Section 1: Import Required Libraries

In [2]:
```
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from category_encoders import TargetEncoder
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, mean_sc
import mlflow
import os
import itertools
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
import holidays
import datetime
```

Section 2: Load and Explore Data

In [3]:
```
# Load raw data
raw_data = pd.read_csv('../data/external/data_2.csv')
raw_data.head(3)
```

Out[3]:

	market_id	created_at	actual_delivery_time	store_primary_category	order_protoco
0	1.0	2015-02-06 22:24:17	2015-02-06 23:11:17	4	1.
1	2.0	2015-02-10 21:49:25	2015-02-10 22:33:25	46	2.
2	2.0	2015-02-16 00:11:35	2015-02-16 01:06:35	36	3.

In [4]:
```
# Display data info
print("Dataset Shape:", raw_data.shape)
print("\nColumn Names:")
print(raw_data.columns.tolist())
```

```
Dataset Shape: (175777, 14)

Column Names:
['market_id', 'created_at', 'actual_delivery_time', 'store_primary_categor
y', 'order_protocol', 'total_items', 'subtotal', 'num_distinct_items', 'min_
item_price', 'max_item_price', 'total_onshift_dashers', 'total_busy_dasher
s', 'total_outstanding_orders', 'estimated_store_to_consumer_driving_duratio
n']
```

In [5]:
```
# Define column types
cat_cols = ['market_id', 'store_primary_category', 'order_protocol']

num_cols = ['total_items', 'subtotal', 'num_distinct_items', 'min_item_price
            'total_onshift_dashers', 'total_busy_dashers',
            'total_outstanding_orders',
            'estimated_store_to_consumer_driving_duration']

created_numcols = ['day_of_week', 'week_of_year', 'hour_of_day', 'minute_of_
                   'is_holiday', 'is_weekend', 'is_long_weekend']

target_col = 'delivery_duration'

print(f"Categorical columns: {cat_cols}")
print(f"Numerical columns: {num_cols}")
print(f"Created numerical columns: {created_numcols}")
print(f"Target column: {target_col}")
```

```
Categorical columns: ['market_id', 'store_primary_category', 'order_protoco
l']
Numerical columns: ['total_items', 'subtotal', 'num_distinct_items', 'min_it
em_price', 'max_item_price', 'total_onshift_dashers', 'total_busy_dashers',
'total_outstanding_orders', 'estimated_store_to_consumer_driving_duration']
Created numerical columns: ['day_of_week', 'week_of_year', 'hour_of_day', 'm
inute_of_hour', 'is_holiday', 'is_weekend', 'is_long_weekend']
Target column: delivery_duration
```

Section 3: Data Preprocessing

In [6]:
```
# Convert datetime columns
raw_data['created_at'] = pd.to_datetime(raw_data['created_at'])
raw_data['actual_delivery_time'] = pd.to_datetime(raw_data['actual_delivery_

# Calculate delivery duration in minutes
raw_data['delivery_duration'] = (raw_data['actual_delivery_time'] - raw_data
raw_data[['created_at', 'actual_delivery_time', 'delivery_duration']].head(3
```

Out[6]:

	created_at	actual_delivery_time	delivery_duration
0	2015-02-06 22:24:17	2015-02-06 23:11:17	47.0
1	2015-02-10 21:49:25	2015-02-10 22:33:25	44.0
2	2015-02-16 00:11:35	2015-02-16 01:06:35	55.0

In [7]:
```
# Feature engineering - extract temporal features
india_holidays = holidays.India()

processed_data = raw_data.copy()

# Extract temporal features
processed_data['day_of_week'] = processed_data['created_at'].dt.dayofweek
processed_data['week_of_year'] = processed_data['created_at'].dt.isocalendar
processed_data['hour_of_day'] = processed_data['created_at'].dt.hour
processed_data['minute_of_hour'] = processed_data['created_at'].dt.minute

# Check if the date is a holiday
processed_data['is_holiday'] = processed_data['created_at'].dt.date.apply(la

# Add weekend feature
processed_data['is_weekend'] = processed_data['day_of_week'].apply(lambda x:

# Add long weekend feature
processed_data['is_long_weekend'] = 0
for idx, row in processed_data.iterrows():
    day = row['day_of_week']
    is_holiday = row['is_holiday']
    if day == 4 and is_holiday:  # Friday holiday
        processed_data.at[idx, 'is_long_weekend'] = 1
    elif day == 0 and is_holiday:  # Monday holiday
        processed_data.at[idx, 'is_long_weekend'] = 1

print("Feature engineering completed")
processed_data[created_numcols + ['delivery_duration']].head(3)
```

Feature engineering completed

Out[7]:

	day_of_week	week_of_year	hour_of_day	minute_of_hour	is_holiday	is_weekend
0	4	6	22	24	0	0
1	1	7	21	49	0	0
2	0	8	0	11	0	0

In [8]:
```
# Data exploration - Check for missing values and data quality
print(f"Original shape: {raw_data.shape}")

# Remove rows with negative values in numerical columns
processed_data_no_neg = processed_data[(processed_data[num_cols] >= 0).all(a
print(f"After removing negative values: {processed_data_no_neg.shape}")

processed_data = processed_data_no_neg.copy()

# Check for missing values
print("\nMissing values per column:")
print(processed_data.isnull().sum())
```

```
Original shape: (175777, 15)
After removing negative values: (175687, 22)

Missing values per column:
market_id                                   0
created_at                                  0
actual_delivery_time                        0
store_primary_category                      0
order_protocol                              0
total_items                                 0
subtotal                                    0
num_distinct_items                          0
min_item_price                              0
max_item_price                              0
total_onshift_dashers                       0
total_busy_dashers                          0
total_outstanding_orders                    0
estimated_store_to_consumer_driving_duration    0
delivery_duration                           0
day_of_week                                 0
week_of_year                                0
hour_of_day                                 0
minute_of_hour                              0
is_holiday                                  0
is_weekend                                  0
is_long_weekend                             0
dtype: int64
```

