APPLIED ANALYTICS PRACTICUM BUAN-690-DB

GROUP 4

SHU VOCA — Data Cleaning & Exploratory Data Analysis(EDA) Report

This notebook shows our work of cleaning and analyzing the SHU VOCA dataset.
The aim was to transform messy multi-sheet Excel data into a clean data, consistent format, perform exploratory data analysis and finally provide actionable recommendations.

Throughout the process we faced several **challenges** like:

- Column names were inconsistent across sheets like (different capitalization, spacing, and symbols).
- Some fields that looked like dates were actually IDs or text which is leading to misclassification.
- Several columns had high missingness that making them unreliable without remediation.
- Duplicate records appeared in the raw dataset.

To address these issues we applied **innovative solutions** like:

- Functions to automatically standardize column names into a uniform `snake_case` style which was easy to understand the data.
- Automated attempts to parse potential dates, with safeguards to avoid false positives.
- Deduplication steps to remove repeated entries.
- Visualization and EDA to highlight gaps and anomalies.

This combination of code, visuals, and narrative helps us bring clarity to the data confusion.

1. Loading data, libraries and Consolidate Excel Sheets

We started by loading all libraries which is required to work with this dataset then loaded the dataset by DATA_PATH method. Then the sheets from the Excel workbook into a unified DataFrame. This step is very crucial because the raw data was spread across different sheets. By adding a source_sheet column, we ensured that we could track the origin of each row while still working with the data as a single dataset as we find this method is right to use for such situation.

In [12]:
```python
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from pandas.api.types import is_numeric_dtype, is_datetime64_any_dtype

DATA_PATH = 'SHU VOCA Data Set 1-1-24 -- 10-25-24.xlsx'  # Adjust path if needed

sheets = pd.read_excel(DATA_PATH, sheet_name=None)
frames = []
for name, df in sheets.items():
    df = df.copy()
    df['source_sheet'] = name
    frames.append(df)
raw = pd.concat(frames, ignore_index=True)
raw.shape
```

Out [12]:
```
(10884, 36)
```

2. Cleaning Utilities

This raw dataset had messy structure: column names varied in style, special characters like `%` and `#` appeared, and potential date fields were stored as inconsistent text.
To be fixed first so we had to built two helper functions like:

1. `standardize_colnames`: this helps in Converting all names to lowercase, replaces spaces with underscores, and normalizes special characters. This ensures consistent naming across sheets.
2. `try_parse_dates`: Tries to parse text columns into dates. We applied this cautiously because some columns (like IDs) looked numeric or date-like but weren't. By coercing invalid formats into `NaT`, we avoided breaking the dataset.

These utilities gave us a 1st step for reliable cleaning in the next step.

In [13]:
```python
def standardize_colnames(cols):
    out = []
    for c in cols:
        if isinstance(c, str):
            cc = c.strip().lower().replace(' ', '_').replace('%','percent').replace('#','num')
            out.append(cc)
        else:
            out.append(f'col_{c}')
    return out

def try_parse_dates(df):
    parsed_cols = []
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = pd.to_datetime(df[col], errors='coerce')
            if is_datetime64_any_dtype(df[col]):
                parsed_cols.append(col)
    return df, parsed_cols
```

3. Apply Cleaning

After building our tools then we cleaned the dataset by:

- **Standardizing columns** into snake_case for consistency.
- **Parsing dates** where applicable, while leaving invalid ones untouched.
- **Removing duplicates** to ensure only unique records remained.

One challenge we faced was that some columns, like `eto_id`, were mistakenly parsed as dates.
We solved this by verifying each parsed column contained valid non-null date values before using them in time-based analysis.
This highlights why data cleaning isn't just mechanical—it requires **critical judgment** at every step.

In [14]:
```python
clean = raw.copy()
clean.columns = standardize_colnames(clean.columns)
clean, parsed_date_cols = try_parse_dates(clean)
dup_mask = clean.duplicated()
clean_no_dups = clean.loc[~dup_mask].copy()
clean_no_dups.shape, parsed_date_cols
```

```
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
```

```
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
C:\Users\HUSSA\AppData\Local\Temp\ipykernel_10168\2268289830.py:15: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected,
please specify a format.
  df[col] = pd.to_datetime(df[col], errors='coerce')
```

```
Out [14]: ((4660, 36),
           ['eto_id',
            'date_taken',
            'services_provided',
            'outcome_of_contact',
            'appointment_outcome',
            'did_you_interact_with_the_person_you_contacted?',
            'source_sheet',
            'city',
            'zipcode',
            'race',
            'ethnicity',
            'gender',
            'sexual_orientation',
            'who_is_making_the_referral?',
            'how_did_you_hear_about_us?',
            'victim_type',
            'type_of_victimization',
            'organization_name',
            'organization_category',
            'organization_location',
            'num_of_people',
            'time_spent'])
```

Data Transformation Pipeline

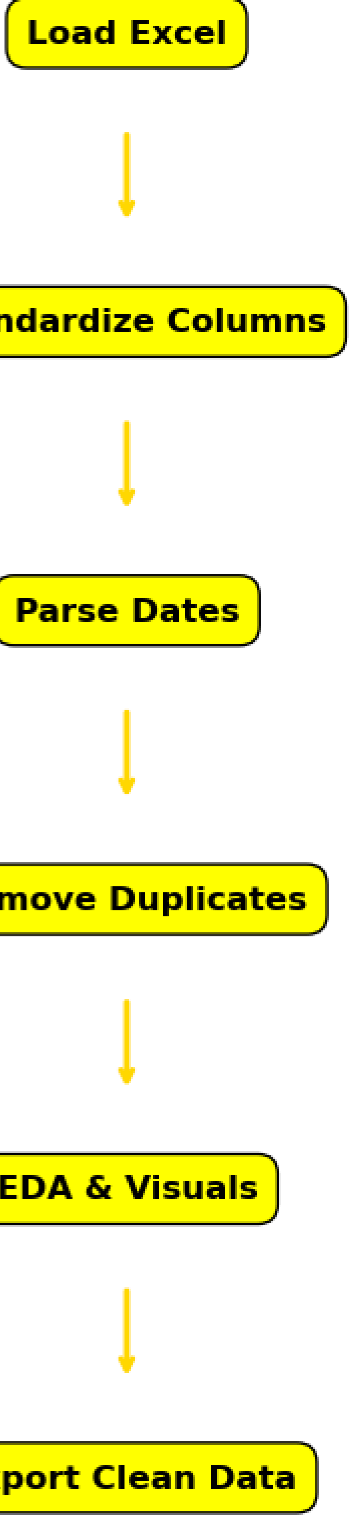
The diagram below summarizes our end-to-end workflow, from raw Excel data to cleaned and export-ready dataset.

In [31]:
```
fig, ax = plt.subplots(figsize=(6, 10))  # tall figure for vertical layout
steps = ['Load Excel',
         'Standardize Columns',
         'Parse Dates',
         'Remove Duplicates',
         'EDA & Visuals',
         'Export Clean Data']

for i, step in enumerate(steps):
    ax.text(0.5, 1 - i*0.15, step, ha='center', va='center',
            fontsize=12, fontweight='bold',
            bbox=dict(boxstyle='round,pad=0.6', fc='yellow', ec='black'))

    # draw arrows pointing down between boxes
    if i < len(steps) - 1:
        ax.annotate('', xy=(0.5, 1 - (i+1)*0.15 + 0.05),
                    xytext=(0.5, 1 - i*0.15 - 0.05),
                    arrowprops=dict(arrowstyle='->', lw=2, color='gold'))

ax.axis('off')
plt.tight_layout()
plt.show()
```



4. Exploratory Data Analysis (EDA)

With a clean dataset prepared, we conducted exploratory data analysis to better understand data quality and patterns. This included examining missing values, distributions of numeric features, and temporal coverage.

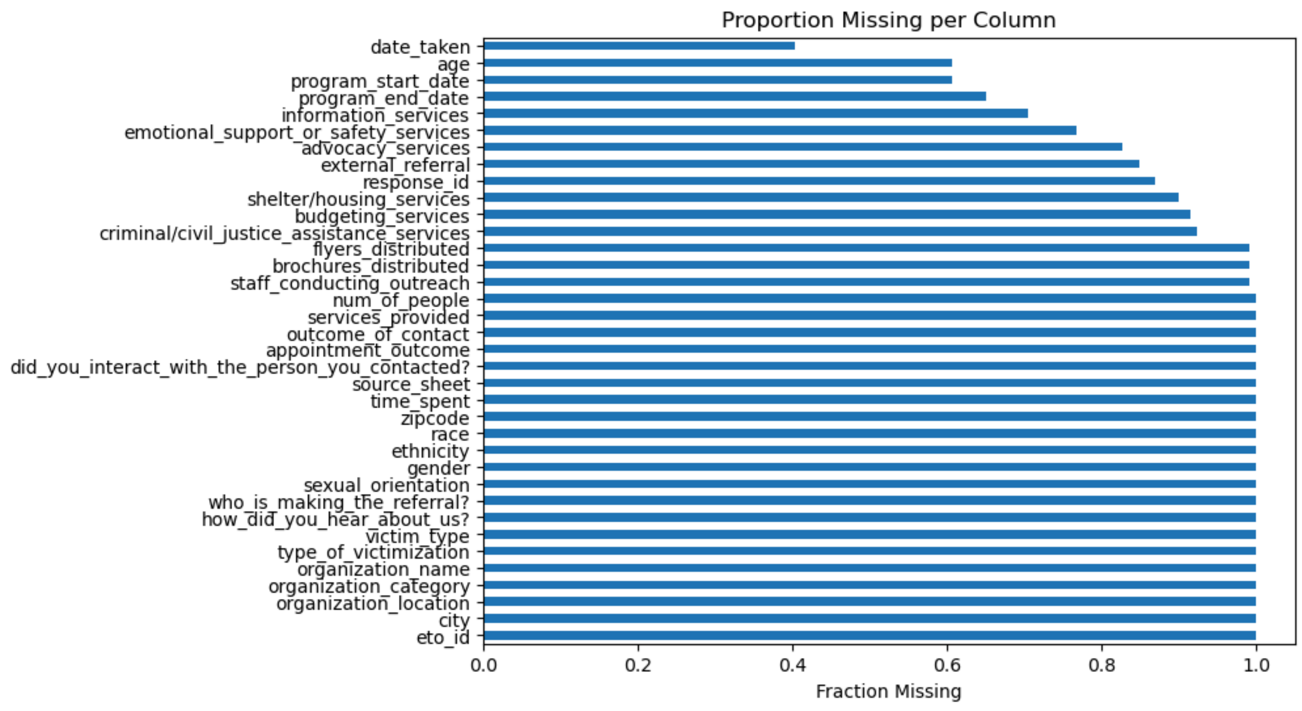
Missingness Analysis

We measured the proportion of missing values in each column. Columns with more than 30% missing values may not be reliable for downstream analysis. The bar chart below highlights which fields suffer the most from missing data.

In [16]:
```
missing_before = clean_no_dups.isna().mean().sort_values(ascending=False)
missing_before.head(10)
```

Out [16]:
```
eto_id                          1.0
city                            1.0
organization_location           1.0
organization_category           1.0
organization_name               1.0
type_of_victimization           1.0
victim_type                     1.0
how_did_you_hear_about_us?      1.0
who_is_making_the_referral?     1.0
sexual_orientation              1.0
dtype: float64
```

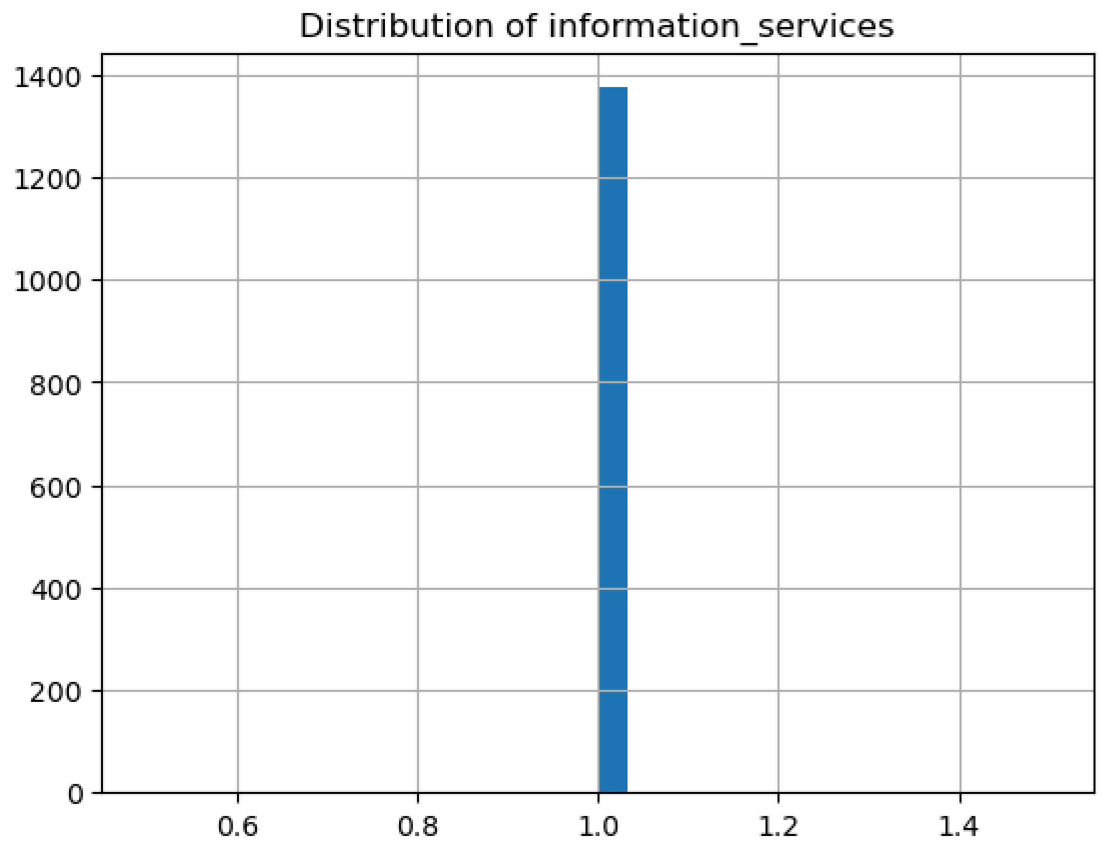
In [17]:
```
missing_before.plot(kind='barh', figsize=(8,6))
plt.title('Proportion Missing per Column')
plt.xlabel('Fraction Missing')
plt.show()
```

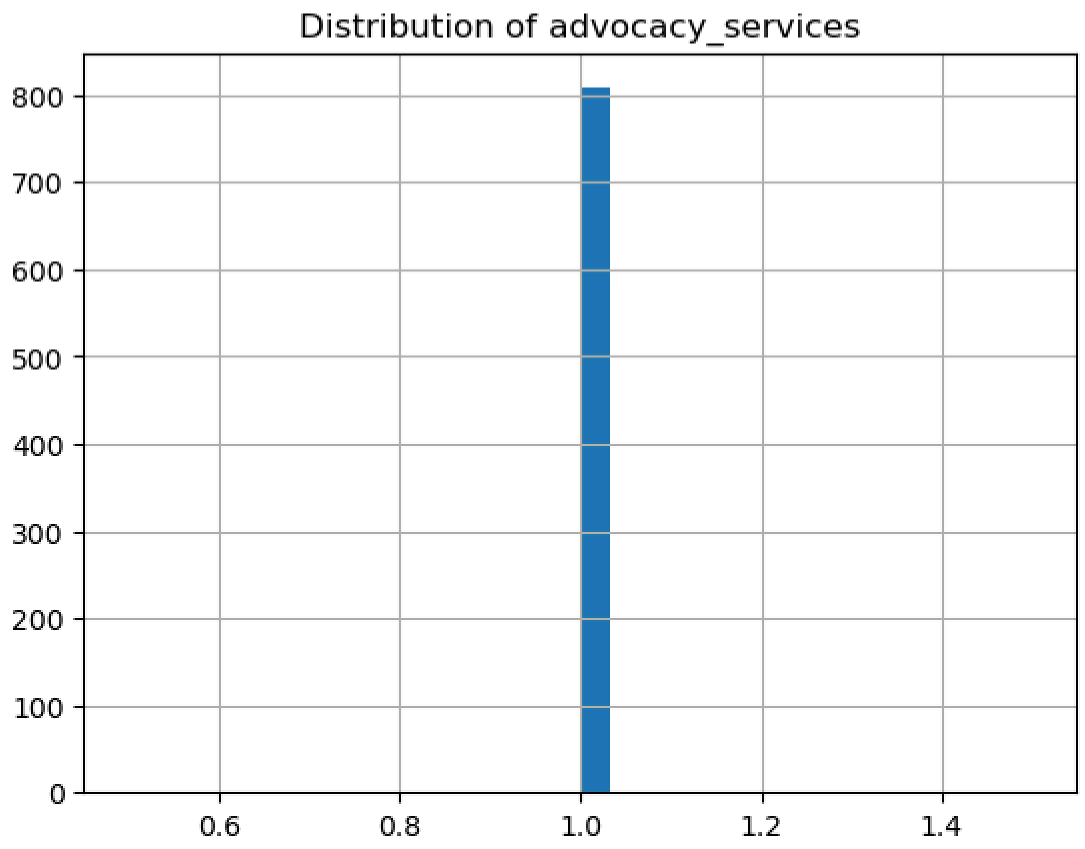


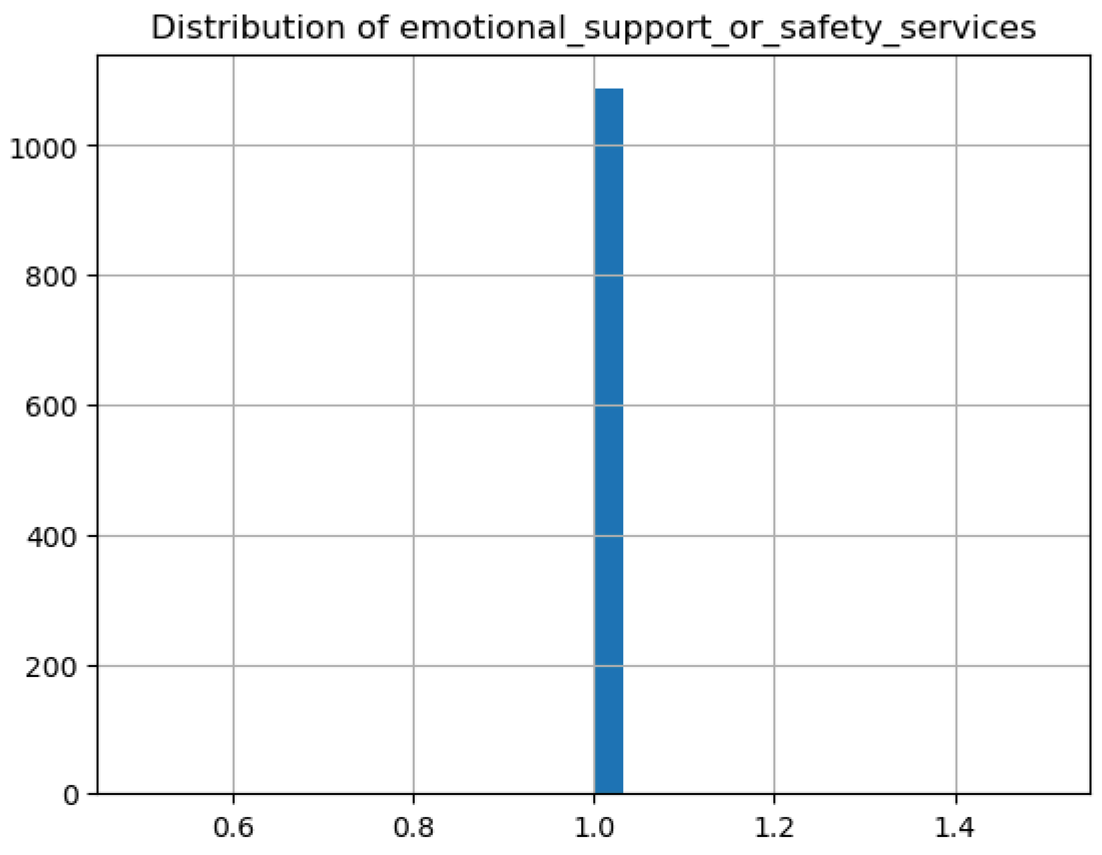
Numeric Distributions

We plotted histograms for several numeric columns to understand the spread of values and spot any outliers. This helps us identify potential data entry errors or skewed distributions.

In [18]:
```
num_cols = [c for c in clean_no_dups.columns if is_numeric_dtype(clean_no_dups[c])]
for c in num_cols[:3]:
    clean_no_dups[c].dropna().hist(bins=30)
    plt.title(f'Distribution of {c}')
    plt.show()
```


Distribution of information_services
1400
1200
1000
800
600
400
200
0
0.6
0.8
1.0
1.2
1.4

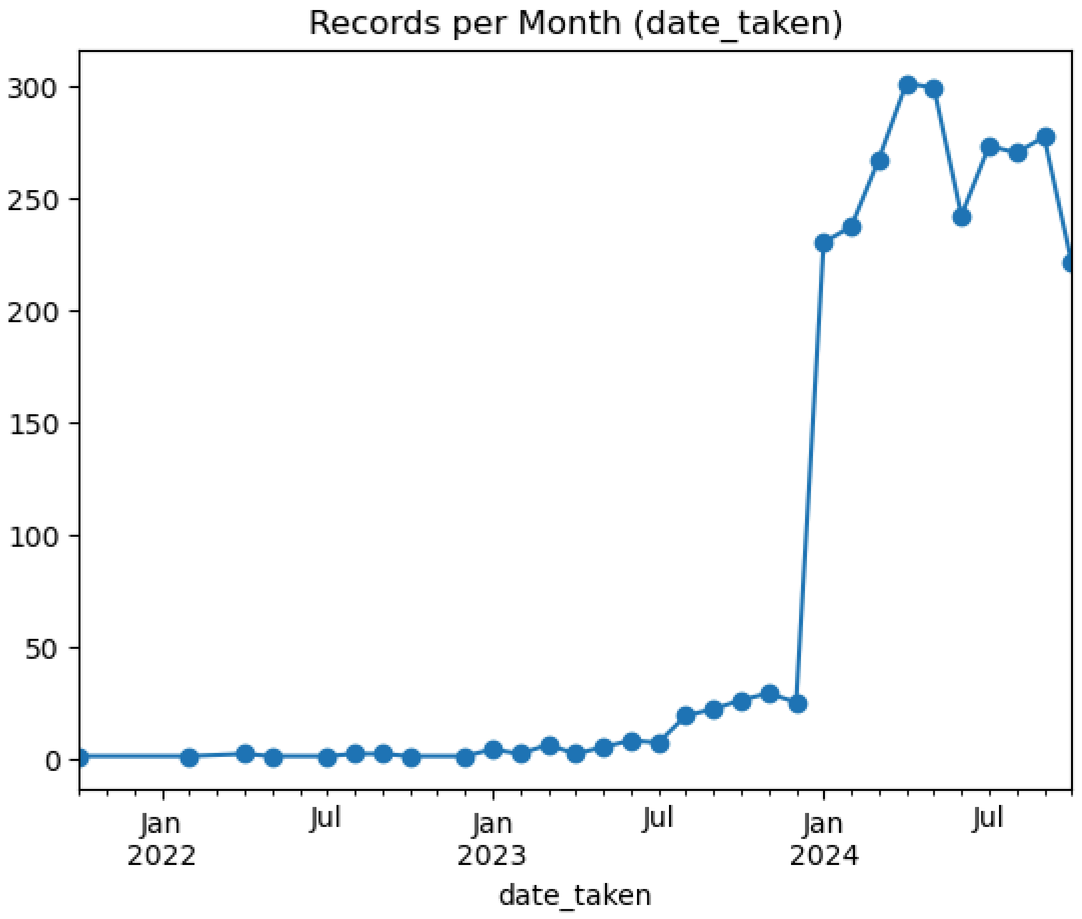

Distribution of advocacy_services
800
700
600
500
400
300
200
100
0
0.6
0.8
1.0
1.2
1.4

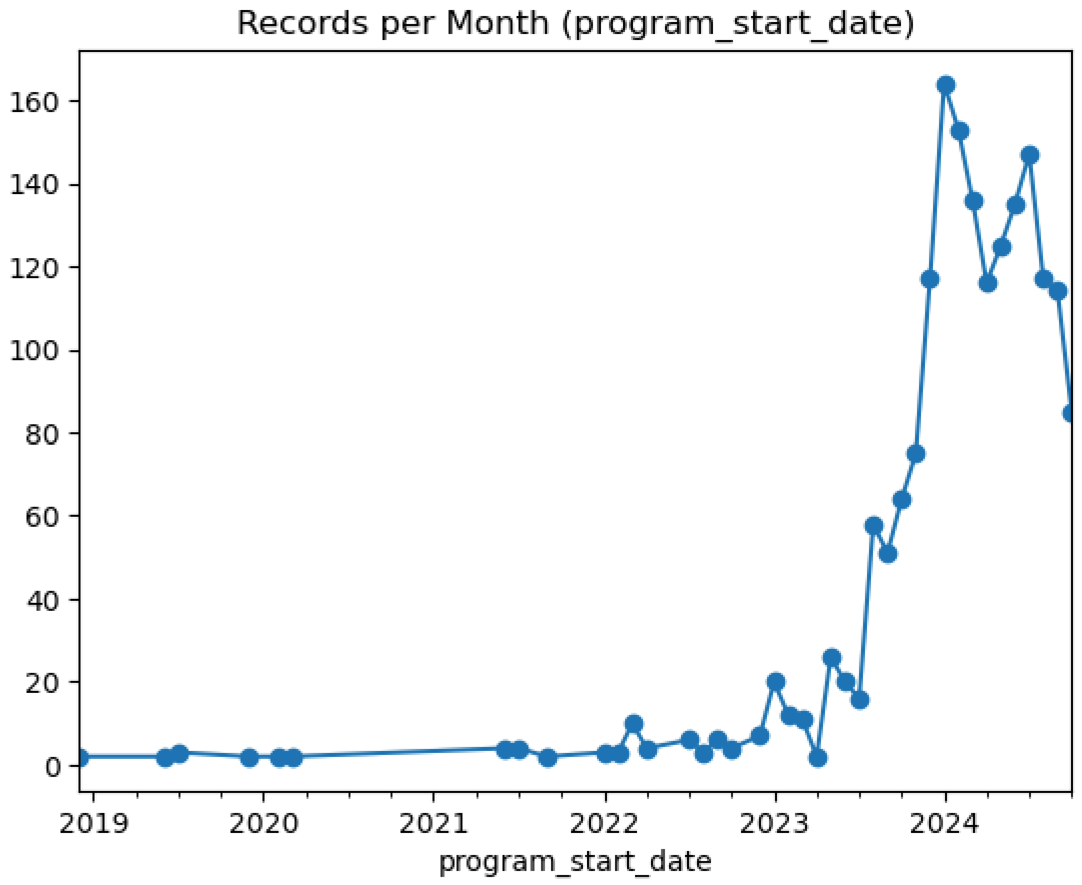


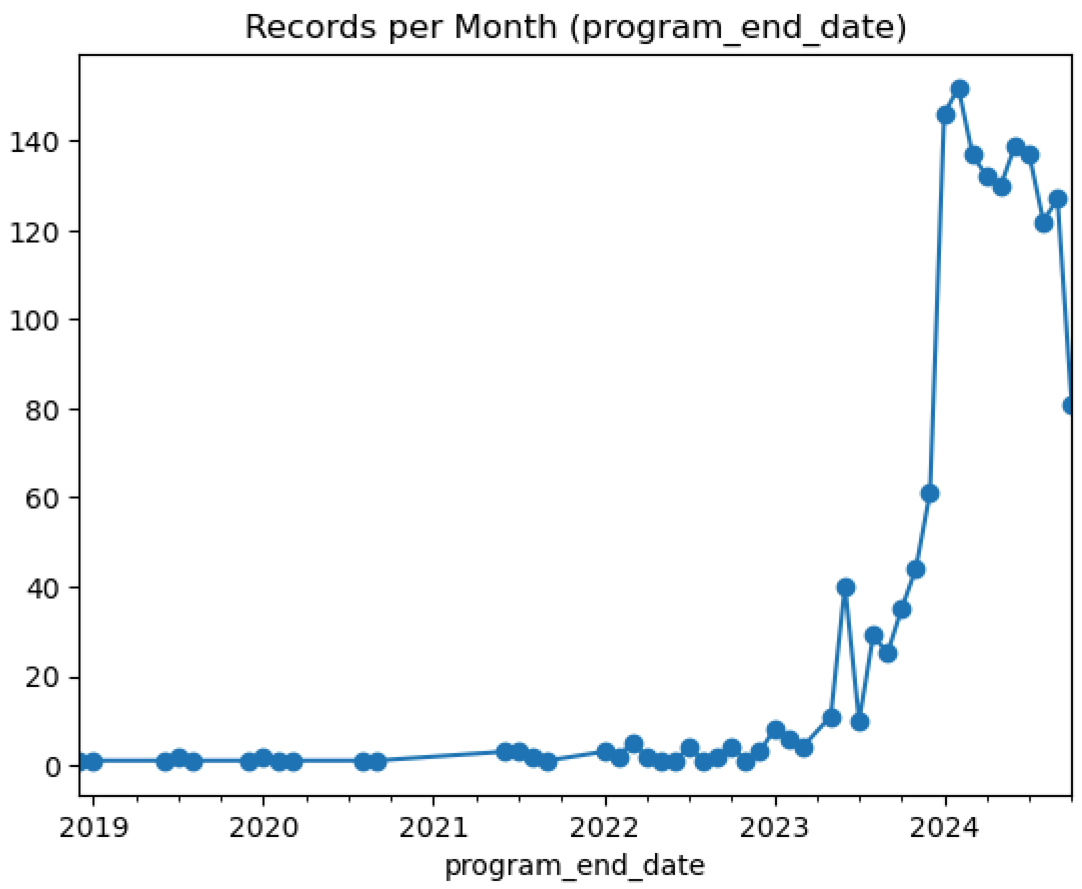
Temporal Trends

For valid datetime columns, we examined how many records appear per month. This shows whether data collection has been consistent over time or if there are gaps.

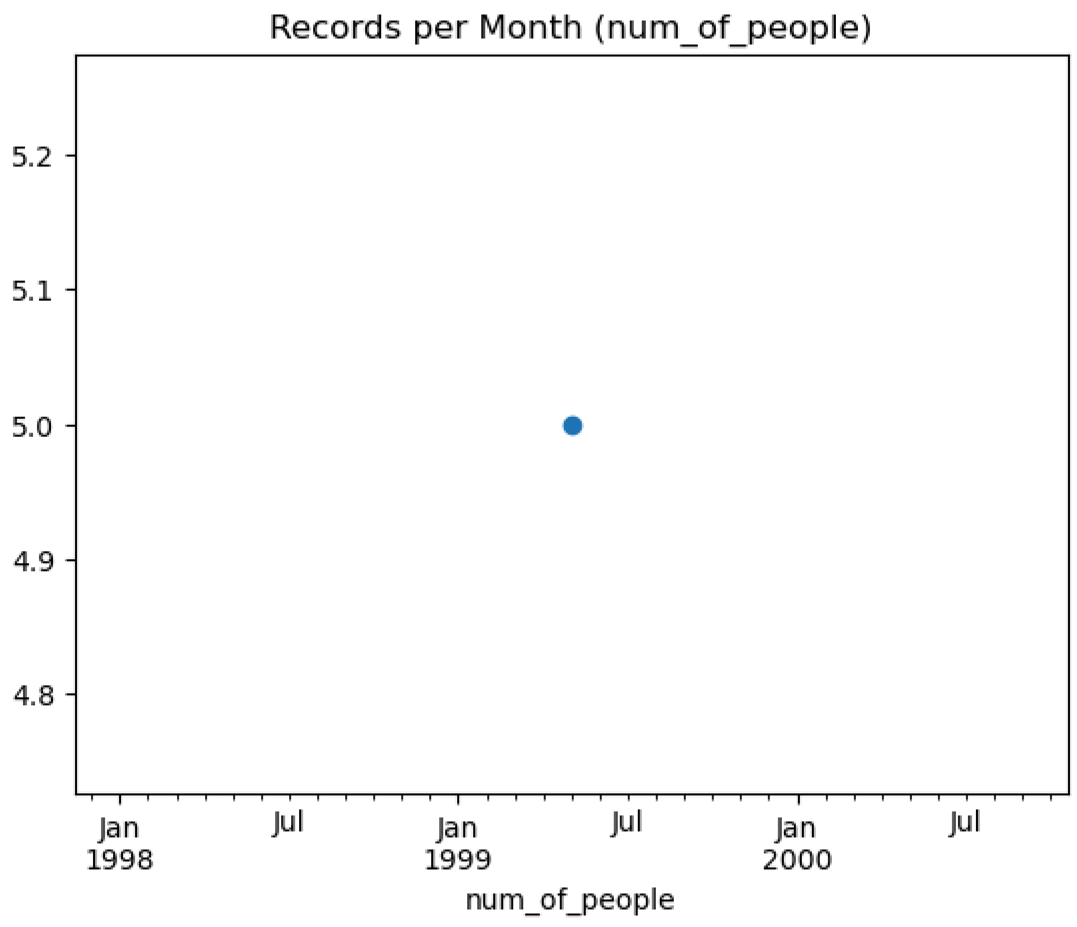
In [19]:
```
valid_date_cols = [c for c in clean_no_dups.columns if is_datetime64_any_dtype(clean_no_dups[c
if valid_date_cols:
    for col in valid_date_cols:
        ts = (clean_no_dups[col].dropna().dt.to_period('M').value_counts().sort_index())
        if not ts.empty:
            ts.plot(kind='line', marker='o', title=f'Records per Month ({col})')
            plt.show()
else:
    print('No usable datetime columns detected.')
```


Records per Month (date_taken)
300
250
200
150
100
50
0
Jan
2022
Jul
Jan
2023
Jul
Jan
2024
Jul
date_taken


Records per Month (program_start_date)
160
140
120
100
80
60
40
20
0
2019
2020
2021
2022
2023
2024
program_start_date



```
C:\Users\HUSSA\AppData\Local\anaconda3\Lib\site-packages\pandas\plotting\_matplotlib\core.py:1561: UserWarning:
Attempting to set identical low and high xlims makes transformation singular; automatically expanding.
  ax.set_xlim(left, right)
```



5. Export Cleaned Data

Finally, we exported the cleaned dataset to CSV format. This ensures the processed data can be reused for further modeling, visualization, or reporting.

In [33]:
```
OUT_DIR = 'outputs'
os.makedirs(OUT_DIR, exist_ok=True)
clean_no_dups.to_csv(os.path.join(OUT_DIR, 'shu_voca_cleaned.csv'), index=False)
print('Exported to outputs/shu_voca_cleaned.csv')
```

```
Exported to outputs/shu_voca_cleaned.csv
```

6.Categorical Breakdown

To explore categorical variables, we plotted the most frequent categories for one example column.
This helps us see whether categories are well-defined or if there is free-text sprawl (e.g., multiple spellings of the same category).

In [21]:
```
cat_cols = [c for c in clean_no_dups.columns if clean_no_dups[c].dtype == "object"]

if cat_cols:
    col = cat_cols[0]  # take first categorical as an example
    top_counts = clean_no_dups[col].value_counts().head(10)
    top_counts.plot(kind="barh", figsize=(8,6))
    plt.title(f"Top 10 Categories in {col}")
    plt.xlabel("Count")
    plt.show()
else:
    print("No categorical columns available for plotting.")
```

```
No categorical columns available for plotting.
```

7. Findings and Recommendations

Key Findings

During the cleaning and EDA process, we observed several important issues in the dataset:

- **High Missingness**: Some columns had more than 30% missing values, which makes them unreliable without remediation.
- **Inconsistent Categories**: Certain categorical fields contained multiple variations of the same label (e.g., spelling differences or free-text entries).
- **Duplicate Records**: Several duplicate rows were found and removed, which suggests that data collection lacked uniqueness checks.
- **Date Parsing Issues**: A few columns were misclassified as dates when they were actually IDs or text fields, which caused parsing errors.
- **Column Naming Variations**: Column names were inconsistent across sheets, using different cases, spaces, and symbols.

Recommendations

To improve the dataset quality going forward, we recommend the following actions:

- **Reduce Missingness at the Source**: Ensure required fields are filled in during data entry rather than relying only on cleaning after collection.
- **Standardize Categorical Data**: Use dropdown menus, controlled vocabularies, or codebooks to avoid inconsistent free-text labels.
- **Prevent Duplicates**: Introduce unique identifiers or primary keys in the data entry process to stop duplicate rows.
- **Adopt Date Standards**: Enforce ISO 8601 (`YYYY-MM-DD`) formatting for all date fields across systems to avoid parsing problems.
- **Consistent Naming Conventions**: Maintain `snake_case` naming across all datasets so that column names remain uniform.

By addressing these issues, the SHU VOCA dataset will become much more reliable, easier to analyze, and better suited for research and decision-making.

Conclusion

Through this project, we learned how important it is to clean and explore data before doing deeper analysis.
Starting from messy Excel sheets, we faced challenges like inconsistent column names, missing values, duplicate rows, and misclassified dates.
By applying step-by-step cleaning, building helper functions, and running EDA with visualizations, we transformed the dataset into a much clearer and more reliable form.

Our findings and recommendations highlight areas that need improvement at the data collection stage, which will make future analysis faster and more accurate.
This matters because well-structured, reliable data not only saves time but also leads to more trustworthy insights for decision-making.

Thank you for reviewing our work!