# ANALYSIS ON TANZANIAN WATER POINTS

---

## BUSINESS UNDERSTANDING

---

### OVERVIEW

---

Tanzania, the fifth most populous country in Africa, has experienced significant economic growth over the years, however despite considerable investment in water supply infrastructure from donor funding and the government, a significant proportion of its population remains without proper access to improved drinking water. With their Millennium Development Goals (MDGs) to halve the proportion of people that do not have access to water services by 2015, Tanzania only increased its access to improved drinking water from 54 percent to 56 percent [JMP, 2015 (https://www.unwater.org/sites/default/files/app/uploads/2020/04/WHOUNICEF-Joint-Monitoring-Program-for-Water-Supply-and-Sanitation-JMP-%e2%80%93-2015-Update_-ENG.pdf)](https://www.unwater.org/sites/default/files/app/uploads/2020/04/WHOUNICEF-Joint-Monitoring-Program-for-Water-Supply-and-Sanitation-JMP-%e2%80%93-2015-Update_-ENG.pdf). The country now faces a difficult task of meeting the Sustainable Development Goals (SDGs) to provide universal coverage of safe water by 2030.

Despite their efforts one persistent problem that has adversely affected the country's effort in increasing access to improved water services is the prevailing high levels of non-functionality or failures of its current water infrastructures and in particular, water points and while this issue is prevalent in Africa, evidence indicates that the problem of water point failures may be relatively more serious in Tanzania with some estimates putting the figure as high as 44 percent Banks & Furey, 2016 (https://www.researchgate.net/publication/312027512_What's_Working_Where_and_for_How_Lo

A holistic approach was used to determine the factors to be considered to be able to predict water point failure. These factors included age of the water point, technology used, the quantity and quality of the water as well as location and management of these water points. We also considered the population that use these water points, and the sources of the water. Using a variety of statistical methods, we seek to understand the impact of these factors on water point's failure and develop a model to predict the possibility of water pump failure with an accuracy of 80% and f1 score of 75%

# PROBLEM STATEMENT

> We have been tasked by World Bank Group together with the Government of Tanzania to seek a better understanding as to why water point failure is significantly high in Tanzania as well as a way to reliably predict when water pumps shall fail as they tackle the difficult task of meeting their 2030 MDG goals in Environmental Sustainability .

# OBJECTIVES

The research seeks to meet the following objectives:

> **1. Analyze the Impact of Age, Technology, and Investment on Water Point Failure**

> **2. Assess the Impact of Socioeconomic and Geographical Factors**

> **3. Develop a Predictive Model for Water Point Failure**

## DATA UNDERSTANDING

This research utilized data from DRIVEN DATA (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/page/23/) about waterpoints. The dataset was split into three CSV files:

- **Training set values**
- **Training set labels**
- **Test set values**

The training and test datasets contained similar columns, while the training set labels dataset included one column, which was the focus of the study.

## Column Information

The following columns were provided in the training and testing datasets:

- **amount_tsh**:Total static head (amount of water available to waterpoint)
- **date_recorded**: The date the row was entered
- **funder**: Who funded the well
- **gps_height**: Altitude of the well
- **installer**: Organization that installed the well
- **longitude**: GPS coordinate
- **latitude**: GPS coordinate
- **wpt_name**: Name of the waterpoint if there is one
- **num_private**: [Missing information]
- **basin**: Geographic water basin
- **subvillage**: Geographic location
- **region**: Geographic location
- **region_code**: Geographic location (coded)
- **district_code**: Geographic location (coded)
- **lga**: Geographic location
- **ward**: Geographic location
- **population**: Population around the well
- **public_meeting**: True/False
- **recorded_by**: Group entering this row of data
- **scheme_management**: Who operates the waterpoint
- **scheme_name**: Who operates the waterpoint
- **permit**: If the waterpoint is permitted
- **construction_year**: Year the waterpoint was constructed
- **extraction_type**: The kind of extraction the waterpoint uses
- **extraction_type_group**: The kind of extraction the waterpoint uses
- **extraction_type_class**: The kind of extraction the waterpoint uses
- **management**: How the waterpoint is managed
- **management_group**: How the waterpoint is managed
- **payment**: What the water costs
- **payment_type**: What the water costs
- **water_quality**: The quality of the water
- **quality_group**: The quality of the water
- **quantity**: The quantity of water
- **quantity_group**: The quantity of water
- **source**: The source of the water
- **source_type**: The source of the water
- **source_class**: The source of the water
- **waterpoint_type**: The kind of waterpoint

- **waterpoint_type_group**: The kind of waterpoint

## Labels Information

The labels in the training set labels contained one column, **status_group**. This column indicates the condition of the waterpoint with the following possible outcomes:

- **functional**: The waterpoint is operational and there are no repairs needed
- **functional needs repair**: The waterpoint is operational, but needs repairs
- **non functional**: The waterpoint is not operational

## DATA PREPARATION

---

The following steps in summary shall be followed in the data preparation stage in preparation for Modeling in later stages

#### 1. Data Loading

- Load the Datasets
- Inspect the Data

#### 2. Data Cleaning

- Validity Check
- Consistency Check
- Uniformity Check
- Completeness Check

#### 3. EXPLORATORY DATA ANALYSIS

- Understand Data Distribution
- Identify Relationships - Univariate and Bivariate Analysis
- Handle High Cardinality Columns

## DATA LOADING

The following was carried out

1. Loading the Datasets
2. Inspecting the Data

In [1]:
```
import os
import numpy as np
import pandas as pd
from itertools import combinations
import warnings
warnings.filterwarnings('ignore')

# Libraries for visualizations
import folium
import seaborn as sns
import plotly.express as px
from IPython.display import display, HTML
import matplotlib.pyplot as plt
%matplotlib inline

# libraries for Model Preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing   import StandardScaler, OneHotEncoder, LabelEncoder
from imblearn.over_sampling  import SMOTE

# Libraries for Modeling
from sklearn.linear_model    import LogisticRegression
from sklearn.naive_bayes     import MultinomialNB
from sklearn.tree            import DecisionTreeClassifier
from sklearn.neighbors       import KNeighborsClassifier
from sklearn.ensemble        import GradientBoostingClassifier,RandomForestClas
from xgboost                 import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV, cross_val_score
from sklearn.metrics         import accuracy_score, f1_score,make_scorer, confu
```

In [2]:

```
class DataLoader:
    def __init__(self):
        pass

    def read_data(self, file_path):
        _, file_ext = os.path.splitext(file_path)
        """
        Load data from a CSV, TSV, JSON or Excel file
        """
        if file_ext == '.csv':
            return pd.read_csv(file_path, index_col=None)

        elif file_ext == '.tsv':
            return pd.read_csv(file_path, sep='\t')

        elif file_ext == '.json':
            return pd.read_json(file_path)

        elif file_ext in ['.xls', '.xlsx']:
            return pd.read_excel(file_path)

        else:
            raise ValueError(f"Unsupported file format:")

class DataFrameMerger:
    def __init__(self):
        pass

    def merge_dataframes(self, df1, df2, on, how='inner'):
        """
        Merge two dataframes on specified columns with the specified method.
        """
        merged_df = pd.merge(df1, df2, on=on, how=how)
        return merged_df.sort_values(by=merged_df.columns.tolist())

class DataInfo:

    def __init__(self,df):
        self.df = df

    def info(self):
        """
        Displaying Relevant Information on the the Dataset Provided
        """
        # Counting no of rows
        print(f'\nTotal Rows : {self.df.shape[0]} \n' + '--'*10 )

        # Counting no of columns
        print(f'\nTotal Columns : {self.df.shape[1]} \n' + '--'*10)

        # Extracting column names
        column_name =  self.df.columns
        print(f'\nColumn Names\n' + '--'*10 +  f'\n{column_name} \n \n')

        # Data type info
        print(f'Data Summary\n' + '--'*10)
```

```
        data_summary = self.df.info()

        # Total null values by each categories
        null_values = self.df.isnull().sum()
        print(f'\nNull values\n' + '--'*10 + f'\n{null_values} \n \n')

        # Descriptive statistics
        describe =  self.df.describe()
        print(f'\nDescriptive Statistics\n' + '--'*10 )
        display(describe)

        #Display the dataset
        print(f'\nDataset Overview\n'+ '--'*10)
        return self.df.head()
```

In [3]:
```
#Instantiate the loader class
data_loader = DataLoader()

# Loading the datasets
train_data=data_loader.read_data("data/train.csv")
train_labels=data_loader.read_data("data/train-labels.csv")
test_data=data_loader.read_data("data/test.csv")

# Instantiate the DF merger class
merger=DataFrameMerger()

# Merge the train data provided
data=merger.merge_dataframes(train_data, train_labels, on="id")

print(f'\nTimeLine of Recorded Data\n' + '--'*10 )
print(f"From:",data['date_recorded'].min(), "To:",  data['date_recorded'].max(
print(f'--'*10 )

# Instantiate the Information class
information=DataInfo(data)

# Getting the info on the training DF
information.info()
```

```
TimeLine of Recorded Data
--------------------
From: 2002-10-14 To: 2013-12-03
--------------------

Total Rows : 59400
--------------------

Total Columns : 41
--------------------

Column Names
--------------------
Index(['id', 'amount_tsh', 'date_recorded', 'funder', 'gps_height',
       'installer', 'longitude', 'latitude', 'wpt_name', 'num_private',
       'basin', 'subvillage', 'region', 'region_code', 'district_code', 'lg
a',
       'ward', 'population', 'public_meeting', 'recorded_by',
       'scheme_management', 'scheme_name', 'permit', 'construction_year',
       'extraction_type', 'extraction_type_group', 'extraction_type_class',
       'management', 'management_group', 'payment', 'payment_type',
       'water_quality', 'quality_group', 'quantity', 'quantity_group',
       'source', 'source_type', 'source_class', 'waterpoint_type',
       'waterpoint_type_group', 'status_group'],
      dtype='object')


Data Summary
--------------------
<class 'pandas.core.frame.DataFrame'>
Int64Index: 59400 entries, 9410 to 39131
Data columns (total 41 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   id                     59400 non-null  int64  
 1   amount_tsh             59400 non-null  float64
 2   date_recorded          59400 non-null  object 
 3   funder                 55765 non-null  object 
 4   gps_height             59400 non-null  int64  
 5   installer              55745 non-null  object 
 6   longitude              59400 non-null  float64
 7   latitude               59400 non-null  float64
 8   wpt_name               59400 non-null  object 
 9   num_private            59400 non-null  int64  
 10  basin                  59400 non-null  object 
 11  subvillage             59029 non-null  object 
 12  region                 59400 non-null  object 
 13  region_code            59400 non-null  int64  
 14  district_code          59400 non-null  int64  
 15  lga                    59400 non-null  object 
 16  ward                   59400 non-null  object 
 17  population             59400 non-null  int64  
 18  public_meeting         56066 non-null  object 
 19  recorded_by            59400 non-null  object 
 20  scheme_management      55523 non-null  object 
 21  scheme_name            31234 non-null  object 
```

```
 22  permit                 56344 non-null  object
 23  construction_year      59400 non-null  int64
 24  extraction_type        59400 non-null  object
 25  extraction_type_group  59400 non-null  object
 26  extraction_type_class  59400 non-null  object
 27  management             59400 non-null  object
 28  management_group       59400 non-null  object
 29  payment                59400 non-null  object
 30  payment_type           59400 non-null  object
 31  water_quality          59400 non-null  object
 32  quality_group          59400 non-null  object
 33  quantity               59400 non-null  object
 34  quantity_group         59400 non-null  object
 35  source                 59400 non-null  object
 36  source_type            59400 non-null  object
 37  source_class           59400 non-null  object
 38  waterpoint_type        59400 non-null  object
 39  waterpoint_type_group  59400 non-null  object
 40  status_group           59400 non-null  object
dtypes: float64(3), int64(7), object(31)
memory usage: 19.0+ MB

Null values
--------------------
id                           0
amount_tsh                   0
date_recorded                0
funder                    3635
gps_height                   0
installer                 3655
longitude                    0
latitude                     0
wpt_name                     0
num_private                  0
basin                        0
subvillage                 371
region                       0
region_code                  0
district_code                0
lga                          0
ward                         0
population                   0
public_meeting            3334
recorded_by                  0
scheme_management         3877
scheme_name              28166
permit                    3056
construction_year            0
extraction_type              0
extraction_type_group        0
extraction_type_class        0
management                   0
management_group             0
payment                      0
payment_type                 0
water_quality                0
quality_group                0
```

```
quantity                  0
quantity_group            0
source                    0
source_type               0
source_class              0
waterpoint_type           0
waterpoint_type_group     0
status_group              0
dtype: int64



Descriptive Statistics
--------------------
```

| | id | amount_tsh | gps_height | longitude | latitude | num_private | r |
|---|---|---|---|---|---|---|---|
| **count** | 59400.000000 | 59400.000000 | 59400.000000 | 59400.000000 | 5.940000e+04 | 59400.000000 | 59 |
| **mean** | 37115.131768 | 317.650385 | 668.297239 | 34.077427 | -5.706033e+00 | 0.474141 | |
| **std** | 21453.128371 | 2997.574558 | 693.116350 | 6.567432 | 2.946019e+00 | 12.236230 | |
| **min** | 0.000000 | 0.000000 | -90.000000 | 0.000000 | -1.164944e+01 | 0.000000 | |
| **25%** | 18519.750000 | 0.000000 | 0.000000 | 33.090347 | -8.540621e+00 | 0.000000 | |
| **50%** | 37061.500000 | 0.000000 | 369.000000 | 34.908743 | -5.021597e+00 | 0.000000 | |
| **75%** | 55656.500000 | 20.000000 | 1319.250000 | 37.178387 | -3.326156e+00 | 0.000000 | |
| **max** | 74247.000000 | 350000.000000 | 2770.000000 | 40.345193 | -2.000000e-08 | 1776.000000 | |

```
Dataset Overview
--------------------
```

Out[3]:

| | id | amount_tsh | date_recorded | funder | gps_height | installer | longitude | latitude | wpt_ |
|---|---|---|---|---|---|---|---|---|---|
| **9410** | 0 | 0.0 | 2012-11-13 | Tasaf | 0 | TASAF | 33.125828 | -5.118154 | M |
| **18428** | 1 | 0.0 | 2011-03-05 | Shipo | 1978 | SHIPO | 34.770717 | -9.395642 | |
| **12119** | 2 | 0.0 | 2011-03-27 | Lvia | 0 | LVIA | 36.115056 | -6.279268 | Bo |
| **10629** | 3 | 10.0 | 2013-06-03 | Germany Republi | 1639 | CES | 37.147432 | -3.187555 | Na |
| **2343** | 4 | 0.0 | 2011-03-22 | Cmsr | 0 | CMSR | 36.164893 | -6.099289 | E: |

5 rows × 41 columns

**Initial Observations:**

- There seem to be unique identifiers in the dataset that need to be investigated eg "id"
- Some columns seem to carry repeated data based on data understanding and column names and should be investigated
- Numeric features seem to be mainly location identifiers

## DATA CLEANING

Data cleaning shall be carried out in the following steps:

1. Validity Check
2. Consistency Check
3. Uniformity Check
4. Completeness Check

#### VALIDITY CHECK

- Check for unique identifiers i.e. distinct elements in a column that are equal to the length of the column value counts
- Check for constant columns i.e. total distinct elements in a column is equal to 1

In [4]:
```
class Validity:
    def __init__(self,df):
        self.df=df
        self.unique_identifier = []

    def find_unique_identifiers(self):
        """
        Identify unique identifiers and constant columns and drop them
        """
        self.df= self.df.copy()
        columns = self.df.columns

        for column in columns:
            if self.df[column].nunique() == len(self.df[column]):
                self.unique_identifier.append(column)
                print(f" Unique Identifier Columns:", column)
            elif self.df[column].nunique() == 1:
                self.unique_identifier.append(column)
                print(f" Constant Columns:", column)

        self.df = self.df.drop(columns=self.unique_identifier)
        display(self.df)

        return self.df
```

