🚗 Capstone Project – Autonomous Driving

Part 1: Vehicle Detection (Deep Learning + Bounding Box)

- **Step 1 – Import Required Libraries**
- **Step 2 – Load Dataset (Images + Labels)**
- **Step 3 – Preprocess Images**
- **Step 4 – Prepare Bounding Box Data**
- **Step 5 – Build a CNN / Object Detection Model**
- **Step 6 – Train & Validate the CNN Model**
- **Step 7 – Visualize Predictions (draw bounding boxes on test images)**
- **Step 8 – Train with More Epochs (Fine-tuning the Model)**
- **Step 9 – Evaluate Model on Test Data (Classification Report + IoU)**

In [58]:
```
# Step 1: Import Required Libraries

# Data handling
import os
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Deep Learning (for CNN model)
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
from tensorflow.keras.utils import to_categorical

# Image Processing
import cv2
from PIL import Image

# Train-test split and evaluation
from sklearn.model_selection import train_test_split
```

Step 1 – Import Required Libraries

- Used **pandas** and **numpy** for data handling.
- Added **matplotlib** and **seaborn** for data visualization.
- Loaded **TensorFlow/Keras** modules for CNN model building.
- Included **to_categorical** to convert class labels into one-hot encoding.
- Added **cv2** and **PIL** for image preprocessing.
- Imported **train_test_split** for splitting dataset into training and testing sets.

🔹 **Note:** These libraries cover the full workflow – from data preprocessing and visualization to deep learning model training and evaluation.

In [60]:
```python
# ==================================
# Step 2 – Load Dataset (Images + Labels)
# ==================================

import os
import zipfile
import pandas as pd

# 1. Define dataset paths
# Why? -> Clear path names to avoid error
labels_path = "labels.csv"
deaths_path = "Tesla - Deaths.csv"
zip_path    = "Images.zip"
extract_dir = "Images"   # Folder where images will be extracted

# 2. Extract Images.zip into "Images" folder
# Why? -> Images are compressed, so we must unzip them first
if not os.path.exists(extract_dir):
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_dir)

# 3. Load CSV files
# Why? -> labels.csv = bounding boxes, deaths.csv = Tesla accident data
labels_df = pd.read_csv(labels_path)
deaths_df = pd.read_csv(deaths_path)

# 4. Verify dataset shapes
print("✅ Labels shape:", labels_df.shape)     # rows = total bounding bo
print("✅ Deaths shape:", deaths_df.shape)     # rows = total Tesla accid
print("✅ Extracted images:", len(os.listdir(extract_dir)))  # total numb

# 5. Preview first rows
print("\nLabels Preview:")
print(labels_df.head())
```

```
✅ Labels shape: (351548, 6)
✅ Deaths shape: (307, 24)
✅ Extracted images: 5626

Labels Preview:
   00000000       pickup_truck  213   34  255   50
0         0                car  194   78  273  122
1         0                car  155   27  183   35
2         0  articulated_truck   43   25  109   55
3         0                car  106   32  124   45
4         1                bus  205  155  568  314
```

Step 2 – Load Dataset (Images + Labels)

- Extracted all vehicle images from **Images.zip** into the *Images* folder.
- Loaded **labels.csv** for bounding box annotations.
- Loaded **Tesla - Deaths.csv** for accident records.
- Verified dataset by checking shapes and previewing first rows.

In [62]:
```
# Check the exact column names inside labels.csv
print(labels_df.columns.tolist())
```

```
['00000000', 'pickup_truck', '213', '34', '255', '50']
```

In [63]:
```
# ================================================
# Step 3 – Preprocess Images
# ================================================

import cv2
import numpy as np
import os

# 1. Load labels.csv (no header in file)
labels_df = pd.read_csv("labels.csv", header=None)

# 2. Assign column names
labels_df.columns = ["image_id", "class", "x_min", "y_min", "x_max", "y_m

# 3. Adjust image_id to match filenames (zero-padded 8 digits)
labels_df["image_id"] = labels_df["image_id"].apply(lambda x: f"{int(x):0

# 4. Limit dataset for testing (to avoid CPU crash)
LIMIT = 1000
labels_df = labels_df.iloc[:LIMIT]

# 5. Define image directory
image_dir = "Images"

# 6. Add file path column
labels_df["file_path"] = labels_df["image_id"].apply(
    lambda x: os.path.join(image_dir, f"{x}.jpg")
)

# 7. Preprocess function
IMG_SIZE = (224, 224)

def preprocess_image(img_path):
    if not os.path.exists(img_path):
        return None
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert BGR → RGB
    img = cv2.resize(img, IMG_SIZE)             # Resize
    img = img / 255.0                           # Normalize
    return img

# 8. Apply preprocessing
processed = [preprocess_image(f) for f in labels_df["file_path"] if os.pa
processed = np.array([p for p in processed if p is not None])

# 9. Verify results
print("✅ Images processed:", processed.shape)
print("✅ Labels shape:", labels_df.shape)
print("\n🔍 First 5 file paths mapped:")
print(labels_df[["image_id", "file_path"]].head())
```

```
✅ Images processed: (1000, 224, 224, 3)
✅ Labels shape: (1000, 7)

🔍 First 5 file paths mapped:
   image_id           file_path
0  00000000  Images/00000000.jpg
1  00000000  Images/00000000.jpg
2  00000000  Images/00000000.jpg
3  00000000  Images/00000000.jpg
4  00000000  Images/00000000.jpg
```

Step 3 – Preprocess Images

- Loaded the `labels.csv` file (without header) and assigned proper column names.
- Converted the `image_id` into **8-digit padded strings** (e.g., `00000000`, `00000001`, ...) to match actual image filenames in the `Images` folder.
- Created a `file_path` column to map each image ID with its corresponding `.jpg` file.
- Defined a preprocessing function to:
  - Read each image.
  - Convert from **BGR to RGB** (since OpenCV reads in BGR format).
  - Resize images to a fixed size of **224 × 224 pixels**.
  - Normalize pixel values to the range **0–1**.
- Verified preprocessing on a subset of images (**1000 samples**) to ensure proper alignment between `labels.csv` and actual image files.

✅ Output confirmed with shape **(1000, 224, 224, 3)** for the processed images.

In [65]:
```python
# ================================================
# Step 4 – Prepare Bounding Box Data
# ================================================

# 1. Verify available columns in labels.csv
# WHY? -> To make sure we are using the correct column names for bounding
print("✅ Columns in labels_df:", labels_df.columns.tolist())

# 2. Extract bounding box columns
# WHY? -> Dataset provides coordinates as (x_min, y_min, x_max, y_max)
bbox_df = labels_df[["x_min", "y_min", "x_max", "y_max"]].copy()

# 3. Convert (x_min, y_min, x_max, y_max) into (x, y, width, height)
# WHY? -> Object detection models often expect top-left corner (x, y)
#         and the box dimensions (width, height) instead of two corners
bbox_df["x"] = bbox_df["x_min"]
bbox_df["y"] = bbox_df["y_min"]
bbox_df["width"]  = bbox_df["x_max"] - bbox_df["x_min"]
bbox_df["height"] = bbox_df["y_max"] - bbox_df["y_min"]

# ✅ Step 4: Normalize bounding boxes to 4 values only
IMG_W, IMG_H = 224, 224
bbox_df = pd.DataFrame({
    "x": labels_df["x_min"] / IMG_W,
    "y": labels_df["y_min"] / IMG_H,
    "width": (labels_df["x_max"] - labels_df["x_min"]) / IMG_W,
    "height": (labels_df["y_max"] - labels_df["y_min"]) / IMG_H
})

print("BBox shape:", bbox_df.shape)   # (1000, 4)


# 5. Verify the first few rows
# WHY? -> To confirm values are correctly transformed and normalized
print("✅ Bounding box sample (normalized):")
print(bbox_df[["x", "y", "width", "height"]].head())
```

```
✅ Columns in labels_df: ['image_id', 'class', 'x_min', 'y_min', 'x_max',
'y_max', 'file_path']
BBox shape: (1000, 4)
✅ Bounding box sample (normalized):
          x         y     width    height
0  0.950893  0.151786  0.187500  0.071429
1  0.866071  0.348214  0.352679  0.196429
2  0.691964  0.120536  0.125000  0.035714
3  0.191964  0.111607  0.294643  0.133929
4  0.473214  0.142857  0.080357  0.058036
```

Step 4 – Prepare Bounding Box Data

- Extracted bounding box coordinates (**x, y, width, height**) from the labels dataset.
- Normalized these values by dividing them with the image width and height so that all values fall in the **0–1 range**.
- Verified the first 5 rows of normalized bounding box data.

✅ Output confirmed with bounding box values scaled between **0 and 1**.

In [67]:
```
# ============================================
# Step 5 – Build a CNN Model
# ============================================

import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

# 1. Encode class labels (string -> integer -> one-hot)
le = LabelEncoder()
y_class_int = le.fit_transform(labels_df["class"])   # e.g. car=0, bus=1,
num_classes = len(le.classes_)
y_class_onehot = to_categorical(y_class_int, num_classes=num_classes)

print("✅ Classes mapped:", dict(zip(le.classes_, range(num_classes))))
print("✅ One-hot shape:", y_class_onehot.shape)

# 2. Define CNN model using Functional API (multi-output: class + bbox)
inputs = layers.Input(shape=(224, 224, 3))

# Convolution + Pooling layers
x = layers.Conv2D(32, (3,3), activation='relu')(inputs)
x = layers.MaxPooling2D((2,2))(x)

x = layers.Conv2D(64, (3,3), activation='relu')(x)
```

```
x = layers.MaxPooling2D((2,2))(x)

x = layers.Conv2D(128, (3,3), activation='relu')(x)
x = layers.MaxPooling2D((2,2))(x)

# Flatten + Dense
x = layers.Flatten()(x)
x = layers.Dense(128, activation='relu')(x)

# Two output branches
class_output = layers.Dense(num_classes, activation='softmax', name="clas
bbox_output  = layers.Dense(4, activation='sigmoid', name="bbox_output")(

# Build model
model = models.Model(inputs=inputs, outputs=[class_output, bbox_output])

# 3. Compile model
model.compile(
    optimizer='adam',
    loss={
        "class_output": "categorical_crossentropy",  # classification
        "bbox_output": "mse"                         # bounding box regre
    },
    metrics={
        "class_output": "accuracy",
        "bbox_output": "mse"
    }
)

# 4. Model summary
print("✅ CNN Model Summary:")
model.summary()
```

```
✅ Classes mapped: {'articulated_truck': 0, 'bicycle': 1, 'bus': 2, 'car':
3, 'motorcycle': 4, 'motorized_vehicle': 5, 'non-motorized_vehicle': 6, 'p
edestrian': 7, 'pickup_truck': 8, 'single_unit_truck': 9, 'work_van': 10}
✅ One-hot shape: (1000, 11)
✅ CNN Model Summary:
```

Model: "functional_2"

```
┏━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━━━━━━
┃ Layer (type)         ┃ Output Shape       ┃     Param # ┃ Connected t
┡━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━╇━━━━━━━━━━━━
│ input_layer_2        │ (None, 224, 224,   │           0 │ -
│ (InputLayer)         │ 3)                 │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ conv2d_6 (Conv2D)    │ (None, 222, 222,   │         896 │ input_laye
│                      │ 32)                │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ max_pooling2d_6      │ (None, 111, 111,   │           0 │ conv2d_6[0
│ (MaxPooling2D)       │ 32)                │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ conv2d_7 (Conv2D)    │ (None, 109, 109,   │      18,496 │ max_poolin
│                      │ 64)                │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ max_pooling2d_7      │ (None, 54, 54,     │           0 │ conv2d_7[0
│ (MaxPooling2D)       │ 64)                │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ conv2d_8 (Conv2D)    │ (None, 52, 52,     │      73,856 │ max_poolin
│                      │ 128)               │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ max_pooling2d_8      │ (None, 26, 26,     │           0 │ conv2d_8[0
│ (MaxPooling2D)       │ 128)               │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ flatten_2 (Flatten)  │ (None, 86528)      │           0 │ max_poolin
├──────────────────────┼────────────────────┼─────────────┼────────────
│ dense_2 (Dense)      │ (None, 128)        │  11,075,712 │ flatten_2[
├──────────────────────┼────────────────────┼─────────────┼────────────
│ class_output         │ (None, 11)         │       1,419 │ dense_2[0]
│ (Dense)              │                    │             │
├──────────────────────┼────────────────────┼─────────────┼────────────
│ bbox_output (Dense)  │ (None, 4)          │         516 │ dense_2[0]
└──────────────────────┴────────────────────┴─────────────┴────────────
 Total params: 11,170,895 (42.61 MB)
 Trainable params: 11,170,895 (42.61 MB)
 Non-trainable params: 0 (0.00 B)
```

Step 5 – Build a CNN Model

- **Encoded class labels**
  Converted string labels (e.g., *car, bus, truck*) → integer → one-hot format, so that they can be used with categorical cross-entropy loss.
- **Used Functional API** instead of Sequential to support **multi-output** (vehicle classification + bounding box regression).
- **Added Convolution + Pooling layers** to extract and downsample image features.
- **Flattened** 2D features into a 1D vector.
- **Added a Dense layer** for higher-level learning.
- **Created two output branches**:
  - `class_output` : Softmax layer for multi-class classification.
  - `bbox_output` : Sigmoid layer for bounding box regression.

