In [ ]:
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from keras.models import Sequential, Model
from keras.layers import Bidirectional, LSTM, Dense, Input, concatenate, Dropout
from keras.optimizers import Adam
import seaborn as sns
import matplotlib.pyplot as plt

# Load your dataset (replace 'deepslice_data.csv' with the actual file path)
dataset = pd.read_csv('deepslice_data.csv')

# Encode categorical features using Label Encoding
label_encoder = LabelEncoder()
categorical_cols = ['Use Case', 'LTE/5g Category', 'Technology Supported', 'Day', 'Time', 'GBR']
for col in categorical_cols:
    dataset[col] = label_encoder.fit_transform(dataset[col])

# Encode the 'slice Type' column
dataset['slice Type'] = label_encoder.fit_transform(dataset['slice Type'])

# Define the features and target variable
X = dataset.drop(columns=['slice Type'])
y = dataset['slice Type']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Bi-LSTM Model for resource allocation and slice selection
bi_lstm_model = Sequential()
bi_lstm_model.add(Bidirectional(LSTM(100, activation='relu'), input_shape=(X_train.shape[1], 1)))
bi_lstm_model.add(Dropout(0.2))  # Add dropout for regularization
bi_lstm_output = Dense(50, activation='relu')(bi_lstm_model.output)
bi_lstm_output = Dense(10, activation='softmax')(bi_lstm_output)

# LSTM Model for statistical information regarding network slices
lstm_model = Sequential()
lstm_model.add(LSTM(100, activation='relu', input_shape=(X_train.shape[1], 1)))
lstm_model.add(Dropout(0.2))  # Add dropout for regularization
lstm_output = Dense(50, activation='relu')(lstm_model.output)
lstm_output = Dense(10, activation='softmax')(lstm_output)
```

```python
# Concatenate the outputs of Bi-LSTM and LSTM
concatenated = concatenate([bi_lstm_model.output, lstm_model.output])

# Dense layer for final classification
dense_layer = Dense(len(np.unique(y)), activation='softmax')(concatenated)

# Create the hybrid model
hybrid_model = Model(inputs=[bi_lstm_model.input, lstm_model.input], outputs=dense_layer)

# Network Slicing Controller (Simulated)
def network_slicing_controller(requested_slices):
    # Simulated logic: Allocate resources based on requested slices
    # In a real-world scenario, this would involve more complex orchestration
    allocated_slices = []

    for slice_request in requested_slices:
        allocated_slice = {
            'slice_id': slice_request['slice_id'],
            'allocated_bandwidth': slice_request['required_bandwidth'] * 1.2,  # Simulated allocation
            'other_resources': slice_request['other_resources']  # Simulated other resources
        }
        allocated_slices.append(allocated_slice)

    return allocated_slices

# Simulate slice requests
slice_requests = [
    {'slice_id': 1, 'required_bandwidth': 10, 'other_resources': '...'},
    {'slice_id': 2, 'required_bandwidth': 5, 'other_resources': '...'},
    # Add more slice requests as needed
]

# Simulate network slicing controller
allocated_slices = network_slicing_controller(slice_requests)

# Print the simulated allocated slices
print("Simulated Allocated Slices:")
for allocated_slice in allocated_slices:
    print(allocated_slice)

# Compile the model
hybrid_model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=0.001), metrics=['accuracy'])
```

```
# Train the model
hybrid_model.fit(
    [X_train.values.reshape((X_train.shape[0], X_train.shape[1], 1)), X_train.values.reshape((X_train.shape[0
    y_train,
    epochs=20,  # Increase the number of epochs
    batch_size=128,  # Adjust batch size based on your data size
    validation_data=(
        [X_test.values.reshape((X_test.shape[0], X_test.shape[1], 1)), X_test.values.reshape((X_test.shape[0]
        y_test
    )
)

# Evaluate the model on the test set
accuracy = hybrid_model.evaluate(
    [X_test.values.reshape((X_test.shape[0], X_test.shape[1], 1)), X_test.values.reshape((X_test.shape[0], X_
    y_test
)[1]

# Print accuracy
print(f"Accuracy: {accuracy}")

# Generate predictions
y_pred_classes = hybrid_model.predict([X_test.values.reshape((X_test.shape[0], X_test.shape[1], 1)), X_test.va
y_pred_classes = np.argmax(y_pred_classes, axis=1)

# Print classification report
print(classification_report(y_test, y_pred_classes, target_names=label_encoder.classes_))

# Confusion matrix for hybrid model
conf_matrix_hybrid = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_hybrid, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, ytickla
plt.title('Confusion Matrix - Hybrid Model')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

#--------------------------------------------------------------------------------------------------------------
# Read the dataset
df = pd.read_csv('Quality of Service 5G.csv')

# Convert 'Resource_Allocation' to a numeric value (percentage to decimal)
```

```python
df['Resource_Allocation'] = df['Resource_Allocation'].str.rstrip('%').astype('float') / 100.0