In [5]:
```
# Checking for Validity on datset

# Instantiate the Validity check class
valid= Validity(data)

# Validate the training dataset
train_data=valid.find_unique_identifiers()
```

```
Unique Identifier Columns: id
Constant Columns: recorded_by
```

| | amount_tsh | date_recorded | funder | gps_height | installer | longitude | latitude | wpt_na |
|---|---|---|---|---|---|---|---|---|
| **9410** | 0.0 | 2012-11-13 | Tasaf | 0 | TASAF | 33.125828 | -5.118154 | Mra |
| **18428** | 0.0 | 2011-03-05 | Shipo | 1978 | SHIPO | 34.770717 | -9.395642 | n |
| **12119** | 0.0 | 2011-03-27 | Lvia | 0 | LVIA | 36.115056 | -6.279268 | Bomb |
| **10629** | 10.0 | 2013-06-03 | Germany Republi | 1639 | CES | 37.147432 | -3.187555 | Are Namb |
| **2343** | 0.0 | 2011-03-22 | Cmsr | 0 | CMSR | 36.164893 | -6.099289 | Ezel |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **15137** | 0.0 | 2013-03-22 | World Vision | 1183 | World vision | 37.007726 | -3.280868 | Upe Prim Sch |
| **8667** | 0.0 | 2011-04-12 | Danida | 0 | DANIDA | 33.724987 | -8.940758 | K Mvu |
| **22584** | 0.0 | 2012-11-13 | Ministry Of Water | 1188 | Hesawa | 33.963539 | -1.429477 | K Wamb Ms |
| **108** | 50.0 | 2011-03-07 | Ruthe | 1428 | Ruthe | 35.630481 | -7.710549 | n |
| **39131** | 50.0 | 2013-02-16 | Mission | 965 | DWE | 35.432998 | -10.639270 | K Mapu |

59400 rows × 39 columns

**Observations:**

- 1 column had unique identifiers i.e. "id" column
- 1 column had constant column identifiers i.e. "recorded_by" column
- Dataset now has 59,400 rows and 39 columns

#### CONSISTENCY CHECK

- Duplicates checked in rows and dropped them
- Duplicates Checked in columns and dropped them

In [6]:
```
class Consistency:
    def __init__(self,df):
        self.df=df
        self.to_drop = []

    def duplicated_rows(self):
        """
        Displaying the duplicated rows for visual assesment
        """
        df_sorted = self.df.sort_values(by=self.df.columns.tolist())

        # Find duplicated rows
        duplicates = df_sorted[df_sorted.duplicated(keep=False)]

        # Display the duplicated rows as HTML
        return display(HTML(duplicates.to_html()))

    def drop_duplicated_rows(self,rows=None):
        """
        Dropping confirmed duplicated rows
        """
        self.df.drop_duplicates(subset=rows, keep= "first", inplace= True)
        display(self.df.shape)
        return self.df

    def find_duplicated_columns(self):
        """
         Displaying the duplicated columns for visual assesment
        """
        duplicated_columns = []
        columns = self.df.columns
        for i, col1 in enumerate(columns):
            for col2 in columns[i + 1:]:
                if self.df[col1].equals(self.df[col2]):
                    duplicated_columns.append((col1, col2))
        for pair in duplicated_columns:
                display(pair[0],(self.df[pair[0]].value_counts()))
                display(pair[1],(self.df[pair[1]].value_counts()))
                self.to_drop.append(pair[1])
        return display(f"Duplicated Columns:",duplicated_columns)

    def drop_duplicate_columns(self, columns=None):
        """
        Dropping confirmed duplicated columns
        """
        if columns is None:
            columns = self.to_drop
        self.df = self.df.drop(columns=columns)
        display(self.df.shape)
        return self.df
```

In [7]:
```
# Checking for Consistency in dataset

# Instantiate the Consistency check class
const= Consistency(train_data)

# Visually Checking for duplicated rows
duplicates= const.duplicated_rows()

# Dropping the duplicated rows
train_data=const.drop_duplicated_rows()
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **34310** | 0.0 | 2011-07-19 | Plan International | 0 | Plan Internationa | 0.00000 | -2.000000e |
| **13355** | 0.0 | 2011-07-19 | Plan International | 0 | Plan Internationa | 0.00000 | -2.000000e |
| **37202** | 0.0 | 2011-07-26 | Hesawa | 0 | DWE | 0.00000 | -2.000000e |
| **8460** | 0.0 | 2011-07-26 | Hesawa | 0 | DWE | 0.00000 | -2.000000e |
| **25300** | 0.0 | 2011-07-27 | Hesawa | 0 | DWE | 0.00000 | -2.000000e |
| **31558** | 0.0 | 2011-07-27 | Hesawa | 0 | DWE | 0.00000 | -2.000000e |
| **7907** | 0.0 | 2011-07-27 | Hesawa | 0 | DWE | 0.00000 | -2.000000e |
| **51183** | 0.0 | 2011-07-28 | Government Of Tanzania | 0 | Government | 0.00000 | -2.000000e |

**Observations:**

---

- 36 rows are confirmed duplicates and have been dropped
- Dataset now has 59,364 rows and 39 columns

In [8]:
```
# Checking for Consistency in columns dataset

# Visually Checking for duplicated columns
const.find_duplicated_columns()

# Dropping the duplicated columns
train_data=const.drop_duplicate_columns()
```

```
'quantity'

enough          33165
insufficient    15119
dry              6243
seasonal         4048
unknown           789
Name: quantity, dtype: int64

'quantity_group'

enough          33165
insufficient    15119
dry              6243
seasonal         4048
unknown           789
Name: quantity_group, dtype: int64

'Duplicated Columns:'

[('quantity', 'quantity_group')]

(59364, 38)
```

#### Observations:

---

- 1 column visually confirmed to be a duplicate i.e. "quantity_group" and dropped
- Dataset now has 59,364 rows and 38 columns

---

#### UNIFORMITY CHECK

---

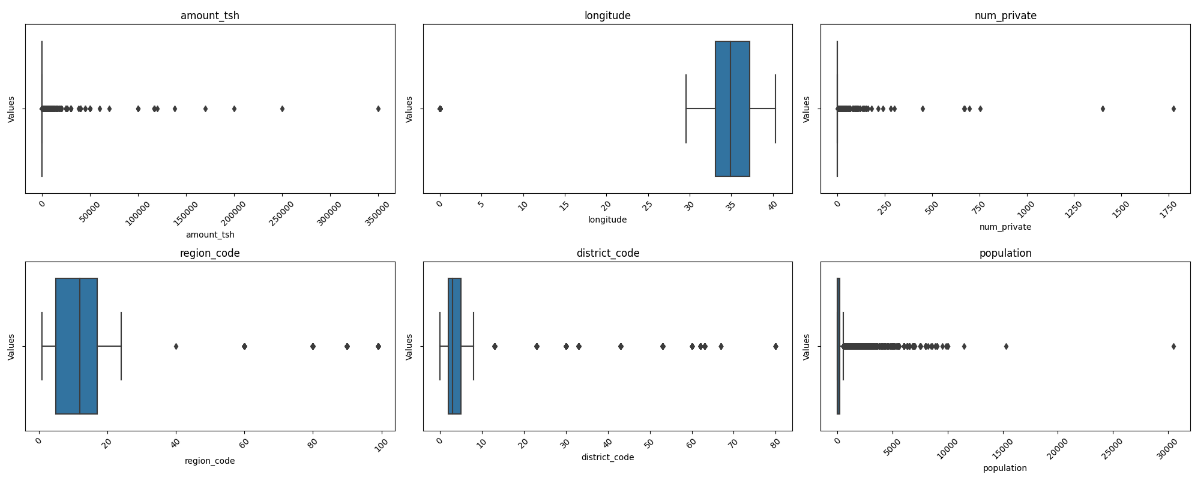
- Assessing Data Distributions i.e. outliers
- Checking Data Types

In [9]:
```
class Uniformity:
    def __init__(self,df):
        self.df=df
        self.categorical_columns = []
        self.numerical_columns =[]

    def column_seperation(self):
        """
        Seperate the columns into Categorical and Numerical Columns
        """
        for col in self.df.columns:
            if self.df[col].dtype == object:
                self.categorical_columns.append(col)
            else:
                self.numerical_columns.append(col)

        return self.numerical_columns, self.categorical_columns

    def detect_outliers_iqr(self):
        """
        Detect outliers in numerical columns using the IQR method.
        """
        outlier_columns = []

        for column in self.numerical_columns:
            Q1 = self.df[column].quantile(0.25)
            Q3 = self.df[column].quantile(0.75)
            IQR = Q3 - Q1
            lower_bound = Q1 - 1.5 * IQR
            upper_bound = Q3 + 1.5 * IQR

            outlier_indices = self.df[(self.df[column] < lower_bound) | (self.

            if outlier_indices:
                outlier_columns.append(column)

        return outlier_columns

    def plot_outliers(self, outlier_columns):
        """
        Plot boxplots for the columns that have outliers using Seaborn.
        """
        num_rows = (len(outlier_columns) + 2) // 3
        num_cols = min(len(outlier_columns), 3)
        fig, axes = plt.subplots(nrows=num_rows, ncols=num_cols, figsize=(20,
        axes = axes.flatten()
        for i, column in enumerate(outlier_columns):
            sns.boxplot(x=self.df[column], ax=axes[i])
            axes[i].set_xlabel(column)
            axes[i].set_ylabel('Values')
            axes[i].set_title(f'{column}')
            axes[i].tick_params(axis='x', rotation=45)

        # Adjust layout to prevent overlapping
        plt.tight_layout()

        # Show the plots
```

```
        plt.show()


    def convert_column_dtype(self, column_name, dtype):
        """
        Convert the data type of a column in a DataFrame.
        """
        self.df[column_name] = self.df[column_name].astype(dtype).astype(dtype)

        return self.df
```

In [10]:
```
#Instantiate the Uniformity class
uniform = Uniformity(train_data)

# Separating the columns into numerical and categorical
numerical_columns, categorical_columns=uniform.column_seperation()
print(f"Numerical Columns:",numerical_columns)

# Looking for Outliers in the numerical_columns
outlier_columns = uniform.detect_outliers_iqr()
print("Columns with outliers detected using IQR method:", outlier_columns)

# Plotting the columns with outliers
uniform.plot_outliers(outlier_columns)
```

```
Numerical Columns: ['amount_tsh', 'gps_height', 'longitude', 'latitude', 'num
_private', 'region_code', 'district_code', 'population', 'construction_year']
Columns with outliers detected using IQR method: ['amount_tsh', 'longitude',
'num_private', 'region_code', 'district_code', 'population']
```




### Observation:

- Region_code and District code are geographical code identifiers and shall be dropped since these codes regularly change and are not the current one based on their associated region columns [reference (https://www.iso.org/obp/ui/#iso:code:3166:TZ)]

In [11]:
```
#Further analysis of the Longitude and Latitude Columns
# check to see if latitude outlier is the same for both regions
mwanza=train_data[train_data["region"]=="Mwanza"]["latitude"].max()
shinyanga=train_data[train_data["region"]=="Shinyanga"]["latitude"].max()
mwanza==shinyanga
```

Out[11]:
```
True
```

In [12]:
```
# Reassigning it as maxim
maxim =mwanza

# Check the outlier value in maxim variable for longitude based on the region c
train_data[train_data["latitude"]==maxim]["region"].value_counts()
```

Out[12]:
```
Shinyanga    1003
Mwanza        774
Name: region, dtype: int64
```

In [13]:
```
# Check the outlier value 0 for longitude based on the region column
train_data[train_data.longitude == 0]["region"].value_counts()
```

Out[13]:
```
Shinyanga    1003
Mwanza        774
Name: region, dtype: int64
```

In [14]:
```
# Function to replace all outlier values with the median for the longitude colu
def replace_zero_longitudes(df, longitude_col='longitude', latitude_col="latitu
    """
    Replace longitude and Latitude Outlier values with the median longitude of
    """
    regions = df[region_col].unique()
    median_longitudes = {}
    median_latitudes={}

    for region in regions:
        median_longitude = df[df[region_col] == region][longitude_col].median()
        median_latitude=df[df[region_col] == region][latitude_col].median()
        median_longitudes[region] = median_longitude
        median_latitudes[region] = median_latitude

        df.loc[(df[region_col] == region) & (df[longitude_col] == 0), longitude
        df.loc[(df[region_col] == region) & (df[latitude_col] == maxim), latitu


    return df, median_longitudes, median_latitudes

train_data, median_longitudes, median_latitudes = replace_zero_longitudes(trai
print("\nMedian Longitude values used:")
print("----------------------------")
print("Shinyanga:",median_longitudes["Shinyanga"])
print("Mwanza:   ",median_longitudes["Mwanza"])
print("\nMedian Latitude values used:")
print("----------------------------")
print("Shinyanga:",median_latitudes["Shinyanga"])
print("Mwanza:   ",median_latitudes["Mwanza"])
train_data
```

```
Median Longitude values used:
----------------------------
Shinyanga: 33.04787608
Mwanza:    32.99787276

Median Latitude values used:
----------------------------
Shinyanga: -3.3524871149999997
Mwanza:    -2.52297834
```

Out[14]:

| | amount_tsh | date_recorded | funder | gps_height | installer | longitude | latitude | wpt_na |
|---|---|---|---|---|---|---|---|---|
| **9410** | 0.0 | 2012-11-13 | Tasaf | 0 | TASAF | 33.125828 | -5.118154 | Mra |
| **18428** | 0.0 | 2011-03-05 | Shipo | 1978 | SHIPO | 34.770717 | -9.395642 | n |
| **12119** | 0.0 | 2011-03-27 | Lvia | 0 | LVIA | 36.115056 | -6.279268 | Bomb |
| **10629** | 10.0 | 2013-06-03 | Germany Republi | 1639 | CES | 37.147432 | -3.187555 | Are Namb |
| **2343** | 0.0 | 2011-03-22 | Cmsr | 0 | CMSR | 36.164893 | -6.099289 | Ezel |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **15137** | 0.0 | 2013-03-22 | World Vision | 1183 | World vision | 37.007726 | -3.280868 | Upe Prim Sch |
| **8667** | 0.0 | 2011-04-12 | Danida | 0 | DANIDA | 33.724987 | -8.940758 | K Mvu |
| **22584** | 0.0 | 2012-11-13 | Ministry Of Water | 1188 | Hesawa | 33.963539 | -1.429477 | K Wamb Ms |
| **108** | 50.0 | 2011-03-07 | Ruthe | 1428 | Ruthe | 35.630481 | -7.710549 | n |
| **39131** | 50.0 | 2013-02-16 | Mission | 965 | DWE | 35.432998 | -10.639270 | K Mapu |

59364 rows × 38 columns

### Observations:

---

- It was evident that the longitude and latitude columns for Mwanza and Shinyanga had been given incorrect values i.e. outlier positions.
- The longitudes and latitudes have been replaced by the median of the regions and is a suitable replacement and can be confirmed from [TZ_Cities_db (https://simplemaps.com/data/tz-cities)](https://simplemaps.com/data/tz-cities)
- Median was used as it is least affected by the outlier values. It would also give the most probable accurate value
- Dataset has 59,364 rows and 38 columns

In [15]:
```
# Convert the list datatype to bool and return info on the data
list_to_bool=["permit", "public_meeting"]
uniform.convert_column_dtype(list_to_bool, bool).info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 59364 entries, 9410 to 39131
Data columns (total 38 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   amount_tsh             59364 non-null  float64
 1   date_recorded          59364 non-null  object 
 2   funder                 55729 non-null  object 
 3   gps_height             59364 non-null  int64  
 4   installer              55709 non-null  object 
 5   longitude              59364 non-null  float64
 6   latitude               59364 non-null  float64
 7   wpt_name               59364 non-null  object 
 8   num_private            59364 non-null  int64  
 9   basin                  59364 non-null  object 
 10  subvillage             58993 non-null  object 
 11  region                 59364 non-null  object 
 12  region_code            59364 non-null  int64  
 13  district_code          59364 non-null  int64  
 14  lga                    59364 non-null  object 
 15  ward                   59364 non-null  object 
 16  population             59364 non-null  int64  
 17  public_meeting         59364 non-null  bool   
 18  scheme_management      55487 non-null  object 
 19  scheme_name            31225 non-null  object 
 20  permit                 59364 non-null  bool   
 21  construction_year      59364 non-null  int64  
 22  extraction_type        59364 non-null  object 
 23  extraction_type_group  59364 non-null  object 
 24  extraction_type_class  59364 non-null  object 
 25  management             59364 non-null  object 
 26  management_group       59364 non-null  object 
 27  payment                59364 non-null  object 
 28  payment_type           59364 non-null  object 
 29  water_quality          59364 non-null  object 
 30  quality_group          59364 non-null  object 
 31  quantity               59364 non-null  object 
 32  source                 59364 non-null  object 
 33  source_type            59364 non-null  object 
 34  source_class           59364 non-null  object 
 35  waterpoint_type        59364 non-null  object 
 36  waterpoint_type_group  59364 non-null  object 
 37  status_group           59364 non-null  object 
dtypes: bool(2), float64(3), int64(6), object(27)
memory usage: 16.9+ MB
```

### Observation:

- Permit and public meeting have now been accurately classified as boolean types
- All other datatypes shall be used as is

**COMPLETENESS CHECK**

- Check for similarity in columns
- Check for Null values in all rows

In [16]:
```
class Completeness:
    def __init__(self,df):
        self.df=df

    def similarity(self,threshold=0.2):
        """
        Identifying columns that have similarity in their value counts
        with a low threshold to avoid missing similarities
        """
        similar_columns = []