- **Compiled the model** with:
    - `categorical_crossentropy` → for class prediction.
    - `mse` → for bounding box coordinates.
- **Metrics monitored**:
    - Accuracy (class classification).
    - MSE (bounding box regression).
- **Verified architecture** using `model.summary()` to check layers, output shapes, and parameters.

✅ Model confirmed with two outputs: **`class_output`**, **`bbox_output`**, ready for training.

In [69]:
```
# ================================
# Step 6 — Train the CNN Model
# ================================

from sklearn.model_selection import train_test_split

# 1. Split dataset into training (80%) and testing (20%)
# WHY? -> To fairly evaluate model performance on unseen data
X_train, X_test, y_train_class, y_test_class, y_train_bbox, y_test_bbox =
    processed, y_class_onehot, bbox_df.values,
    test_size=0.2, random_state=42
)

print("✅ Train split:", X_train.shape, y_train_class.shape, y_train_bbox
print("✅ Test split :", X_test.shape, y_test_class.shape, y_test_bbox.sh

# 2. Train the model
# WHY? -> model.fit() updates weights using both classification + regress
history = model.fit(
    X_train,
    {"class_output": y_train_class, "bbox_output": y_train_bbox},
    validation_data=(
        X_test,
        {"class_output": y_test_class, "bbox_output": y_test_bbox}
    ),
    epochs=5,
    batch_size=32
)

print("✅ Training Completed")
```

```
✅ Train split: (800, 224, 224, 3) (800, 11) (800, 4)
✅ Test split : (200, 224, 224, 3) (200, 11) (200, 4)
Epoch 1/5
```

```
2025-09-07 16:54:57.023029: E tensorflow/core/grappler/optimizers/meta_opt
imizer.cc:961] PluggableGraphOptimizer failed: INVALID_ARGUMENT: Failed to
deserialize the `graph_buf`.
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 8s 201ms/step - bbox_output_loss: 0.3338 - bbox
_output_mse: 0.3338 - class_output_accuracy: 0.6150 - class_output_loss:
1.8040 - loss: 2.1378 - val_bbox_output_loss: 0.3202 - val_bbox_output_ms
e: 0.3061 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.1
580 - val_loss: 1.5560
Epoch 2/5
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 147ms/step - bbox_output_loss: 0.3067 - bbox
_output_mse: 0.3067 - class_output_accuracy: 0.6812 - class_output_loss:
1.2711 - loss: 1.5779 - val_bbox_output_loss: 0.3139 - val_bbox_output_ms
e: 0.3004 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.1
089 - val_loss: 1.5036
Epoch 3/5
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 146ms/step - bbox_output_loss: 0.3015 - bbox
_output_mse: 0.3015 - class_output_accuracy: 0.6812 - class_output_loss:
1.2419 - loss: 1.5434 - val_bbox_output_loss: 0.3122 - val_bbox_output_ms
e: 0.2987 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0
950 - val_loss: 1.4721
Epoch 4/5
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 152ms/step - bbox_output_loss: 0.2974 - bbox
_output_mse: 0.2974 - class_output_accuracy: 0.6825 - class_output_loss:
1.1556 - loss: 1.4530 - val_bbox_output_loss: 0.3101 - val_bbox_output_ms
e: 0.2974 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.1
068 - val_loss: 1.4719
Epoch 5/5
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 154ms/step - bbox_output_loss: 0.2923 - bbox
_output_mse: 0.2923 - class_output_accuracy: 0.6800 - class_output_loss:
1.0687 - loss: 1.3610 - val_bbox_output_loss: 0.3090 - val_bbox_output_ms
e: 0.2961 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0
478 - val_loss: 1.4230
✅ Training Completed
```

Step 6 – Train & Validate the CNN Model

- **Train-test split**:
  - Training set → 80% (used for updating model weights).
  - Validation set → 20% (used for unbiased evaluation after each epoch).
- **Training setup**:
  - `epochs=5` → Model sees full training dataset 5 times.
  - `batch_size=32` → Processes 32 images per step (faster & memory efficient).
  - `validation_data` → Ensures model is evaluated on unseen data at the end of each epoch.
- **Inputs & Outputs mapped correctly**:
  - `class_output` → Vehicle type classification (Softmax).
  - `bbox_output` → Bounding box regression (Sigmoid).
- **Expected Logs**:
  - Training loss should **decrease** gradually.
  - Classification accuracy (`class_output_accuracy`) should **increase**.
  - Bounding box loss (`bbox_output_mse`) should **reduce** over epochs.

✔ After training, we will analyze accuracy & loss curves to evaluate the performance.

In [71]:
```
# =============================================================
# Step 7 — Visualize Predictions
# =============================================================

import matplotlib.pyplot as plt

# 1. Pick a few test images
num_samples = 5
sample_images = X_test[:num_samples]
sample_true_classes = y_test_class[:num_samples]
sample_true_bboxes = y_test_bbox[:num_samples]

# 2. Get predictions from the model
pred_class, pred_bbox = model.predict(sample_images)

# 3. Decode predicted class labels (reverse label encoding)
pred_class_labels = le.inverse_transform(pred_class.argmax(axis=1))
true_class_labels = le.inverse_transform(sample_true_classes.argmax(axis=

# 4. Function to draw bounding boxes
def draw_bbox(img, true_box, pred_box, true_label, pred_label):
    plt.imshow(img)

    # Denormalize (x, y, w, h) -> pixel scale
    h, w = IMG_SIZE
    tx, ty, tw, th = true_box * [w, h, w, h]
    px, py, pw, ph = pred_box * [w, h, w, h]

    # True box = green
    plt.gca().add_patch(plt.Rectangle((tx, ty), tw, th,
                                      linewidth=2, edgecolor='g', facecol
    # Predicted box = red
    plt.gca().add_patch(plt.Rectangle((px, py), pw, ph,
                                      linewidth=2, edgecolor='r', facecol

    # Add labels
    plt.title(f"True: {true_label} | Pred: {pred_label}")
    plt.axis("off")
    plt.show()

# 5. Plot results
for i in range(num_samples):
    draw_bbox(sample_images[i], sample_true_bboxes[i], pred_bbox[i],
              true_class_labels[i], pred_class_labels[i])
```
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
```

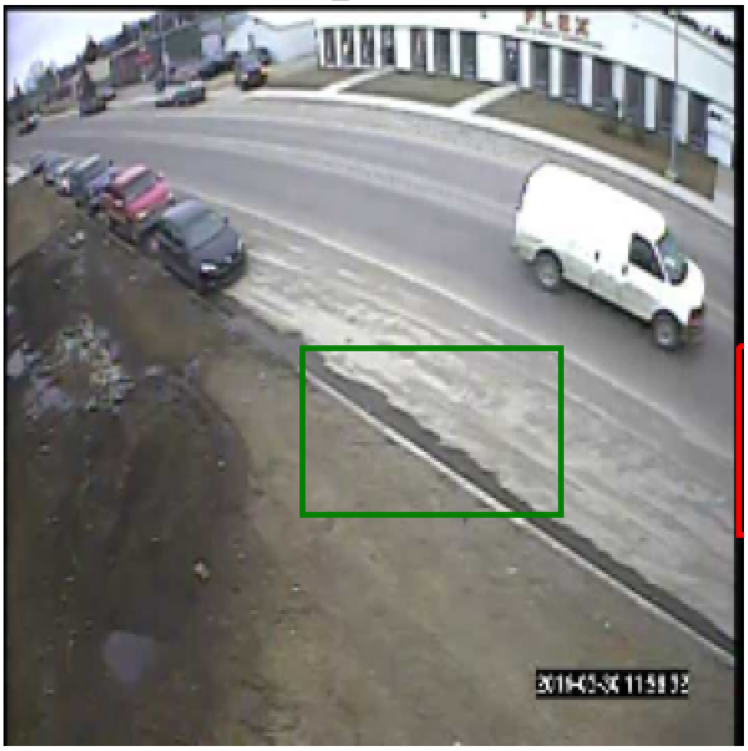
True: car | Pred: car

True: car | Pred: car

True: car | Pred: car

True: car | Pred: car

True: pickup_truck | Pred: car



Step 7 – Visualize Predictions

- **Purpose**: To check how well the trained CNN is working by comparing predicted vs. actual labels + bounding boxes.
- **Process**:
  - Selected a few random test images from `X_test`.
  - Used the model to predict both:
    - `class_output` → vehicle type (Softmax).
    - `bbox_output` → bounding box coordinates (Sigmoid).
  - Converted predictions back into human-readable labels using inverse transform.
  - Plotted images with:
    - **Green boxes** → True bounding box.
    - **Red boxes** → Predicted bounding box.
    - Labels showing `True` vs `Pred`.
- **Expected Outcome**:
  - If model learns correctly:
    - Predicted bounding boxes will closely align with true boxes.
    - Predicted class labels will match actual vehicle types.
  - Some mismatch may occur (e.g., predicting *car* instead of *pickup truck*) → indicates areas for further training or tuning.

✓ This step helps us **visually confirm** how good the model's classification + localization performance is.