# Extract numeric values from 'Required_Bandwidth' considering both 'Kbps' and 'Mbps'
df['Required_Bandwidth'] = df['Required_Bandwidth'].str.extract('(\d+\.*\d*)')
df['Required_Bandwidth'] = pd.to_numeric(df['Required_Bandwidth'], errors='coerce')

# Function to perform resource allocation
def allocate_resources(row):
    required_bandwidth = row['Required_Bandwidth']
    resource_allocation_percentage = row['Resource_Allocation']

    # Check if 'Required_Bandwidth' is a valid numeric value
    if pd.notna(required_bandwidth):
        # Calculate the allocated bandwidth based on the resource allocation percentage
        allocated_bandwidth_calculated = required_bandwidth * resource_allocation_percentage
        return allocated_bandwidth_calculated
    else:
        return None  # Return None for rows where 'Required_Bandwidth' is not numeric

# Apply the resource allocation function to each row
df['Allocated_Bandwidth_Calculated'] = df.apply(allocate_resources, axis=1)

# Display the updated DataFrame with calculated allocated bandwidth
print(df[['User_ID', 'Required_Bandwidth', 'Allocated_Bandwidth', 'Resource_Allocation', 'Allocated_Bandwidth_

#--------------------------------------------------------------------------------------------------------------

# Visualization for resource allocation
plt.figure(figsize=(10, 6))
plt.scatter(df['Required_Bandwidth'], df['Allocated_Bandwidth_Calculated'], alpha=0.5)
plt.title('Resource Allocation - Actual vs Calculated')
plt.xlabel('Required Bandwidth')
plt.ylabel('Allocated Bandwidth (Calculated)')
plt.show()
#--------------------------------------------------------------------------------------------------------------
# OGD+ non optimized scatter point+ scatter point
class OnlineGradientDescent:
    def __init__(self, learning_rate):
        self.learning_rate = learning_rate
        self.weights = None

    def initialize_weights(self, num_features):
        self.weights = np.zeros(num_features)
```

```python
    def update_weights(self, gradient):
        self.weights -= self.learning_rate * gradient

    def predict(self, features):
        return np.dot(features, self.weights)

    def compute_gradient(self, features, prediction, target):
        error = prediction - target
        gradient = error * features
        return gradient

# Read the dataset
df = pd.read_csv('Quality of Service 5G.csv')

# Convert 'Resource_Allocation' to a numeric value (percentage to decimal)
df['Resource_Allocation'] = df['Resource_Allocation'].str.rstrip('%').astype('float') / 100.0

# Extract numeric values from 'Required_Bandwidth' considering both 'Kbps' and 'Mbps'
df['Required_Bandwidth'] = df['Required_Bandwidth'].str.extract('(\d+\.*\d*)')
df['Required_Bandwidth'] = pd.to_numeric(df['Required_Bandwidth'], errors='coerce')

# Function to perform resource allocation
def allocate_resources(row):
    required_bandwidth = row['Required_Bandwidth']
    resource_allocation_percentage = row['Resource_Allocation']

    # Check if 'Required_Bandwidth' is a valid numeric value
    if pd.notna(required_bandwidth):
        # Calculate the allocated bandwidth based on the resource allocation percentage
        allocated_bandwidth_calculated = required_bandwidth * resource_allocation_percentage
        return allocated_bandwidth_calculated
    else:
        return None  # Return None for rows where 'Required_Bandwidth' is not numeric

# Apply the resource allocation function to each row
df['Allocated_Bandwidth_Calculated'] = df.apply(allocate_resources, axis=1)

# Initialize OGD with a learning rate
ogd = OnlineGradientDescent(learning_rate=0.01)

# Initialize weights
ogd.initialize_weights(1)  # We have only one feature for simplicity
```

```python
# Online training (Optimize resource allocation)
for i, row in df.iterrows():
    # Predict
    prediction = ogd.predict(row['Required_Bandwidth'])

    # Update weights based on gradient
    gradient = ogd.compute_gradient(row['Required_Bandwidth'], prediction, row['Allocated_Bandwidth_Calculate
    ogd.update_weights(gradient)

# Apply the optimized resource allocation
df['Allocated_Bandwidth_Optimized'] = df['Required_Bandwidth'] * ogd.weights[0]

# Print the dataset with both non-optimized and optimized resource allocation
print(df[['User_ID', 'Required_Bandwidth', 'Allocated_Bandwidth_Calculated', 'Allocated_Bandwidth_Optimized']

# Visualization for resource allocation

plt.figure(figsize=(10, 6))
plt.scatter(df['Required_Bandwidth'], df['Allocated_Bandwidth_Calculated'], label='Non-Optimized', alpha=0.5)
plt.scatter(df['Required_Bandwidth'], df['Allocated_Bandwidth_Optimized'], label='Optimized', alpha=0.5)
plt.title('Resource Allocation - Non-Optimized vs Optimized')
plt.xlabel('Required Bandwidth')
plt.ylabel('Allocated Bandwidth')
plt.legend()
plt.show()
```