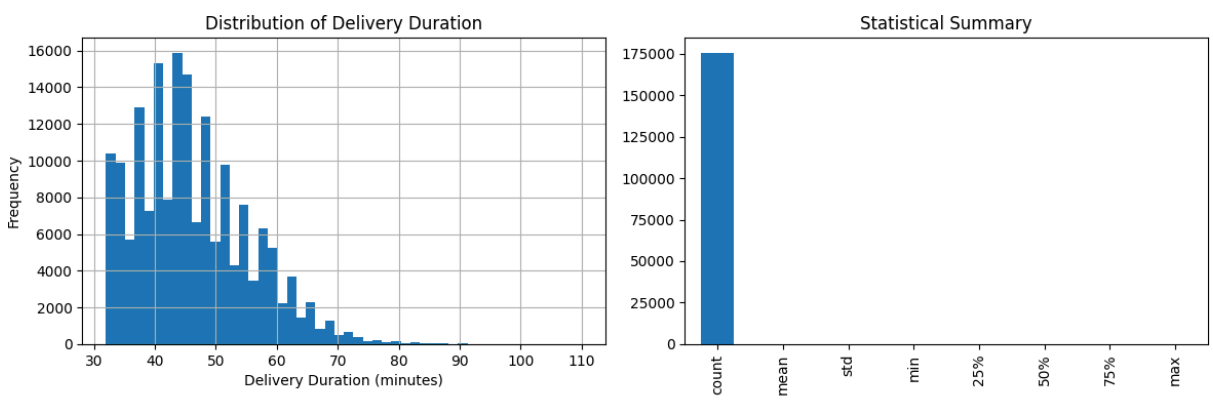
In [9]:

```
# Convert categorical columns to category dtype
for col in cat_cols:
    processed_data[col] = processed_data[col].astype('category')

print("Data types after conversion:")
print(processed_data.dtypes)
```

```
Data types after conversion:
market_id                                          category
created_at                                   datetime64[ns]
actual_delivery_time                         datetime64[ns]
store_primary_category                             category
order_protocol                                     category
total_items                                           int64
subtotal                                              int64
num_distinct_items                                    int64
min_item_price                                        int64
max_item_price                                        int64
total_onshift_dashers                               float64
total_busy_dashers                                  float64
total_outstanding_orders                            float64
estimated_store_to_consumer_driving_duration        float64
delivery_duration                                   float64
day_of_week                                           int32
week_of_year                                         UInt32
hour_of_day                                           int32
minute_of_hour                                        int32
is_holiday                                            int64
is_weekend                                            int64
is_long_weekend                                       int64
dtype: object
```

In [10]:

```
# Analyze target variable distribution
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
processed_data['delivery_duration'].hist(bins=50)
plt.xlabel('Delivery Duration (minutes)')
plt.ylabel('Frequency')
plt.title('Distribution of Delivery Duration')

plt.subplot(1, 2, 2)
processed_data['delivery_duration'].describe().plot(kind='bar')
plt.title('Statistical Summary')
plt.tight_layout()
plt.show()

print(processed_data['delivery_duration'].describe())
```



```
count    175687.000000
mean         46.202656
std           9.327784
min          32.000000
25%          39.000000
50%          45.000000
75%          52.000000
max         110.000000
Name: delivery_duration, dtype: float64
```

In [11]:
```
# Analyze categorical columns
print("Categorical Columns Analysis:")
for col in cat_cols:
    print(f"\n{col}:")
    print(processed_data[col].value_counts())
```

```
Categorical Columns Analysis:

market_id:
market_id
2.0    53462
4.0    46220
1.0    37070
3.0    21044
5.0    17253
6.0      638
Name: count, dtype: int64

store_primary_category:
store_primary_category
4     18174
55    15740
46    15583
13     9911
58     8988
      ...
1       10
43       9
8        2
3        1
21       1
Name: count, Length: 73, dtype: int64

order_protocol:
order_protocol
1.0    48367
3.0    47109
5.0    41403
2.0    20887
4.0    17226
6.0      676
7.0       19
Name: count, dtype: int64
```

Section 4: Train-Test Split and Target Encoding

In [12]:
```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    processed_data.drop(columns=[target_col]),
    processed_data[target_col],
    test_size=0.2,
    random_state=42
)

print(f"Training set size: {X_train.shape}")
print(f"Testing set size: {X_test.shape}")
```

```
Training set size: (140549, 21)
Testing set size: (35138, 21)
```

In [13]:
```
# Apply target encoding to categorical columns
te = TargetEncoder(cols=cat_cols)
X_train_enc = te.fit_transform(X_train, y_train)
X_test_enc = te.transform(X_test)

print("Target encoding completed")
print(f"Encoded training set shape: {X_train_enc.shape}")
print(f"Encoded testing set shape: {X_test_enc.shape}")
```

```
Target encoding completed
Encoded training set shape: (140549, 21)
Encoded testing set shape: (35138, 21)
```

In [14]:
```
# Drop unnecessary columns
drop_cols = set(X_train_enc.columns) - set(num_cols) - set(created_numcols)

print("Dropping columns:", drop_cols)

X_train_final = X_train_enc.drop(columns=drop_cols)
X_test_final = X_test_enc.drop(columns=drop_cols)

print(f"Final training set shape: {X_train_final.shape}")
print(f"Final testing set shape: {X_test_final.shape}")
```

```
Dropping columns: {'actual_delivery_time', 'created_at'}
Final training set shape: (140549, 19)
Final testing set shape: (35138, 19)
```

Section 5: Feature Scaling

In [15]:
```
# Feature scaling for numerical features
X_scaler = StandardScaler()
X_train_scaled = X_scaler.fit_transform(X_train_final)
X_test_scaled = X_scaler.transform(X_test_final)

# Target scaling (important for neural networks)
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).ravel()
```

```
y_test_scaled = y_scaler.transform(
    y_test.values.reshape(-1, 1)
).ravel()

print("Feature and target scaling completed")
print(f"X_train_scaled shape: {X_train_scaled.shape}")
print(f"X_test_scaled shape: {X_test_scaled.shape}")
```

```
Feature and target scaling completed
X_train_scaled shape: (140549, 19)
X_test_scaled shape: (35138, 19)
```

In [16]:
```
# Log transformation of target variable for improved model performance
y_train_log = np.log1p(y_train)
y_test_log = np.log1p(y_test)

print("Log transformation of target variable completed")
```

