining.

☐ Evaluation

After retraining, the model will be evaluated using metrics like recall, precision, accuracy, and confusion matrices. If performance meets predefined thresholds, the new model will be approved for deployment.

☐ Deployment

Once validated, the retrained model will be saved as new artifacts. These new artifacts will then be committed to the GitHub repository, after which the FastAPI service will be restarted to load the updated model.

☐ Artifact Management

Model artifacts (autoencoder.keras, xgb_model.pkl, scaler.pkl, and the feature metadata) will be versioned and will be stored in the repository. This is to ensure reproducibility and allow rollback if necessary.

Monitoring, Debugging, and Error Handling

1) System Monitoring
   - Logging: Both FastAPI and Streamlit produce logs that will be monitored in Render and Streamlit Community Cloud dashboards.
   - Health Checks: Regular health checks (HTTP ping requests) will be performed to ensure that the backend service is responding.
   - Performance Metrics: Response times, error rates, and system resource usage will be monitored.

2) Debugging:
   - Logs from Render and Streamlit dashboards will be reviewed regularly.
   - Will implement interactive debuggers locally before deployment.
   - Will implement exception handling in FastAPI (using middleware or try/except blocks) to capture and log errors.

3) Handling Unexpected Errors:

   - Will implement fallback mechanisms (default error responses) and proper exception logging.
   - Will design the system to be stateless where possible, allowing easy restart or redeployment.
   - Will utilize cloud monitoring services (Render's built-in monitoring) to alert me to issues.

Required Tools and Technologies

1) Backend:
    - FastAPI: For building the RESTful API.
    - Uvicorn: ASGI server to run FastAPI.
    - Pydantic: Underlying frameworks for web and validation.

2) Frontend:
    - Streamlit: For building interactive data applications.

3) Machine Learning:
    - TensorFlow/Keras: For building and training the autoencoder.
    - XGBoost: For the classification model.
    - Scikit-learn: For data preprocessing (StandardScaler).

4) Data Storage:
    - GitHub Repository: For storing code, datasets, and model artifacts.

5) Deployment:
    - Render: For hosting the FastAPI backend.
    - Streamlit Community Cloud: For hosting the Streamlit frontend.

6) Other:
    - Joblib and Pickle: For saving/loading model artifacts.
    - Logging: For system monitoring and debugging.

Estimated Implementation Cost and Time

1) Computing Resources
    - FastAPI will be hosted on Render using Render's free tiers since this can handle moderate loads. I will consider the monthly $7–$20 plan, depending on future traffic.
    - Streamlit Community Cloud is free to the public with paid options available for private use. Since my target is the public, I will maintain the free option.

2) Time
    - Development and testing might take a few weeks depending on complexity.
    - Model retraining and validation can be scheduled (monthly or quarterly) with minimal overhead.