ODP Endpoint Status Summary Generator

This script produces a CSV file (`odp-status.csv`) comparing **expected dataset provisions** with the **actual presence and status of endpoints** across local planning authorities, as published through the Open Digital Planning (ODP) platform.

Purpose:

To help programme and data managers quickly identify:

- Which organisations have active ODP endpoints for each expected dataset.
- Which pipelines are missing or unresponsive.
- Licensing, exception, and logging details associated with each endpoint.

How It Works:

1. **Fetches expected provisions** from the `provision` table (`digital-land` database).
2. **Retrieves current endpoint metadata** from `reporting_latest_endpoints` (`performance` database).
3. **Normalises organisation codes** (removes `-eng` suffix).
4. **Matches each expected pipeline** against live endpoints.
5. **Outputs a row per match**, or a 'No endpoint added' row if none is found.

In [ ]:

```
"""
Script to generate an ODP (Open Digital Planning) status CSV, summarising
endpoint presence and conformance against expected dataset provisions.

The script:
- Retrieves all organisations expected to provide datasets ("provisions")
- Fetches endpoint status from the reporting_latest_endpoints table
- Matches expected datasets (via pipelines) against actual endpoints
- Outputs a detailed CSV of provision vs. actual endpoint status
"""

import os
import pandas as pd
import requests
from requests.adapters import HTTPAdapter
from urllib3.util import Retry
import argparse

# Dataset to Pipeline Map
ALL_PIPELINES = {
    "article-4-direction": ["article-4-direction", "article-4-direction-area"],
    "conservation-area": ["conservation-area", "conservation-area-document"],
    "listed-building": ["listed-building-outline"],
    "tree-preservation-order": [
        "tree-preservation-order",
        "tree-preservation-zone",
        "tree",
    ],
}
```

```python
# Datasette Query Helpers
def get_datasette_http():
    """
    Returns a requests session with retry logic to handle larger Datasette queries.

    Returns:
        requests.Session: Session with retry strategy enabled.
    """
    retry_strategy = Retry(total=3, status_forcelist=[400], backoff_factor=0.2)
    adapter = HTTPAdapter(max_retries=retry_strategy)
    http = requests.Session()
    http.mount("https://", adapter)
    return http


def get_datasette_query(db: str, sql: str, url="https://datasette.planning.data.gov
    """
    Executes SQL against a Datasette database and returns the result as a DataFrame

    Args:
        db (str): The name of the Datasette database (e.g., 'digital-land').
        sql (str): SQL query string to run.
        url (str): Base URL of the Datasette instance.

    Returns:
        pd.DataFrame: The result set, or empty DataFrame on error.
    """
    full_url = f"{url}/{db}.json"
    params = {"sql": sql, "_shape": "array", "_size": "max"}

    try:
        http = get_datasette_http()
        response = http.get(full_url, params=params)
        response.raise_for_status()
        return pd.DataFrame.from_dict(response.json())
    except Exception as e:
        print(f"Datasette query failed: {e}")
        return pd.DataFrame()

# Data Retrieval Functions
def get_provisions():
    """
    Retrieves provision records showing which organisations are expected to
    provide datasets for each cohort.

    Returns:
        pd.DataFrame: Provision table including cohort and organisation names.
    """
    sql = """
        SELECT
            p.cohort,
            p.organisation,
            c.start_date as cohort_start_date,
            org.name as name
        FROM provision p
        INNER JOIN cohort c ON c.cohort = p.cohort
        INNER JOIN organisation org ON org.organisation = p.organisation
        WHERE p.provision_reason = "expected"
          AND p.project = "open-digital-planning"
        GROUP BY p.organisation, p.cohort
    """
    return get_datasette_query("digital-land", sql)
```

```
def get_endpoints():
    """
    Retrieves latest reporting data for all active endpoints.

    Returns:
        pd.DataFrame: Table of endpoint metadata and status.
    """
    sql = """
        SELECT
            rle.organisation,
            rle.collection,
            rle.pipeline,
            rle.endpoint,
            rle.endpoint_url,
            rle.licence,
            rle.latest_status as status,
            rle.days_since_200,
            rle.latest_exception as exception,
            rle.resource,
            rle.latest_log_entry_date,