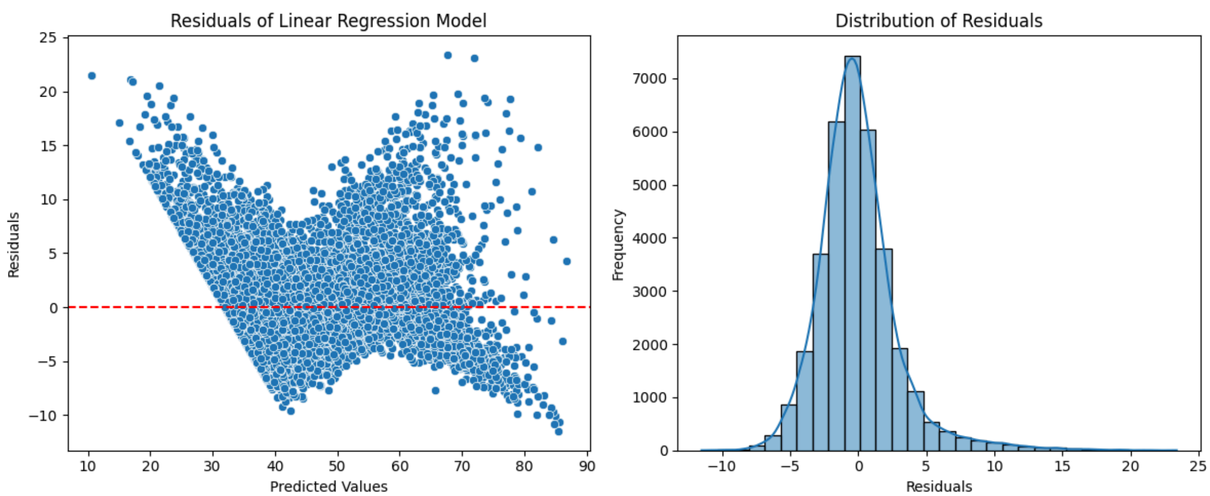
```
Log transformation of target variable completed
```

Section 6: Baseline Model - Linear Regression

In [30]:
```
# Set up MLflow tracking
mlflow.set_tracking_uri("/Users/pramodkumar/ML Learning/SCALAR/Business Case
mlflow.set_experiment("delivery_time_prediction")

# Train baseline linear regression model
with mlflow.start_run(run_name="Linear_Regression_Model"):
    lr_model = LinearRegression()
    lr_model.fit(X_train_scaled, y_train_scaled)
    y_pred_train = lr_model.predict(X_train_scaled)
    y_pred_test = lr_model.predict(X_test_scaled)

    # Inverse scale predictions
    y_pred_train = y_scaler.inverse_transform(y_pred_train.reshape(-1, 1)).r
    y_pred_test = y_scaler.inverse_transform(y_pred_test.reshape(-1, 1)).rav

    # Calculate metrics
    train_mae = mean_absolute_error(y_train, y_pred_train)
    test_mae = mean_absolute_error(y_test, y_pred_test)
    train_rmse = mean_squared_error(y_train, y_pred_train)
    test_rmse = mean_squared_error(y_test, y_pred_test)

    print(f"Train MAE: {train_mae:.4f}, Test MAE: {test_mae:.4f}")
    print(f"Train RMSE: {train_rmse:.4f}, Test RMSE: {test_rmse:.4f}")

    # Log to MLflow
    mlflow.log_param("model_type", "Linear Regression")
    mlflow.log_metric("train_mae", train_mae)
    mlflow.log_metric("test_mae", test_mae)
    mlflow.log_metric("train_rmse", train_rmse)
    mlflow.log_metric("test_rmse", test_rmse)
    mlflow.sklearn.log_model(lr_model, "linear_regression_model")
```

```
2026/01/05 07:09:23 WARNING mlflow.models.model: `artifact_path` is deprecat
ed. Please use `name` instead.
Train MAE: 2.1235, Test MAE: 2.0987
Train RMSE: 9.0844, Test RMSE: 8.7333
```

In [31]:
```
# Residual analysis
residuals_lr = y_test - y_pred_test

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(x=y_pred_test, y=residuals_lr)
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residuals of Linear Regression Model')
plt.axhline(y=0, color='r', linestyle='--')

plt.subplot(1, 2, 2)
sns.histplot(residuals_lr, bins=30, kde=True)
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.title('Distribution of Residuals')
plt.tight_layout()
plt.show()
```



Section 7: Linear Regression with Log-Transformed Target

In [17]:
```
# Train linear regression with log-transformed target
with mlflow.start_run(run_name="Linear_Regression_Model_Log_Target"):
    lr_model_log = LinearRegression()
    lr_model_log.fit(X_train_scaled, y_train_log)
    y_pred_train_log = lr_model_log.predict(X_train_scaled)
    y_pred_test_log = lr_model_log.predict(X_test_scaled)

    # Inverse log transformation
    y_pred_train_log = np.expm1(y_pred_train_log)
    y_pred_test_log = np.expm1(y_pred_test_log)
```

```
    # Calculate metrics
    train_mae_log = mean_absolute_error(y_train, y_pred_train_log)
    test_mae_log = mean_absolute_error(y_test, y_pred_test_log)
    train_rmse_log = mean_squared_error(y_train, y_pred_train_log)
    test_rmse_log = mean_squared_error(y_test, y_pred_test_log)

    print(f"Log Target - Train MAE: {train_mae_log:.4f}, Test MAE: {test_mae
    print(f"Log Target - Train RMSE: {train_rmse_log:.4f}, Test RMSE: {test_

    # Log to MLflow
    mlflow.log_param("model_type", "Linear Regression with Log Target")
    mlflow.log_metric("train_mae", train_mae_log)
    mlflow.log_metric("test_mae", test_mae_log)
    mlflow.log_metric("train_rmse", train_rmse_log)
    mlflow.log_metric("test_rmse", test_rmse_log)
    mlflow.sklearn.log_model(lr_model_log, "linear_regression_model_log_targ
```

```
2026/01/05 07:02:15 INFO mlflow.store.db.utils: Creating initial MLflow data
base tables...
2026/01/05 07:02:15 INFO mlflow.store.db.utils: Updating database tables
2026/01/05 07:02:15 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Will assume non-transact
ional DDL.
2026/01/05 07:02:15 INFO mlflow.store.db.utils: Updating database tables
2026/01/05 07:02:15 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Will assume non-transact
ional DDL.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Will assume non-transact
ional DDL.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/05 07:02:15 INFO alembic.runtime.migration: Will assume non-transact
ional DDL.
2026/01/05 07:02:16 WARNING mlflow.models.model: `artifact_path` is deprecat
ed. Please use `name` instead.
2026/01/05 07:02:16 WARNING mlflow.models.model: `artifact_path` is deprecat
ed. Please use `name` instead.
Log Target - Train MAE: 2.1067, Test MAE: 2.0723
Log Target - Train RMSE: 9.1930, Test RMSE: 8.7242
```

