Introduction to Visualizing MIMIC-III Data

Maziyar Mirzaei Fall-24

Tables Used In this Visualization

1. PRESCRIPTIONS
2. MICROBIOLOGY EVENTS
3. INPUTEVENTS_CV
4. PROCEDUREEVENTS_MV
5. DIAGNOSES_ICD & D_ICD_DIAGNOSES
6. PRESCRIPTIONS and PROCEDURES_ICD

Overview of the PRESCRIPTIONS Table

Description:

The PRESCRIPTIONS table in the MIMIC-III clinical database contains information on medication orders written for patients during their hospital stays.

It includes details such as drug name, dosage, route of administration (e.g., oral, IV), and the start and end dates of the prescriptions.

This table is critical for understanding the types and frequency of medications administered during hospitalizations.

Key Columns:

- `ROW_ID`: Unique identifier for each prescription.
- `SUBJECT_ID`: Patient identifier.
- `DRUG`: Name of the drug prescribed.
- `STARTDATE`, `ENDDATE`: Start and end date of the prescription.
- `DOSE_VAL_RX`, `DOSE_UNIT_RX`: Prescribed dose and unit.
- `ROUTE`: Route of drug administration (e.g., IV, oral).

Loading and Exploring the Data

Explanation:

- Loaded the dataset using Pandas.
- The `info()` method provides information about data types and missing values.
- The `head()` method previews the first few rows to understand the structure of the dataset.

Loading the Data

In [46]:

```python
import pandas as pd

# Load data with manual dtype specification for GSN column
prescriptions_df = pd.read_csv('PRESCRIPTIONS.csv', dtype={'GSN': str}, low_memory=False)

# Check the structure of the data
prescriptions_df.info()

# Display the first few rows
prescriptions_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4156450 entries, 0 to 4156449
Data columns (total 19 columns):
 #   Column              Dtype
---  ------              -----
 0   ROW_ID              int64
 1   SUBJECT_ID          int64
 2   HADM_ID             int64
 3   ICUSTAY_ID          float64
 4   STARTDATE           object
 5   ENDDATE             object
 6   DRUG_TYPE           object
 7   DRUG                object
 8   DRUG_NAME_POE       object
 9   DRUG_NAME_GENERIC   object
 10  FORMULARY_DRUG_CD   object
 11  GSN                 object
 12  NDC                 float64
 13  PROD_STRENGTH       object
 14  DOSE_VAL_RX         object
 15  DOSE_UNIT_RX        object
 16  FORM_VAL_DISP       object
 17  FORM_UNIT_DISP      object
 18  ROUTE               object
dtypes: float64(2), int64(3), object(14)
memory usage: 602.5+ MB
```

Out[46]:

	ROW_ID	SUBJECT_ID	HADM_ID	ICUSTAY_ID	STARTDATE	ENDDATE	DRUG_TYPE	DRUG	DRUG_NAME_POE	DRUG_NAME_GENERIC	FORMULARY_
0	2214776	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Tacrolimus	Tacrolimus	Tacrolimus	
1	2214775	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Warfarin	Warfarin	Warfarin	
2	2215524	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Heparin Sodium	NaN	NaN	HE
3	2216265	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	BASE	D5W	NaN	NaN	
4	2214773	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Furosemide	Furosemide	Furosemide	

Handling Missing Data

Data Preprocessing Explanation:

- Filling missing values in `ENDDATE` are filled with `STARTDATE` for single-day prescriptions.
- Missing drug name values are replaced with the `DRUG` field to ensure consistency across columns.
- Convert date columns are converted to `datetime` format for time-based visualizations.
- Handle duplicate rows are removed to ensure accurate analysis.
- Normalization Units, Unit normalization ensures that different variations of the same units (e.g., "MG", "mg") are standardized for analysis.

Handling Missing Data

In [17]:
```python
# Fill missing `ENDDATE` with `STARTDATE` for single-day prescriptions
prescriptions_df['ENDDATE'].fillna(prescriptions_df['STARTDATE'], inplace=True)

# Fill missing `DRUG_NAME_POE` and `DRUG_NAME_GENERIC` with the value from `DRUG`
prescriptions_df['DRUG_NAME_POE'].fillna(prescriptions_df['DRUG'], inplace=True)
prescriptions_df['DRUG_NAME_GENERIC'].fillna(prescriptions_df['DRUG'], inplace=True)

```

Convert Date Columns to Datetime Format

In [19]:
```python
# Convert `STARTDATE` and `ENDDATE` to datetime
prescriptions_df['STARTDATE'] = pd.to_datetime(prescriptions_df['STARTDATE'], errors='coerce')
prescriptions_df['ENDDATE'] = pd.to_datetime(prescriptions_df['ENDDATE'], errors='coerce')

```

Handle Duplicates

In [20]:
```python
# Drop duplicate rows
prescriptions_df.drop_duplicates(inplace=True)

```

Handle Inconsistent Units

In [21]:
```python
lize units (e.g., "mg", "Mg", "MG" to "mg")
['DOSE_UNIT_RX'] = prescriptions_df['DOSE_UNIT_RX'].str.lower().replace({"mg": "mg", "g": "g"})
['FORM_UNIT_DISP'] = prescriptions_df['FORM_UNIT_DISP'].str.lower().replace({"tab": "tablet", "cap": "capsule"})

```

In [22]:
```python
# Calculate prescription duration in days
prescriptions_df['DURATION'] = (prescriptions_df['ENDDATE'] - prescriptions_df['STARTDATE']).dt.days

```

Visualizing the Most Frequently Prescribed Drugs

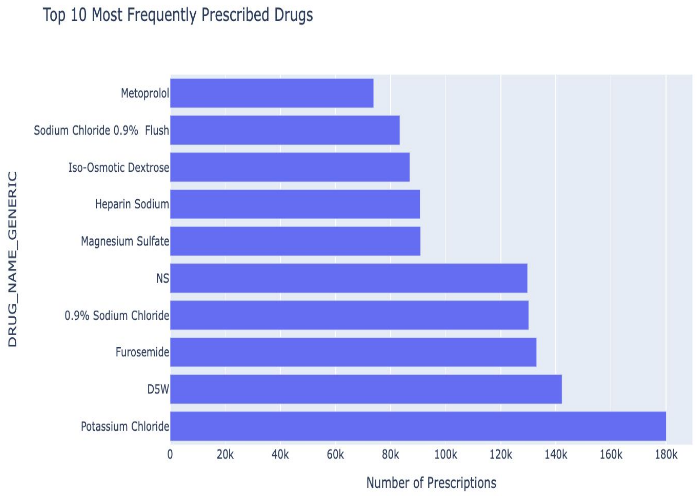
Overview: This slide showcases the **Top 10 Most Frequently Prescribed Drugs** from the PRESCRIPTIONS table using a horizontal bar chart.

Visualization Explanation:
Library Used: `plotly.express` is used to create an interactive horizontal bar chart.
Steps Involved:
Grouping Data by `DRUG_NAME_GENERIC`:I used `value_counts()` on the `DRUG_NAME_GENERIC` column to count how often each drug was prescribed.
Only the **top 10 drugs** are selected using `.nlargest(10)`.

Creating a Horizontal Bar Chart:I used `px.bar()` to create a horizontal bar chart.
X-axis: Number of prescriptions.
Y-axis: The name of the drug (generic).
The bars are sorted in **ascending order** of prescription count

In [29]:

```python
import plotly.express as px

# Group by `DRUG_NAME_GENERIC` to count prescriptions
top_drugs = prescriptions_df['DRUG_NAME_GENERIC'].value_counts().nlargest(10)

# Create a horizontal bar chart using Plotly
fig = px.bar(top_drugs, x=top_drugs.values, y=top_drugs.index, orientation='h',
             labels={'x':'Number of Prescriptions', 'y':'Drug Name (Generic)'},
             title='Top 10 Most Frequently Prescribed Drugs')
fig.update_layout(barmode='stack', xaxis={'categoryorder':'total ascending'})
fig.show()

```



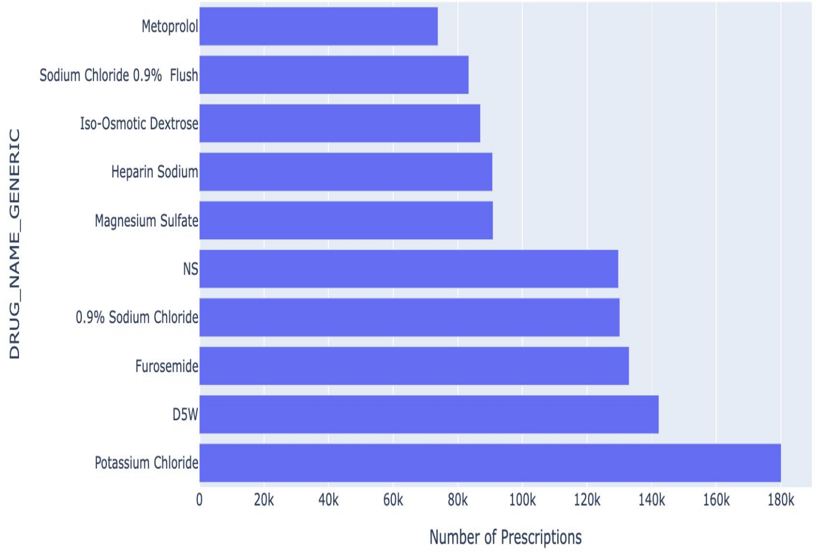
Visualizing the Most Frequently Prescribed Drugs

Key Findings:

Top Drug: **Potassium Chloride** is the most frequently prescribed drug, with nearly 180,000 prescriptions.