        # Calculate the similarity between each pair of columns
        for col1, col2 in combinations(self.df.columns, 2):
            set1, set2 = set(self.df[col1]), set(self.df[col2])
            intersection = len(set1 & set2)
            union = len(set1 | set2)
            similarity = intersection / union if union != 0 else 0

        # Check if similarity exceeds threshold
            if similarity > threshold:
                similar_columns.append((col1, col2))

        return similar_columns

    def similar_columns(self,similar_columns):
        """
        Visually representing these similar columns for inspection
        """
        for col1, col2 in similar_columns:
            print(f"Value counts for columns '{col1}' and '{col2}':")
            print("\nColumn '{}' value counts:".format(col1))
            print(self.df[col1].value_counts())
            print("\nColumn '{}' value counts:".format(col2))
            print(self.df[col2].value_counts())
            print("\n")

    def null_values(self):
        """
        Identify Null values in dataset as value count and percentage
        """
        # Get features with null values
        null_features = self.df.columns[self.df.isnull().any()].tolist()

        # Calculate the number of missing values for each feature
        null_counts = self.df[null_features].isnull().sum()

        # Calculate the percentage of missing data for each feature
        null_percentages = self.df[null_features].isnull().mean() * 100

        # Create a DataFrame to display the results
        null_info = pd.DataFrame({
                                'Column Names': null_features,
                                'Missing Values': null_counts,
                                'Percentage Missing': null_percentages
        }).reset_index(drop=True)

        return null_info
```

```
    def handle_missing_values(self):
        """
        Handle missing values in the DataFrame.
        """
        null_info = self.null_values()

        # Apply conditions for handling missing values
        for index, row in null_info.iterrows():
            if row['Percentage Missing'] < 5:
                # Drop rows with missing values
                self.df.dropna(subset=[row['Column Names']], inplace= True)
            elif 5 <= row['Percentage Missing'] <= 10:
                # Replace missing values with "Unknown"
                self.df[row['Column Names']].fillna("Unknown", inplace=True)
            else:
                # More than 10% missing, mark column for dropping
                print(f"Column '{row['Column Names']}' has more than 10% missi

        return self.df
```

In [17]:

```
# Checking for completeness in the dataset

# Instantiate the Completeness check class
comp= Completeness(train_data)

# Find columns that have similar distinct elements using a small threshold of 0
similar_columns=comp.similarity(threshold=0.2)
display(similar_columns)

# Visualling comparing similar columns
comp.similar_columns(similar_columns)
```

```
[('gps_height', 'population'),
 ('num_private', 'region_code'),
 ('region_code', 'district_code'),
 ('public_meeting', 'permit'),
 ('extraction_type', 'extraction_type_group'),
 ('extraction_type_group', 'extraction_type_class'),
 ('management', 'management_group'),
 ('payment', 'payment_type'),
 ('water_quality', 'quality_group'),
 ('source', 'source_type'),
 ('waterpoint_type', 'waterpoint_type_group')]

Value counts for columns 'gps_height' and 'population':

Column 'gps_height' value counts:
 0       20402
-15         60
-13         55
-16         55
 1290       52
```

**Observations:**

- After visually inspecting the column pairs, some columns bare similarity (i.e. a column is a subset of another column) to each other and were dropped to avoid repeated data points.
- The columns are:

```
                'waterpoint_type_group', 'source','water
_quality', 'payment',
                'management', 'extraction_type_group',
'extraction_type'
```

In [18]:
```
# Inspecting for null values
comp.null_values()
```

Out[18]:

| | Column Names | Missing Values | Percentage Missing |
|---|---|---|---|
| **0** | funder | 3635 | 6.123240 |
| **1** | installer | 3655 | 6.156930 |
| **2** | subvillage | 371 | 0.624958 |
| **3** | scheme_management | 3877 | 6.530894 |
| **4** | scheme_name | 28139 | 47.400782 |

In [19]:
```
# Handling with null values
comp.handle_missing_values()
```

```
Column 'scheme_name' has more than 10% missing values.
```

Out[19]:

| | amount_tsh | date_recorded | funder | gps_height | installer | longitude | latitude | wpt_na |
|---|---|---|---|---|---|---|---|---|
| **9410** | 0.0 | 2012-11-13 | Tasaf | 0 | TASAF | 33.125828 | -5.118154 | Mra |
| **18428** | 0.0 | 2011-03-05 | Shipo | 1978 | SHIPO | 34.770717 | -9.395642 | n |
| **12119** | 0.0 | 2011-03-27 | Lvia | 0 | LVIA | 36.115056 | -6.279268 | Bomb |
| **10629** | 10.0 | 2013-06-03 | Germany Republi | 1639 | CES | 37.147432 | -3.187555 | Are Namb |
| **2343** | 0.0 | 2011-03-22 | Cmsr | 0 | CMSR | 36.164893 | -6.099289 | Ezel |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **15137** | 0.0 | 2013-03-22 | World Vision | 1183 | World vision | 37.007726 | -3.280868 | Uper Prim Sch |
| **8667** | 0.0 | 2011-04-12 | Danida | 0 | DANIDA | 33.724987 | -8.940758 | K Mvu |
| **22584** | 0.0 | 2012-11-13 | Ministry Of Water | 1188 | Hesawa | 33.963539 | -1.429477 | K Wamb Ms |
| **108** | 50.0 | 2011-03-07 | Ruthe | 1428 | Ruthe | 35.630481 | -7.710549 | n |
| **39131** | 50.0 | 2013-02-16 | Mission | 965 | DWE | 35.432998 | -10.639270 | K Mapu |

58993 rows × 38 columns

**Observation:**

---

- The null values have been handled according to their percentage of missing values

> - Less than 5% values have had those rows dropped i.e. subvillage column(371 rows)
> - Between 5-10% have had their Nan Values replaced by Unknown i.e. scheme_management,funder and installer columns
> - Greater than 10% will be added to a features drop list to be dropped i,e, scheme_name column

In [20]:
```
# After visual inspection, generating list of similar columns to drop
features_to_drop=['waterpoint_type', 'source','water_quality', 'payment',
                  'management', 'extraction_type_group', 'extraction_type',
                  'scheme_name',"region_code","district_code","num_private",
                 ]

# recalling drop duplicates from the consistency class
clean_data=const.drop_duplicate_columns(features_to_drop)
```

```
(58993, 27)
```

**Observation:**

---

- The dataset now has 58,993 columns and 27 rows
- The dataset still has columns with very high cardinality. This shall be taken care of before modelling as they are still important for bivariate analysis

---

# EXPLORATORY DATA ANALYSIS

---

In this section consideration was given to:

> - Feature Engineering
> - Univariate Analysis
> - Bivariate Analysis
> - Handling High Cardinality Columns

---

## FEATURE ENGINEERING

---

Before continuining with the EDA process, the following needs to be engineered

> - Target variable shall be converted to a binary classification problem.
> - Age of the water points is of vital importance and so feature engineering shall also be carried out to ascertain the age of these points and only those with a viable age were used

In [21]:
```
# Define the names to be changed and their new values
name_changes = {
    "functional needs repair": 'functional',
}

# Replace the names in the status_group column
clean_data['status_group'] = clean_data['status_group'].replace(name_changes)
clean_data.status_group.value_counts()
```

Out[21]:
```
functional        36345
non functional    22648
Name: status_group, dtype: int64
```

In [22]:
```
# Function to calculate the age of the water points
def calculate_age(df, date_col='date_recorded', year_col='construction_year', n
    """
    Processes the date column to extract the year and calculates the age based
    """
    # Extract the year from the date column and replace the column with the yea
    df[date_col] = pd.to_datetime(df[date_col]).dt.year

    # Calculate the age and create a new column
    df[new_col] = df[date_col]-df[year_col]

    return df

# Process the DataFrame and calculate age
clean_data = calculate_age(clean_data)

#Filter the dataset to only have data with relevant age
aged_data = clean_data[(clean_data["age"] >= 0) & (clean_data["age"] <= 100)]
print(f"Shape of the cleaned_data:",clean_data.shape)
print(f"Shape of the aged_data:",aged_data.shape)
```

```
Shape of the cleaned_data: (58993, 28)
Shape of the aged_data: (38672, 28)
```

In [23]:
```
# Value_counts to see distribution of data without construction year
deleted_data = clean_data[ (clean_data["age"] >= 100)]
deleted_data["region"].value_counts()
```

Out[23]:
```
Shinyanga        4816
Mbeya            4639
Kagera           3315
Mwanza           2714
Tabora           1959
Dodoma           1840
Iringa            372
Pwani             176
Lindi             113
Tanga              94
Mtwara             87
Arusha             51
Kilimanjaro        35
Morogoro           18
Dar es Salaam      17
Mara               17
Ruvuma             16
Manyara            13
Kigoma             10
Singida             8
Rukwa               2
Name: region, dtype: int64
```

**Observations:**

---

> - The dataset target variable now has 2 distinct elements.
> - Being a vital predictor, age is missing several rows of construction year data and this reduced the size of the dataset to 38,672 rows and 28 columns

**NOTE:** As evident from the counts, the missing construction years are mainly found in the following areas i.e. Shinyanga(4816), Mbeya(4639), Kagera(3315), Mwanza(2714), Tabora(1959), Dodoma(1840), Iringa(372), Pwani(176) and Lindi(113). with other all other areas missing less than 100 entries each. Analysis carried out with dataset with available age only.

---

**UNIVARIATE ANALYSIS**

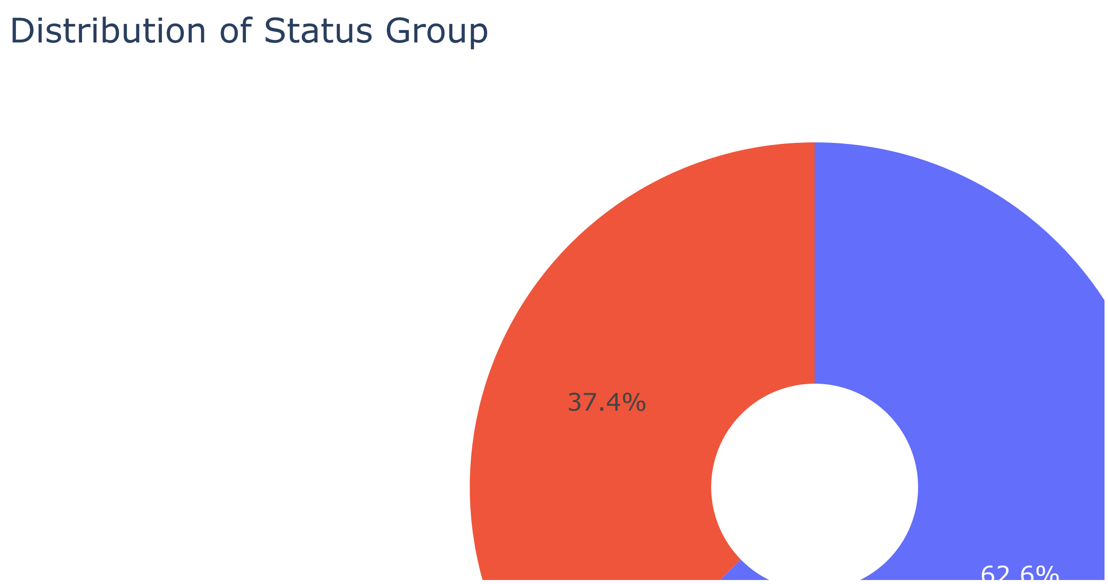
---

For this analysis we were interested in the following

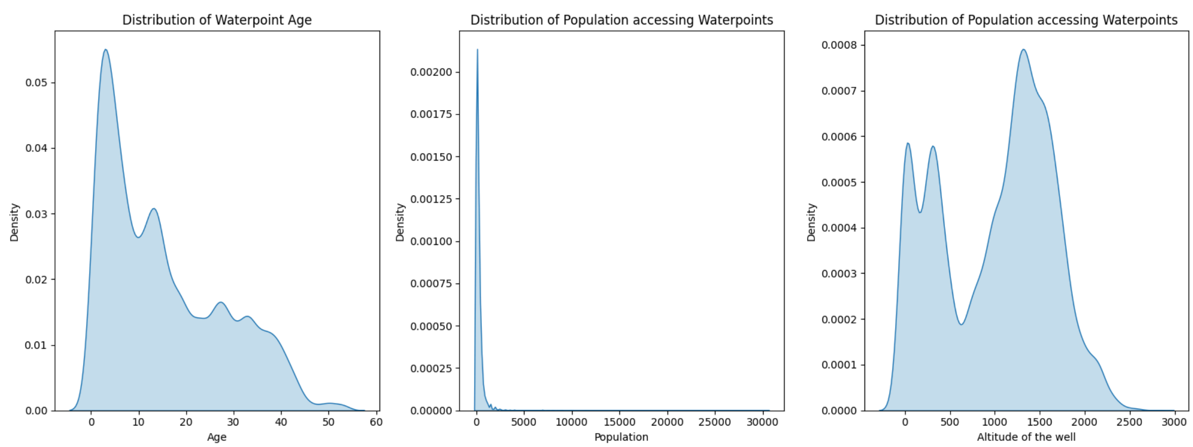
- Target Variable Distribution to determine class imbalance
- Age and population distribution of the dataset

In [24]:
```
# Get the value counts of the 'status_group' column
value_counts = aged_data['status_group'].value_counts()

# Create an interactive pie chart
fig = px.pie(
            value_counts,
            values=value_counts.values,
            names=value_counts.index,
#             color_discrete_sequence=["cyan","pink","yellow"],
            title='Distribution of Status Group',
            hole=0.3
            )

display(value_counts)
fig.show()
```

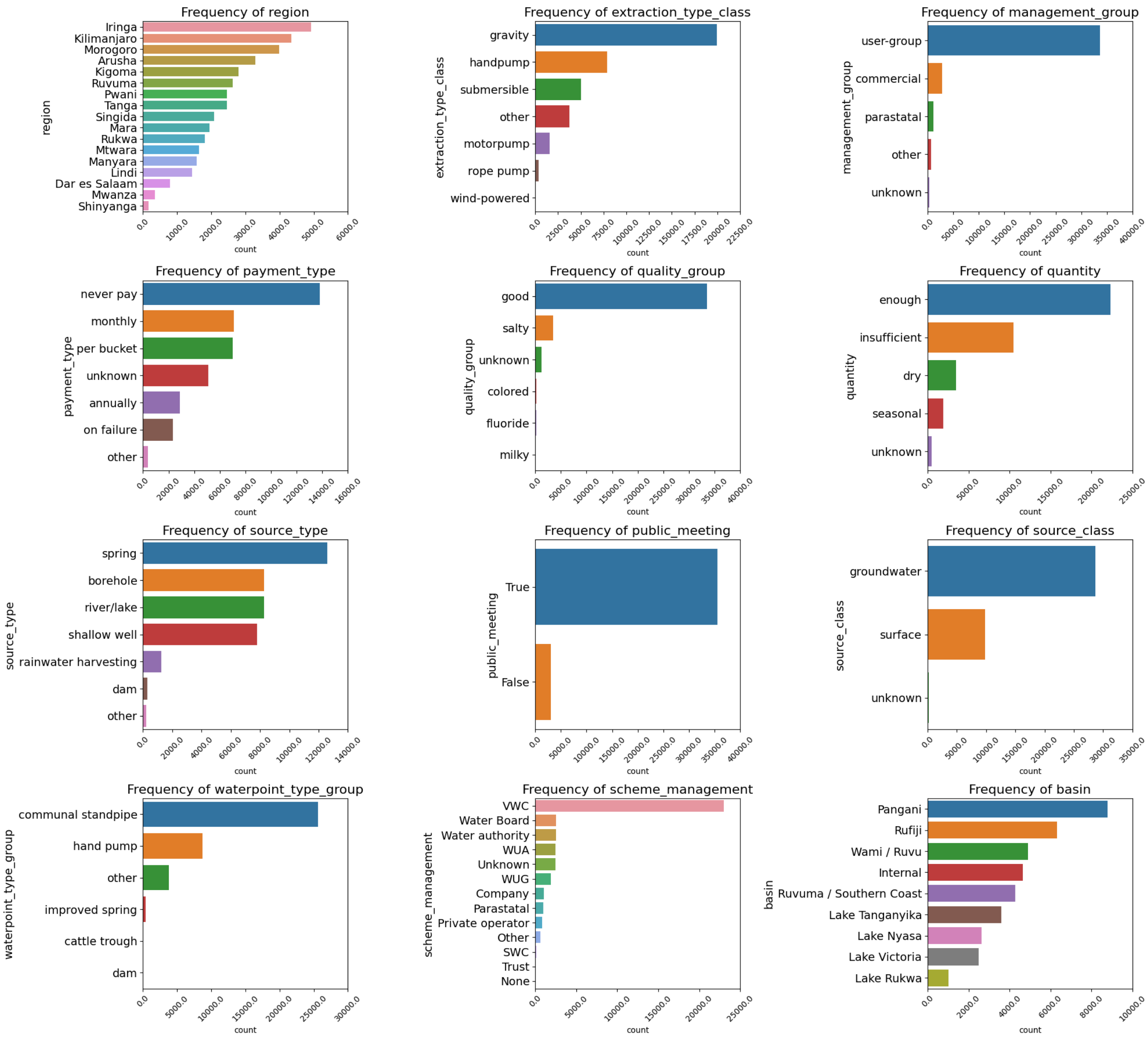
```
functional        24225
non functional    14447
Name: status_group, dtype: int64
```



In [25]:
```
# Create subplots with 3 columns
fig, (ax1, ax2, ax3) = plt.subplots(figsize=(16, 6), ncols=3)