Section 8: Build Neural Network Model with Hyperparameter Tuning

In [18]:
```
# Define function to build neural network model
def build_nn_model_hp(input_dim,
                      neurons=[24, 32],
                      dropout_rate=0.2,
                      use_batchnorm=True,
                      learning_rate=1e-3):

    model = Sequential()

    # Input layer
    model.add(Dense(neurons[0], activation='relu', input_dim=input_dim))
```

```
    if use_batchnorm:
        model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))

    # Hidden layers
    for n in neurons[1:]:
        model.add(Dense(n, activation='relu'))
        if use_batchnorm:
            model.add(BatchNormalization())
        model.add(Dropout(dropout_rate))

    # Output layer (linear for regression)
    model.add(Dense(1, activation='linear'))

    model.compile(
        optimizer=Adam(learning_rate=learning_rate),
        loss='mse',
        metrics=['mae']
    )

    return model

print("Neural network building function defined")
```

```
Neural network building function defined
```

In [19]:

```
# Define hyperparameter grid for tuning
param_grid = {
    "neurons": [
        [24, 32],
        [16, 32],
        [24, 16]
    ],
    "dropout_rate": [0.1, 0.2, 0.3],
    "use_batchnorm": [True, False],
    "batch_size": [256, 512],
    "learning_rate": [1e-3, 5e-4]
}

print(f"Total hyperparameter combinations: {3 * 3 * 2 * 2 * 2} = 72")
```

```
Total hyperparameter combinations: 72 = 72
```

In [ ]:

```
# Hyperparameter tuning for neural network
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

best_test_rmse = np.inf
best_run_id = None

for params in itertools.product(
        param_grid["neurons"],
        param_grid["dropout_rate"],
        param_grid["use_batchnorm"],
```

```
        param_grid["batch_size"],
        param_grid["learning_rate"]):

    neurons, dropout_rate, use_batchnorm, batch_size, lr = params

    with mlflow.start_run(run_name="NN_LogTarget_HP"):
        log_dir = "tb_logs/nn/" + neurons.__str__() + "_" + str(dropout_rate
        os.makedirs(log_dir, exist_ok=True)

        tensorboard_cb = TensorBoard(
            log_dir=log_dir,
            histogram_freq=1
        )

        mlflow.log_params({
            "neurons": str(neurons),
            "dropout_rate": dropout_rate,
            "batchnorm": use_batchnorm,
            "batch_size": batch_size,
            "learning_rate": lr,
            "target_transform": "log1p"
        })

        model = build_nn_model_hp(
            input_dim=X_train_scaled.shape[1],
            neurons=neurons,
            dropout_rate=dropout_rate,
            use_batchnorm=use_batchnorm,
            learning_rate=lr
        )

        history = model.fit(
            X_train_scaled, y_train_log,
            epochs=100,
            batch_size=batch_size,
            validation_split=0.2,
            callbacks=[early_stop, tensorboard_cb],
            verbose=0
        )

        # Predictions (inverse log transformation)
        y_train_pred = np.expm1(model.predict(X_train_scaled, verbose=0)).ra
        y_test_pred = np.expm1(model.predict(X_test_scaled, verbose=0)).rave

        # Calculate metrics
        train_mae = mean_absolute_error(y_train, y_train_pred)
        test_mae = mean_absolute_error(y_test, y_test_pred)

        train_rmse = mean_squared_error(y_train, y_train_pred)
        test_rmse = mean_squared_error(y_test, y_test_pred)

        if (y_test_pred.min() < 0) and (y_train_pred.min() < 0):
            train_msle = np.nan
            test_msle = np.nan
        else:
            train_msle = mean_squared_log_error(y_train, y_train_pred)
```

```
            test_msle = mean_squared_log_error(y_test, y_test_pred)

        mlflow.log_metrics({
            "train_mae": train_mae,
            "test_mae": test_mae,
            "train_rmse": train_rmse,
            "test_rmse": test_rmse,
            "train_msle": train_msle,
            "test_msle": test_msle
        })

        # Save best model
        if test_rmse < best_test_rmse:
            best_test_rmse = test_rmse
            best_run_id = mlflow.active_run().info.run_id
            mlflow.tensorflow.log_model(model, "best_nn_log_target")

        mlflow.end_run()

print(f"Hyperparameter tuning completed. Best test RMSE: {best_test_rmse:.4f
```

Section 9: Evaluate and Rank Models

In [20]:

```
# Function to evaluate model runs and create rankings
from tensorboard.backend.event_processing.event_accumulator import EventAccu

def read_tensor_scalar(ea, tag):
    return np.array([
        tf.make_ndarray(e.tensor_proto).item()
        for e in ea.Tensors(tag)
    ])

def read_tb_scalars(run_dir):
    ea_train = EventAccumulator(run_dir + '/train/')
    ea_val = EventAccumulator(run_dir + '/validation/')
    ea_train.Reload()
    ea_val.Reload()

    train_loss = read_tensor_scalar(ea_train, 'epoch_loss')
    val_loss = read_tensor_scalar(ea_val, 'epoch_loss')
    return train_loss, val_loss

def evaluate_run(run_dir, model_params, alpha=0.1, beta=0.1):
    train_loss, val_loss = read_tb_scalars(run_dir)

    best_epoch = np.argmin(val_loss)
    best_val = val_loss[best_epoch]
    train_at_best = train_loss[best_epoch]

    # Generalization gap
    gap = abs(best_val - train_at_best)

    # Stability
    start = max(0, best_epoch - 3)
```

```
    end = min(len(val_loss), best_epoch + 4)
    stability = np.std(val_loss[start:end])

    # Final score
    score = (
        best_val
        + alpha * gap
        + beta * stability
    )

    return {
        "run": os.path.basename(run_dir),
        "best_val_loss": best_val,
        "gap": gap,
        "stability": stability,
        "model_params": model_params,
        "score": score
    }

print("Evaluation functions defined")
```

```
Evaluation functions defined
```

