# Phase 1: Dataset Processing

```
def prepare_input_data(path):
  raw = pd.read_csv(path)
  raw["Label"] = raw["Label"].map({"BENIGN": 0}).fillna(1).astype(int)

# Isolate numeric attributes using boolean masking
numeric_mask = raw.applymap(np.isreal).all()
numeric_data = raw.loc[:, numeric_mask]

# Cleanse data: substitute infinite values with NaN, then fill missing entries with zero
cleaned_data = numeric_data.applymap(lambda x: 0 if pd.isna(x) or x in [np.inf, -np.inf] else x)

  labels = numerical_only.pop("Label").values
  features = numerical_only.values

  scaled = RobustScaler().fit_transform(features)
  reshaped = scaled[:, np.newaxis, :]

  return reshaped, labels, numerical_only.columns
```

# Phase 2: Handling Imbalanced Distributions

```
def resample_data(data_3d, targets):
  flat_data = data_3d[:, 0, :]
  sampler = SMOTE(random_state=101)
  resampled_data, resampled_labels = sampler.fit_resample(flat_data, targets)
  resampled_3d = resampled_data[:, np.newaxis, :]
  return resampled_3d, resampled_labels
```

# Phase 3: Constructing a CNN-LSTM Fusion Model

```
def architect_flow(dimensions):
  inlet = I(shape=dimensions)

  # Inline convolution and pooling pipeline
  layer_stream = C1(64, 1, activation='relu')(inlet)
  layer_stream = MP1(1)(layer_stream)

  # Sequence learning component
  sequence_logic = LS(64)(layer_stream)

  # Fully connected decision layers
  decision_core = D(32, activation='relu')(sequence_logic)
  binary_gate = D(1, activation='sigmoid')(decision_core)
  # Build and compile
  construct = Model(inputs=inlet, outputs=binary_gate)
  construct.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

  return construct
```

# Phase 4: Model Training & Assessment via K-Fold

```
def cross_validate_model(features, labels, column_names):
  kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

  for fold_id, (train_idxs, test_idxs) in enumerate(kfold.split(features, labels), start=1):
    print(f"\n--- Fold {fold_id} ---")

train_features = np.take(features, train_idxs, axis=0)
test_features = np.take(features, test_idxs, axis=0)

train_labels = np.take(labels, train_idxs, axis=0)
test_labels = np.take(labels, test_idxs, axis=0)

    # Resampling training set
    train_flat = X_train[:, 0, :]
    X_aug, y_aug = SMOTE(random_state=42).fit_resample(train_flat, y_train)
    X_aug = X_aug[:, np.newaxis, :]

    # Build and train model
    net = define_model((1, X_aug.shape[2]))
    net.fit(X_aug, y_aug, epochs=3, batch_size=64, verbose=0)

    # Predict & Report
    predictions = (net.predict(X_test) > 0.5).astype(int)
    print("Evaluation:")
    print(classification_report(y_test, predictions))

    # Phase 5: Interpretation via LIME
    from lime.lime_tabular import LimeTabularExplainer as Interpreter

def interpret_instance(model_core, tensorized_data, label_headers, fold_tag=0):
  # Derive sample context for LIME initialization
  scaffold = tensorized_data[:, 0, :]
  meta_labels = ["Benign", "Hostile"]

  # Configure interpreter with abstracted parameters
  oracle = Interpreter(
    training_data=scaffold,
    feature_names=list(label_headers),
    class_names=meta_labels,
    mode='classification'
  )

  # Define subject and adapter for LIME's prediction bridge
  focus_sample = scaffold[0]
  bridge_fn = lambda batch: model_core.predict(batch[:, None, :])

  # Generate surrogate-based explanation with ranked attributions
  breakdown = oracle.explain_instance(
    data_row=focus_sample,
    predict_fn=bridge_fn,
    num_features=10
  )

  # Render insight to an HTML artifact
  output_artifact = f"lime_expl_{fold_tag}.html"
  breakdown.save_to_file(output_artifact)
```

# Execution Pipeline

```
if __name__ == "__main__":
    feature_tensor, label_array, column_headers = prepare_input_data("your_dataset.csv")
    balanced_data, balanced_labels = resample_data(feature_tensor, label_array)
    cross_validate_model(balanced_data, balanced_labels, column_headers)
```