# Plot the density of age on the first subplot
sns.kdeplot(aged_data['age'], shade=True, ax=ax1)
ax1.set_title('Distribution of Waterpoint Age')
ax1.set_xlabel('Age')
ax1.set_ylabel('Density')

# Plot the density of population on the second subplot
sns.kdeplot(aged_data['population'], shade=True, ax=ax2)
ax2.set_title('Distribution of Population accessing Waterpoints')
ax2.set_xlabel('Population')
ax2.set_ylabel('Density')

# Plot the density of Altitude of the Well on the second subplot
sns.kdeplot(aged_data['gps_height'], shade=True, ax=ax3)
ax3.set_title('Distribution of Population accessing Waterpoints')
ax3.set_xlabel('Altitude of the well')
ax3.set_ylabel('Density')

# Adjust layout to prevent overlap
plt.tight_layout()

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



In [26]:
```
def plot_countplots(df, columns):
    """
    Plot count plots for each column in the list
    """
    fig, axes = plt.subplots(4,3, figsize=(20, 18))

    for i in range(len(columns)):
        row = i // 3
        col = i % 3
        ax = axes[row, col]
        sorted_counts = df[columns[i]].value_counts().sort_values(ascending=Fa
        sns.countplot(data=df, y=columns[i], order=sorted_counts.index, ax=ax)
        ax.set_title(f'Frequency of {columns[i]}', fontsize=16)
        ax.set_xticks(ax.get_xticks())
        ax.set_xticklabels(ax.get_xticks(), rotation=45, fontsize=10)
        ax.set_yticklabels(ax.get_yticklabels(), fontsize=14)
        ax.set_ylabel(columns[i], fontsize=14)
    plt.tight_layout()
    plt.show()
#List of features to be plotted
columns= ['region','extraction_type_class',
          'management_group', 'payment_type',
          'quality_group', 'quantity','source_type','public_meeting',
          'source_class', 'waterpoint_type_group','scheme_management','basin' 
#plotting countplots
plot_countplots(aged_data, columns, )
```




## Observations:

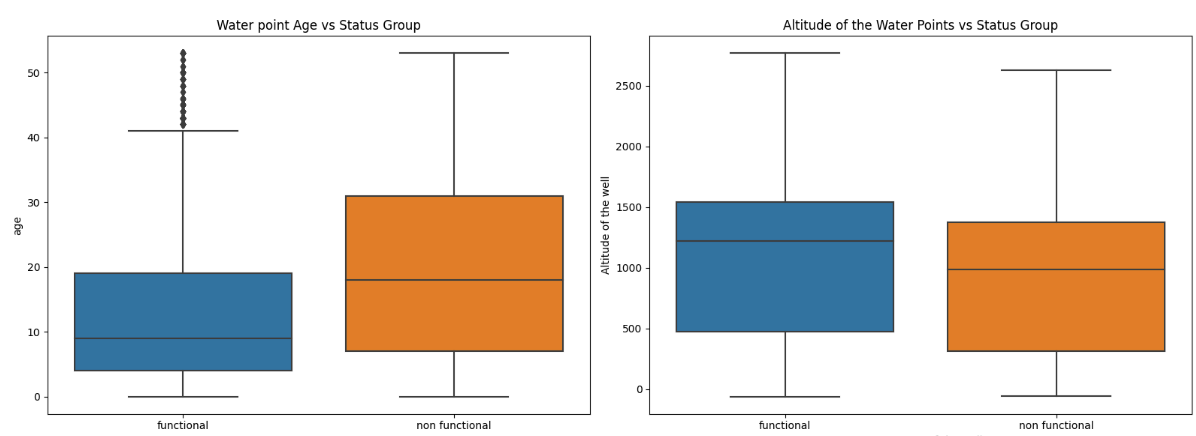
- Our status group show 62.6% functional and 37.4% non functional water points. There is some slight class imbalance

# BIVARIATE ANALYSIS

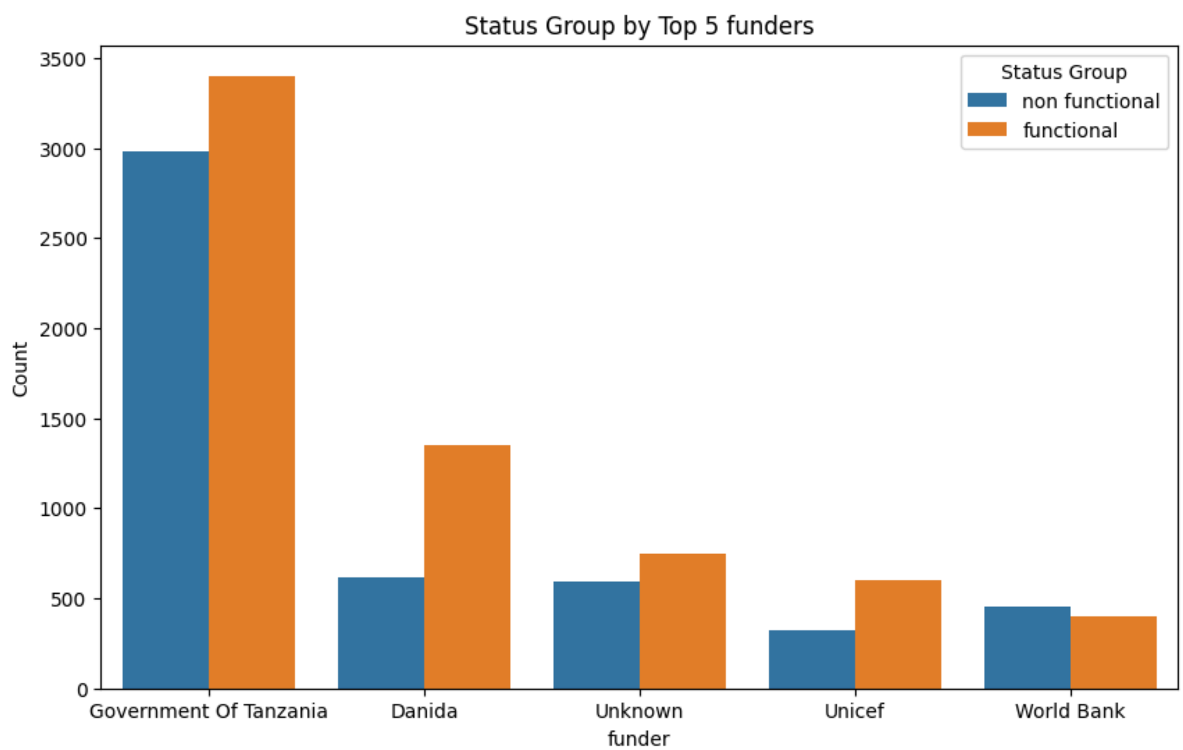
In this section we will consider the relationship between:

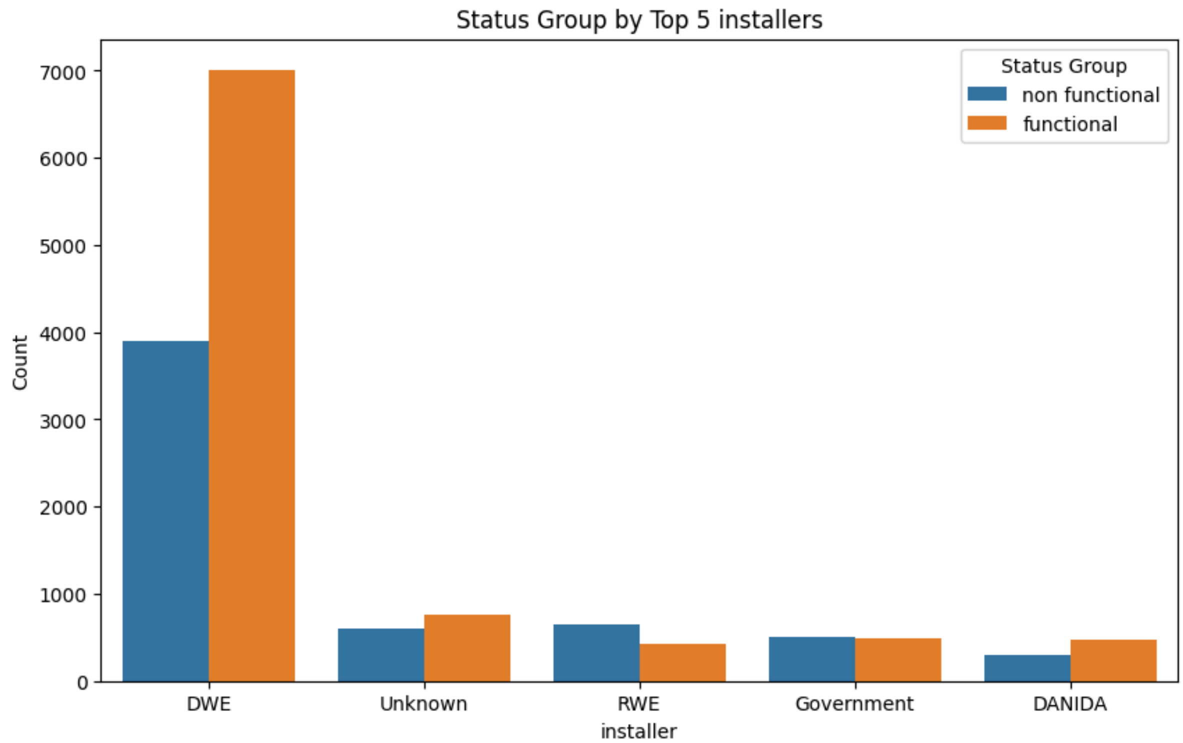
- Age and well altitude and the target variable
- Select Features and the target variables
- Deep dive into age, technology, and investment factors relating to water point failure
- Assess socioeconomic and geographic factors relating to water point failure

In [27]:
```
# Create subplots with 3 columns
fig, (ax1, ax2) = plt.subplots(figsize=(16, 6), ncols=2)

# Plot the density of age
sns.boxplot(x='status_group', y='age', data=aged_data,  ax=ax1)
ax1.set_title('Water point Age vs Status Group')
ax1.set_xlabel('status_group')
ax1.set_ylabel('age')

# Plot the density of Altitude of the Well
sns.boxplot(x='status_group', y='gps_height', data=aged_data,  ax=ax2)
ax2.set_title('Altitude of the Water Points vs Status Group')
ax2.set_xlabel('status group of the well')
ax2.set_ylabel('Altitude of the well')

# Adjust layout to prevent overlap
plt.tight_layout()

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

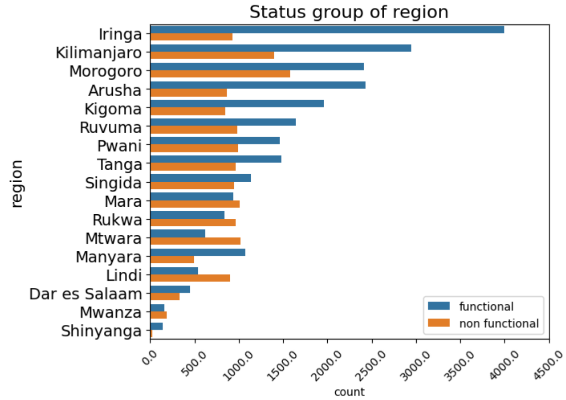


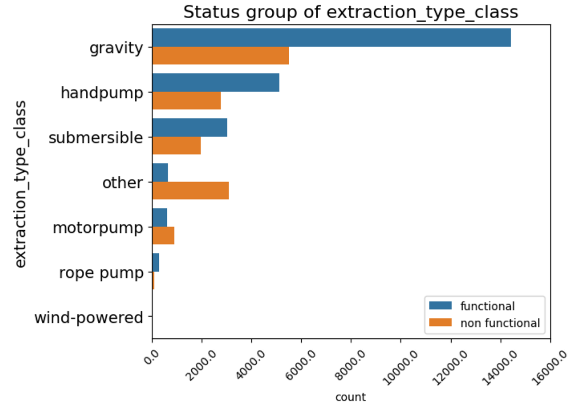
In [28]:
```
def plot_count_plots(df, columns):
    """
    Plot count plots for the top five values of each column while keeping hue a
    """
    for column in columns:
        # Get the top n values of the column in descending order
        top_values = df[column].value_counts().head().index

        # Filter the DataFrame to include only the top n values
        df_top_values = df[df[column].isin(top_values)]

        # Create the count plot for the top n values of the column
        plt.figure(figsize=(10, 6))
        sns.countplot(data=df_top_values, x=column, hue="status_group", order=t
        plt.title(f"Status Group by Top 5 {column}s")
        plt.xlabel(column)
        plt.ylabel("Count")
        plt.xticks()
        plt.legend(title="Status Group")
        plt.show()
plot_count_plots(aged_data, columns=["funder", "installer"])
```

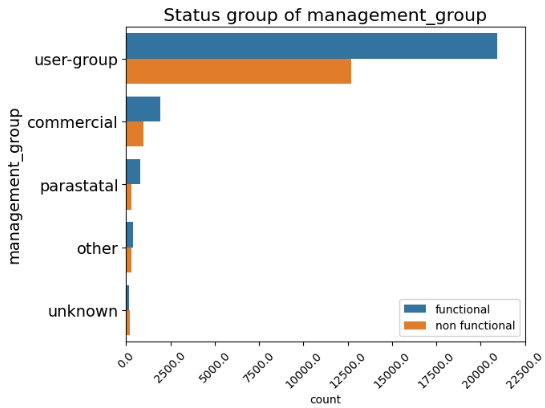


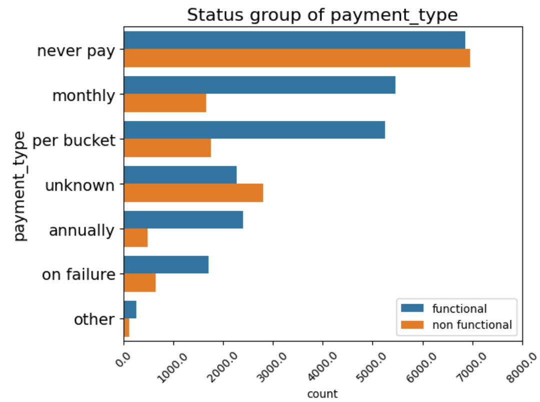

Status Group by Top 5 installers
Status Group
non functional
functional
Count
7000
6000
5000
4000
3000
2000
1000
0
DWE
Unknown
RWE
Government
DANIDA
installer

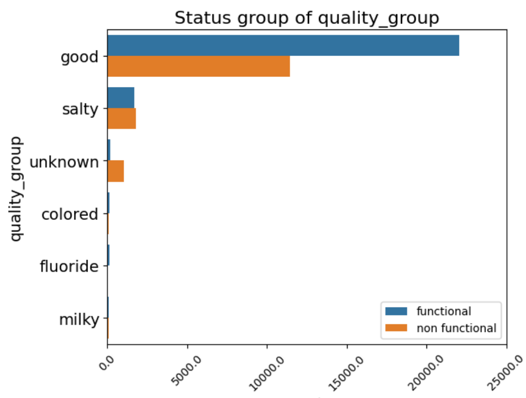
In [29]:
```
def plot_count_plots(df, columns, ):
    """
    Plot count plots for each column in the list while keeping hue as "status_g
    """
    num_rows=6
    num_cols=2

    fig, axes = plt.subplots(num_rows,num_cols, figsize=(15, 30))
    for i in range(len(columns)):
        row = i // 2
        col = i % 2
        ax = axes[row, col]
        sorted_counts = df[columns[i]].value_counts().sort_values(ascending=Fal
        sns.countplot(data=df, y=columns[i], order=sorted_counts.index, ax=ax,
        ax.set_title(f'Status group of {columns[i]}', fontsize=16)
        ax.set_xticks(ax.get_xticks())
        ax.set_xticklabels(ax.get_xticks(), rotation=45, fontsize=10)
        ax.set_yticklabels(ax.get_yticklabels(), fontsize=14)
        ax.set_ylabel(columns[i], fontsize=14)
        ax.legend(loc='lower right')