In [22]:

```
# Evaluate all runs and create rankings
all_ranks = []

for params in itertools.product(
        param_grid["neurons"],
        param_grid["dropout_rate"],
        param_grid["use_batchnorm"],
        param_grid["batch_size"],
        param_grid["learning_rate"]):

    neurons, dropout_rate, use_batchnorm, batch_size, lr = params

    log_dir = "tb_logs/nn/" + neurons.__str__() + "_" + str(dropout_rate) +
    if os.path.exists(log_dir):
        eval_metrics = evaluate_run(
            log_dir,
            model_params={
                "neurons": neurons,
                "dropout_rate": dropout_rate,
                "use_batchnorm": use_batchnorm,
                "batch_size": batch_size,
                "learning_rate": lr
            }
        )
        all_ranks.append(eval_metrics)

ranked_df = pd.DataFrame(all_ranks)
ranked_df = ranked_df.sort_values('score')
print("\nTop 10 Models by Score:")
ranked_df.head(10)
```

```
Top 10 Models by Score:
```

Out[22]:

	run	best_val_loss	gap	stability	model_params	s
0	[24, 32]_0.1_True_256_0.001	0.004015	0.008800	0.000183	{'neurons': [24, 32], 'dropout_rate': 0.1, 'us...	0.00
24	[16, 32]_0.1_True_256_0.001	0.009583	0.094720	0.002186	{'neurons': [16, 32], 'dropout_rate': 0.1, 'us...	0.01
48	[24, 16]_0.1_True_256_0.001	0.011577	0.106163	0.001780	{'neurons': [24, 16], 'dropout_rate': 0.1, 'us...	0.02
8	[24, 32]_0.2_True_256_0.001	0.012231	0.127186	0.002438	{'neurons': [24, 32], 'dropout_rate': 0.2, 'us...	0.02
32	[16, 32]_0.2_True_256_0.001	0.014412	0.130416	0.002679	{'neurons': [16, 32], 'dropout_rate': 0.2, 'us...	0.02
50	[24, 16]_0.1_True_512_0.001	0.011450	0.177045	0.011904	{'neurons': [24, 16], 'dropout_rate': 0.1, 'us...	0.03
2	[24, 32]_0.1_True_512_0.001	0.010724	0.210763	0.005735	{'neurons': [24, 32], 'dropout_rate': 0.1, 'us...	0.03
1	[24, 32]_0.1_True_256_0.0005	0.011507	0.219008	0.004719	{'neurons': [24, 32], 'dropout_rate': 0.1, 'us...	0.03
26	[16, 32]_0.1_True_512_0.001	0.014750	0.192103	0.004206	{'neurons': [16, 32], 'dropout_rate': 0.1, 'us...	0.03
25	[16, 32]_0.1_True_256_0.0005	0.013119	0.224137	0.002538	{'neurons': [16, 32], 'dropout_rate': 0.1, 'us...	0.03

In [23]:

```
# Display best model parameters
print("Best Model Parameters:")
best_params = ranked_df.loc[0, 'model_params']
for key, value in best_params.items():
    print(f"  {key}: {value}")
```

```
Best Model Parameters:
  neurons: [24, 32]
  dropout_rate: 0.1
  use_batchnorm: True
  batch_size: 256
  learning_rate: 0.001
```

Section 10: Train Final Model with Best Hyperparameters

In [24]:
```
# Build and train final model with best hyperparameters
best_model_params = ranked_df.loc[0, 'model_params']

final_model = build_nn_model_hp(
    input_dim=X_train_scaled.shape[1],
    neurons=best_model_params['neurons'],
    dropout_rate=best_model_params['dropout_rate'],
    use_batchnorm=best_model_params['use_batchnorm'],
    learning_rate=best_model_params['learning_rate']
)

# Display model architecture
final_model.summary()
```

```
/Users/pramodkumar/ML Learning/SCALAR/Business Case Studies/NN_Regression_Po
rter/.venv/lib/python3.11/site-packages/keras/src/layers/core/dense.py:106:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W
hen using Sequential models, prefer using an `Input(shape)` object as the fi
rst layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2026-01-05 07:03:14.888274: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M4
2026-01-05 07:03:14.888325: I metal_plugin/src/device/metal_device.cc:296] s
ystemMemory: 16.00 GB
2026-01-05 07:03:14.888339: I metal_plugin/src/device/metal_device.cc:313] m
axCacheSize: 5.92 GB
2026-01-05 07:03:14.888478: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:305] Could not identify NUMA node of platform
GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA sup
port.
2026-01-05 07:03:14.888509: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhos
t/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevic
e (device: 0, name: METAL, pci bus id: <undefined>)
```

Model: "sequential"

Layer (type)	Output Shape	Par
dense (Dense)	(None, 24)	
batch_normalization (BatchNormalization)	(None, 24)	
dropout (Dropout)	(None, 24)	
dense_1 (Dense)	(None, 32)	
batch_normalization_1 (BatchNormalization)	(None, 32)	
dropout_1 (Dropout)	(None, 32)	
dense_2 (Dense)	(None, 1)	

Total params: 1,537 (6.00 KB)

Trainable params: 1,425 (5.57 KB)

Non-trainable params: 112 (448.00 B)