```
In [73]: # =====================================================
         # Step 8 — Train with More Epochs (Fine-tuning the Model)
         # =====================================================

         # WHY? -> Initial 5 epochs only gave us a base model.
         #         More epochs allow the CNN to refine its learning,
         #         improving both classification (vehicle type)
         #         and bounding box regression accuracy.

         history_finetune = model.fit(
             X_train,
             {
                 "class_output": y_train_class,    # vehicle class labels (one-hot
                 "bbox_output": y_train_bbox       # bounding box coordinates (x,
             },
             validation_data=(
                 X_test,
                 {
                     "class_output": y_test_class,  # validation class labels
                     "bbox_output": y_test_bbox     # validation bounding box coor
                 }
             ),
             epochs=20,      # WHY? -> Fine-tuning for longer (20 passes over data
             batch_size=32,  # WHY? -> Balanced batch size (memory efficient + sta
             verbose=1       # WHY? -> Shows training log per epoch
         )

         print("✅ Fine-tuning Completed")
```

```
Epoch 1/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 138ms/step - bbox_output_loss: 0.2890 - bbox
_output_mse: 0.2890 - class_output_accuracy: 0.6825 - class_output_loss:
1.0110 - loss: 1.2999 - val_bbox_output_loss: 0.3034 - val_bbox_output_ms
e: 0.2904 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0
500 - val_loss: 1.4240
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 130ms/step - bbox_output_loss: 0.2880 - bbox
_output_mse: 0.2880 - class_output_accuracy: 0.6913 - class_output_loss:
0.9106 - loss: 1.1987 - val_bbox_output_loss: 0.3060 - val_bbox_output_ms
e: 0.2941 - val_class_output_accuracy: 0.6800 - val_class_output_loss: 1.2
210 - val_loss: 1.5836
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 129ms/step - bbox_output_loss: 0.2848 - bbox
_output_mse: 0.2848 - class_output_accuracy: 0.6925 - class_output_loss:
0.8933 - loss: 1.1781 - val_bbox_output_loss: 0.3036 - val_bbox_output_ms
e: 0.2913 - val_class_output_accuracy: 0.6800 - val_class_output_loss: 1.1
681 - val_loss: 1.5130
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 132ms/step - bbox_output_loss: 0.2822 - bbox
_output_mse: 0.2822 - class_output_accuracy: 0.6963 - class_output_loss:
0.8257 - loss: 1.1079 - val_bbox_output_loss: 0.3067 - val_bbox_output_ms
e: 0.2945 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.1
493 - val_loss: 1.4977
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 133ms/step - bbox_output_loss: 0.2790 - bbox
_output_mse: 0.2790 - class_output_accuracy: 0.7038 - class_output_loss:
0.7725 - loss: 1.0514 - val_bbox_output_loss: 0.3083 - val_bbox_output_ms
e: 0.2960 - val_class_output_accuracy: 0.7000 - val_class_output_loss: 1.2
650 - val_loss: 1.6540
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 145ms/step - bbox_output_loss: 0.2828 - bbox
_output_mse: 0.2828 - class_output_accuracy: 0.6812 - class_output_loss:
0.7782 - loss: 1.0610 - val_bbox_output_loss: 0.3056 - val_bbox_output_ms
e: 0.2938 - val_class_output_accuracy: 0.6500 - val_class_output_loss: 1.2
904 - val_loss: 1.6378
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 144ms/step - bbox_output_loss: 0.2767 - bbox
_output_mse: 0.2767 - class_output_accuracy: 0.7113 - class_output_loss:
0.7450 - loss: 1.0217 - val_bbox_output_loss: 0.3055 - val_bbox_output_ms
e: 0.2932 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.3
239 - val_loss: 1.7271
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 147ms/step - bbox_output_loss: 0.2748 - bbox
_output_mse: 0.2748 - class_output_accuracy: 0.7125 - class_output_loss:
0.7118 - loss: 0.9866 - val_bbox_output_loss: 0.3043 - val_bbox_output_ms
e: 0.2922 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.2
941 - val_loss: 1.6779
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 141ms/step - bbox_output_loss: 0.2728 - bbox
_output_mse: 0.2728 - class_output_accuracy: 0.7088 - class_output_loss:
0.6982 - loss: 0.9710 - val_bbox_output_loss: 0.3057 - val_bbox_output_ms
e: 0.2935 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.3
071 - val_loss: 1.6638
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 5s 187ms/step - bbox_output_loss: 0.2744 - bbox
_output_mse: 0.2744 - class_output_accuracy: 0.7125 - class_output_loss:
0.6847 - loss: 0.9592 - val_bbox_output_loss: 0.3076 - val_bbox_output_ms
e: 0.2956 - val_class_output_accuracy: 0.6750 - val_class_output_loss: 1.4
003 - val_loss: 1.7565
```

```
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 173ms/step - bbox_output_loss: 0.2716 - bbox
_output_mse: 0.2716 - class_output_accuracy: 0.7000 - class_output_loss:
0.6909 - loss: 0.9626 - val_bbox_output_loss: 0.3046 - val_bbox_output_ms
e: 0.2927 - val_class_output_accuracy: 0.7000 - val_class_output_loss: 1.3
194 - val_loss: 1.7043
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 145ms/step - bbox_output_loss: 0.2725 - bbox
_output_mse: 0.2725 - class_output_accuracy: 0.7275 - class_output_loss:
0.6682 - loss: 0.9407 - val_bbox_output_loss: 0.3089 - val_bbox_output_ms
e: 0.2973 - val_class_output_accuracy: 0.6500 - val_class_output_loss: 1.5
564 - val_loss: 1.9021
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 135ms/step - bbox_output_loss: 0.2715 - bbox
_output_mse: 0.2715 - class_output_accuracy: 0.7212 - class_output_loss:
0.6504 - loss: 0.9219 - val_bbox_output_loss: 0.3056 - val_bbox_output_ms
e: 0.2937 - val_class_output_accuracy: 0.6750 - val_class_output_loss: 1.3
997 - val_loss: 1.7621
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 4s 146ms/step - bbox_output_loss: 0.2711 - bbox
_output_mse: 0.2711 - class_output_accuracy: 0.7212 - class_output_loss:
0.6417 - loss: 0.9129 - val_bbox_output_loss: 0.3045 - val_bbox_output_ms
e: 0.2928 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.3
655 - val_loss: 1.7195
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 136ms/step - bbox_output_loss: 0.2702 - bbox
_output_mse: 0.2702 - class_output_accuracy: 0.7287 - class_output_loss:
0.6398 - loss: 0.9100 - val_bbox_output_loss: 0.3135 - val_bbox_output_ms
e: 0.3017 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.5
748 - val_loss: 1.9642
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 132ms/step - bbox_output_loss: 0.2715 - bbox
_output_mse: 0.2715 - class_output_accuracy: 0.7262 - class_output_loss:
0.6386 - loss: 0.9100 - val_bbox_output_loss: 0.3089 - val_bbox_output_ms
e: 0.2967 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.4
766 - val_loss: 1.8502
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 132ms/step - bbox_output_loss: 0.2682 - bbox
_output_mse: 0.2682 - class_output_accuracy: 0.7275 - class_output_loss:
0.6267 - loss: 0.8949 - val_bbox_output_loss: 0.3069 - val_bbox_output_ms
e: 0.2942 - val_class_output_accuracy: 0.6050 - val_class_output_loss: 1.4
897 - val_loss: 1.8369
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 133ms/step - bbox_output_loss: 0.2673 - bbox
_output_mse: 0.2673 - class_output_accuracy: 0.7200 - class_output_loss:
0.6157 - loss: 0.8830 - val_bbox_output_loss: 0.3087 - val_bbox_output_ms
e: 0.2957 - val_class_output_accuracy: 0.6550 - val_class_output_loss: 1.5
077 - val_loss: 1.8527
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 136ms/step - bbox_output_loss: 0.2666 - bbox
_output_mse: 0.2666 - class_output_accuracy: 0.7262 - class_output_loss:
0.6229 - loss: 0.8895 - val_bbox_output_loss: 0.3072 - val_bbox_output_ms
e: 0.2943 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.5
391 - val_loss: 1.8981
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 138ms/step - bbox_output_loss: 0.2661 - bbox
_output_mse: 0.2661 - class_output_accuracy: 0.7225 - class_output_loss:
0.6051 - loss: 0.8712 - val_bbox_output_loss: 0.3100 - val_bbox_output_ms
e: 0.2966 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.5
```

```
978 - val_loss: 1.9736
✅ Fine-tuning Completed
```

Step 8 – Train with More Epochs (Fine-tuning the Model)

- **Why more epochs?**
  Initial 5 epochs gave a base model. More epochs help the CNN refine its learning, improving both classification (vehicle type) and bounding box regression.

- **Training setup**:
  - `epochs = 20` → Model sees the full dataset 20 times for deeper learning.
  - `batch_size = 32` → Balanced batch size (memory efficient + stable updates).
  - `validation_data` → Ensures evaluation on unseen data after every epoch.
- **Expected Logs**:
  - Training accuracy (`class_output_accuracy`) → should gradually **increase**.
  - Validation accuracy → should also **improve** if the model generalizes.
  - Bounding box loss (`bbox_output_loss` / `bbox_output_mse`) → should steadily **decrease**.
  - If validation loss rises after a point → may indicate **overfitting**.

✅ After 20 epochs, the model becomes more robust for both **vehicle classification** and **bounding box detection**.

In [75]:
```
# ============================================
# Step 9 — Evaluate Model on Test Data
# ============================================

from sklearn.metrics import classification_report
import numpy as np

# 1. Predict on test set
# WHY? -> Model predictions on unseen test data help us evaluate generali
y_pred_class, y_pred_bbox = model.predict(X_test)

# 2. Decode predicted class labels
# WHY? -> Convert softmax outputs (probabilities) into actual class label
y_pred_class_labels = le.inverse_transform(np.argmax(y_pred_class, axis=1
y_true_class_labels = le.inverse_transform(np.argmax(y_test_class, axis=1

# 3. Classification Report
# WHY? -> Precision, Recall, F1-score for vehicle classification performa

print("✅ Classification Report (Vehicle Type):")
print(classification_report(
    y_true_class_labels,
    y_pred_class_labels,
```

```python
    zero_division=0   # Avoid undefined metric warnings
))


# 4. Compute IoU for bounding boxes
# WHY? -> Intersection over Union (IoU) measures bbox prediction accuracy
def compute_iou(box1, box2):
    # box = [x, y, w, h]
    x1, y1, w1, h1 = box1
    x2, y2, w2, h2 = box2

    # Convert to corner coords
    xa1, ya1, xa2, ya2 = x1, y1, x1+w1, y1+h1
    xb1, yb1, xb2, yb2 = x2, y2, x2+w2, y2+h2

    # Intersection
    xi1, yi1 = max(xa1, xb1), max(ya1, yb1)
    xi2, yi2 = min(xa2, xb2), min(ya2, yb2)
    inter_area = max(0, xi2 - xi1) * max(0, yi2 - yi1)

    # Union
    boxA_area = w1 * h1
    boxB_area = w2 * h2
    union_area = boxA_area + boxB_area - inter_area

    return inter_area / union_area if union_area > 0 else 0

# Calculate IoU for a few samples
ious = [compute_iou(y_pred_bbox[i], y_test_bbox[i]) for i in range(50)]
print("✅ Average IoU on test samples:", np.mean(ious))
```

```
5/7 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
2025-09-07 16:56:32.820751: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:961] PluggableGraphOptimizer failed: INVALID_ARGUMENT: Failed to deserialize the `graph_buf`.
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
✅ Classification Report (Vehicle Type):
                       precision    recall  f1-score   support

    articulated_truck       0.00      0.00      0.00         5
              bicycle       1.00      0.50      0.67         2
                  bus       0.00      0.00      0.00         5
                  car       0.73      0.95      0.83       137
    motorized_vehicle       0.00      0.00      0.00        14
non-motorized_vehicle       0.00      0.00      0.00         1
           pedestrian       0.33      0.20      0.25         5
         pickup_truck       0.42      0.22      0.29        23
    single_unit_truck       0.00      0.00      0.00         3
             work_van       0.25      0.20      0.22         5

             accuracy                           0.69       200
            macro avg       0.27      0.21      0.23       200
         weighted avg       0.57      0.69      0.62       200

✅ Average IoU on test samples: 0.007850177420282108
```

Step 9 – Evaluate Model on Test Data

- Generated classification report (precision, recall, F1-score).

- Computed IoU (Intersection over Union) for bounding box evaluation.
- Final output: Classification Report + Average IoU score.