    # removing extra empty axis
    for i in range(len(columns), 6 * 2):
        axes.flatten()[i].remove()

    plt.tight_layout()
    plt.show()
# Columns to be plotted
cols= ['region','extraction_type_class',
        'management_group', 'payment_type',
        'quality_group', 'quantity','source_type',
        'source_class', 'waterpoint_type_group',"public_meeting"]
plot_count_plots(aged_data, columns=cols )
```

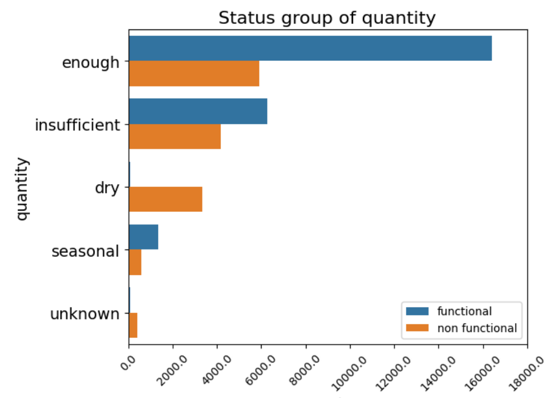

Status group of region
region
Iringa
Kilimanjaro
Morogoro
Arusha
Kigoma
Ruvuma
Pwani
Tanga
Singida
Mara
Rukwa
Mtwara
Manyara
Lindi
Dar es Salaam
Mwanza
Shinyanga
functional
non functional
0.0 500.0 1000.0 1500.0 2000.0 2500.0 3000.0 3500.0 4000.0 4500.0
count

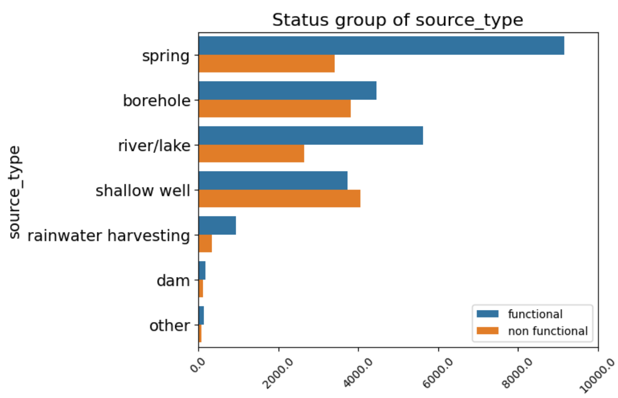

Status group of extraction_type_class
extraction_type_class
gravity
handpump
submersible
other
motorpump
rope pump
wind-powered
functional
non functional
0.0 2000.0 4000.0 6000.0 8000.0 10000.0 12000.0 14000.0 16000.0
count

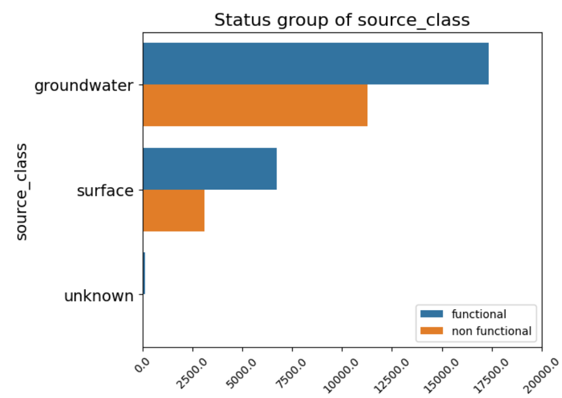

Status group of management_group
management_group
user-group
commercial
parastatal
other
unknown
functional
non functional
0.0 2500.0 5000.0 7500.0 10000.0 12500.0 15000.0 17500.0 20000.0 22500.0
count

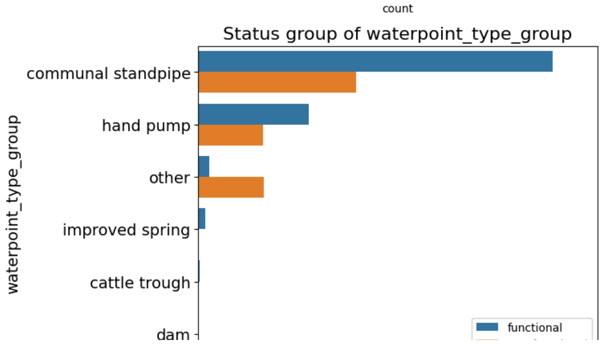

Status group of payment_type
payment_type
never pay
monthly
per bucket
unknown
annually
on failure
other
functional
non functional
0.0 1000.0 2000.0 3000.0 4000.0 5000.0 6000.0 7000.0 8000.0
count

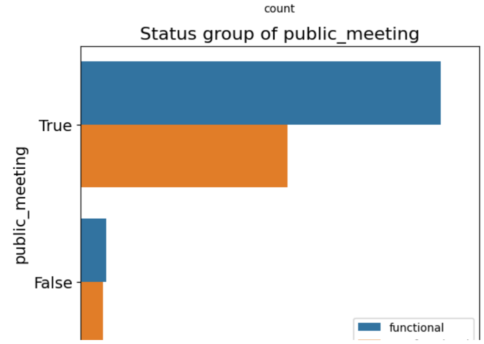

Status group of quality_group
quality_group
good
salty
unknown
colored
fluoride
milky
functional
non functional
0.0 5000.0 10000.0 15000.0 20000.0 25000.0
count

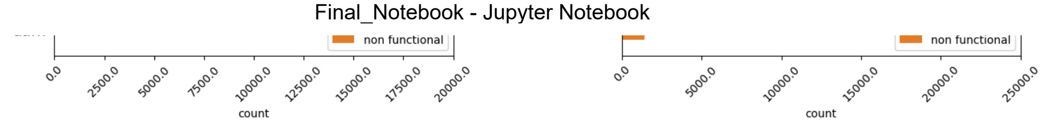

Status group of quantity
quantity
enough
insufficient
dry
seasonal
unknown
functional
non functional
0.0 2000.0 4000.0 6000.0 8000.0 10000.0 12000.0 14000.0 16000.0 18000.0
count


Status group of source_type
source_type
spring
borehole
river/lake
shallow well
rainwater harvesting
dam
other
functional
non functional
0.0 2000.0 4000.0 6000.0 8000.0 10000.0
count


Status group of source_class
source_class
groundwater
surface
unknown
functional
non functional
0.0 2500.0 5000.0 7500.0 10000.0 12500.0 15000.0 17500.0 20000.0
count


Status group of waterpoint_type_group
waterpoint_type_group
communal standpipe
hand pump
other
improved spring
cattle trough
dam
functional


Status group of public_meeting
public_meeting
True
False
functional



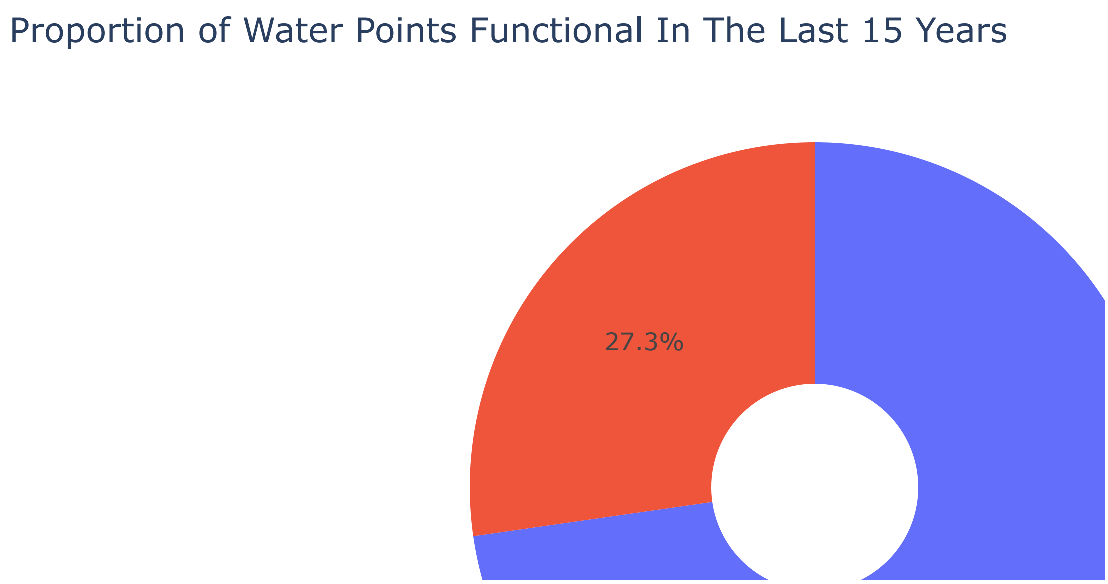

### Observations

- We can see that the lifespan of these pumps is roughly 18-20 years with some fairly new pumps failing within the first 10 years
- Waterpoints at a higher altitude generally seem to be fail less as compared to those in lower altitudes
- The local government is the greatest funder of these water points with the distric water engineeer mainly incharge of installing them with most of these waterpoints still in operation
- Key observations from the features are:
  1. Water points where people never pay, have more none functional water points.
  2. Areas with salty water are prone to have more none functional water points
  3. Shallow well source types also seem to have more none functional water points.
  4. Areas with quantity classification as dry have primarily none functional water points.
  5. Motor Pumps equipment are more likely to have none functional water points.
  6. The areas of Mara, Rukwa, Mtwara, Lindi and Mwanza have more none functional waterpoints.

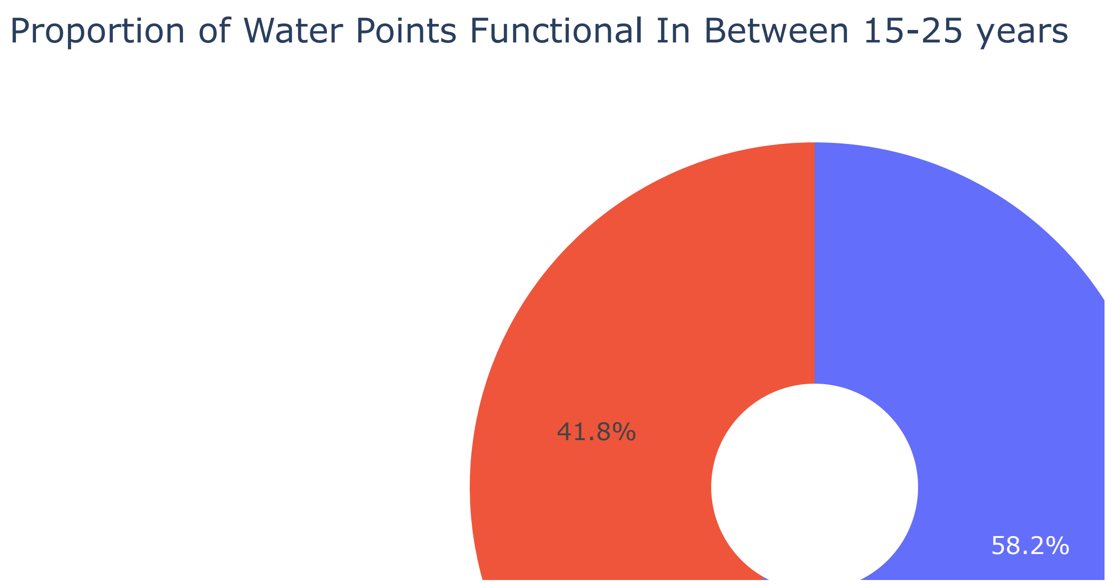
### AGE FACTORS

In [30]:
```
# Get the value counts of the age below 20 years
value_counts = aged_data[aged_data.age<15]["status_group"].value_counts()

# Create an interactive pie chart
fig = px.pie(
            value_counts,
            values=value_counts.values,
            names=value_counts.index,
            title='Proportion of Water Points Functional In The Last 15 Years',
            hole=0.3
            )

display(value_counts)
fig.show()

# Get the value counts of the age ranges
value_counts = aged_data[(aged_data.age>15)&(aged_data.age<25)]["status_group"]

# Create an interactive pie chart
fig = px.pie(
            value_counts,
            values=value_counts.values,
            names=value_counts.index,
            title='Proportion of Water Points Functional In Between 15-25 years
            hole=0.3
            )

display(value_counts)
fig.show()
```

```
functional        16139
non functional     6050
Name: status_group, dtype: int64
```




```
functional        3252
non functional    2337
Name: status_group, dtype: int64
```

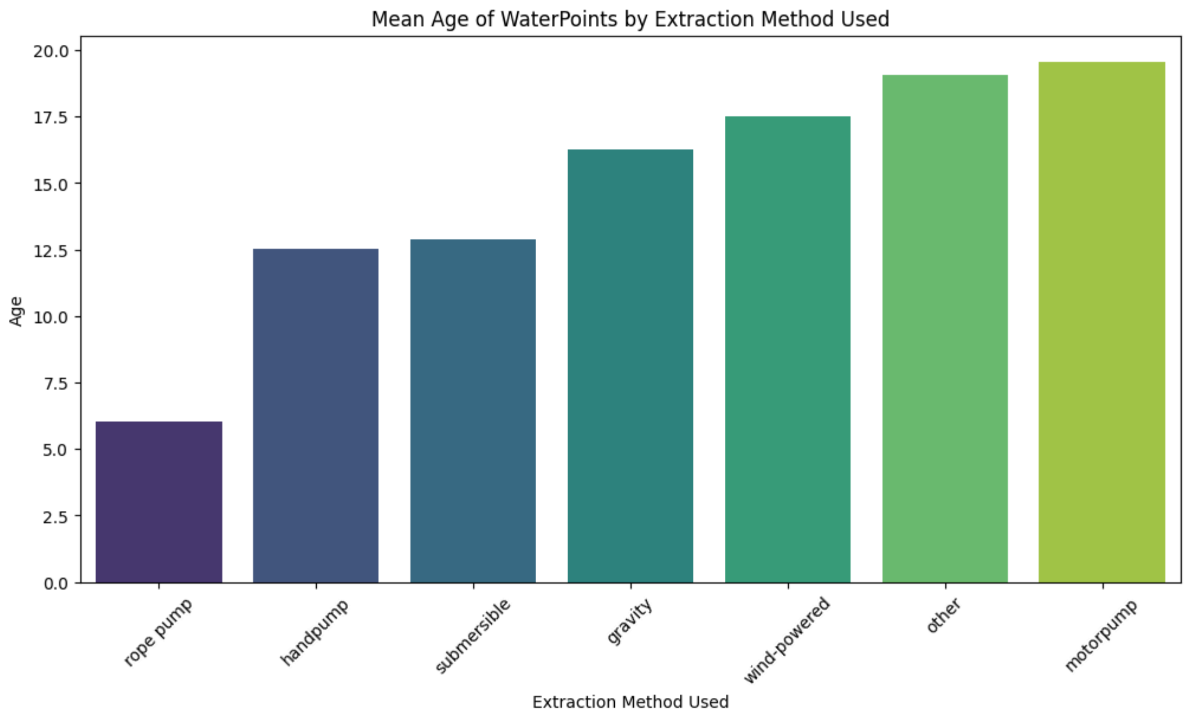



---

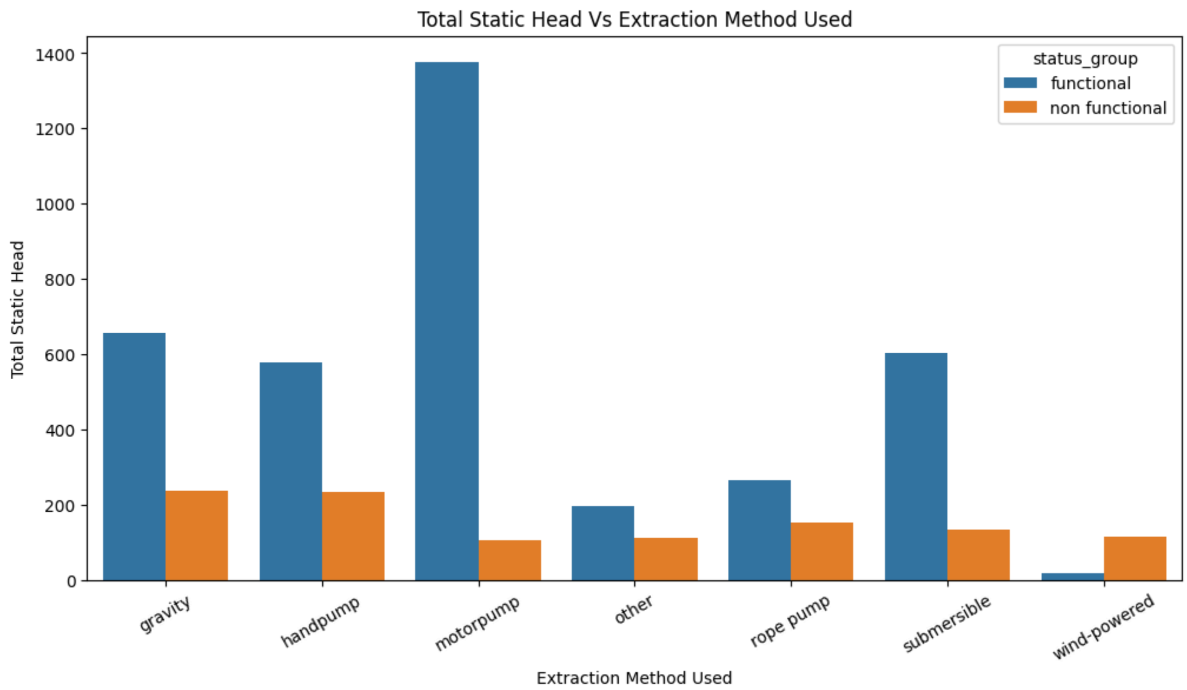
**TECHNOLOGICAL FACTORS**

---

In [31]:
```
# Group by 'extraction type', calculate the mean 'age', and sort in descending
sorted_data = aged_data.groupby('extraction_type_class')["age"].mean().sort_va

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='extraction_type_class', y='age', data=sorted_data_df, palette='v
plt.title('Mean Age of WaterPoints by Extraction Method Used')
plt.xlabel('Extraction Method Used')
plt.ylabel('Age')
plt.xticks(rotation=45)
plt.show()
```



In [32]:
```
# Group by 'extraction type', calculate the mean 'total static head'
sorted_data = aged_data.groupby(['extraction_type_class', 'status_group'])['amo
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='extraction_type_class', y='amount_tsh', data=sorted_data_df, hue
plt.title('Total Static Head Vs Extraction Method Used')
plt.xlabel('Extraction Method Used')
plt.ylabel('Total Static Head')
plt.xticks(rotation=30)
plt.show()
```

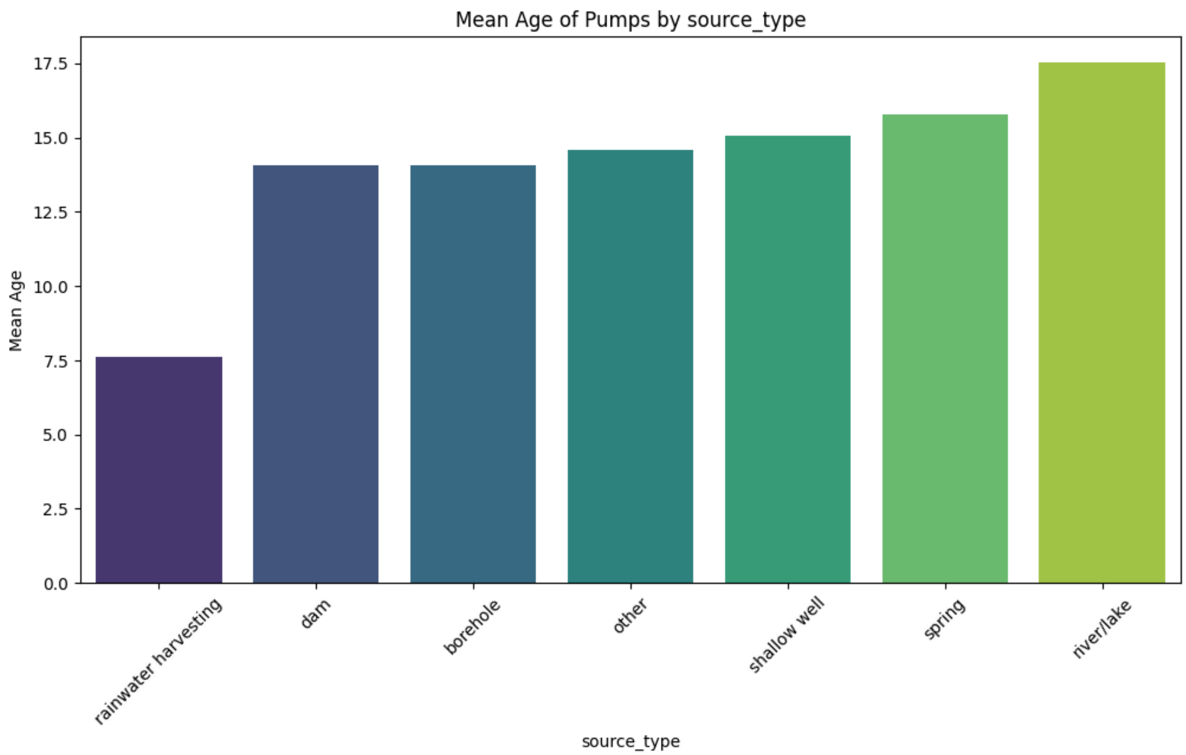



---

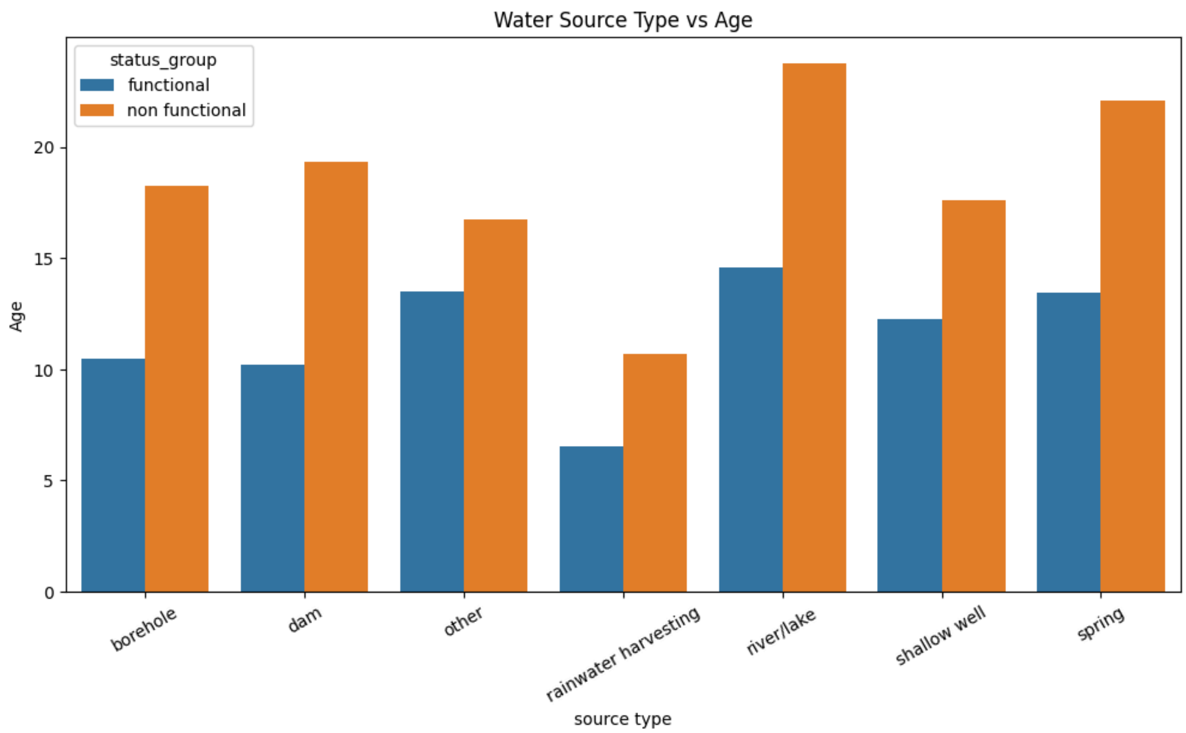
### INVESTMENT

---

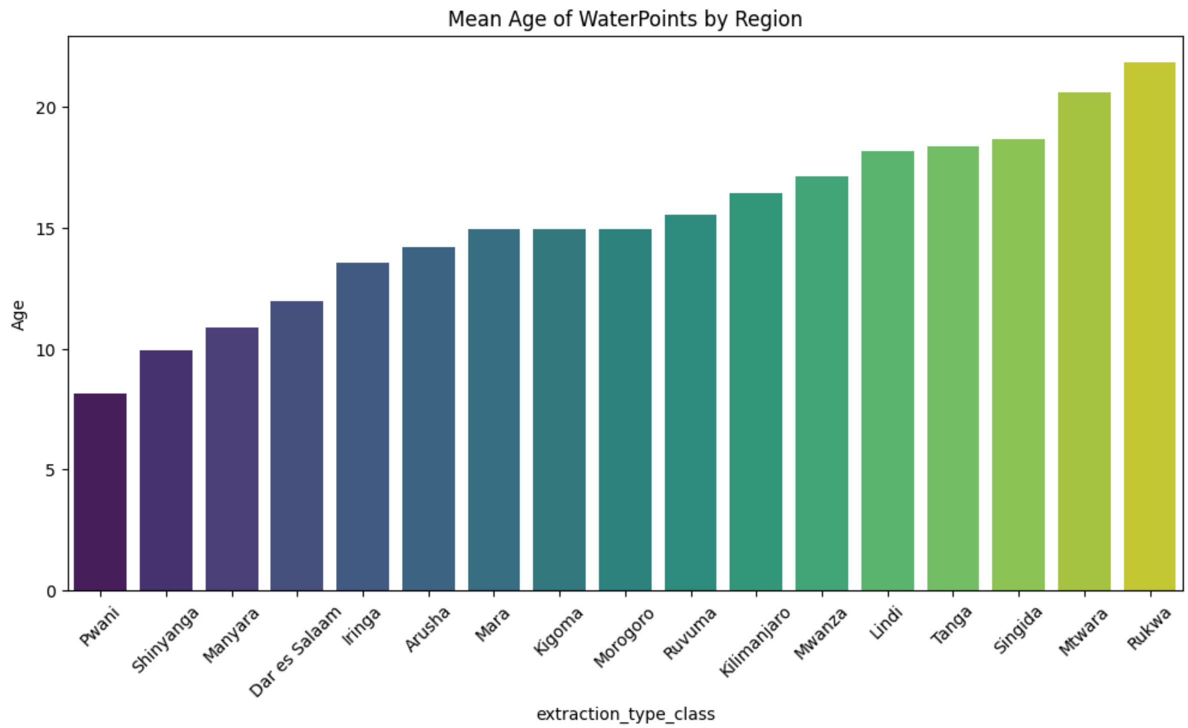
In [33]:
```
# Group by 'source type', calculate the mean 'age'
sorted_data = aged_data.groupby('source_type')["age"].mean().sort_values(ascend

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='source_type', y='age', data=sorted_data_df, palette='viridis')
plt.title('Mean Age of Pumps by source_type')
plt.xlabel('source_type')
plt.ylabel('Mean Age')
plt.xticks(rotation=45)
plt.show()
```



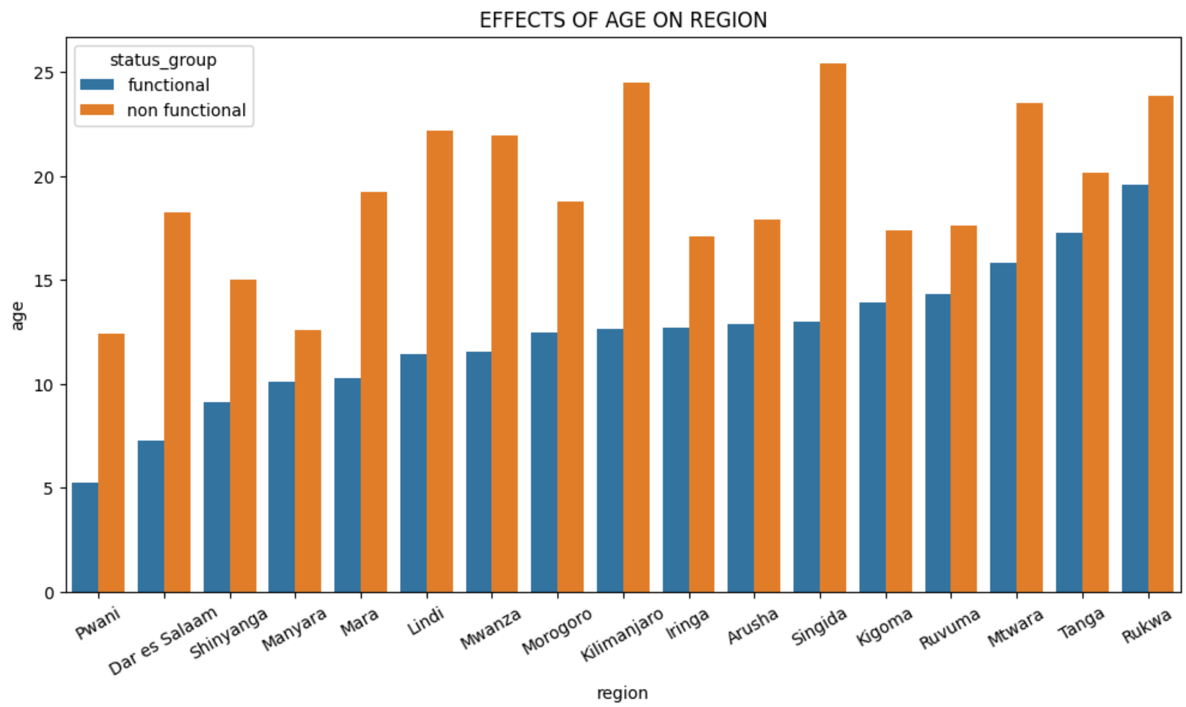
In [34]:
```
# Group by 'source type',then by 'status group' calculate the mean 'age'
sorted_data = aged_data.groupby(['source_type', 'status_group'])['age'].mean()
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='source_type', y='age', data=sorted_data_df, hue="status_group")
plt.title('Water Source Type vs Age')
plt.xlabel('source type')
plt.ylabel('Age')
plt.xticks(rotation=30)
plt.show()
```



In [35]:
```
# Group by 'region', calculate the mean 'age', and sort in descending order
sorted_data = aged_data.groupby('region')["age"].mean().sort_values(ascending=

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='region', y='age', data=sorted_data_df, palette='viridis')
plt.title('Mean Age of WaterPoints by Region')
plt.xlabel('extraction_type_class')
plt.ylabel('Age')
plt.xticks(rotation=45)
plt.show()
```



In [36]:
```
# Group by 'region',then by 'status group' calculate the mean 'age'
sorted_data = aged_data.groupby(['region', 'status_group'])['age'].mean().sort_
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='region', y='age', data=sorted_data_df, hue="status_group")
plt.title('EFFECTS OF AGE ON REGION')
plt.xlabel('region')
plt.ylabel('age')
plt.xticks(rotation=30)
plt.show()
```




---

**FINDINGS AND RESULTS**

---

1. Age Factors

> - Majority of the Water Points have been constructed in the last 20 years and are primarily functional i.e. 26,163 water points constructed and 18561 functional.
> - On average the life span of these water points is 20 years.After which we see the number of none functional points

2. Technology Factors

> - The most common extraction method is gravity type for water access with majority of them functional however we see a majority of motor pumps are non functional.
> - Recently there has been a shift to rope pumps and hand pumps as well as submersible pumps. Motor pumps are rarely installed despite motor pumps giving the most amount of water from a waterpoint on average

3. Investment Factors

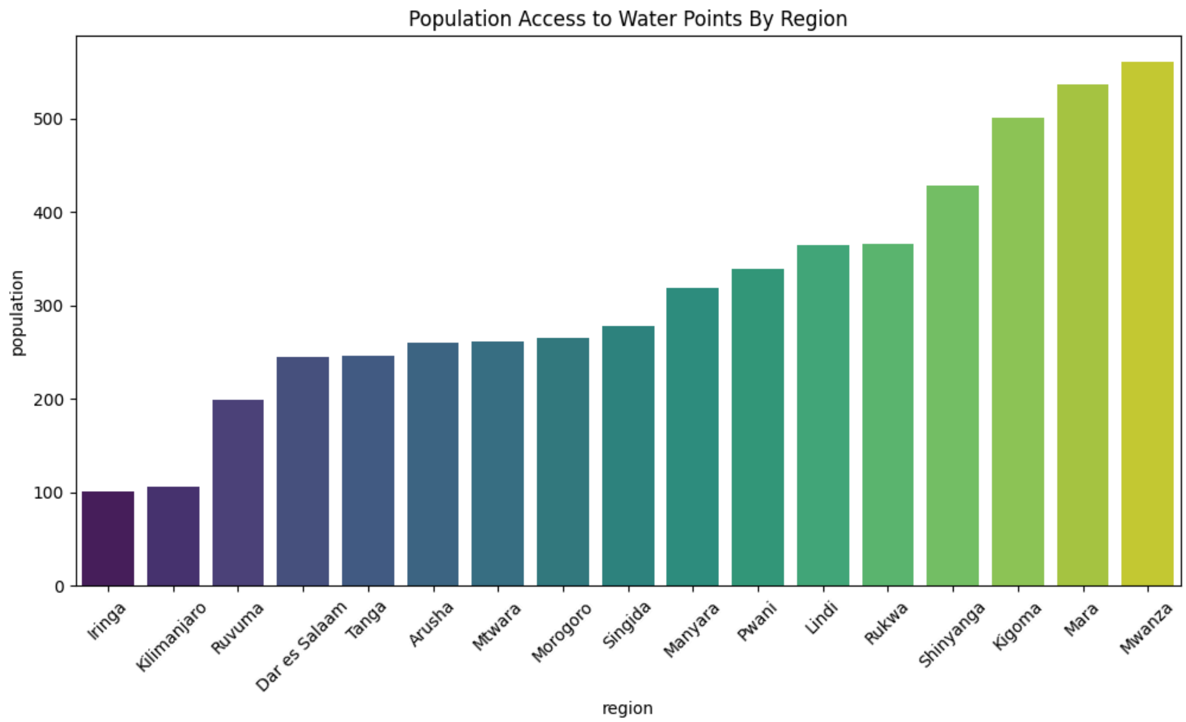
> - Investement has recently been focused on rainwater harvesting, dam and borehole construction in recent years, however it can be seen that despite this investment, majority of these water points are non functional.
> - There has been an increased focus around Pwani, Shinyanga, Manyara Regions in construction of water points.

---

**SOCIOECONOMIC FACTORS**

---

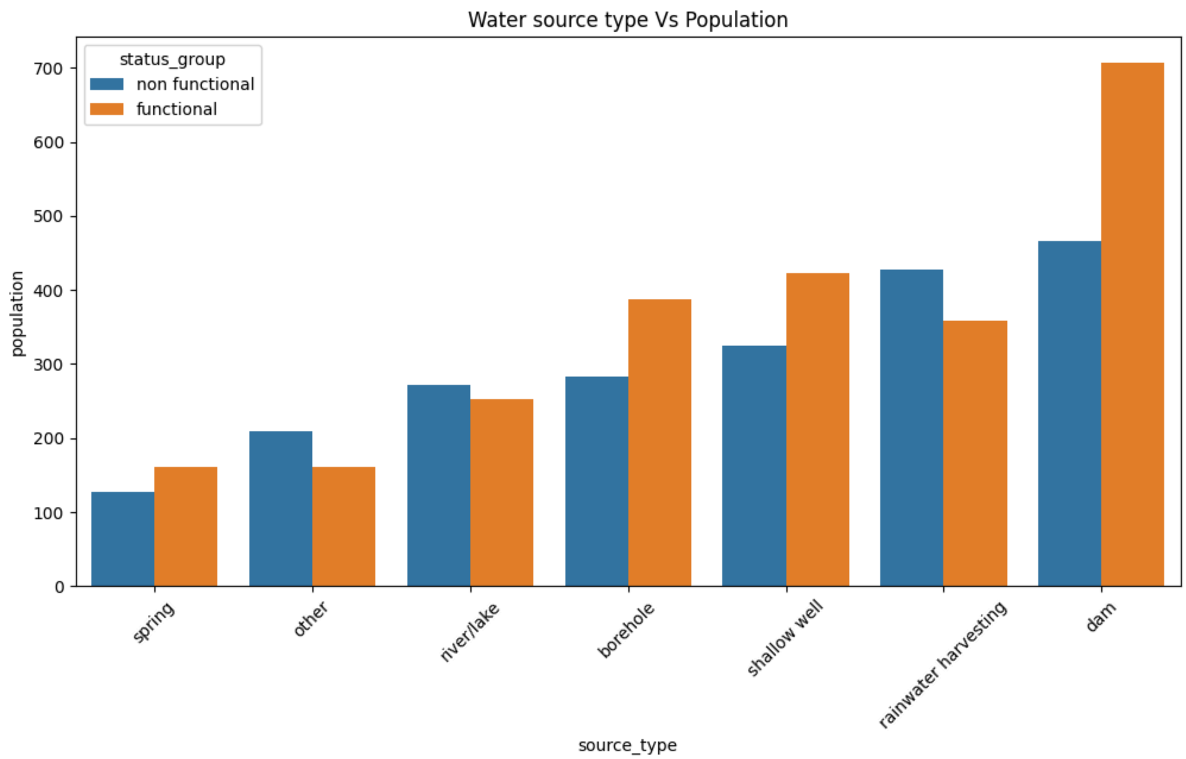
In [37]:
```
# Group by 'region'then calculate the mean 'population'
sorted_data = aged_data.groupby('region')["population"].mean().sort_values(asce

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='region', y='population', data=sorted_data_df, palette='viridis')
plt.title('Population Access to Water Points By Region')
plt.xlabel('region')
plt.ylabel('population')
plt.xticks(rotation=45)
plt.show()
```



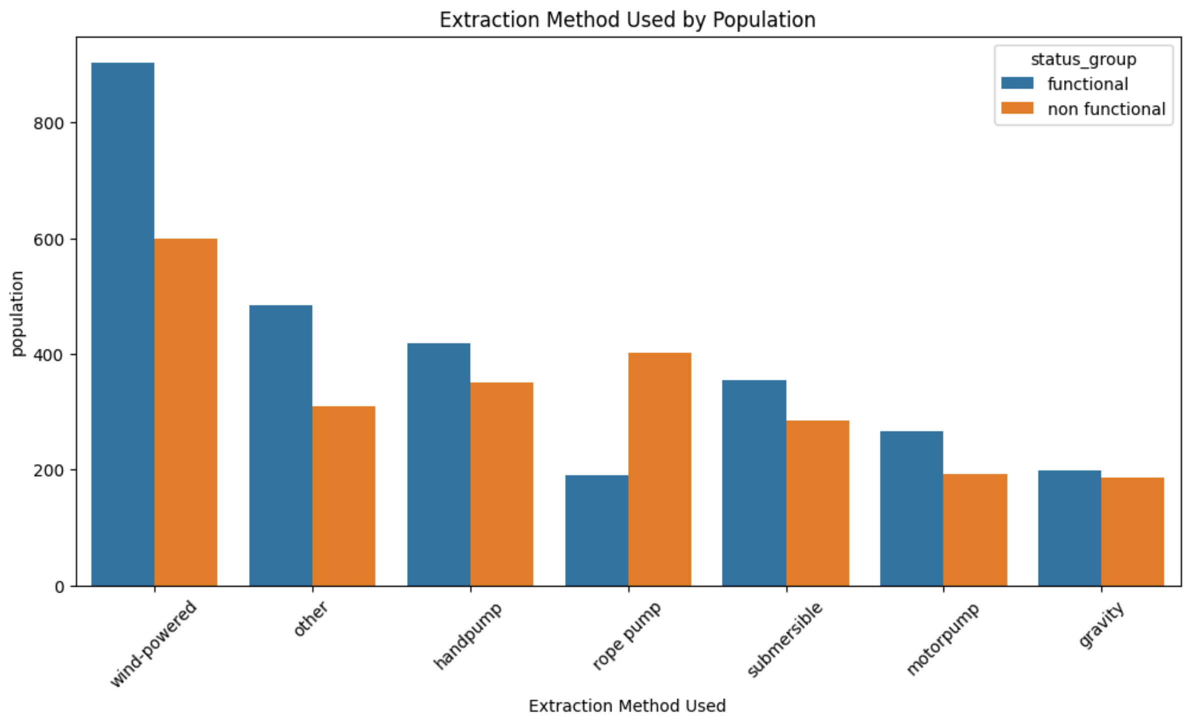

In [38]:
```
aged_data.columns
```

Out[38]:
```
Index(['amount_tsh', 'date_recorded', 'funder', 'gps_height', 'installer',
       'longitude', 'latitude', 'wpt_name', 'basin', 'subvillage', 'region',
       'lga', 'ward', 'population', 'public_meeting', 'scheme_management',
       'permit', 'construction_year', 'extraction_type_class',
       'management_group', 'payment_type', 'quality_group', 'quantity',
       'source_type', 'source_class', 'waterpoint_type_group', 'status_grou
p',
       'age'],
      dtype='object')
```

In [39]:
```
# Group by 'source_type', calculate the mean 'age', and sort in descending orde
sorted_data = aged_data.groupby(['source_type', 'status_group'])['population']
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='source_type', y='population', data=sorted_data_df, hue="status_g
plt.title('Water source type Vs Population')
plt.xlabel('source_type')
plt.ylabel('population')
plt.xticks(rotation=45)
plt.show()
```

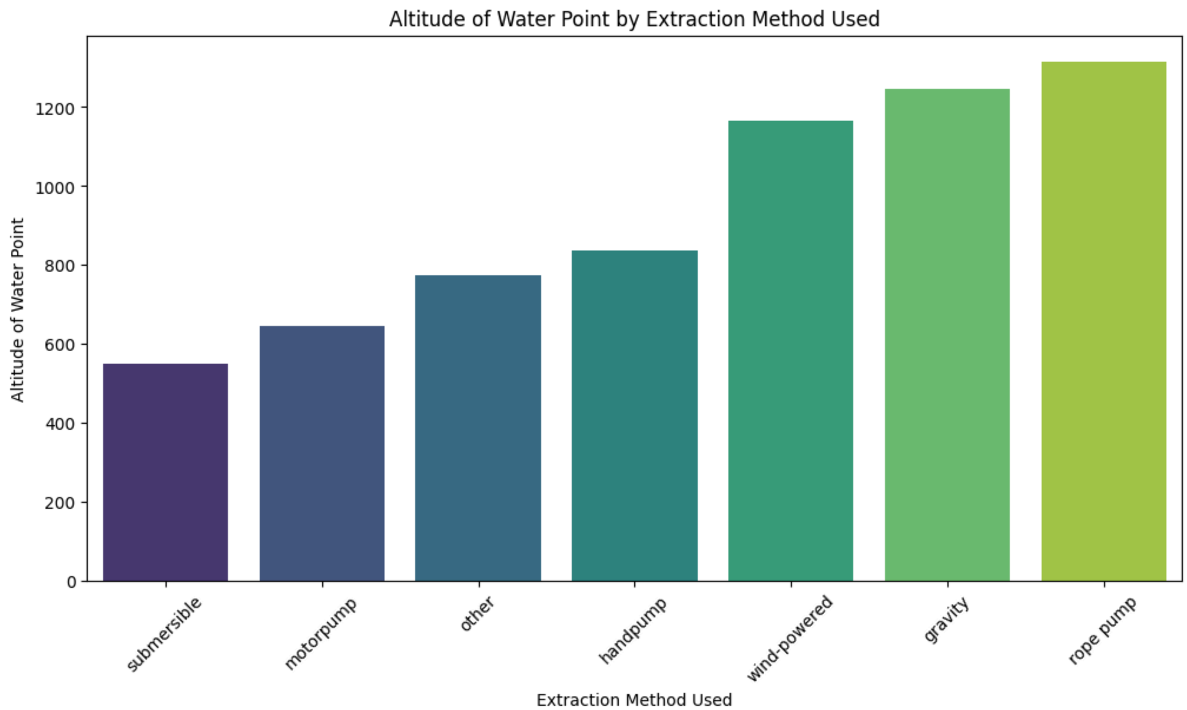


In [40]:
```
# Group by 'extraction type', calculate the mean 'age', and sort in descending
sorted_data = aged_data.groupby(['extraction_type_class', 'status_group'])['pop
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='extraction_type_class', y='population', data=sorted_data_df, hue
plt.title('Extraction Method Used by Population')
plt.xlabel('Extraction Method Used')
plt.ylabel('population')
plt.xticks(rotation=45)
plt.show()
```

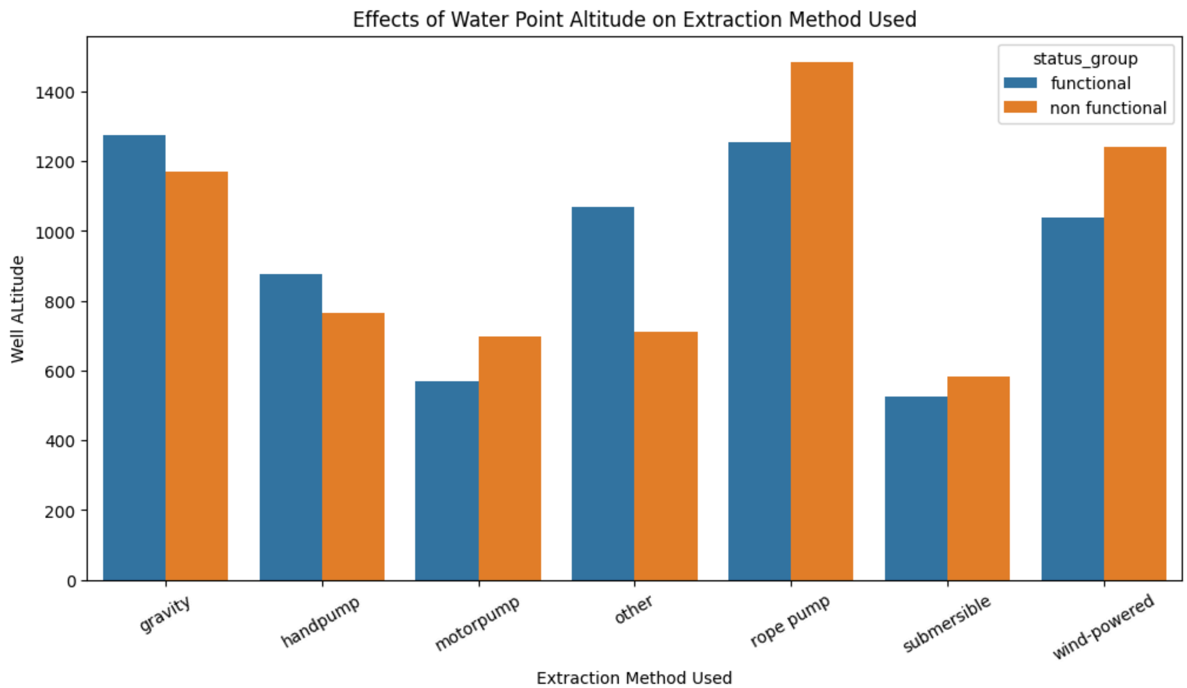



## GEOGRAPHICAL FACTORS

In [41]:
```
# Group by 'extraction type', calculate the mean 'well altitude', and sort in a
sorted_data = aged_data.groupby('extraction_type_class')["gps_height"].mean().s