In [27]:
```
# Train final model
final_history = final_model.fit(
    X_train_scaled, y_train_log,
    epochs=100,
    batch_size=best_model_params['batch_size'],
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=1
)

print("Final model training completed")
```

```
Epoch 1/100
2026-01-05 07:04:06.426836: I tensorflow/core/grappler/optimizers/custom_gra
ph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enable
d.
```

```
440/440 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 4.5985 - mae: 1.6746 - val
_loss: 0.0278 - val_mae: 0.1351
Epoch 2/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.3415 - mae: 0.4364 - val
_loss: 0.0144 - val_mae: 0.0964
Epoch 3/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.2003 - mae: 0.3381 - val
_loss: 0.0125 - val_mae: 0.0856
Epoch 4/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.1318 - mae: 0.2787 - val
_loss: 0.0089 - val_mae: 0.0742
Epoch 5/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 0.0946 - mae: 0.2394 - val
_loss: 0.0079 - val_mae: 0.0709
Epoch 6/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0746 - mae: 0.2139 - val
_loss: 0.0074 - val_mae: 0.0681
Epoch 7/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0642 - mae: 0.1999 - val
_loss: 0.0064 - val_mae: 0.0626
Epoch 8/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0569 - mae: 0.1889 - val
_loss: 0.0060 - val_mae: 0.0606
Epoch 9/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0532 - mae: 0.1829 - val
_loss: 0.0051 - val_mae: 0.0554
Epoch 10/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0508 - mae: 0.1789 - val
_loss: 0.0046 - val_mae: 0.0514
Epoch 11/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0484 - mae: 0.1749 - val
_loss: 0.0045 - val_mae: 0.0507
Epoch 12/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0469 - mae: 0.1722 - val
_loss: 0.0046 - val_mae: 0.0518
Epoch 13/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0451 - mae: 0.1687 - val
_loss: 0.0047 - val_mae: 0.0521
Epoch 14/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0436 - mae: 0.1658 - val
_loss: 0.0045 - val_mae: 0.0505
Epoch 15/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0412 - mae: 0.1612 - val
_loss: 0.0043 - val_mae: 0.0493
Epoch 16/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0397 - mae: 0.1585 - val
_loss: 0.0041 - val_mae: 0.0475
Epoch 17/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0380 - mae: 0.1544 - val
_loss: 0.0046 - val_mae: 0.0513
Epoch 18/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0354 - mae: 0.1495 - val
_loss: 0.0040 - val_mae: 0.0468
Epoch 19/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0329 - mae: 0.1441 - val
_loss: 0.0040 - val_mae: 0.0470
```

```
Epoch 20/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - loss: 0.0304 - mae: 0.1384 - val
_loss: 0.0046 - val_mae: 0.0515
Epoch 21/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0277 - mae: 0.1322 - val
_loss: 0.0041 - val_mae: 0.0464
Epoch 22/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0252 - mae: 0.1258 - val
_loss: 0.0041 - val_mae: 0.0470
Epoch 23/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0226 - mae: 0.1192 - val
_loss: 0.0039 - val_mae: 0.0458
Epoch 24/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 0.0204 - mae: 0.1129 - val
_loss: 0.0039 - val_mae: 0.0450
Epoch 25/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0183 - mae: 0.1071 - val
_loss: 0.0039 - val_mae: 0.0452
Epoch 26/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0166 - mae: 0.1018 - val
_loss: 0.0044 - val_mae: 0.0501
Epoch 27/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0150 - mae: 0.0965 - val
_loss: 0.0039 - val_mae: 0.0453
Epoch 28/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0135 - mae: 0.0915 - val
_loss: 0.0039 - val_mae: 0.0456
Epoch 29/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0123 - mae: 0.0871 - val_
loss: 0.0040 - val_mae: 0.0451
Epoch 30/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0112 - mae: 0.0830 - val
_loss: 0.0038 - val_mae: 0.0451
Epoch 31/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0103 - mae: 0.0794 - val
_loss: 0.0041 - val_mae: 0.0472
Epoch 32/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0094 - mae: 0.0756 - val
_loss: 0.0040 - val_mae: 0.0460
Epoch 33/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0088 - mae: 0.0727 - val
_loss: 0.0040 - val_mae: 0.0464
Epoch 34/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0082 - mae: 0.0704 - val
_loss: 0.0040 - val_mae: 0.0461
Epoch 35/100
440/440 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - loss: 0.0076 - mae: 0.0674 - val
_loss: 0.0040 - val_mae: 0.0460
Final model training completed
```

Section 11: Model Evaluation and Comparison

In [28]:
```
# Generate predictions from final model
y_train_pred_final = np.expm1(final_model.predict(X_train_scaled, verbose=0)
y_test_pred_final = np.expm1(final_model.predict(X_test_scaled, verbose=0)).
```

```
# Calculate metrics for final model
final_train_mae = mean_absolute_error(y_train, y_train_pred_final)
final_test_mae = mean_absolute_error(y_test, y_test_pred_final)

final_train_rmse = mean_squared_error(y_train, y_train_pred_final)
final_test_rmse = mean_squared_error(y_test, y_test_pred_final)

print("\n=== FINAL NEURAL NETWORK MODEL ===")
print(f"Training MAE: {final_train_mae:.4f}")
print(f"Testing MAE:  {final_test_mae:.4f}")
print(f"Training RMSE: {final_train_rmse:.4f}")
print(f"Testing RMSE:  {final_test_rmse:.4f}")
```

```
=== FINAL NEURAL NETWORK MODEL ===
Training MAE: 2.1284
Testing MAE:  2.0905
Training RMSE: 9.4324
Testing RMSE:  8.9138
```

In [33]:

```
# Compare all three models
comparison_df = pd.DataFrame({
    'Model': ['Linear Regression', 'Linear Regression (Log)', 'Neural Networ
    'Train MAE': [train_mae, train_mae_log, final_train_mae],
    'Test MAE': [test_mae, test_mae_log, final_test_mae],
    'Train RMSE': [train_rmse, train_rmse_log, final_train_rmse],
    'Test RMSE': [test_rmse, test_rmse_log, final_test_rmse]
})

print("\n=== MODEL COMPARISON ===")
comparison_df
```

```
=== MODEL COMPARISON ===
```