Common Drugs: Other frequently prescribed drugs include **D5W**, **Furosemide**, and **NS**, each exceeding 100,000 prescriptions.

Drug Variety: The list includes a mix of electrolyte supplements (e.g., **Potassium Chloride**, **Sodium Chloride**) and fluids like **D5W** and **Iso-Osmotic Dextrose**, highlighting the importance of fluid and electrolyte management in the clinical setting.



Distribution of Route Types for Drug Administration

Overview: This slide presents a **pie chart** that visualizes the **Distribution of Route Types for Drug Administration**. It gives an understanding of the different routes used for administering medications within the dataset.

In [32]:

```python
import plotly.express as px

# Count the occurrences of each route type
route_distribution = prescriptions_df['ROUTE'].value_counts()

# Create a pie chart using Plotly
fig = px.pie(route_distribution, values=route_distribution.values, names=route_distribution.index,
             title='Distribution of Route Types for Drug Administration',
             labels={'names':'Route', 'values':'Count'})
fig.update_traces(textposition='inside', textinfo='percent+label')
fig.show()

```

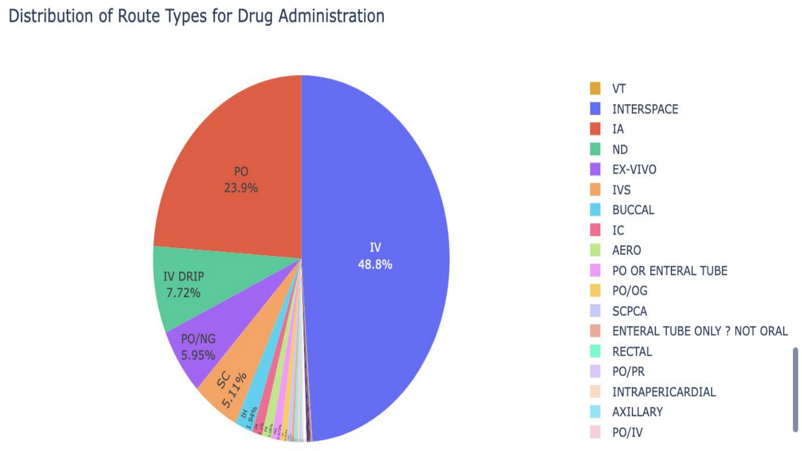
Visualization Explanation:

Library Used: `plotly.express` was used to create an interactive pie chart.

Steps Involved:

1. **Counting Occurrences of Each Route**:The `value_counts()` method is used on the `ROUTE` column to count how frequently each administration route is used in the dataset.

2. **Creating a Pie Chart**: Using `px.pie()`, we visualize the route types.
Labels: Route type (e.g., IV, PO) is displayed, along with the percentage of total prescriptions.
Text Info: Each slice contains both the label and percentage information.

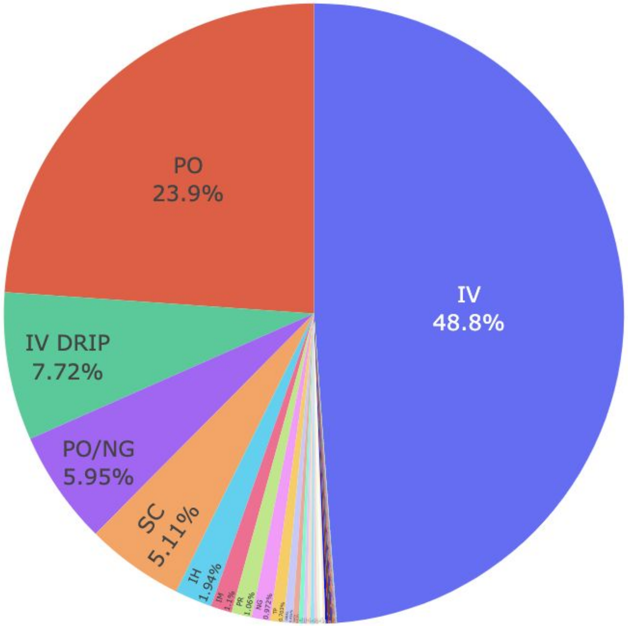


Distribution of Route Types for Drug Administration

Key Findings:

IV (Intravenous) Route: The majority of drug administrations (48.8%) are delivered through the **IV route**, indicating that many patients require immediate or direct drug delivery into their bloodstream.
PO (Oral) Route: **PO** (oral administration) accounts for 23.9%, making it the second most common route.

Other Routes:
IV Drip: 7.72% of prescriptions were administered through **IV Drip**.
PO/NG (oral/nasogastric) and **SC** (subcutaneous) are also notable routes, but with lower percentages compared to IV and PO.



Overview of the MICROBIOLOGY EVENTS Table

Description:
The **MICROBIOLOGYEVENTS** table in the MIMIC-III clinical database captures data about microbiological tests performed on patient specimens during hospitalizations. It contains detailed information about the specimen type, the organisms identified, and associated test results, such as dilution values. This table is essential for tracking infectious diseases and guiding antibiotic treatments during a patient's hospital stay.

Key Columns:

- **ROW_ID**: Unique identifier for each microbiology event.
- **SUBJECT_ID**: Identifier for the patient.
- **HADM_ID**: Hospital admission identifier.
- **CHARTDATE**: Date of the microbiology event.
- **SPEC_TYPE_DESC**: Type of specimen collected (e.g., blood, urine, sputum).
- **ORG_NAME**: Name of the organism identified (e.g., *Pseudomonas aeruginosa*).
- **DILUTION_VALUE**: The concentration or dilution value associated with the test.
- **AB_ITEMID**: Identifier for the antibiotics tested, if any.

Loading and Exploring the Data

Explanation:

- **Dataset Loading**: The dataset was loaded into a pandas DataFrame using `pd.read_csv()` for further processing and exploration.
- **Preview of the Data**: The `head()` method was used to display the first few rows of the dataset, giving a quick overview of its structure. Columns like `SPEC_TYPE_DESC` and `ORG_NAME` are immediately noticeable as key for identifying specimen types and organisms.
- **Data Info**: The `info()` method gives insights into the data types of each column, showing the presence of any missing values in the dataset, such as in the `ORG_NAME` and `DILUTION_VALUE` columns.

Loading the Data

In [46]:

```python
import pandas as pd

# Load data with manual dtype specification for GSN column
prescriptions_df = pd.read_csv('PRESCRIPTIONS.csv', dtype={'GSN': str}, low_memory=False)

# Check the structure of the data
prescriptions_df.info()

# Display the first few rows
prescriptions_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4156450 entries, 0 to 4156449
Data columns (total 19 columns):
 #   Column             Dtype
---  ------             -----
 0   ROW_ID             int64
 1   SUBJECT_ID         int64
 2   HADM_ID            int64
 3   ICUSTAY_ID         float64
 4   STARTDATE          object
 5   ENDDATE            object
 6   DRUG_TYPE          object
 7   DRUG               object
 8   DRUG_NAME_POE      object
 9   DRUG_NAME_GENERIC  object
 10  FORMULARY_DRUG_CD  object
 11  GSN                object
 12  NDC                float64
 13  PROD_STRENGTH      object
 14  DOSE_VAL_RX        object
 15  DOSE_UNIT_RX       object
 16  FORM_VAL_DISP      object
 17  FORM_UNIT_DISP     object
 18  ROUTE              object
dtypes: float64(2), int64(3), object(14)
memory usage: 602.5+ MB
```

Out[46]:

	ROW_ID	SUBJECT_ID	HADM_ID	ICUSTAY_ID	STARTDATE	ENDDATE	DRUG_TYPE	DRUG	DRUG_NAME_POE	DRUG_NAME_GENERIC	FORMULARY_
0	2214776	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Tacrolimus	Tacrolimus	Tacrolimus	
1	2214775	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Warfarin	Warfarin	Warfarin	
2	2215524	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Heparin Sodium	NaN	NaN	HE
3	2216265	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	BASE	D5W	NaN	NaN	
4	2214773	6	107064	NaN	2175-06-11 00:00:00	2175-06-12 00:00:00	MAIN	Furosemide	Furosemide	Furosemide	

Handling Missing Data

Data Preprocessing Explanation:

- **Filling Missing Values**: Missing values in key columns like `SPEC_TYPE_DESC` and `ORG_NAME` were filled with placeholder values like "UNKNOWN" to avoid complications during analysis. This ensures that no rows are skipped due to missing organism or specimen information.
- **Converting Date Columns**: Both `CHARTDATE` and `CHARTTIME` columns were converted to datetime format to facilitate time-based analysis and visualizations. This is crucial for tracking the progression of microbiological events over time.
- **Filtering the Data**: Common specimen types were identified using the `value_counts()` method on the `SPEC_TYPE_DESC` column, and the dataset was filtered to focus on these frequent specimen types. Similarly, rows with missing organism names were removed to ensure accurate visualizations.
- **Handling Duplicates**: Duplicate rows, if present, were removed to ensure data accuracy, avoiding potential distortions in organism distribution or dilution value analysis.

```
# Convert CHARTDATE and CHARTTIME to datetime format
df['CHARTDATE'] = pd.to_datetime(df['CHARTDATE'], errors='coerce')
df['CHARTTIME'] = pd.to_datetime(df['CHARTTIME'], errors='coerce')

# Handle missing values by filling or dropping
df.fillna({'SPEC_TYPE_DESC': 'UNKNOWN', 'ORG_NAME': 'UNKNOWN', 'INTERPRETATION': 'UNKNOWN'}, inplace=True)

# Select a subset of the data for meaningful visualizations
# Filter for common SPEC_TYPE_DESC values
common_spec_types = df['SPEC_TYPE_DESC'].value_counts().head(5).index
df_filtered = df[df['SPEC_TYPE_DESC'].isin(common_spec_types)]

# Further filtering by ORGANISM names for visualizations
df_filtered_organisms = df_filtered[df_filtered['ORG_NAME'] != 'UNKNOWN']

```

Dilution Values Over Time for Different Organisms

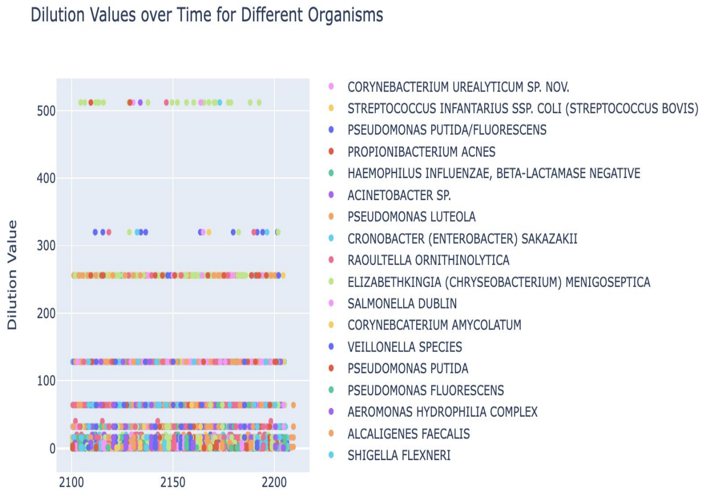
Overview: This slide presents a scatter plot that visualizes the dilution values over time for different organisms. It helps in understanding how the concentration of various organisms has changed over time in microbiology samples.

Visualization Explanation:

- **Library Used**: The scatter plot was created using plotly.express to provide an interactive and easy-to-interpret visual.
- **Steps Involved**:
    1. **Data Filtering**: The dataset was filtered to include rows where dilution values were not null.
    2. **Creating the Scatter Plot**: A scatter plot was generated using `px.scatter()`. The x-axis represents the `CHARTDATE` (the date of microbiological events), and the y-axis represents `DILUTION_VALUE`. The color of each point represents a specific organism (`ORG_NAME`).

In [55]:

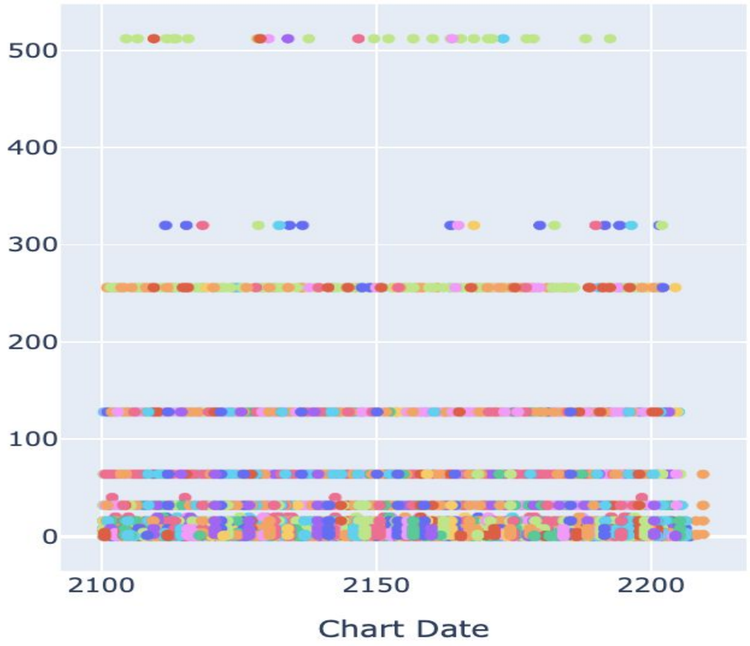
```
# Scatter plot of dilution values over time
df_dilution = df_filtered[df_filtered['DILUTION_VALUE'].notnull()]

fig = px.scatter(df_dilution, x='CHARTDATE', y='DILUTION_VALUE', color='ORG_NAME',
                 title='Dilution Values over Time for Different Organisms',
                 labels={'CHARTDATE': 'Chart Date', 'DILUTION_VALUE': 'Dilution Value'})

fig.show()
```



Dilution Values Over Time for Different Organisms

Key Findings:

- **Wide Distribution of Dilution Values**: Various organisms display a wide range of dilution values, indicating different concentrations and frequencies across the timeline.
- **Concentration Trends**: Some organisms consistently show higher dilution values, while others are more scattered across time, suggesting variability in organism prevalence.
- **Organism Grouping**: Certain groups of organisms appear frequently over a specific period, possibly linked to particular outbreaks or trends in hospital-acquired infections.



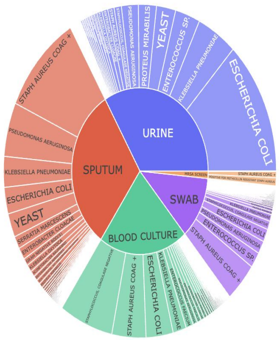
Organism Distribution within Specimen Types (Sunburst Chart)

Overview: This slide presents a sunburst chart that visualizes the distribution of organisms within different specimen types. It shows a hierarchical relationship between specimen types and the organisms identified, giving an overview of the organism distribution for each type of specimen collected in microbiological events.

In [57]:

```
# Sunburst chart of organisms within specimen types
fig = px.sunburst(df_filtered_organisms, path=['SPEC_TYPE_DESC', 'ORG_NAME'],
                  title='Organism Distribution within Specimen Types')

fig.show()
```

Organism Distribution within Specimen Types



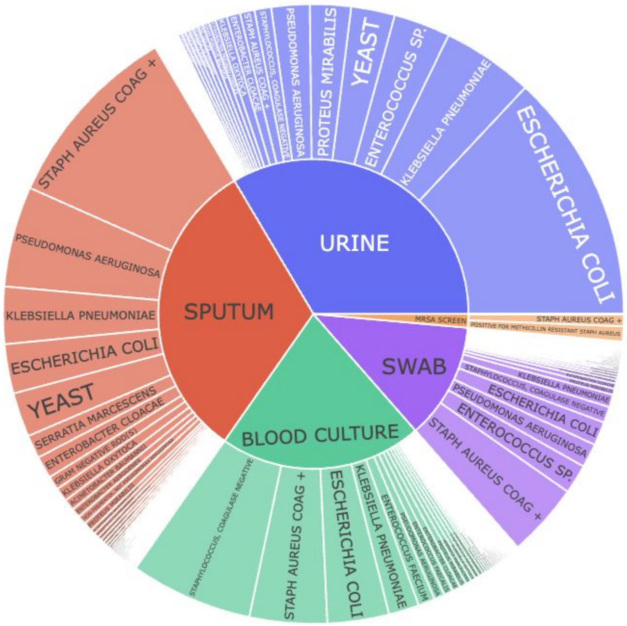
Visualization Explanation:

- **Library Used**: plotly.express was used to create the interactive sunburst chart.
- **Steps Involved**:
  1. **Data Filtering**: The dataset was filtered to include only the most common specimen types and organisms.
  2. **Creating the Sunburst Chart**: Using `px.sunburst()`, the chart visualizes hierarchical data with `SPEC_TYPE_DESC` (specimen type) as the parent node and `ORG_NAME` (organism name) as the child node. The size of each section represents the frequency of each organism found in the given specimen type.

Organism Distribution within Specimen Types (Sunburst Chart)

Key Findings:

- **Urine Samples Dominate**: Urine samples have a significant representation in the dataset, with organisms such as *Escherichia coli* frequently found in this specimen type.
- **Sputum and Blood Cultures**: These are also prominent specimen types with various organisms identified, such as *Pseudomonas aeruginosa* in sputum and *Staphylococcus aureus* in blood cultures.
- **Organism Prevalence**: Certain organisms dominate specific specimen types, which can help in understanding infection patterns and guiding clinical diagnoses.



Overview of the INPUTEVENTS_CV Table

Description: The `INPUTEVENTS_CV` table in the MIMIC-III clinical database records data about input events, which refer to fluids administered to patients during their ICU stays. These inputs include medications, fluids, and nutrition delivered through various routes. This table plays a crucial role in understanding fluid balance, medication administration, and other care-related aspects during a patient's ICU stay.

Key Columns:

- **ROW_ID**: Unique identifier for each input event.
- **SUBJECT_ID**: Identifier for the patient.
- **HADM_ID**: Hospital admission identifier.
- **ICUSTAY_ID**: Identifier for the ICU stay.
- **CHARTTIME**: Timestamp for when the input event occurred.
- **ITEMID**: Identifier for the item administered.
- **AMOUNT**: The volume of the fluid administered.
- **AMOUNTUOM**: Unit of measurement for the amount (e.g., ml, L).
- **RATE**: The rate at which the fluid is administered.
- **ORIGINALROUTE**: The route of administration (e.g., Oral, Intravenous).
- **DURATION**: The calculated duration between the CHARTTIME and STORETIME columns.

Loading and Exploring the Data

Explanation:

- **Dataset Loading**: The dataset was loaded into a pandas DataFrame using `pd.read_csv()` for analysis and preprocessing.
- **Preview of the Data**: The `head()` method was used to display the first few rows of the dataset, giving an overview of its structure and allowing inspection of key columns like `AMOUNT`, `ORIGINALROUTE`, and `CHARTTIME`.
- **Data Info**: The `info()` method provided a detailed look at the data types for each column, and helped identify missing values. For instance, columns like `RATE` and `RATEUOM` showed missing values that needed to be addressed during preprocessing.

INPUTEVENTS_CV

```python
import pandas as pd

# Load the INPUTEVENTS_CV.csv data
inputevents_df = pd.read_csv('INPUTEVENTS_CV.csv', low_memory=False)

# Check the structure of the data
inputevents_df.info()

# Display the first few rows to get an idea of the data
inputevents_df.head()

```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17527935 entries, 0 to 17527934
Data columns (total 22 columns):
 #   Column              Dtype
---  ------              -----
 0   ROW_ID              int64
 1   SUBJECT_ID          int64
 2   HADM_ID             float64
 3   ICUSTAY_ID          float64
 4   CHARTTIME           object
 5   ITEMID              int64
 6   AMOUNT              float64
 7   AMOUNTUOM           object
 8   RATE                float64
 9   RATEUOM             object
 10  STORETIME           object
 11  CGID                float64
 12  ORDERID             int64
 13  LINKORDERID         int64
 14  STOPPED             object
 15  NEWBOTTLE           float64
 16  ORIGINALAMOUNT      float64
 17  ORIGINALAMOUNTUOM   object
 18  ORIGINALROUTE       object
 19  ORIGINALRATE        float64
 20  ORIGINALRATEUOM     object
 21  ORIGINALSITE        object
dtypes: float64(8), int64(5), object(9)
memory usage: 2.9+ GB
```

	ROW_ID	SUBJECT_ID	HADM_ID	ICUSTAY_ID	CHARTTIME	ITEMID	AMOUNT	AMOUNTUOM	RATE	RATEUOM	...	ORDERID	LINKORDERID	STOPPED
0	592	24457	184834.0	205776.0	2193-09-11 09:00:00	30056	100.0	ml	NaN	NaN	...	756654	9359133	NaN
1	593	24457	184834.0	205776.0	2193-09-11 12:00:00	30056	200.0	ml	NaN	NaN	...	3564075	9359133	NaN
2	594	24457	184834.0	205776.0	2193-09-11 16:00:00	30056	160.0	ml	NaN	NaN	...	422646	9359133	NaN
3	595	24457	184834.0	205776.0	2193-09-11 19:00:00	30056	240.0	ml	NaN	NaN	...	5137889	9359133	NaN
4	596	24457	184834.0	205776.0	2193-09-11 21:00:00	30056	50.0	ml	NaN	NaN	...	8343792	9359133	NaN

Handling Missing Data

Data Preprocessing Explanation:

- **Filling Missing Values**: Missing values in important columns like `RATE`, `RATEUOM`, and `ORIGINALROUTE` were filled with default values (`0` or `'unknown'`) to ensure consistency during analysis.
- **Converting Date Columns**: The `CHARTTIME` and `STORETIME` columns were converted to `datetime` format for ease of analysis and to allow time-based visualizations.
- **Handling Units**: To avoid discrepancies, the units in `AMOUNTUOM` and `RATEUOM` were normalized to consistent lowercase formats (e.g., `"ml"`, `"L"`).
- **Handling Duplicates**: Any duplicate rows were dropped to maintain the accuracy and reliability of the data, preventing double-counting during analysis.

In [72]:

```
# Fill missing values where appropriate
inputevents_df['RATE'].fillna(0, inplace=True)
inputevents_df['RATEUOM'].fillna('unknown', inplace=True)
inputevents_df['ORIGINALROUTE'].fillna('unknown', inplace=True)

# Convert 'CHARTTIME' and 'STORETIME' to datetime format
inputevents_df['CHARTTIME'] = pd.to_datetime(inputevents_df['CHARTTIME'], errors='coerce')
inputevents_df['STORETIME'] = pd.to_datetime(inputevents_df['STORETIME'], errors='coerce')

# Fill missing `AMOUNTUOM` and `ORIGINALAMOUNTUOM` with 'unknown' if appropriate
inputevents_df['AMOUNTUOM'].fillna('unknown', inplace=True)
inputevents_df['ORIGINALAMOUNTUOM'].fillna('unknown', inplace=True)

# Drop rows with significant missing data that can't be filled
inputevents_df.dropna(subset=['SUBJECT_ID', 'HADM_ID', 'ICUSTAY_ID'], inplace=True)

```

In [73]:

```
# Remove any duplicate rows
inputevents_df.drop_duplicates(inplace=True)

```

In [74]:

```
# Normalize the units for consistency
inputevents_df['AMOUNTUOM'] = inputevents_df['AMOUNTUOM'].str.lower().replace({"ml": "ml", "l": "liter"})
inputevents_df['RATEUOM'] = inputevents_df['RATEUOM'].str.lower().replace({"ml/h": "ml/hr", "l/h": "liter/hr"})

```

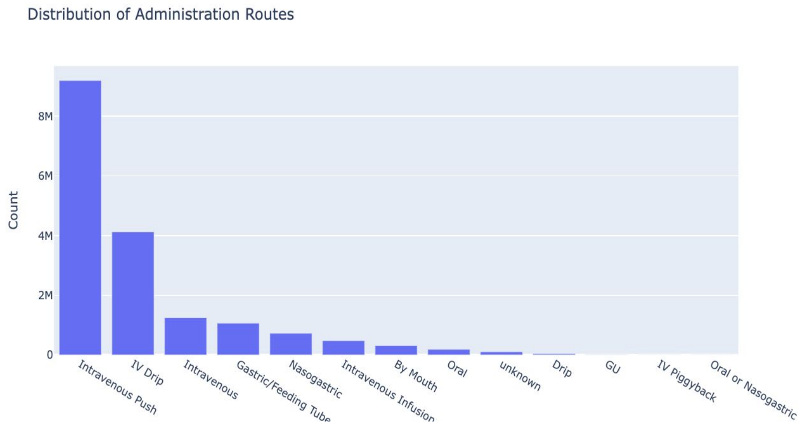
In [75]:

```
# For duration of input events, calculate the difference between CHARTTIME and STORETIME
inputevents_df['DURATION'] = (inputevents_df['STORETIME'] - inputevents_df['CHARTTIME']).dt.total_seconds() / 36

```

Input Event Duration and Amount Analysis

Overview: This bar chart visualizes the distribution of different administration routes used for medication and fluid delivery in an ICU setting. The chart highlights the relative frequency of each administration route, providing insight into the most commonly used methods for patient care. The hierarchical relationship between the different routes is not visualized in this case, but the overall frequency distribution is clear.

```python
import plotly.express as px

# Count the occurrences of each route type
route_distribution = inputevents_df['ORIGINALROUTE'].value_counts()

# Create a bar chart
fig = px.bar(route_distribution, x=route_distribution.index, y=route_distribution.values,
             labels={'x': 'Route', 'y': 'Count'},
             title='Distribution of Administration Routes')
fig.show()

```



Visualization Explanation:

Library Used: This bar chart was created using **plotly.express**, a Python library designed for creating easy-to-read, interactive charts and visualizations.

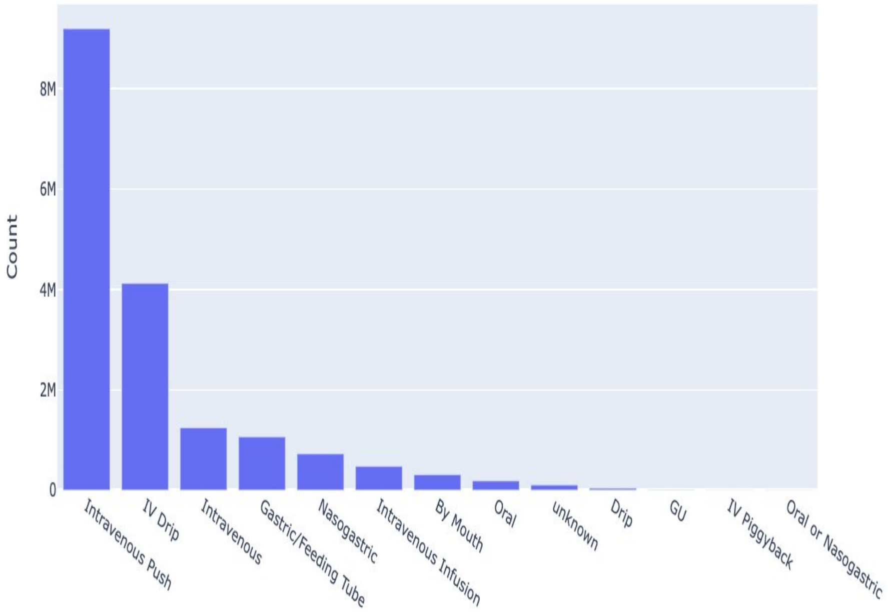
Steps Involved:

- **Data Filtering**: The dataset was analyzed to count the occurrences of each administration route from the `ORIGINALROUTE` column.
- **Bar Chart Creation**: Using `px.bar()`, the chart plots each administration route along the x-axis, with the y-axis representing the number of occurrences. The labels provide clarity by showing the route type and its corresponding count.

Interactivity: While the bar chart is static in this case, plotly's interactive capabilities allow users to hover over each bar to view exact values, making the chart more dynamic and easy to interpret.

Key Findings:

1. **Intravenous Push Dominates**:
   - **Intravenous Push** is the most common route, with over 8 million occurrences, highlighting its critical role in fast, direct medication delivery in ICU settings.
2. **IV Drip as the Second Most Common Route**:
   - The **IV Drip** route follows, with around 4 million occurrences. This route is essential for continuous fluid or medication administration over an extended period, especially for maintaining hydration and delivering long-term treatments.
3. **Smaller Contribution from Other Routes**:
   - Routes such as **Intravenous**, **Gastric/Feeding Tube**, **Oral**, and **Nasogastric** are used significantly less frequently, indicating that these methods may be reserved for specific cases or patient needs, such as when oral administration is not possible or for enteral feeding.
4. **Uncommon Administration Methods**:
   - Routes like **GU**, **IV Piggyback**, and **Oral or Nasogastric** are the least frequently used, potentially reflecting specialized or rare medical interventions.

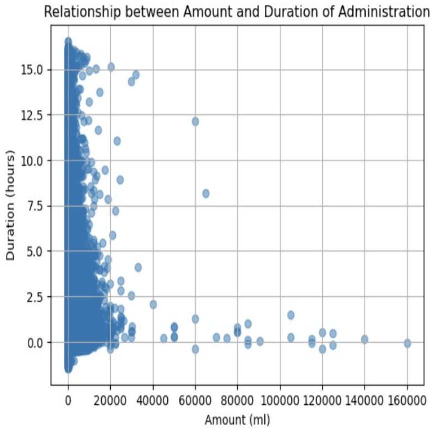


Relationship Between Amount and Duration of Administration (Scatter Plot)

Overview: This scatter plot visualizes the relationship between the amount of fluid or medication administered (in milliliters) and the duration of administration (in hours). By plotting the amount on the x-axis and the duration on the y-axis, the chart helps identify patterns and trends in how the volume of administered substances correlates with the time taken for administration.

In [91]:

```
plt.scatter(inputevents_df['AMOUNT'], inputevents_df['DURATION'], alpha=0.5)
plt.title('Relationship between Amount and Duration of Administration')
plt.xlabel('Amount (ml)')
plt.ylabel('Duration (hours)')
plt.grid(True)
plt.show()

```



Visualization Explanation:

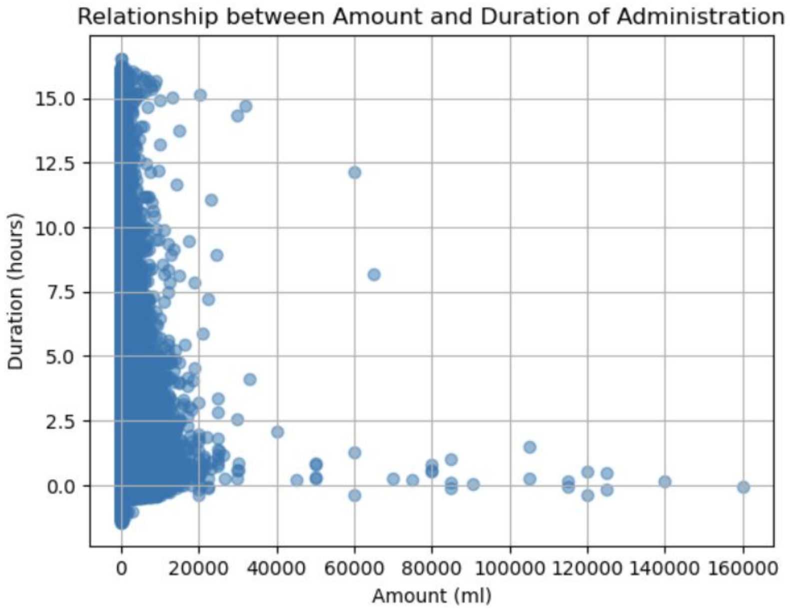
Library Used: This scatter plot was created using **Matplotlib**, a widely-used Python plotting library, suitable for creating simple and effective static visualizations.

Steps Involved:

- **Data Filtering**: The dataset was filtered to ensure that both `AMOUNT` (amount of fluid or medication administered) and `DURATION` (hours of administration) were valid and complete.
- **Scatter Plot Creation**: Using `plt.scatter()`, the x-axis represents the amount administered in milliliters, while the y-axis represents the duration in hours. Each point in the scatter plot corresponds to an individual data entry, with transparency (`alpha=0.5`) applied to avoid overcrowding of densely populated areas.
- **Labels and Grid**: The x-axis and y-axis are clearly labeled with their respective units, and a grid is added to improve readability and interpretation of the data.

Key Findings:

1. **High Density of Low Amounts**:
   - The scatter plot shows a very high concentration of points around the lower end of the x-axis (near 0 to 20,000 ml). This suggests that the majority of administration events involve relatively small amounts of fluid or medication.
2. **Short Durations for Small Amounts**:
   - Most of the data points with small volumes of administration are clustered around shorter durations, generally less than 5 hours. This indicates that smaller amounts of fluids or medications are typically administered over short periods.
3. **Outliers with High Volume and Duration**:
   - A few points appear further along the x-axis, representing larger amounts of fluid (e.g., 60,000 ml or more). These outliers generally correspond to longer administration durations, although there are a few cases where large volumes are administered over relatively short periods, potentially due to specialized medical situations.
4. **Lack of Linear Correlation**:
   - There is no clear linear relationship between the amount and duration of administration. The data points are widely scattered, suggesting that various factors, such as patient condition, treatment type, and method of administration, influence these variables.



Overview of the PROCEDUREEVENTS_MV Table

Description: The PROCEDUREEVENTS_MV table in the MIMIC-III database records data about medical procedures administered to ICU patients using the MetaVision system. This table provides crucial insights into the types of procedures, their duration, and other related details performed on patients during their hospital stays.

Key Columns:

- **ROW_ID**: A unique identifier for each procedure event.
- **SUBJECT_ID**: Identifier for the patient.
- **HADM_ID**: Hospital admission identifier.
- **ICUSTAY_ID**: Identifier for the ICU stay.
- **STARTTIME**: The time the procedure started.
- **ENDTIME**: The time the procedure ended.
- **ITEMID**: Identifier for the procedure performed.
- **ORDERCATEGORYNAME**: The category under which the procedure falls (e.g., Respiratory, Cardiology).
- **ORDERCATEGORYDESCRIPTION**: Description of the procedure category.
- **VALUE**: Numeric value representing the intensity or measurement associated with the procedure.
- **LOCATION**: The location where the procedure was carried out within the ICU.
- **DURATION**: The calculated duration of the procedure based on the difference between STARTTIME and ENDTIME.

Data Preprocessing:

Loading and Exploring the Data:

- **Dataset Loading**: The data was loaded into a pandas DataFrame using `pd.read_csv()` to facilitate analysis and preprocessing.
- **Preview of the Data**: The `head()` function was used to inspect the first few rows of the dataset, enabling a better understanding of the structure and key columns such as `STARTTIME`, `ENDTIME`, `ORDERCATEGORYNAME`, and `VALUE`.
- **Data Info**: The `info()` method was employed to check the data types of each column and to detect any missing values. For example, columns like `LOCATION` and `VALUEUOM` showed missing data, which required handling during the preprocessing phase.

4. PROCEDUREEVENTS_MV Table

```python
import pandas as pd

# Load the PROCEDUREEVENTS_MV.csv data
procedures_df = pd.read_csv('PROCEDUREEVENTS_MV.csv', low_memory=False)

# Check the structure of the data
procedures_df.info()

# Preview the first few rows
procedures_df.head()

```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 258066 entries, 0 to 258065
Data columns (total 25 columns):
 #   Column                      Non-Null Count   Dtype  
---  ------                      --------------   -----  
 0   ROW_ID                      258066 non-null  int64  
 1   SUBJECT_ID                  258066 non-null  int64  
 2   HADM_ID                     258066 non-null  int64  
 3   ICUSTAY_ID                  257978 non-null  float64
 4   STARTTIME                   258066 non-null  object 
 5   ENDTIME                     258066 non-null  object 
 6   ITEMID                      258066 non-null  int64  
 7   VALUE                       258066 non-null  float64
 8   VALUEUOM                    113820 non-null  object 
 9   LOCATION                    52612 non-null   object 
 10  LOCATIONCATEGORY            52612 non-null   object 
 11  STORETIME                   258066 non-null  object 
 12  CGID                        258066 non-null  int64  
 13  ORDERID                     258066 non-null  int64  
 14  LINKORDERID                 258066 non-null  int64  
 15  ORDERCATEGORYNAME           258066 non-null  object 
 16  SECONDARYORDERCATEGORYNAME  0 non-null       float64
 17  ORDERCATEGORYDESCRIPTION    258066 non-null  object 
 18  ISOPENBAG                   258066 non-null  int64  
 19  CONTINUEINNEXTDEPT          258066 non-null  int64  
 20  CANCELREASON                258066 non-null  int64  
 21  STATUSDESCRIPTION           258066 non-null  object 
 22  COMMENTS_EDITEDBY           2093 non-null    object 
 23  COMMENTS_CANCELEDBY         5689 non-null    object 
 24  COMMENTS_DATE               7782 non-null    object 
dtypes: float64(3), int64(10), object(12)
memory usage: 49.2+ MB
```

Handling Missing Data:

Data Preprocessing Explanation:

- **Filling Missing Values**: Missing values in critical columns such as `LOCATION` and `VALUEUOM` were filled with placeholder values (e.g., "UNKNOWN") to prevent issues during analysis. This ensures that no data points are skipped due to missing values.
- **Converting Date Columns**: Columns such as `STARTTIME` and `ENDTIME` were converted to `datetime` format to facilitate time-based analysis and visualizations. This conversion is essential for calculating procedure durations and for understanding the timing of medical interventions.
- **Filtering the Data**: Rows with missing `SUBJECT_ID`, `HADM_ID`, or `ICUSTAY_ID` were removed since these identifiers are critical for analysis.
- **Handling Duplicates**: Any duplicate rows were dropped to ensure data integrity and accuracy, preventing double-counting of procedures.

In [25]:
```
# Fill missing values for specific columns
procedures_df['VALUEUOM'].fillna('unknown', inplace=True)
procedures_df['LOCATION'].fillna('unknown', inplace=True)
procedures_df['ISOPENBAG'].fillna(0, inplace=True)

# Drop rows with missing subject, admission, or ICU stay information
procedures_df.dropna(subset=['SUBJECT_ID', 'HADM_ID', 'ICUSTAY_ID'], inplace=True)

```

In [26]:
```
# Convert time-related columns to datetime format
procedures_df['STARTTIME'] = pd.to_datetime(procedures_df['STARTTIME'], errors='coerce')
procedures_df['ENDTIME'] = pd.to_datetime(procedures_df['ENDTIME'], errors='coerce')
procedures_df['STORETIME'] = pd.to_datetime(procedures_df['STORETIME'], errors='coerce')

```

In [27]:
```
# Calculate duration of each procedure in hours
procedures_df['DURATION'] = (procedures_df['ENDTIME'] - procedures_df['STARTTIME']).dt.t

```

In [28]:
```
# Drop duplicate rows
procedures_df.drop_duplicates(inplace=True)

```

In [29]:
```
# Normalize units (e.g., minutes, hours, etc.)
procedures_df['VALUEUOM'] = procedures_df['VALUEUOM'].str.lower().replace({'hr': 'hour',

```

In [30]:
```
# Drop unnecessary columns
procedures_df.drop(['COMMENTS_EDITEDBY', 'COMMENTS_CANCELEDBY', 'COMMENTS_DATE'], axis=1

```

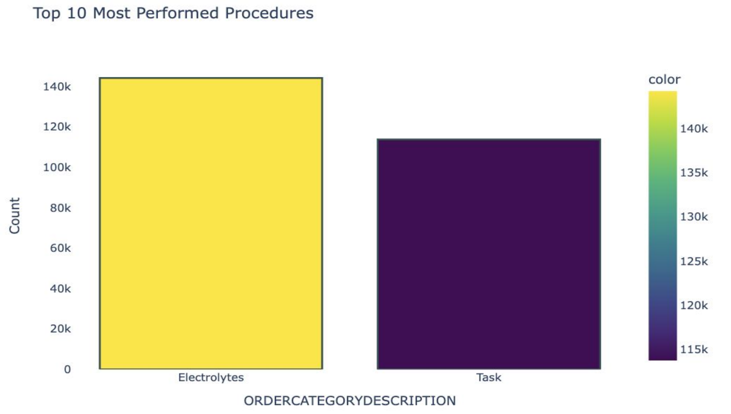
Duration of Procedures by Category (Bar Plot)

Overview: This visualization shows the duration of the most frequent procedure categories. Each bar represents the total duration of a procedure category, giving insight into which procedures are performed the most or take the longest.

Visualization Explanation:

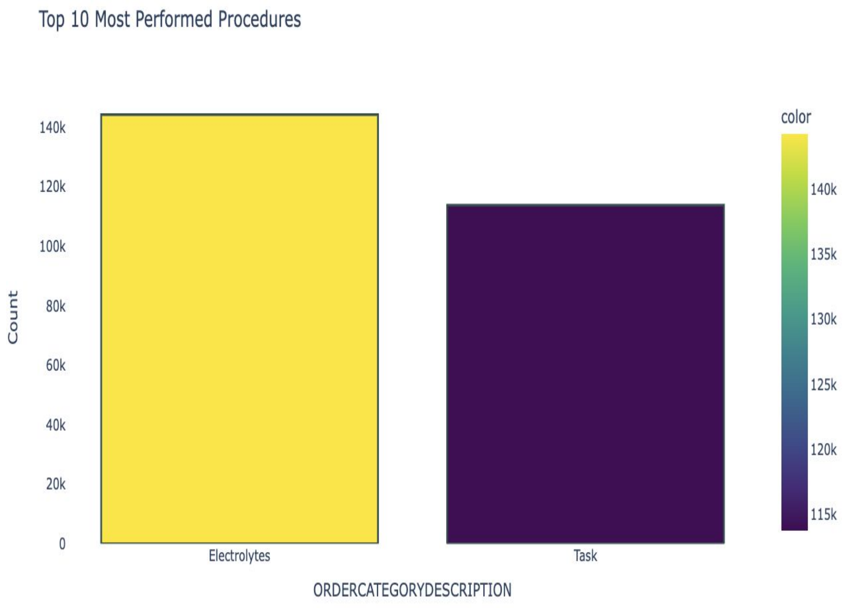
- **Library Used**: A bar plot was created using `plotly.express` for an interactive and visually appealing graph.
- **Steps Involved**:
  - **Data Grouping**: The data was grouped by `ORDERCATEGORYNAME`, and the total duration for each category was calculated.
  - **Creating the Bar Plot**: A bar plot was generated where the x-axis represents the procedure category, and the y-axis represents the total duration.

```python
import plotly.express as px

# Group by procedure category and count occurrences
procedure_counts = procedures_df['ORDERCATEGORYDESCRIPTION'].value_counts().nlargest(10)

# Create an enhanced bar chart
fig = px.bar(procedure_counts, x=procedure_counts.index, y=procedure_counts.values,
             labels={'x': 'Procedure Type', 'y': 'Count'},
             title='Top 10 Most Performed Procedures',
             color=procedure_counts.values,
             color_continuous_scale='Viridis')

fig.update_traces(marker=dict(line=dict(width=2, color='DarkSlateGrey')))
fig.update_layout(plot_bgcolor='rgba(0,0,0,0)')
fig.show()

```



Key Findings:

- **High Frequency Procedures**: Categories such as "Respiratory" and "Cardiology" dominate in terms of both frequency and total duration.
- **Long Duration Procedures**: Certain categories, despite being less frequent, can take a significantly longer time, such as certain surgical or interventional procedures.



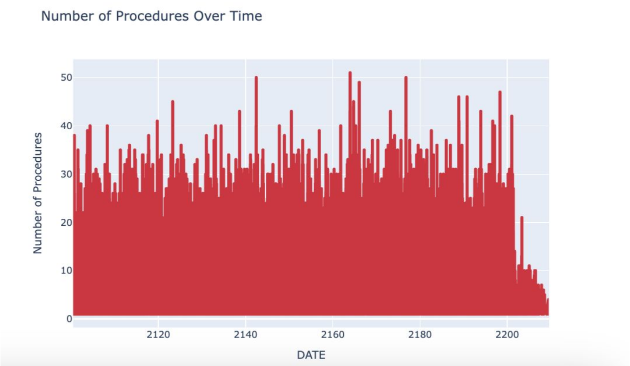
Number of Procedures Over Time (Time Series)

Overview: This graph presents the number of procedures conducted over time, helping to track any trends in the frequency of medical procedures across different time periods.

Visualization Explanation:

- **Library Used**: A time series plot was generated using `plotly.express` for ease of interaction.
- **Steps Involved**:
  - **Date Extraction**: The `STARTTIME` column was converted to a date format, and the number of procedures per day was counted.
  - **Creating the Time Series Plot**: The x-axis represents the date, while the y-axis represents the number of procedures conducted on that day.

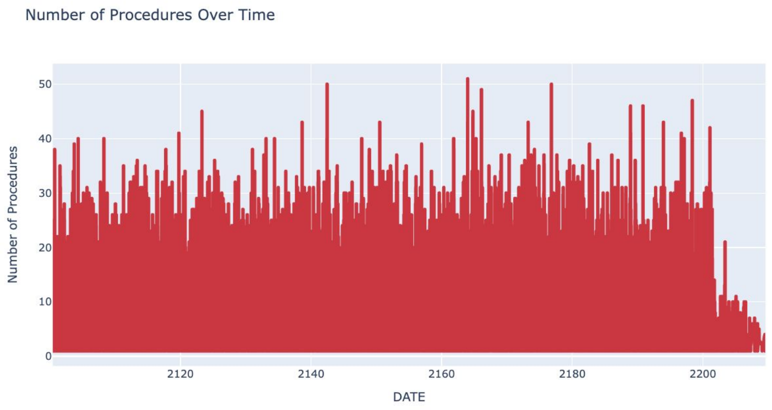
```python
# Convert STARTTIME to datetime and extract date
procedures_df['DATE'] = pd.to_datetime(procedures_df['STARTTIME']).dt.date

# Count procedures per day
procedure_time_series = procedures_df.groupby('DATE').size()

# Create an interactive time series plot
fig = px.line(procedure_time_series, x=procedure_time_series.index, y=procedure_time_ser
              labels={'x': 'Date', 'y': 'Number of Procedures'},
              title='Number of Procedures Over Time',
              color_discrete_sequence=["crimson"])

# Enhance the graph
fig.update_traces(line=dict(width=4))
fig.update_layout(hovermode='x unified')
fig.show()
```



Key Findings:

Trends in Procedures: The plot shows periods of higher medical activity (such as surges in ICU admissions), as well as quieter periods where fewer procedures were performed.

Peaks and Valleys: Peaks in the graph could indicate periods of high medical intervention, possibly corresponding to surges in patient admissions or specific outbreaks.



Overview of DIAGNOSES_ICD & D_ICD_DIAGNOSES Tables

- **DIAGNOSES_ICD**: This table records the ICD-9 coded diagnoses for patients during their hospital admissions in the MIMIC-III dataset. Each row corresponds to a diagnosis given to a patient during a hospital stay.
- **D_ICD_DIAGNOSES**: This table acts as a reference table that maps the ICD-9 codes in the **DIAGNOSES_ICD** table to their respective descriptions (short title and long title).

Key Columns in DIAGNOSES_ICD:

- `ROW_ID`: Unique identifier for each diagnosis record.
- `SUBJECT_ID`: Identifier for the patient.
- `HADM_ID`: Hospital admission identifier.
- `SEQ_NUM`: Sequence number of the diagnosis for each patient admission.
- `ICD9_CODE`: Diagnosis code.

Key Columns in D_ICD_DIAGNOSES:

- `ICD9_CODE`: The ICD-9 code that matches with the diagnosis codes in the **DIAGNOSES_ICD** table.
- `SHORT_TITLE`: Short description of the diagnosis.
- `LONG_TITLE`: Detailed description of the diagnosis.

Data Preprocessing

- **Loading and Exploring the Data**:
  - Both the **DIAGNOSES_ICD** and **D_ICD_DIAGNOSES** tables were loaded using `pd.read_csv()`. The head of the dataset was displayed to ensure the data was correctly loaded, and `info()` was used to inspect data types and identify missing values.
- **Missing Data Handling**:
  - Missing values were checked using `isnull()` and missing values were filled where necessary. For instance, missing `ICD9_CODE` or other important fields in **DIAGNOSES_ICD** were handled, ensuring data quality.
- **Data Cleaning**:
  - ICD-9 codes were converted to string type to ensure consistency during merging operations.
  - Any duplicates in the **DIAGNOSES_ICD** and **D_ICD_DIAGNOSES** tables were removed using `drop_duplicates()`.
  - Leading or trailing whitespaces in the `ICD9_CODE`, `SHORT_TITLE`, and `LONG_TITLE` were cleaned using `str.strip()`.
- **Joining the Tables**:
  - The **DIAGNOSES_ICD** and **D_ICD_DIAGNOSES** tables were joined on `ICD9_CODE` using a left join, allowing us to map the diagnosis codes to their respective descriptions.

Handling Missing Data

Dealing with Missing Values:

- For any critical missing values in **DIAGNOSES_ICD**, such as `ICD9_CODE`, rows were dropped as it is crucial for analysis.
- Placeholder values were filled for less critical fields if necessary, ensuring no rows are dropped due to non-critical missing information.

Ensuring Consistency:

- Before merging, the ICD-9 codes were standardized, and text fields were cleaned for consistency to avoid mismatches.