            rle.endpoint_entry_date,
            rle.endpoint_end_date,
            rle.resource_start_date,
            rle.resource_end_date
        FROM reporting_latest_endpoints rle
    """
    df = get_datasette_query("performance", sql)

    # Normalise organisation codes (remove -eng suffix)
    df["organisation"] = df["organisation"].str.replace("-eng", "", regex=False)
    return df

# CSV Export Logic
def generate_odp_summary_csv(output_dir: str) -> str:
    """
    Generates a CSV file showing provision status by dataset, pipeline, and endpoir

    Args:
        output_dir (str): Directory to save the CSV output.

    Returns:
        str: Path to the saved CSV file.
    """
    provisions = get_provisions()
    endpoints = get_endpoints()
    output_rows = []

    for _, row in provisions.iterrows():
        organisation = row["organisation"]
        cohort = row["cohort"]
        name = row["name"]
        cohort_start_date = row["cohort_start_date"]

        for collection, pipelines in ALL_PIPELINES.items():
            for pipeline in pipelines:
                match = endpoints[
                    (endpoints["organisation"] == organisation) &
                    (endpoints["pipeline"] == pipeline)
                ]

                if not match.empty:
                    # Endpoint(s) exist — add one row per match
                    for _, ep in match.iterrows():
```

```
                        output_rows.append({
                            "organisation": organisation,
                            "cohort": cohort,
                            "name": name,
                            "collection": collection,
                            "pipeline": pipeline,
                            "endpoint": ep["endpoint"],
                            "endpoint_url": ep["endpoint_url"],
                            "licence": ep["licence"],
                            "status": ep["status"],
                            "days_since_200": ep["days_since_200"],
                            "exception": ep["exception"],
                            "resource": ep["resource"],
                            "latest_log_entry_date": ep["latest_log_entry_date"],
                            "endpoint_entry_date": ep["endpoint_entry_date"],
                            "endpoint_end_date": ep["endpoint_end_date"],
                            "resource_start_date": ep["resource_start_date"],
                            "resource_end_date": ep["resource_end_date"],
                            "cohort_start_date": cohort_start_date,
                        })
                else:
                    # No endpoint — mark as missing
                    output_rows.append({
                        "organisation": organisation,
                        "cohort": cohort,
                        "name": name,
                        "collection": collection,
                        "pipeline": pipeline,
                        "endpoint": "No endpoint added",
                        "endpoint_url": "",
                        "licence": "",
                        "status": "",
                        "days_since_200": "",
                        "exception": "",
                        "resource": "",
                        "latest_log_entry_date": "",
                        "endpoint_entry_date": "",
                        "endpoint_end_date": "",
                        "resource_start_date": "",
                        "resource_end_date": "",
                        "cohort_start_date": cohort_start_date,
                    })

    # Convert output to DataFrame and save as CSV
    df_final = pd.DataFrame(output_rows)
    os.makedirs(output_dir, exist_ok=True)
    output_path = os.path.join(output_dir, "odp-status.csv")
    df_final.to_csv(output_path, index=False)
    print(f"CSV generated at {output_path} with {len(df_final)} rows")
    return output_path

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

    Returns:
        argparse.Namespace: Parsed args containing the output path.
    """
    parser = argparse.ArgumentParser(description="Datasette batch exporter")
    parser.add_argument(
        "--output-dir",
        type=str,
        required=True,
```

```
        help="Directory to save exported CSVs"
    )
    return parser.parse_args()

# Script Entry Point
if __name__ == "__main__":
    # Parse CLI arguments
    args = parse_args()
    output_directory = args.output_dir

    # Generate and save ODP endpoint summary
    generate_odp_summary_csv(output_directory)
```