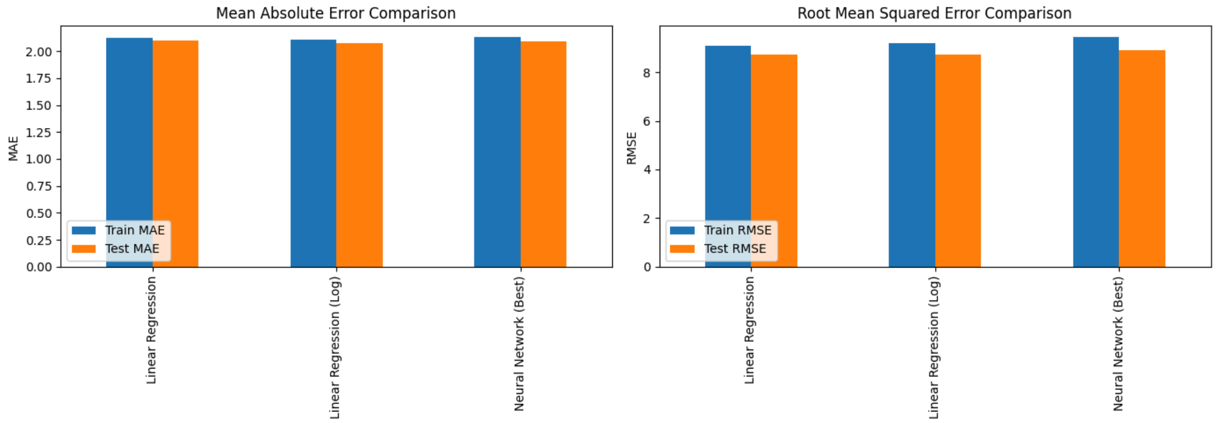
Out[33]:

	Model	Train MAE	Test MAE	Train RMSE	Test RMSE
0	Linear Regression	2.123481	2.098723	9.084379	8.733328
1	Linear Regression (Log)	2.106685	2.072298	9.192999	8.724192
2	Neural Network (Best)	2.128378	2.090494	9.432387	8.913790

In [35]:

```
# Visualize model comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# MAE Comparison
comparison_df.plot(x='Model', y=['Train MAE', 'Test MAE'], kind='bar', ax=ax
axes[0].set_title('Mean Absolute Error Comparison')
axes[0].set_ylabel('MAE')
axes[0].set_xlabel('')
axes[0].legend(loc='lower left')

# RMSE Comparison
comparison_df.plot(x='Model', y=['Train RMSE', 'Test RMSE'], kind='bar', ax=
axes[1].set_title('Root Mean Squared Error Comparison')
axes[1].set_ylabel('RMSE')
axes[1].set_xlabel('')
```

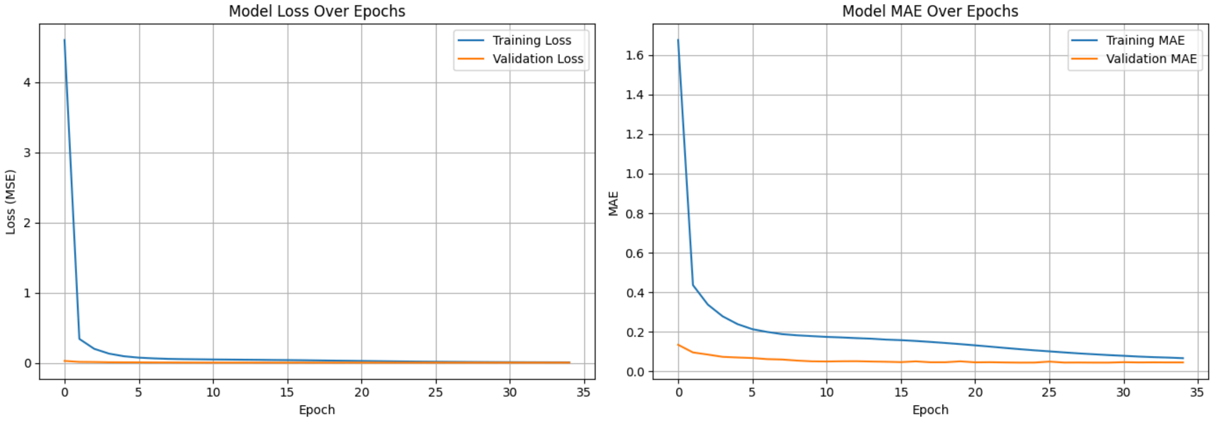
```
axes[1].legend(loc='lower left')

plt.tight_layout()
plt.show()
```



Section 12: Visualize Training History

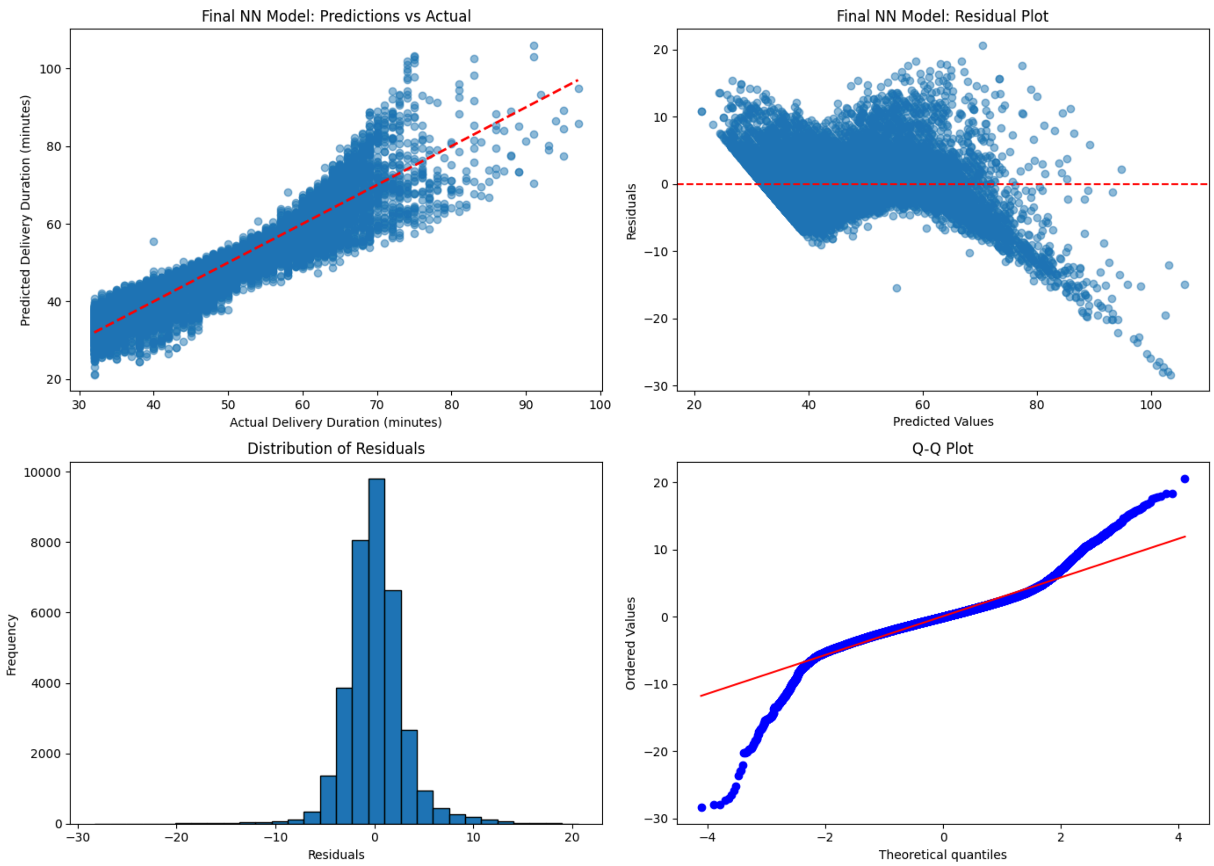
In [36]:

```
# Plot training history
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Loss
axes[0].plot(final_history.history['loss'], label='Training Loss')
axes[0].plot(final_history.history['val_loss'], label='Validation Loss')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss (MSE)')
axes[0].set_title('Model Loss Over Epochs')
axes[0].legend()
axes[0].grid(True)

# MAE
axes[1].plot(final_history.history['mae'], label='Training MAE')
axes[1].plot(final_history.history['val_mae'], label='Validation MAE')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('MAE')
axes[1].set_title('Model MAE Over Epochs')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.show()
```