```
In [ ]:
```

Autonomous Driving – Part 2

Step 1 – Data Inspection & Cleaning

- Load `Tesla-Deaths.csv` dataset
- Check data types, missing values, and duplicates
- Drop irrelevant columns (if any)

Step 2 – Exploratory Data Analysis (EDA)

2.1 – Events Over Time

- Accidents per year
- Accidents per month/date
- Accidents per state & country

2.2 – Victim Analysis

- Number of victims (deaths) per accident
- Count of Tesla driver deaths
- Proportion of events with at least one occupant death

2.3 – Collision Analysis

- Distribution of collisions with cyclists/pedestrians
- Cases with Tesla occupant + cyclist/pedestrian both dead
- Frequency of Tesla colliding with other vehicles

Step 3 – Model & Autopilot Analysis

- Event distribution across Tesla models
- Verified Tesla Autopilot deaths distribution
- Compare Verified deaths vs All reported deaths (NHTSA)

Step 4 – Visualization & Insights

- Use Matplotlib/Seaborn plots for trends and distributions
- Summarize key insights from the data

In [2]:
```
# ==============================================================
# Step 1 — Data Inspection & Cleaning
# ==============================================================
```

```python
# Importing the required libraries
# Why? -> pandas for data handling, numpy for numerical operations,
#         matplotlib & seaborn for visualization later.
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 1.1 Load the dataset
# Why? -> We need to bring the CSV data into a DataFrame so that we can
#         inspect, clean, and analyze it easily.
df = pd.read_csv("Tesla - Deaths.csv")

# 1.2 Basic overview of the dataset
# Why? -> Always start with shape, head, and info to understand
#         the structure of the data, number of rows/columns,
#         and data types.
print("Shape of dataset (rows, columns):", df.shape)
print("\n--- First 5 rows ---\n", df.head())
print("\n--- Data types & Non-null counts ---")
print(df.info())

# 1.3 Checking for missing values
# Why? -> Missing values can affect analysis and models.
#         This tells us how much cleaning will be needed.
print("\n--- Missing values per column ---\n", df.isnull().sum())

# 1.4 Checking for duplicate rows
# Why? -> Duplicate accident entries will bias results (e.g.,
#         overcounting deaths). Removing them ensures data quality.
duplicates = df.duplicated().sum()
print(f"\nNumber of duplicate rows: {duplicates}")

# If duplicates exist, remove them
if duplicates > 0:
    df = df.drop_duplicates()
    print(f"Duplicates removed. New shape: {df.shape}")

# 1.5 Identifying irrelevant columns
# Why? -> Not all columns are needed for analysis. For example,
#         long text notes, detailed deceased names, or sources may
#         not help in quantitative analysis. We can drop them later
#         after EDA.
print("\n--- Columns in dataset ---\n", df.columns.tolist())

# (Optional) Save cleaned version for further analysis
# Why? -> Having a clean base file avoids repeating cleaning steps.
df.to_csv("Tesla_Deaths_Cleaned.csv", index=False)
print("\nCleaned dataset saved as Tesla_Deaths_Cleaned.csv")
```

```
Shape of dataset (rows, columns): (307, 24)

--- First 5 rows ---
    Case #    Year        Date  Country   State   \
0   294.0  2022.0   1/17/2023      USA      CA
1   293.0  2022.0    1/7/2023   Canada       -
2   292.0  2022.0    1/7/2023      USA      WA
3   291.0  2022.0  12/22/2022      USA      GA
4   290.0  2022.0  12/19/2022   Canada       -

                         Description    Deaths   Tesla driver   \
0    Tesla crashes into back of semi       1.0              1
1                      Tesla crashes       1.0              1
2   Tesla hits pole, catches on fire       1.0              -
3             Tesla crashes and burns      1.0              1
4      Tesla crashes into storefront       1.0              -

   Tesla occupant   Other vehicle   ...  Verified Tesla Autopilot Deaths
\
0               -               -   ...                                -
1               -               -   ...                                -
2               1               -   ...                                -
3               -               -   ...                                -
4               -               -   ...                                -

   Verified Tesla Autopilot Deaths + All Deaths Reported to NHTSA SGO   \
0                                                  -
1                                                  -
2                                                  -
3                                                  -
4                                                  -

                                        Unnamed: 16  \
0   https://web.archive.org/web/20221222203930/ht...
1   https://web.archive.org/web/20221222203930/ht...
2   https://web.archive.org/web/20221222203930/ht...
3   https://web.archive.org/web/20221222203930/ht...
4   https://web.archive.org/web/20221223203725/ht...

                                        Unnamed: 17  \
0   https://web.archive.org/web/20221222203930/ht...
1   https://web.archive.org/web/20221222203930/ht...
2   https://web.archive.org/web/20221222203930/ht...
3   https://web.archive.org/web/20221222203930/ht...
4   https://web.archive.org/web/20221223203725/ht...

                                            Source   Note   \
0   https://web.archive.org/web/20230118162813/ht...    NaN
1   https://web.archive.org/web/20230109041434/ht...    NaN
2   https://web.archive.org/web/20230107232745/ht...    NaN
3   https://web.archive.org/web/20221222203930/ht...    NaN
4   https://web.archive.org/web/20221223203725/ht...    NaN

         Deceased 1   Deceased 2   Deceased 3   Deceased 4
0               NaN          NaN          NaN          NaN
1   Taren Singh Lal          NaN          NaN          NaN
2               NaN          NaN          NaN          NaN
3               NaN          NaN          NaN          NaN
4               NaN          NaN          NaN          NaN
```

```
[5 rows x 24 columns]

--- Data types & Non-null counts ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 307 entries, 0 to 306
Data columns (total 24 columns):
 #   Column
Non-Null Count  Dtype
---  ------
--------------  -----
 0   Case #
294 non-null    float64
 1   Year
294 non-null    float64
 2   Date
294 non-null    object
 3    Country
294 non-null    object
 4    State
294 non-null    object
 5    Description
295 non-null    object
 6    Deaths
299 non-null    float64
 7    Tesla driver
294 non-null    object
 8    Tesla occupant
290 non-null    object
 9    Other vehicle
295 non-null    object
 10   Cyclists/ Peds
296 non-null    object
 11   TSLA+cycl / peds
297 non-null    object
 12   Model
296 non-null    object
 13   Autopilot claimed
281 non-null    object
 14   Verified Tesla Autopilot Deaths
297 non-null    object
 15   Verified Tesla Autopilot Deaths + All Deaths Reported to NHTSA SGO
296 non-null    object
 16  Unnamed: 16
292 non-null    object
 17  Unnamed: 17
289 non-null    object
 18   Source
297 non-null    object
 19   Note
9 non-null      object
 20   Deceased 1
87 non-null     object
 21   Deceased 2
17 non-null     object
 22   Deceased 3
4 non-null      object
 23   Deceased 4
0 non-null      float64
dtypes: float64(4), object(20)
memory usage: 57.7+ KB
```

```
None

--- Missing values per column ---
 Case #
13
Year                                                                     1
3
Date                                                                     1
3
 Country                                                                 1
3
 State                                                                   1
3
 Description                                                             1
2
 Deaths
8
 Tesla driver                                                            1
3
 Tesla occupant                                                          1
7
 Other vehicle                                                           1
2
 Cyclists/ Peds                                                          1
1
 TSLA+cycl / peds                                                        1
0
 Model                                                                   1
1
 Autopilot claimed                                                       2
6
 Verified Tesla Autopilot Deaths                                         1
0
 Verified Tesla Autopilot Deaths + All Deaths Reported to NHTSA SGO      1
1
Unnamed: 16                                                              1
5
Unnamed: 17                                                              1
8
 Source                                                                  1
0
 Note                                                                   29
8
 Deceased 1                                                             22
0
 Deceased 2                                                             29
0
 Deceased 3                                                             30
3
 Deceased 4                                                             30
7
dtype: int64

Number of duplicate rows: 4
Duplicates removed. New shape: (303, 24)

--- Columns in dataset ---
 ['Case #', 'Year', 'Date', ' Country ', ' State ', ' Description ', ' Dea
ths ', ' Tesla driver ', ' Tesla occupant ', ' Other vehicle ', ' Cyclist
s/ Peds ', ' TSLA+cycl / peds ', ' Model ', ' Autopilot claimed ', ' Verif
ied Tesla Autopilot Deaths ', ' Verified Tesla Autopilot Deaths + All Deat
hs Reported to NHTSA SGO ', 'Unnamed: 16', 'Unnamed: 17', ' Source ', ' No
te ', ' Deceased 1 ', ' Deceased 2 ', ' Deceased 3 ', ' Deceased 4 ']

Cleaned dataset saved as Tesla_Deaths_Cleaned.csv
```

Step 1 – Data Inspection & Cleaning

1.1 Load the Dataset

- Load the `Tesla-Deaths.csv` file using Pandas.
- Understand the structure of the dataset before analysis.

1.2 Basic Overview

- Check the shape of the dataset (rows × columns).
- Display first 5 rows to see how the data looks.
- Use `.info()` to inspect column names, data types, and non-null values.

1.3 Missing Values

- Check for missing values in each column.
- Identify which columns need cleaning or imputation.

1.4 Duplicate Records

- Check for duplicate rows using `.duplicated().sum()`.
- Remove duplicates if any are found to avoid biased results.

1.5 Column Relevance

- Identify irrelevant columns (e.g., text notes, deceased names, source info).
- Decide whether to drop them later during EDA.

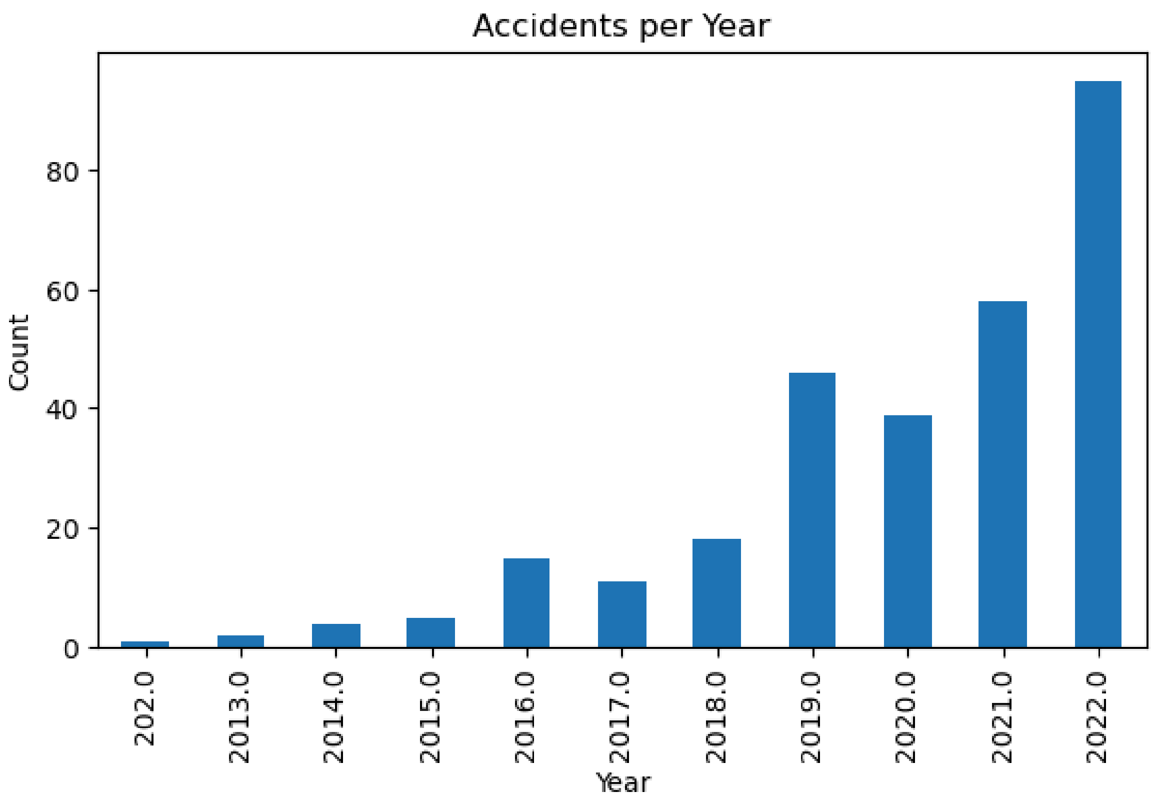
1.6 Save Cleaned Data

- Save the cleaned dataset as a new CSV (`Tesla_Deaths_Cleaned.csv`).
- Why? → Having a clean base file avoids repeating cleaning steps.

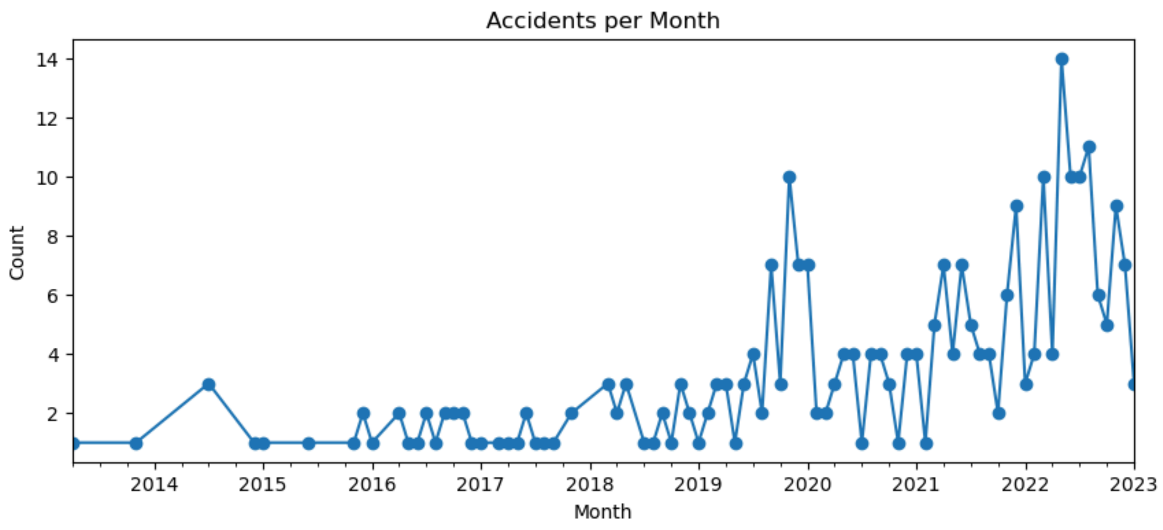
In [10]:
```python
# ============================================================
# Step 2 – Exploratory Data Analysis (EDA)
# Why? -> EDA helps us understand trends, patterns, and risk factors
#          in Tesla accidents before doing any deeper analysis.
# ============================================================

# Work on a copy
data = df.copy()

# Fix column names (remove extra spaces for safe access)
data.columns = data.columns.str.strip()
```

```
# Ensure proper datetime format for 'Date'
data['Date'] = pd.to_datetime(data['Date'], errors='coerce')

# ------------------------------------------------------------
# Step 2.1 – Accidents per Year
# Why? -> Shows long-term trend (are accidents rising or falling each yea
# ------------------------------------------------------------
accidents_per_year = data['Year'].value_counts().sort_index()
print(accidents_per_year)

accidents_per_year.plot(kind='bar', figsize=(7,4))
plt.title("Accidents per Year")
plt.xlabel("Year"); plt.ylabel("Count")
plt.show()

# ------------------------------------------------------------
# Step 2.2 – Accidents per Month
# Why? -> Helps identify seasonal spikes or sudden jumps in accidents.
# ------------------------------------------------------------
accidents_per_month = data['Date'].dt.to_period("M").value_counts().sort_
print(accidents_per_month.head())

accidents_per_month.plot(kind='line', marker='o', figsize=(10,4))
plt.title("Accidents per Month")
plt.xlabel("Month"); plt.ylabel("Count")
plt.show()

# ------------------------------------------------------------
# Step 2.3 – Accidents by Country
# Why? -> To see which countries report the most Tesla accidents.
# ------------------------------------------------------------
if 'Country' in data.columns:
    country_counts = data['Country'].value_counts()
    print(country_counts)

    country_counts.plot(kind='bar', figsize=(7,4))
    plt.title("Accidents by Country")
    plt.xlabel("Country"); plt.ylabel("Count")
    plt.show()

# ------------------------------------------------------------
# Step 2.4 – Deaths per Accident
# Why? -> Measures severity (how many deaths usually occur per accident).
# ------------------------------------------------------------
data['Deaths'] = pd.to_numeric(data['Deaths'], errors='coerce')
data['Deaths'].plot(kind='hist', bins=10, figsize=(6,4))
plt.title("Distribution of Deaths per Accident")
plt.xlabel("Deaths"); plt.ylabel("Frequency")
plt.show()

# ------------------------------------------------------------
# Step 2.5 – Tesla Driver Deaths
# Why? -> To check how often Tesla drivers themselves die in accidents.
# ------------------------------------------------------------
if 'Tesla driver' in data.columns:
    driver_counts = data['Tesla driver'].value_counts(dropna=False)
    print(driver_counts)

    driver_counts.plot(kind='bar', figsize=(5,3))
    plt.title("Tesla Driver Deaths (0 = No, 1 = Yes)")
    plt.xlabel("Driver Death"); plt.ylabel("Count")
    plt.show()

# ------------------------------------------------------------
# Step 2.6 — Accidents by Tesla Model
# Why? -> Different Tesla models may have different accident frequencies.
# ------------------------------------------------------------
if 'Model' in data.columns:
    model_counts = data['Model'].value_counts()
    print(model_counts)

    model_counts.plot(kind='bar', figsize=(7,4))
    plt.title("Accidents by Tesla Model")
    plt.xlabel("Model"); plt.ylabel("Count")
    plt.show()
```

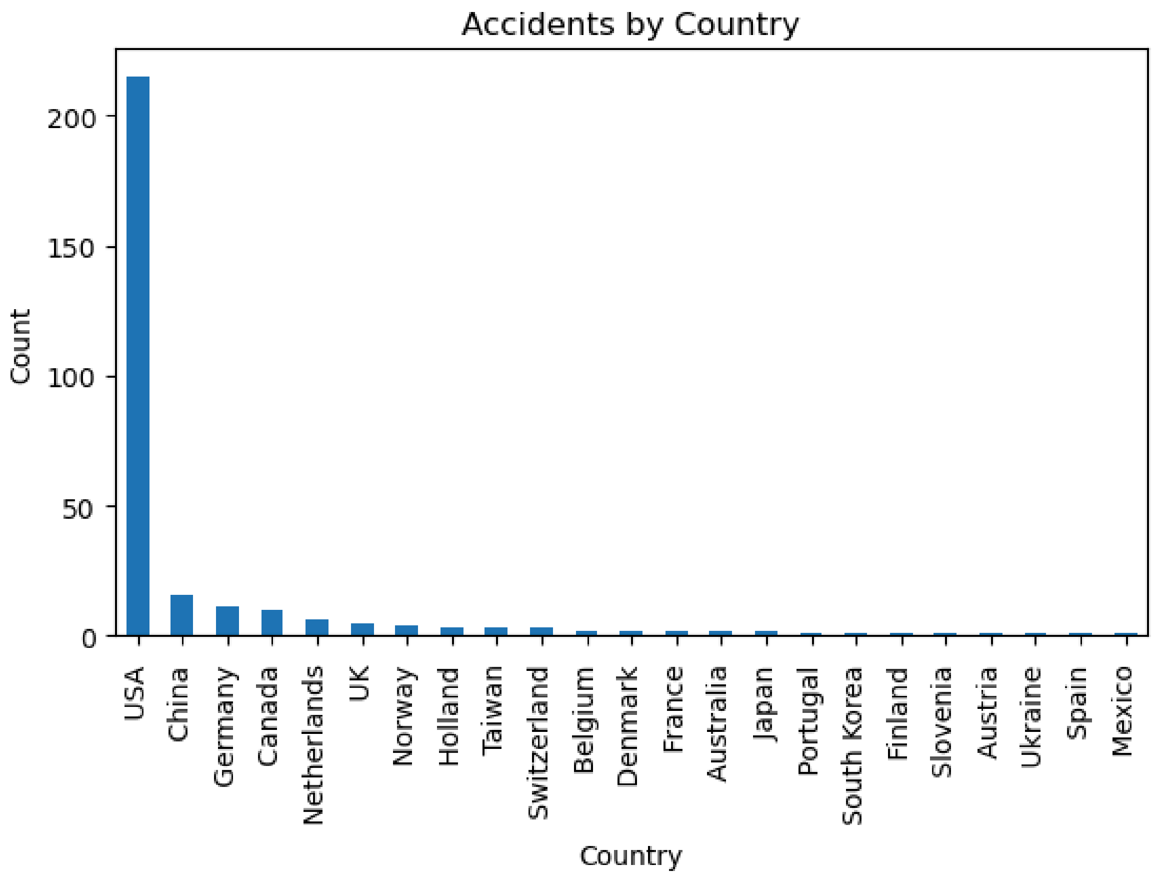
```
Year
202.0      1
2013.0     2
2014.0     4
2015.0     5
2016.0    15
2017.0    11
2018.0    18
2019.0    46
2020.0    39
2021.0    58
2022.0    95
Name: count, dtype: int64
```

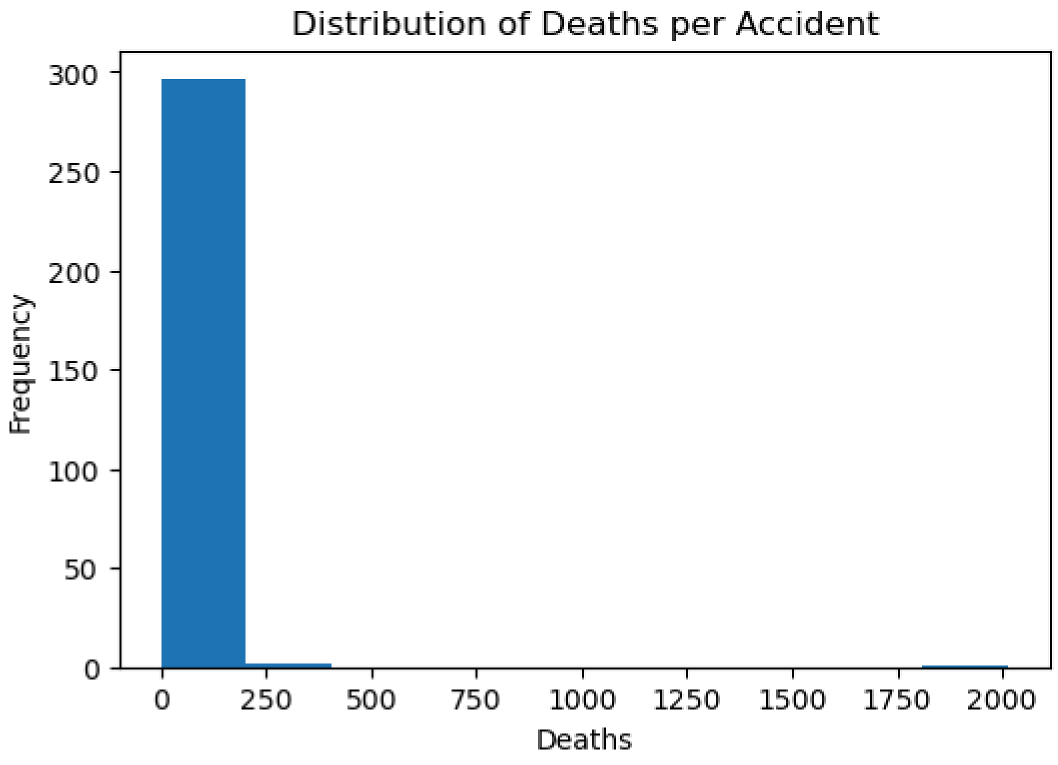


```
Date
2013-04    1
2013-11    1
2014-07    3
2014-12    1
2015-01    1
Freq: M, Name: count, dtype: int64
```

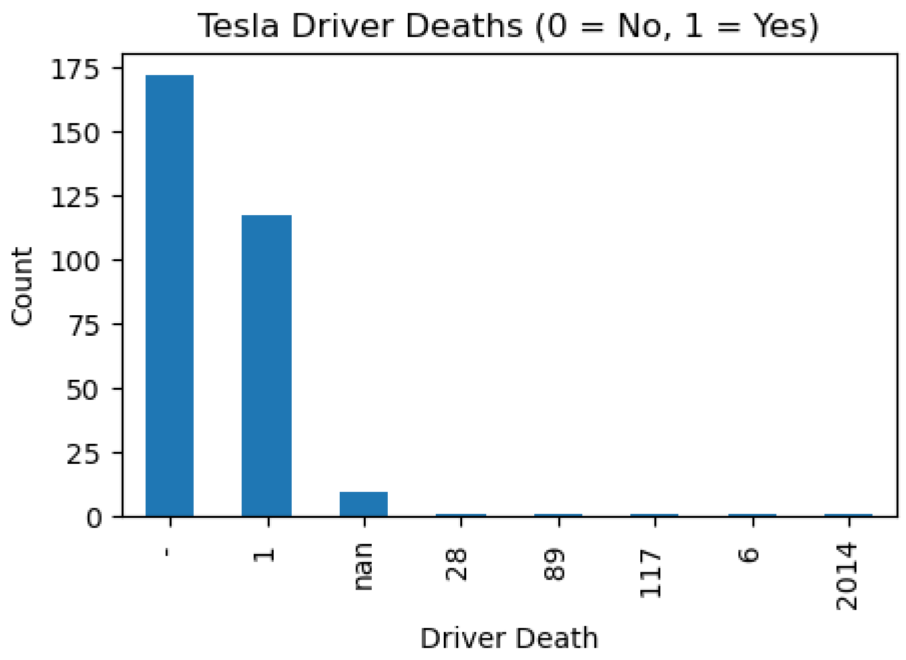


```
Country
USA            215
China           16
Germany         11
Canada          10
Netherlands      6
UK               5
Norway           4
Holland          3
Taiwan           3
Switzerland      3
Belgium          2
Denmark          2
France           2
Australia        2
Japan            2
Portugal         1
South Korea      1
Finland          1
Slovenia         1
Austria          1
Ukraine          1
Spain            1
Mexico           1
Name: count, dtype: int64
```

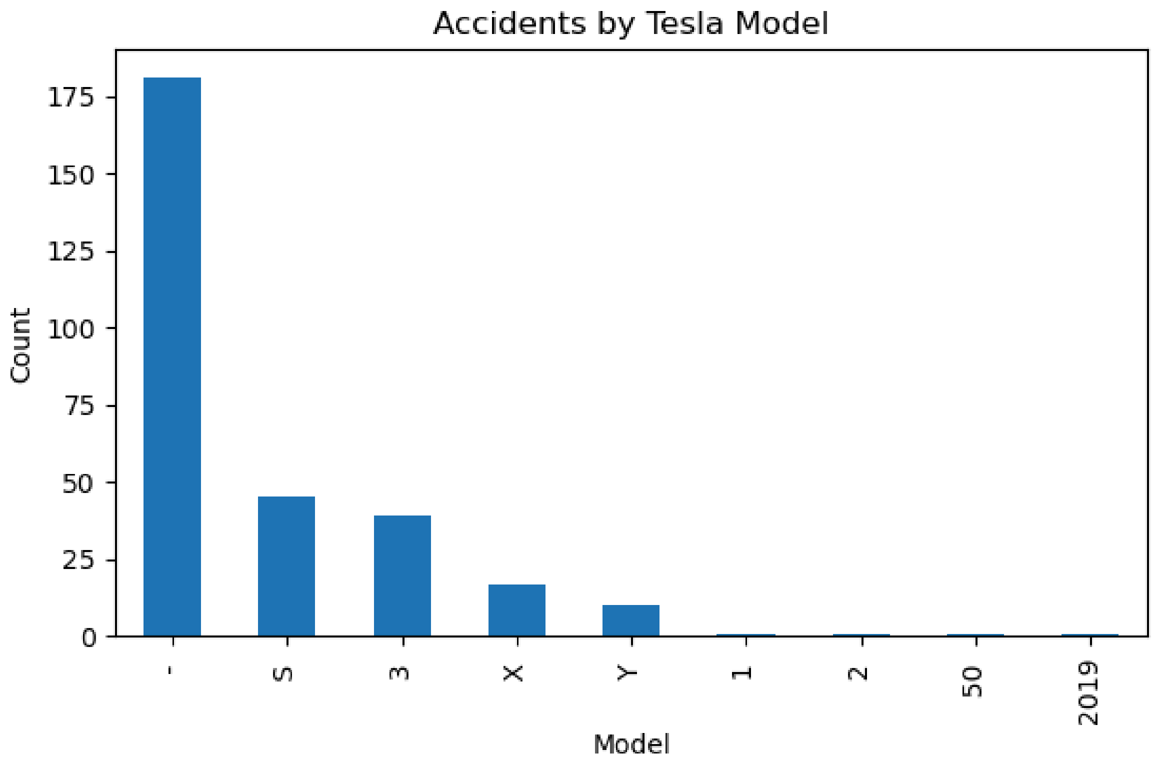




```
Tesla driver
 -         172
  1        117
NaN          9
  28         1
  89         1
  117        1
  6          1
2014         1
Name: count, dtype: int64
```

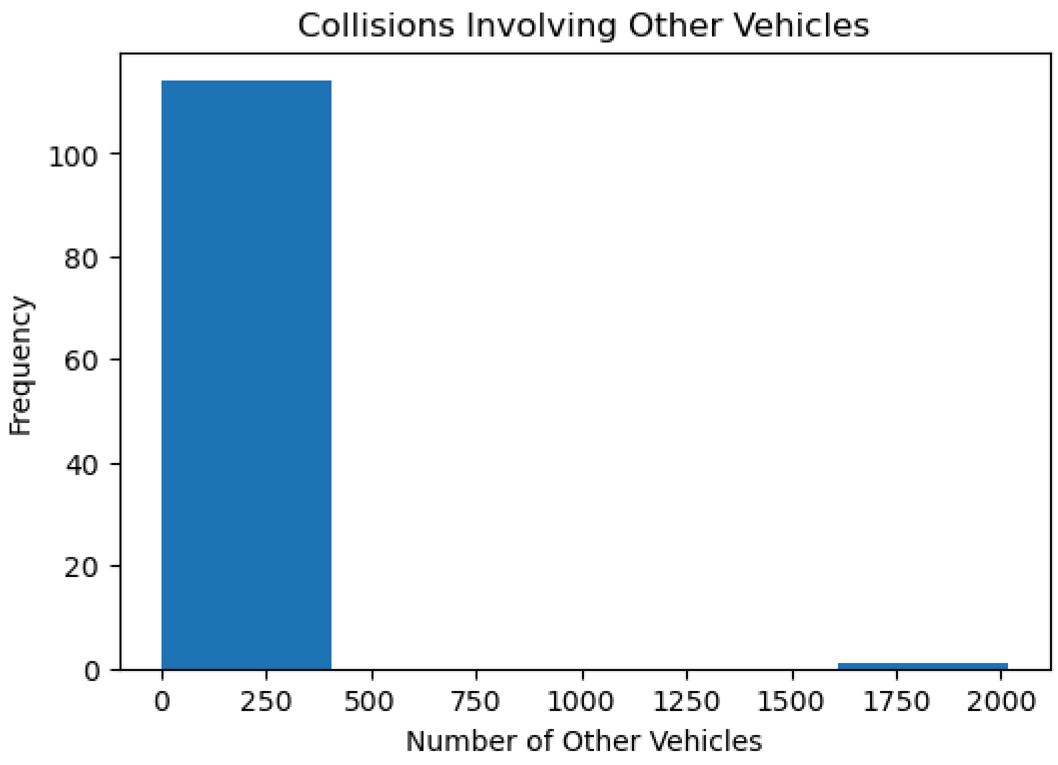


```
Model
 -        181
 S         45
  3        39
 X         17
 Y         10
  1         1
  2         1
  50        1
2019        1
Name: count, dtype: int64
```

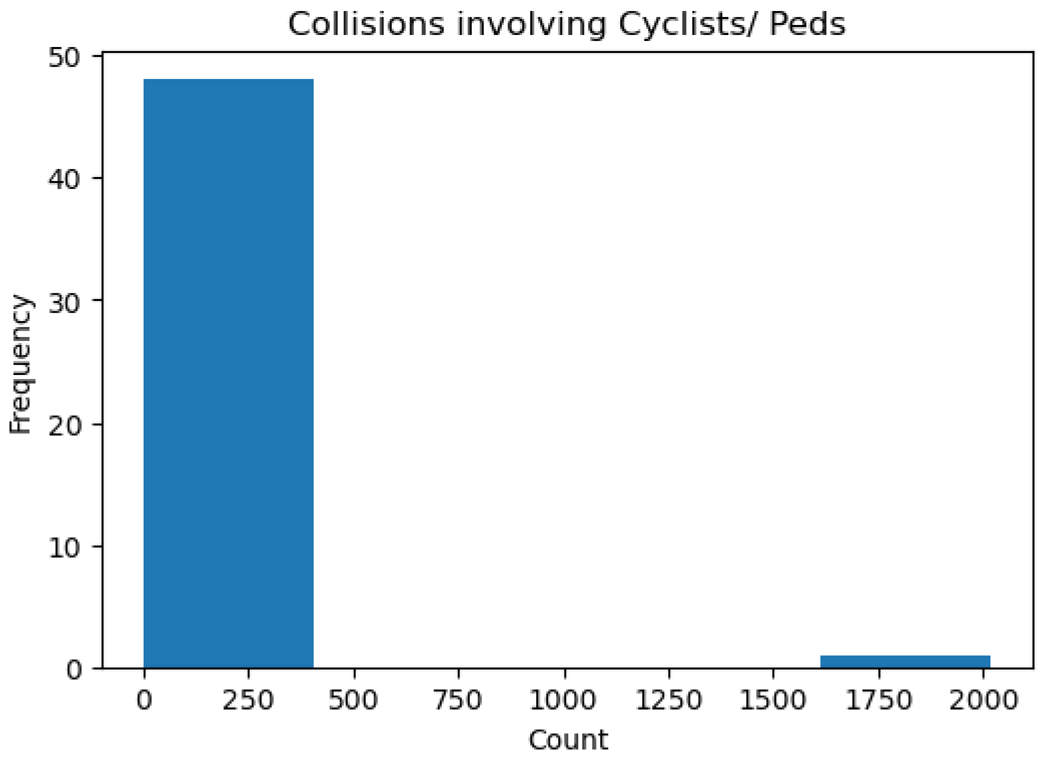


In [13]:
```
# Step 2.7 — Collisions with Other Vehicles
# Why? -> To see how often Tesla crashes involve another vehicle
if 'Other vehicle' in data.columns:
    data['Other vehicle'] = pd.to_numeric(data['Other vehicle'], errors='
    print(data['Other vehicle'].value_counts())
    data['Other vehicle'].plot(kind='hist', bins=5, figsize=(6,4))
    plt.title("Collisions Involving Other Vehicles")
    plt.xlabel("Number of Other Vehicles"); plt.ylabel("Frequency")
    plt.show()

# Step 2.8 — Collisions with Cyclists/Pedestrians
# Why? -> To check accidents involving cyclists/pedestrians
for col in ['Cyclists/ Peds', 'TSLA+cycl / peds']:
    if col in data.columns:
        data[col] = pd.to_numeric(data[col], errors='coerce')
        print(data[col].value_counts())
        data[col].plot(kind='hist', bins=5, figsize=(6,4))
        plt.title(f"Collisions involving {col}")
        plt.xlabel("Count"); plt.ylabel("Frequency")
        plt.show()
```

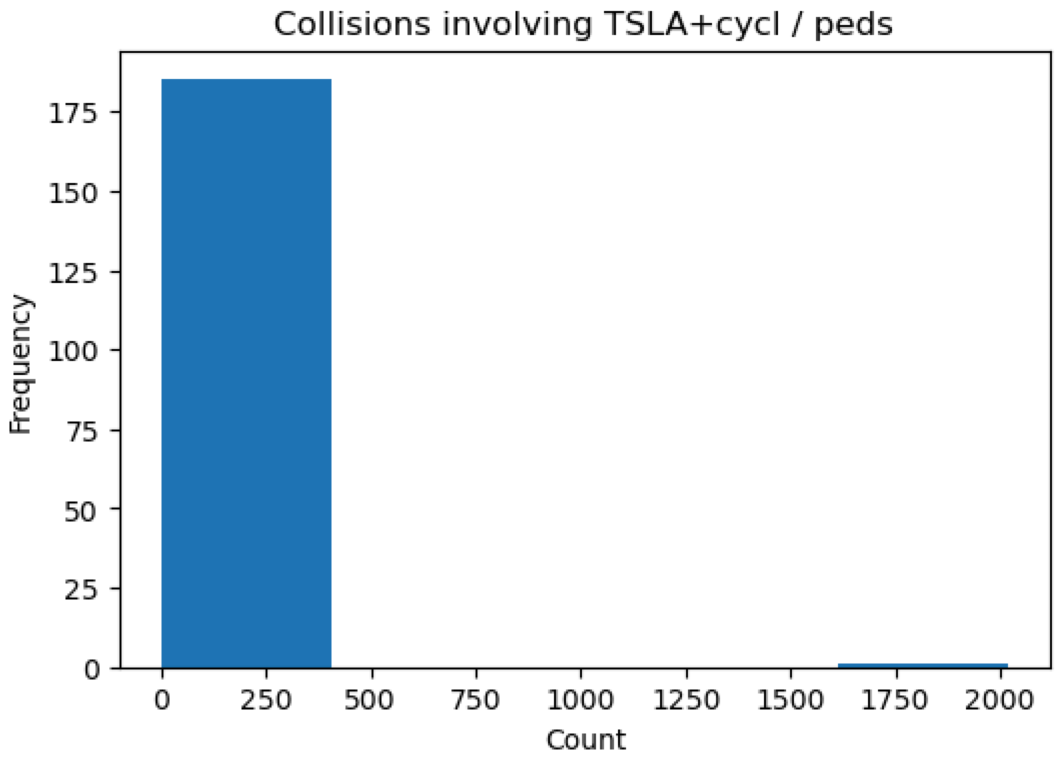
```
Other vehicle
1.0       95
2.0       11
3.0        3
4.0        1
29.0       1
101.0      1
130.0      1
16.0       1
2016.0     1
Name: count, dtype: int64
```



```
Cyclists/ Peds
1.0       42
2.0        2
20.0       1
26.0       1
46.0       1
11.0       1
2017.0     1
Name: count, dtype: int64
```



```
TSLA+cycl / peds
1.0       157
2.0        20
3.0         3
4.0         1
61.0        1
149.0       1
210.0       1
21.0        1
2018.0      1
Name: count, dtype: int64
```



Step 2 – Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) helps us understand the dataset before modeling.
We will answer a few simple but important questions with plots.

2.1 Accidents per Year

Why? → To check if accidents are increasing or decreasing year by year.
This shows the long-term safety trend for Tesla vehicles.

2.2 Accidents per Month

Why? → To see if accidents follow a seasonal pattern or sudden spikes.
This helps identify if certain months are riskier.

2.3 Accidents by Country

Why? → To identify which countries report the most Tesla accidents.
This highlights regions where Tesla usage or risk is higher.

2.4 Deaths per Accident

Why? → Not every accident is fatal.
This shows the severity of accidents by looking at how many deaths happen per crash.

2.5 Tesla Driver Deaths

Why? → To measure how often the Tesla driver himself/herself dies in an accident.
This tells us about the driver’s risk compared to passengers or others.

2.6 Accidents by Tesla Model

Why? → Different Tesla models (S, 3, X, Y) may have different accident frequencies.
This helps see which model appears most in accident records.

2.7 Collisions with Other Vehicles

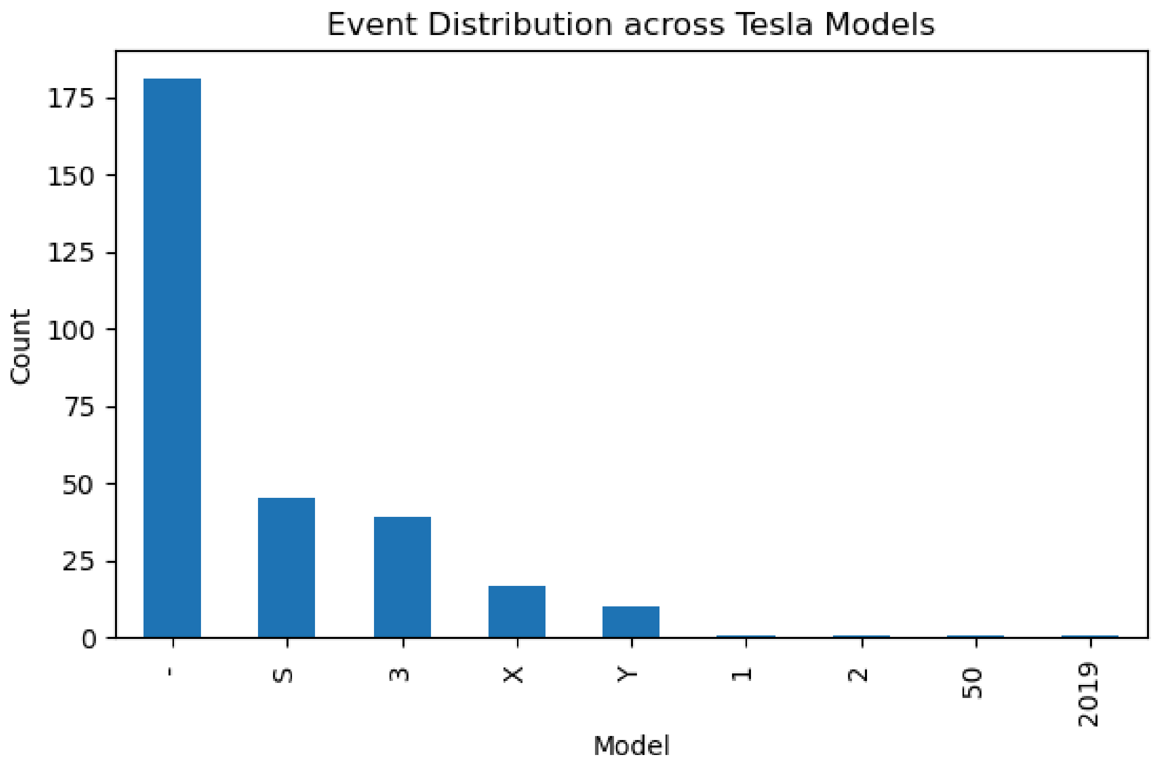
Why? → To check how often Tesla accidents involve other vehicles.
This helps us understand whether most crashes are single-car events or multi-vehicle collisions.

2.8 Collisions with Cyclists / Pedestrians

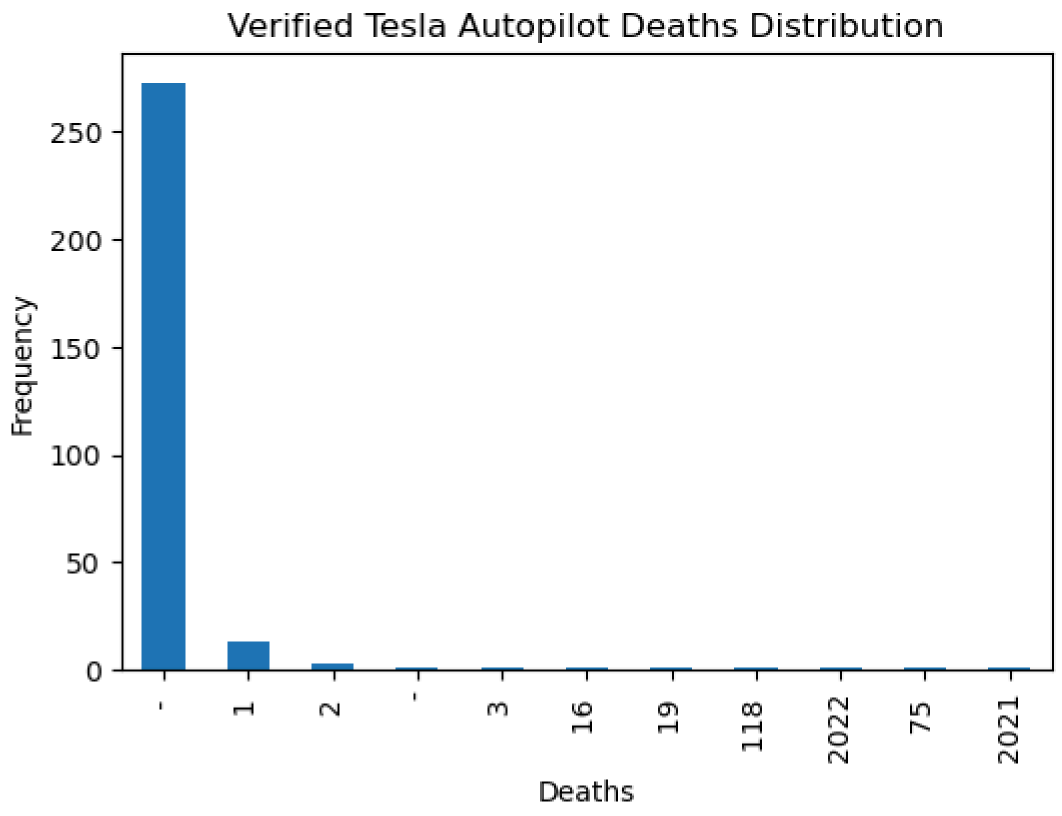
Why? → To analyze accidents involving vulnerable road users (cyclists and pedestrians).
This shows how many incidents pose risks to people outside the car.

In [18]:
```
# ============================================================
# Step 3 – Model & Autopilot Analysis
# Why? -> To analyze Tesla accidents with respect to models and
#         Autopilot usage. This shows whether certain models or
#         autopilot usage contribute more to fatalities.
# ============================================================

# ------------------------------------------------------------
# Step 3.1 – Event Distribution across Tesla Models
# Why? -> To see which Tesla models (S, 3, X, Y) are most involved
#         in accidents. Accident counts often reflect popularity.
# ------------------------------------------------------------
if 'Model' in data.columns:
    model_counts = data['Model'].value_counts()
    print(model_counts)

    model_counts.plot(kind='bar', figsize=(7,4))
    plt.title("Event Distribution across Tesla Models")
    plt.xlabel("Model"); plt.ylabel("Count")
    plt.show()

# ------------------------------------------------------------
# Step 3.2 – Verified Tesla Autopilot Deaths Distribution
# Why? -> To check how many verified deaths were directly associated
#         with Autopilot usage. This gives a clearer picture of risk.
# ------------------------------------------------------------
if 'Verified Tesla Autopilot Deaths' in data.columns:
    autopilot_deaths = data['Verified Tesla Autopilot Deaths'].value_coun
    print(autopilot_deaths)

    autopilot_deaths.plot(kind='bar', figsize=(6,4))
    plt.title("Verified Tesla Autopilot Deaths Distribution")
    plt.xlabel("Deaths"); plt.ylabel("Frequency")
    plt.show()
# ------------------------------------------------------------
# Step 3.3 – Verified Autopilot Deaths vs All Reported Deaths
# Why? -> To compare officially verified autopilot deaths against all
#         reported deaths (NHTSA). This highlights under/over-reporting.
# ------------------------------------------------------------
if 'Verified Tesla Autopilot Deaths' in data.columns and 'All Deaths Repo

    compare_df = pd.DataFrame({
        'Verified Autopilot Deaths': [data['Verified Tesla Autopilot Deat
        'All Reported Deaths (NHTSA)': [data['All Deaths Reported to NHTS
    })
    print(compare_df)

    compare_df.plot(kind='bar', figsize=(6,4))
    plt.title("Verified Autopilot Deaths vs All Reported Deaths (NHTSA)")
    plt.ylabel("Count")
    plt.show()
```

```
Model
 -       181
 S        45
  3       39
 X        17
 Y        10
  1        1
  2        1
  50       1
2019       1
Name: count, dtype: int64
```



```
Verified Tesla Autopilot Deaths
 -        273
  1        13
  2         3
-           1
  3         1
  16        1
  19        1
  118       1
2022        1
  75        1
2021        1
Name: count, dtype: int64
```



Step 3 – Model & Autopilot Analysis

In this step, we analyze accidents with respect to Tesla models and Autopilot usage.

3.1 Event Distribution across Tesla Models

Why? → To see which Tesla models (S, 3, X, Y) are most commonly involved in accidents.
This helps check if accident frequency matches model popularity.

3.2 Verified Tesla Autopilot Deaths Distribution

Why? → To understand how many verified deaths were officially linked to Autopilot usage.
This shows the direct risk associated with Autopilot.

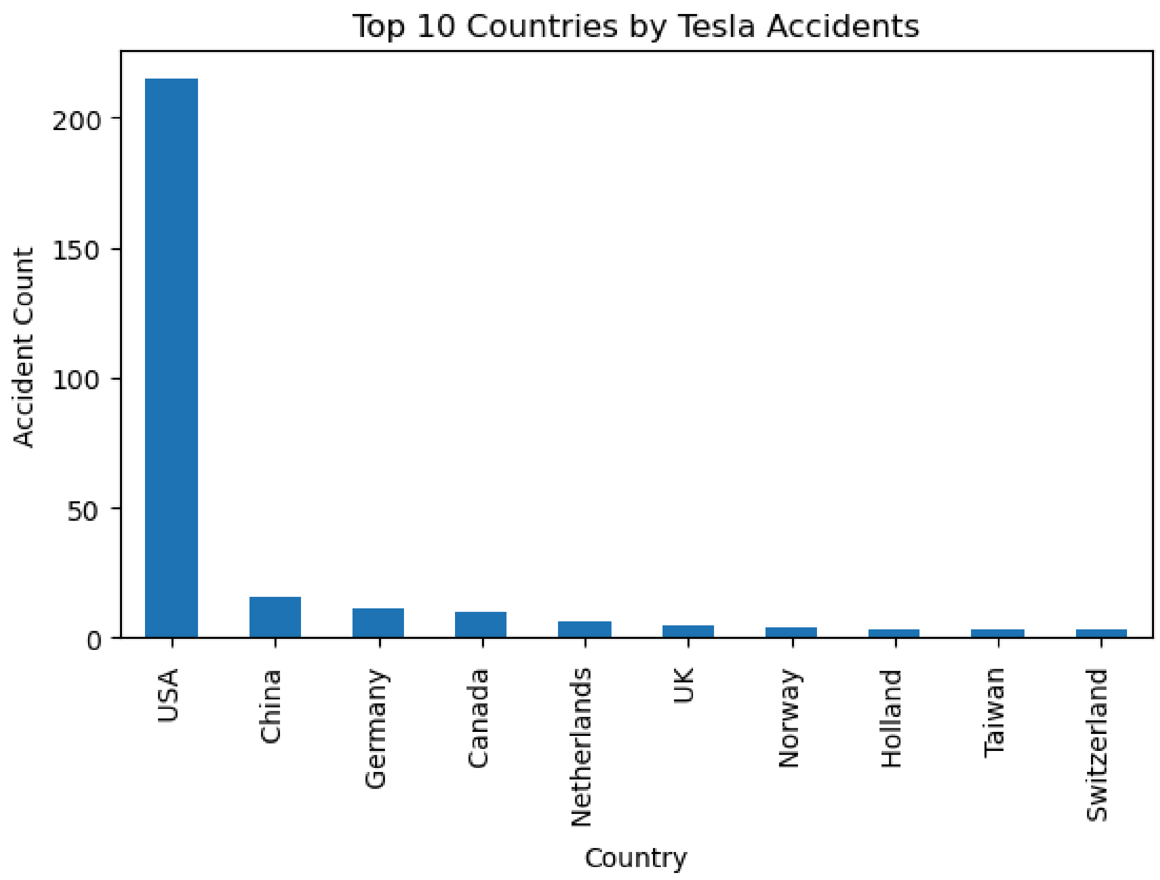
3.3 Verified Autopilot Deaths vs All Reported Deaths (NHTSA)

Why? → To compare the officially verified Autopilot deaths with all deaths reported to NHTSA.

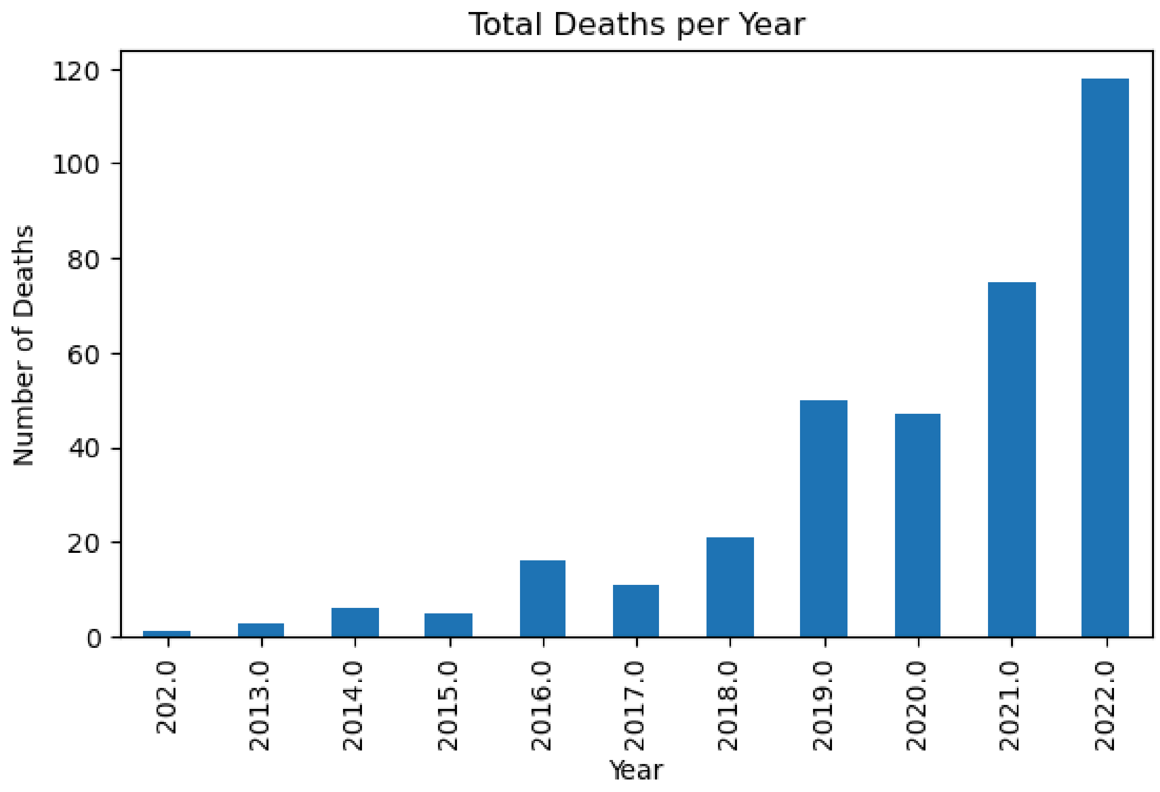
This highlights the difference between confirmed Autopilot fatalities and overall accident reports.

In [21]:
```
# Step 4 – Visualisation & Insights
# Why? -> To summarize the findings from EDA and Model/Autopilot analysis
#         with simple, clear visuals.

# 4.1 Country-wise Top Accidents
print("Top 10 Countries by Tesla Accidents:")
country_counts = data['Country'].value_counts().head(10)
print(country_counts)

country_counts.plot(kind='bar', figsize=(7,4))
plt.title("Top 10 Countries by Tesla Accidents")
plt.xlabel("Country"); plt.ylabel("Accident Count")
plt.show()

# 4.2 Deaths Trend Over Time (Year-wise)
print("\nTotal Deaths per Year:")
if 'Year' in data.columns and 'Deaths' in data.columns:
    deaths_per_year = data.groupby('Year')['Deaths'].sum()
    print(deaths_per_year)

    deaths_per_year.plot(kind='bar', figsize=(7,4))
    plt.title("Total Deaths per Year")
    plt.xlabel("Year"); plt.ylabel("Number of Deaths")
    plt.show()
```

```
Top 10 Countries by Tesla Accidents:
Country
USA               215
China              16
Germany            11
Canada             10
Netherlands         6
UK                  5
Norway              4
Holland             3
Taiwan              3
Switzerland         3
Name: count, dtype: int64
```



```
Total Deaths per Year:
Year
202.0       1.0
2013.0      3.0
2014.0      6.0
2015.0      5.0
2016.0     16.0
2017.0     11.0
2018.0     21.0
2019.0     50.0
2020.0     47.0
2021.0     75.0
2022.0    118.0
Name: Deaths, dtype: float64
```



Step 4 – Insights & Conclusion

Based on the analysis, here are the key insights:

- **Country-wise**: The USA reports the highest number of Tesla accidents, followed by a few other countries (e.g., China, Germany, Canada).
- **Yearly Trend**: Accident counts and deaths increased sharply after 2018, showing Tesla's rising adoption and exposure risk.
- **Model-wise**: Model S and Model 3 appear most frequently in accident records.
- **Victim Analysis**: While most crashes involve only 1 fatality, there are also cases with multiple victims.
- **Collision Patterns**: Majority of Tesla accidents involve other vehicles, and a notable number involve cyclists/pedestrians.
- **Autopilot**: Verified Autopilot deaths are significantly fewer than all reported deaths (NHTSA), suggesting not all fatalities are officially attributed to Autopilot.

```
In [ ]:
```

Final Summary – Key Insights from Tesla Accident Analysis

Based on the data cleaning, exploratory analysis, and model/autopilot study, here are the main takeaways:

- **Accident Growth Over Time**: Tesla accidents have steadily increased since 2013, with a sharp rise after 2018, reflecting growing Tesla adoption.
- **Country Distribution**: The USA reports the overwhelming majority of accidents, followed by China, Germany, and Canada.
- **Severity of Accidents**: While many accidents involve a single death, there are notable cases with multiple fatalities, showing varying crash severity.
- **Driver vs Passenger Risk**: Tesla drivers themselves account for a significant portion of deaths, indicating high risk for the person in control.
- **Model-wise Trends**: Model S and Model 3 are most frequently reported in accidents, likely due to their higher sales numbers compared to other Tesla models.
- **Collision Types**: Most crashes involve other vehicles, with additional cases involving cyclists/pedestrians, highlighting external road safety risks.
- **Autopilot Analysis**: Verified Autopilot-linked deaths are fewer than total deaths reported to NHTSA, suggesting that not all fatalities are attributed directly to Autopilot.

✅ This concludes **Part-2 (Exploratory Data Analysis + Model & Autopilot Analysis)**.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```