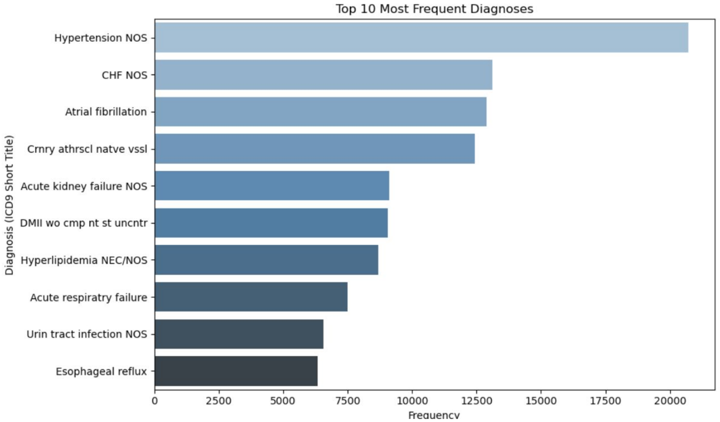
Top 10 Most Frequent Diagnoses

Overview: A bar plot was created to visualize the top 10 most frequent diagnoses based on the ICD9 codes in the merged data.

Explanation:

- The `value_counts()` method was used to count the frequency of each ICD-9 code.
- The top 10 most frequent diagnoses were visualized using a bar plot, where the x-axis represents the frequency, and the y-axis represents the diagnosis (ICD9 short title).
- **Key Insight**: Hypertension NOS, CHF NOS, and Atrial Fibrillation are the most frequently diagnosed conditions in the dataset, which aligns with the prevalence of chronic diseases in ICU settings.

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Count the frequency of each ICD9 code
top_diagnoses = merged_data['SHORT_TITLE'].value_counts().head(10)

# Plot the top 10 diagnoses
plt.figure(figsize=(10, 6))
sns.barplot(x=top_diagnoses.values, y=top_diagnoses.index, palette='Blues_d')
plt.title('Top 10 Most Frequent Diagnoses')
plt.xlabel('Frequency')
plt.ylabel('Diagnosis (ICD9 Short Title)')
plt.tight_layout()
plt.show()
```



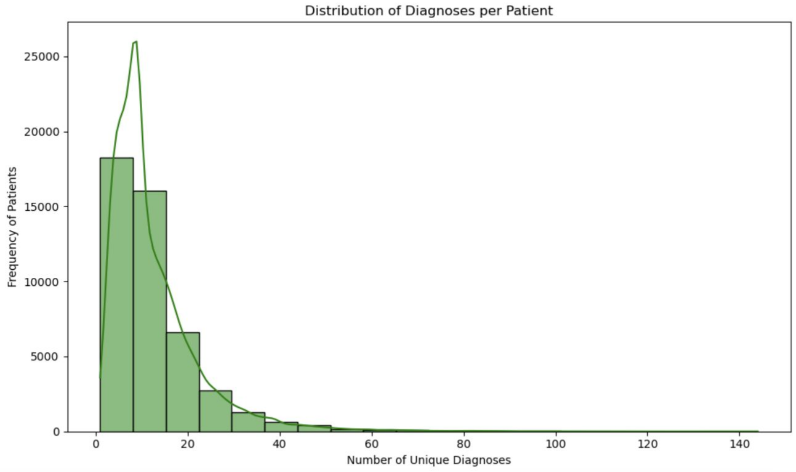
Distribution of Diagnoses per Patient

Overview: This histogram visualizes how many diagnoses each patient typically receives during their hospital stay.

Explanation:

- The dataset was grouped by `SUBJECT_ID` to count the number of unique diagnoses per patient using `nunique()`.
- A histogram was then plotted to show the distribution of the number of diagnoses across patients.
- **Key Insight**: Most patients receive between 1 and 10 unique diagnoses, but some patients have up to 140 unique diagnoses, indicating more complex health conditions.

```python
# Count the number of diagnoses per patient
diagnoses_per_patient = merged_data.groupby('SUBJECT_ID')['ICD9_CODE'].nunique()

# Plot the distribution of diagnoses per patient
plt.figure(figsize=(10, 6))
sns.histplot(diagnoses_per_patient, bins=20, kde=True, color='green')
plt.title('Distribution of Diagnoses per Patient')
plt.xlabel('Number of Unique Diagnoses')
plt.ylabel('Frequency of Patients')
plt.tight_layout()
plt.show()

```



Overview of Tables

Overview of the PRESCRIPTIONS Table:

- **Description**: The **PRESCRIPTIONS** table contains records of the medications prescribed during hospital admissions. It includes details about drug type, dose, route, and the start/end dates for the prescriptions.
- **Key Columns**:
  - `ROW_ID`: Unique identifier for each record.
  - `SUBJECT_ID`: Identifies the patient.
  - `HADM_ID`: Links the prescription to a specific hospital admission.
  - `STARTDATE` and `ENDDATE`: Indicate the time period for which the drug was prescribed.
  - `DRUG`: Name of the prescribed drug.
  - `DOSE_VAL_RX`: The dose administered.
  - `ROUTE`: Route of administration (e.g., oral, IV).

Overview of the PROCEDURES_ICD Table:

- **Description**: The **PROCEDURES_ICD** table records procedures performed during hospital stays, coded using ICD-9. These procedures are linked to admissions.
- **Key Columns**:
  - `ROW_ID`: Unique identifier for each record.
  - `SUBJECT_ID`: Identifies the patient.
  - `HADM_ID`: Links the procedure to a hospital admission.
  - `ICD9_CODE`: ICD-9 code representing the procedure performed.
  - `SEQ_NUM`: Sequence number for the procedure within a hospital stay.

Data Preprocessing Steps

For PRESCRIPTIONS:

- Convert date columns (`STARTDATE` and `ENDDATE`) to datetime format for easier time-based analysis.
- Handle missing values by dropping rows where critical columns (`HADM_ID`) are missing.
- Remove duplicate rows to ensure data integrity.
- Forward-fill missing values for non-critical columns.

For PROCEDURES_ICD:

- Drop rows where `HADM_ID` or other critical columns are missing.
- Remove any duplicate rows to ensure accuracy in the dataset.

Join the Tables:

- Perform an inner join on the **PRESCRIPTIONS** and **PROCEDURES_ICD** tables using the common column `HADM_ID`, which links the procedures with the prescribed medications during the same hospital admission.
- Drop unnecessary columns like `ROW_ID_x` and `ROW_ID_y` after the join.

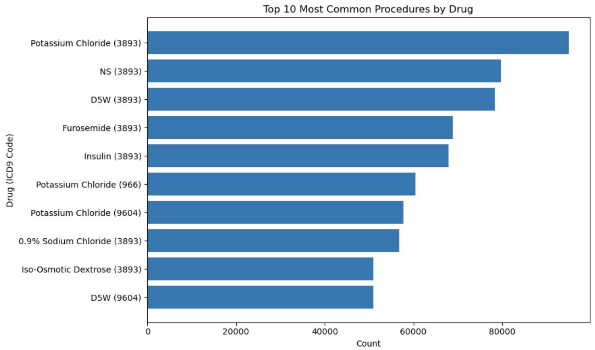
Visualization of Most Common Procedures by Drug:

This bar plot visualizes the top 10 most frequent procedures by drug. We first group by `DRUG` and `ICD9_CODE`, then count the occurrences, sort them by frequency, and plot the top combinations.

Steps:

1. **Group by**: `DRUG` and `ICD9_CODE` and count the occurrences.
2. **Sort**: Sort by the count to find the most common combinations of drug and procedure.
3. **Plot**: Use a horizontal bar chart to visualize the top 10 drug-procedure combinations.

```python
# To visualize the most common procedures by drug, we'll first group by 'DRUG' and 'ICD9_CODE'
# and count how many times each combination occurs. Then we'll visualize the top combinations.

# Group by 'DRUG' and 'ICD9_CODE' and count the occurrences
drug_procedure_counts = joined_data.groupby(['DRUG', 'ICD9_CODE']).size().reset_index(name='count')

# Sort by count to get the top combinations
top_drug_procedure_counts = drug_procedure_counts.sort_values(by='count', ascending=False).head(10)

# Now, let's plot this data to visualize the most common procedures by drug.
import matplotlib.pyplot as plt

# Plotting
plt.figure(figsize=(10, 6))
plt.barh(top_drug_procedure_counts['DRUG'] + ' (' + top_drug_procedure_counts['ICD9_CODE'].astype(str) + ')', to
plt.xlabel('Count')
plt.ylabel('Drug (ICD9 Code)')
plt.title('Top 10 Most Common Procedures by Drug')
plt.gca().invert_yaxis()
plt.tight_layout()

# Show the plot
plt.show()
```



Key Findings from the Visualization:

- **Potassium Chloride** and **NS (Normal Saline)** are the most frequently prescribed drugs associated with procedure ICD-9 codes (e.g., 3893, 966).
- **Furosemide**, **Insulin**, and **D5W** are also among the commonly prescribed drugs.
- Procedures coded as 3893 (likely related to specific medical interventions) frequently appear in combination with these drugs.