Section 13: Predictions and Residual Analysis

In [37]:
```
# Calculate residuals for final model
residuals_nn_final = y_test - y_test_pred_final

# Plot predictions vs actual values
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Predictions vs Actual (Final Model)
axes[0, 0].scatter(y_test, y_test_pred_final, alpha=0.5)
axes[0, 0].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
axes[0, 0].set_xlabel('Actual Delivery Duration (minutes)')
axes[0, 0].set_ylabel('Predicted Delivery Duration (minutes)')
axes[0, 0].set_title('Final NN Model: Predictions vs Actual')

# Residuals (Final Model)
axes[0, 1].scatter(y_test_pred_final, residuals_nn_final, alpha=0.5)
axes[0, 1].axhline(y=0, color='r', linestyle='--')
axes[0, 1].set_xlabel('Predicted Values')
axes[0, 1].set_ylabel('Residuals')
axes[0, 1].set_title('Final NN Model: Residual Plot')

# Residuals Distribution
axes[1, 0].hist(residuals_nn_final, bins=30, edgecolor='black')
axes[1, 0].set_xlabel('Residuals')
axes[1, 0].set_ylabel('Frequency')
axes[1, 0].set_title('Distribution of Residuals')

# Q-Q Plot
from scipy import stats
stats.probplot(residuals_nn_final, dist="norm", plot=axes[1, 1])
axes[1, 1].set_title('Q-Q Plot')

plt.tight_layout()
plt.show()

print(f"Mean Residual: {residuals_nn_final.mean():.4f}")
print(f"Std Dev of Residuals: {residuals_nn_final.std():.4f}")
```




```
Mean Residual: 0.0619
Std Dev of Residuals: 2.9850
```

Section 14: Summary and Insights

In [38]:
```python
# Summary statistics
print("\n" + "="*60)
print("ANALYSIS SUMMARY - DELIVERY DURATION PREDICTION")
print("="*60)

print("\n1. DATA OVERVIEW:")
print(f"   - Total samples: {len(processed_data)}")
print(f"   - Training samples: {len(X_train_final)}")
print(f"   - Testing samples: {len(X_test_final)}")
print(f"   - Number of features: {X_train_final.shape[1]}")
print(f"   - Target variable mean: {y_train.mean():.2f} minutes")
print(f"   - Target variable std: {y_train.std():.2f} minutes")

print("\n2. MODEL PERFORMANCE (Test Set):")
print(f"   Linear Regression:")
print(f"      MAE: {test_mae:.4f}, RMSE: {test_rmse:.4f}")
print(f"   Linear Regression (Log Target):")
print(f"      MAE: {test_mae_log:.4f}, RMSE: {test_rmse_log:.4f}")
print(f"   Best Neural Network:")
print(f"      MAE: {final_test_mae:.4f}, RMSE: {final_test_rmse:.4f}")

print("\n3. BEST NN MODEL CONFIGURATION:")
for key, value in best_model_params.items():
    print(f"   - {key}: {value}")
```

```
print("\n4. KEY FINDINGS:")
improvement = ((test_rmse - final_test_rmse) / test_rmse) * 100
print(f"   - NN model improved baseline LR by {improvement:.2f}% in RMSE")
print(f"   - Residuals have mean: {residuals_nn_final.mean():.4f} (near 0 is
print(f"   - Heteroscedasticity noted; butterfly pattern in residuals")

print("\n" + "="*60)
```

```
============================================================
ANALYSIS SUMMARY - DELIVERY DURATION PREDICTION
============================================================

1. DATA OVERVIEW:
   - Total samples: 175687
   - Training samples: 140549
   - Testing samples: 35138
   - Number of features: 19
   - Target variable mean: 46.22 minutes
   - Target variable std: 9.36 minutes

2. MODEL PERFORMANCE (Test Set):
   Linear Regression:
      MAE: 2.0987, RMSE: 8.7333
   Linear Regression (Log Target):
      MAE: 2.0723, RMSE: 8.7242
   Best Neural Network:
      MAE: 2.0905, RMSE: 8.9138

3. BEST NN MODEL CONFIGURATION:
   - neurons: [24, 32]
   - dropout_rate: 0.1
   - use_batchnorm: True
   - batch_size: 256
   - learning_rate: 0.001

4. KEY FINDINGS:
   - NN model improved baseline LR by -2.07% in RMSE
   - Residuals have mean: 0.0619 (near 0 is good)
   - Heteroscedasticity noted; butterfly pattern in residuals

============================================================
```

In [ ]: