Runaway Resource Detector

This script analyses the `reporting_historic_endpoints` table from the Open Digital Planning Datasette to identify endpoints that:

- Are generating **new resources daily** over a 7-day period
- Have produced **more than 20 resources** in the last 30 days
- Are showing signs of **stale resource end dates**
- Contain **single-day resources** (where start date equals end date)

Logic Overview

1. Loads the full `reporting_historic_endpoints` table via streaming.
2. Filters out endpoints that have already ended.
3. Groups by `endpoint` and derives:
   - Total resource count
   - Metadata columns
   - First and last resource start dates
   - Count of single-day resources
4. Flags endpoints based on:
   - 7-day streak of daily resources
   - 30-day resource activity (>20)
   - Stale `resource_end_date`

Output

Saves a CSV named `runaway_resources.csv` to the provided `--output-dir`, containing flagged and summary metadata.

Run via CLI

```bash python runaway_resources.py --output-dir ./outputs

In [ ]:
```python
import pandas as pd
from datetime import datetime, timedelta
import argparse
import os

def main(output_dir):
    # Load Data
    base_url = "https://datasette.planning.data.gov.uk/digital-land"
    table = "reporting_historic_endpoints"
    full_url = f"{base_url}/{table}.csv?_stream=on"

    df = pd.read_csv(full_url)

    # Filter and convert dates
    df = df[df["endpoint_end_date"].isna()].copy()
    df["resource_start_date"] = pd.to_datetime(df["resource_start_date"])
    df["resource_end_date"] = pd.to_datetime(df["resource_end_date"])
```
```

```python
# Build summary dataframe
summary_df = (
    df.groupby("endpoint")
    .size()
    .reset_index(name="resource_count")
    .query("resource_count > 1")
    .sort_values("resource_count", ascending=False)
    .reset_index(drop=True)
)

# Add metadata columns
meta_cols = ["organisation_name", "dataset", "collection", "pipeline", "endpoir
metadata = df.groupby("endpoint")[meta_cols].first().reset_index()
summary_df = summary_df.merge(metadata, on="endpoint", how="left")

# First and last start dates
first_dates = df.groupby("endpoint")["resource_start_date"].min().dt.date
last_dates = df.groupby("endpoint")["resource_start_date"].max().dt.date

# Count single-day resources (same start and end date)
single_day_counts = df[df["resource_start_date"] == df["resource_end_date"]]
single_day_summary = single_day_counts.groupby("endpoint").size()

# Insert derived columns
summary_df.insert(1, "first_resource_start_date", summary_df["endpoint"].map(fi
summary_df.insert(2, "last_resource_start_date", summary_df["endpoint"].map(las
summary_df.insert(9, "single_day_resources", summary_df["endpoint"].map(single_

# Flagging logic
today = datetime.today().date()
recent_7_days = today - timedelta(days=7)
recent_30_days = today - timedelta(days=30)

# Daily streak for last 7 days
seven_day_df = df[df["resource_start_date"].dt.date >= recent_7_days]
start_dates_7 = seven_day_df.groupby("endpoint")["resource_start_date"].apply(l
expected_last_7 = set(today - timedelta(days=i) for i in range(1, 8))

summary_df["daily_for_7_days"] = summary_df["endpoint"].apply(
    lambda ep: "yes" if expected_last_7.issubset(start_dates_7.get(ep, set()))
)

# More than 20 new resources in last 30 days
thirty_day_df = df[df["resource_start_date"].dt.date >= recent_30_days]
resource_count_30 = thirty_day_df.groupby("endpoint").size()

summary_df[">20_instances_in_30_day_period"] = summary_df["endpoint"].apply(
    lambda ep: "yes" if resource_count_30.get(ep, 0) > 20 else "no"
)

# Flag endpoints with stale resource end dates
last_end_dates = df.groupby("endpoint")["resource_end_date"].max().dt.date
stale_cutoff = today - timedelta(days=30)

summary_df["stale_resource"] = summary_df["endpoint"].apply(
    lambda ep: "yes" if pd.notnull(last_end_dates.get(ep)) and last_end_dates.g
)

# Output
csv_name = "runaway_resources.csv"
save_path = os.path.join(output_dir, csv_name)
summary_df.to_csv(save_path, index=False)  # Save to CSV without index
print(f"Saved: {save_path}")
```

```
def parse_args():
    """
    Parses command-line arguments for specifying the output directory.

    Returns:
        argparse.Namespace: Parsed arguments containing the output directory path.
    """
    parser = argparse.ArgumentParser(description="runaway_resources")
    parser.add_argument(
        "--output-dir",
        type=str,
        required=True,
        help="Directory to save exported CSVs"
    )
    return parser.parse_args()

if __name__ == "__main__":
    # Parse command-line arguments
    args = parse_args()
    main(args.output_dir)
```