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='extraction_type_class', y='gps_height', data=sorted_data_df, pal
plt.title('Altitude of Water Point by Extraction Method Used')
plt.xlabel('Extraction Method Used')
plt.ylabel('Altitude of Water Point')
plt.xticks(rotation=45)
plt.show()
```



In [42]:
```
# Group by 'extraction type',then by 'status group' then calculate the mean 'ag
sorted_data = aged_data.groupby(['extraction_type_class', 'status_group'])['gps
sorted_data

# Convert the Series to a DataFrame
sorted_data_df = sorted_data.reset_index()

# Plot the data using seaborn
plt.figure(figsize=(12, 6))
sns.barplot(x='extraction_type_class', y='gps_height', data=sorted_data_df, hue
plt.title('Effects of Water Point Altitude on Extraction Method Used')
plt.xlabel('Extraction Method Used')
plt.ylabel('Well ALtitude')
plt.xticks(rotation=30)
plt.show()
```




## FINDINGS AND RESULTS

1. Social Economic Factors

> - Focus needs to be meet in regions such as Lindi, Mwanza, Mara and Rukwa which have high population accessing fewer water points.
> - It is also evident that wind powered pumps serve the most population while being less prone to failure

2. Geographic Location Factors

- It is evident that areas with water points in higher altitudes on average have more functional water points.
- The most common technology in higher altitude areas is the rope pump with submersible pumps primarily used in areas of lower altitude.
- We however can see that the rope pumps are prone to failure. Hand Pumps

## HANDLING HIGH CARDINALITY COLUMNS

In [43]:
```
def high_cardinality_columns(df, threshold=10):
    """
    Display columns with high cardinality based on the threshold.
    """
    high_card_cols = [col for col in df.columns if df[col].nunique() > thresho

    return print(f"Columns with high cardinality (threshold={threshold}): {high
high_cardinality_columns(aged_data)
```

```
Columns with high cardinality (threshold=10): ['amount_tsh', 'funder', 'gps_h
eight', 'installer', 'longitude', 'latitude', 'wpt_name', 'subvillage', 'regi
on', 'lga', 'ward', 'population', 'scheme_management', 'construction_year',
'age']
```

In [44]:
```
# Specific columns choosen to be dropped due to high cardinality and irrelevant
cols=['funder', 'installer', 'longitude', 'latitude',
      'wpt_name', 'subvillage', 'lga', 'ward','scheme_management','source_class
      'construction_year', 'date_recorded', 'basin','permit','public_meeting']
aged_data.drop(columns=cols, inplace= True)
aged_data.shape
```

Out[44]:
```
(38672, 13)
```

**Observations:**

- Specific columns with high cardinality were removed as this would cause overfitting in the model
- The dataset has 38,672 rows and 13 rows that shall be used for modelling.

# MODELING

The following steps were carried out:

> - Data Preprocessing
> - Modeling using
>
> ```
>                  - Logistic Regression Classifier
>                  - Decision Tree Classifier
>                  - Random Forest Regression Classifier
>                  - XGBOOST Classifier
> ```
>
> - Hyper Parameter Tuning

---

#### DATA PREPROCESSING

---

A function was created to undertake the following steps for data preprocessing:

> - Splitting data
> - Onehot encoding the features data
> - Label encoding the target data
> - Class Imbalance Correction

In [45]:
```
def preprocessing(df, test_size=0.25):
    """
    Performs Preprocessing of the data received
    """
    # Splitting the dataset into train and validation sets
    X = df.drop(columns=["status_group"], axis=1)
    y = df["status_group"]

    # Performing train test splits
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=test_size

    # Extract categorical columns
    X_train_cat = X_train.select_dtypes(exclude='number')
    X_val_cat = X_val.select_dtypes(exclude='number')
    X_train_num = X_train.select_dtypes(include='number')
    X_val_num = X_val.select_dtypes(include='number')

    # One-hot encode categorical features
    X_train_enc = pd.get_dummies(X_train_cat)
    X_val_enc = pd.get_dummies(X_val_cat)

    # Instantiate the Label Encoder class
    label_encoder = LabelEncoder()

    # Fit and transform the data
    y_train_lb = label_encoder.fit_transform(y_train)
    y_val_lb = label_encoder.transform(y_val)

    # Return array back into a dataframe
    y_train_enc = pd.Series(y_train_lb, index=y_train.index, name='status_grou
    y_val_enc = pd.Series(y_val_lb, index=y_val.index, name='status_group')

    # Instantiate the SMOTE class
    smote = SMOTE(random_state=42)

    # Fit and resample the data
    X_train_enc, y_train_enc = smote.fit_resample(X_train_enc, y_train_enc)

    return X_train_enc, X_val_enc, y_train_enc, y_val_enc
```

---

### MODELING

---

A function was created that would undertake classifciation using different models.

In [46]:
```
def modeling(X_train_enc, X_val_enc, y_train_enc, y_val_enc, classifiers):
    # Define empty lists to store results in a table
    table = []
    """
    Inputs different classifiers and return a table with the results for train
    """
    # Iterate over classifiers
    for clf_name, clf in classifiers.items():
        # Train the model
        clf.fit(X_train_enc, y_train_enc)

        # Make predictions
        y_pred_val = clf.predict(X_val_enc)
        y_pred_train = clf.predict(X_train_enc)

        # Calculate accuracy and F1 score for training set
        train_accuracy = accuracy_score(y_train_enc, y_pred_train)
        train_f1 = f1_score(y_train_enc, y_pred_train)
        train_cross_val = cross_val_score(clf, X_train_enc, y_train_enc, cv=5)

        # Calculate accuracy and F1 score for validation set
        val_accuracy = accuracy_score(y_val_enc, y_pred_val)
        val_f1 = f1_score(y_val_enc, y_pred_val)
        val_cross_val = cross_val_score(clf, X_val_enc, y_val_enc, cv=5).mean(

        # Append results to the list
        table.append({
            'Classifier': clf_name,
            'Data': 'Training',
            'Accuracy': train_accuracy,
            'F1 Score': train_f1,
            'Cross Val Score': train_cross_val
        })
        table.append({
            'Classifier': clf_name,
            'Data': 'Validation',
            'Accuracy': val_accuracy,
            'F1 Score': val_f1,
            'Cross Val Score': val_cross_val
        })

    # Create DataFrame from results
    model_df = pd.DataFrame(table)

    return model_df
```

Prior to modeling, we looked at the best size we should use for the project as below

**DETERMINING TEST SIZE**

In [47]:
```
# list of test sizes to choose from
list_sizes=[0.15,0.2,0.25,0.3]

for size in list_sizes:
    results=[]
    X_train_enc, X_val_enc, y_train_enc,y_val_enc= preprocessing(aged_data, te
    # Define classifiers to build
    classifiers = {
                'Logistic Regression':   LogisticRegression(random_state=42),
                }
    # Modeling
    model_results=modeling(X_train_enc, X_val_enc, y_train_enc, y_val_enc, cla
    display(size,model_results)
```

0.15

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| **0** | Logistic Regression | Training | 0.763835 | 0.741219 | 0.762455 |
| **1** | Logistic Regression | Validation | 0.777452 | 0.687636 | 0.773487 |

0.2

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| **0** | Logistic Regression | Training | 0.761543 | 0.738197 | 0.761002 |
| **1** | Logistic Regression | Validation | 0.775178 | 0.682026 | 0.776988 |

0.25

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| **0** | Logistic Regression | Training | 0.764495 | 0.741892 | 0.763371 |
| **1** | Logistic Regression | Validation | 0.775962 | 0.683888 | 0.775134 |

0.3

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| **0** | Logistic Regression | Training | 0.765048 | 0.741946 | 0.763787 |
| **1** | Logistic Regression | Validation | 0.774263 | 0.682507 | 0.778658 |

### Observation

While using logistic regression with its default hyper parameters we can see that a test size of 0.15 gives us the least variance between the training and test data. Due to the size of the dataset,we do not have enough features to reduce the overfitting issue and will hence use a test size of 0.15 for training to reduce the overfitting

In [48]:

```
#Setting the varibales with a test size of 0.15
X_train_enc, X_val_enc, y_train_enc,y_val_enc= preprocessing(aged_data, test_s

# Define classifiers to build
classifiers = {
                'Logistic Regression':   LogisticRegression(random_state=42),
                'KNN':                   KNeighborsClassifier(),
                'Decision Tree':         DecisionTreeClassifier(random_state=4
               }
# Modeling
modeling(X_train_enc, X_val_enc, y_train_enc, y_val_enc, classifiers)
```

Out[48]:

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| 0 | Logistic Regression | Training | 0.763835 | 0.741219 | 0.762455 |
| 1 | Logistic Regression | Validation | 0.777452 | 0.687636 | 0.773487 |
| 2 | KNN | Training | 0.802287 | 0.803657 | 0.787968 |
| 3 | KNN | Validation | 0.779348 | 0.724257 | 0.773483 |
| 4 | Decision Tree | Training | 0.834319 | 0.824532 | 0.804178 |
| 5 | Decision Tree | Validation | 0.813480 | 0.743359 | 0.794348 |

---

**Observation:**

---

The models were evaluated on their default hyper parameters.

- The best performing model was the decision tree on the accuracy score at 81.34%
- None of the model achieved an f1 score above 75% and so we shall try modeling using ensemble methods

---

**BUILDING MODELS USING ENSEMBLE METHODS**

---

In [49]:
```
# Define classifiers to build
classifiers = {
                'Random Forest':     RandomForestClassifier(random_state=42),
                'XGBoost':           XGBClassifier()
              }
# Modeling
modeling(X_train_enc, X_val_enc, y_train_enc, y_val_enc, classifiers)
```

Out[49]:

| | Classifier | Data | Accuracy | F1 Score | Cross Val Score |
|---|---|---|---|---|---|
| **0** | Random Forest | Training | 0.834270 | 0.825759 | 0.808442 |
| **1** | Random Forest | Validation | 0.817618 | 0.751293 | 0.803655 |
| **2** | XGBoost | Training | 0.807594 | 0.793090 | 0.799308 |
| **3** | XGBoost | Validation | 0.816928 | 0.745446 | 0.807102 |

#### Observation:

---

> XGBoost though not provide the best scores across the board it has the least variance between train and validation scores.
>
> It was hence choosen for further hyper parameter tuning to try and increase its performance

---

#### HYPERPARAMETER TUNING XGBOOST MODEL

---

To perform some hyper parameter tuning, GridsearchCV was considered but due to the large size of the dataset and the complexity of XGBoost, it was decided to use RandomizedSearchCV for parameter tuning

In [50]:
```
# Define the parameter distribution
param_dist = {
            'n_estimators': [100, 200, 500],
            'max_depth': [7, 8, 9],
            'learning_rate': [0.05, 0.1, 0.2],
            'colsample_bytree': [0.3, 0.7, 1.0]
            }

# Initialize XGBoost classifier
xgb_clf = XGBClassifier(random_state=42)

# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(estimator=xgb_clf, param_distributions=paran

# Fit the model
random_search.fit(X_train_enc, y_train_enc)

# Get the best parameters
best_params = random_search.best_params_
print(f"Best parameters found: {best_params}")
```

```
Fitting 3 folds for each of 81 candidates, totalling 243 fits
Best parameters found: {'n_estimators': 500, 'max_depth': 9, 'learning_rate':
0.2, 'colsample_bytree': 0.3}
```

In [51]:
```
# Assuming best_model is already defined
best_model = random_search.best_estimator_

# Predictions and scores for training data
y_train_pred = best_model.predict(X_train_enc)
f1_train = f1_score(y_train_enc, y_train_pred)
accuracy_train = accuracy_score(y_train_enc, y_train_pred)
cross_val_train= cross_val_score(best_model,y_train_enc, y_train_pred).mean()

# Predictions and scores for validation data
y_val_pred = best_model.predict(X_val_enc)
f1_val = f1_score(y_val_enc, y_val_pred)
accuracy_val = accuracy_score(y_val_enc, y_val_pred)
cross_val_val= cross_val_score(best_model,y_val_enc, y_val_pred).mean()


# Create a DataFrame to store the results
results = pd.DataFrame({
    'Data': ['Training', 'Validation'],
    'F1 Score': [f1_train, f1_val],
    'Accuracy': [accuracy_train, accuracy_val],
    'Cross Validation': [cross_val_train,cross_val_val]

})

# Display the results
results
```

Out[51]:

| | Data | F1 Score | Accuracy | Cross Validation |
|---|---|---|---|---|
| **0** | Training | 0.820346 | 0.829957 | 0.829957 |
| **1** | Validation | 0.754307 | 0.820548 | 0.820547 |

In [52]:
```
# Assuming rf.feature_importances_ contains the feature importances
feature_importances = best_model.feature_importances_

# Assuming X_train.columns contains the feature labels
feature_labels = X_train_enc.columns

# Create a DataFrame to store feature importances with their corresponding labe
feature_importance_df = pd.DataFrame({'Feature': feature_labels, 'Importance':

# Optionally, you can sort the DataFrame by importance to visualize the most in
feature_importance_df = feature_importance_df.sort_values(by='Importance', asce

# Display the DataFrame
feature_importance_df[:5]
```

Out[52]:

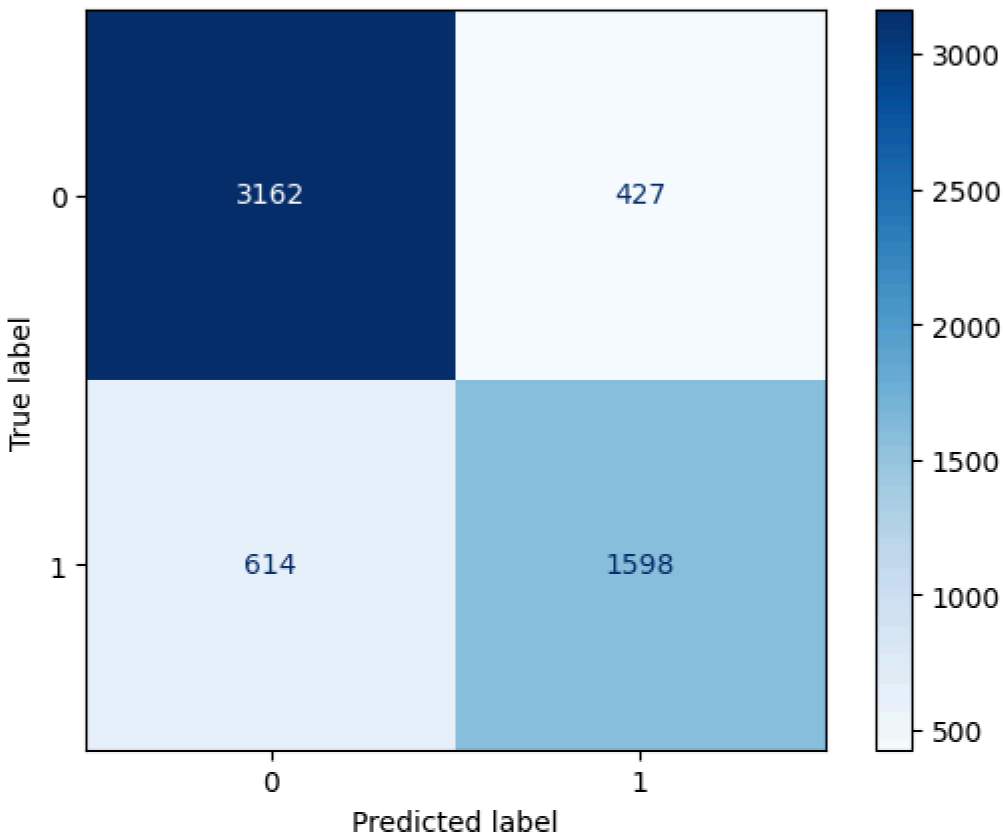
| | Feature | Importance |
|---|---|---|
| **42** | quantity_dry | 0.337651 |
| **20** | extraction_type_class_other | 0.052474 |
| **2** | region_Iringa | 0.050788 |
| **58** | waterpoint_type_group_improved spring | 0.023161 |
| **31** | payment_type_never pay | 0.021514 |

In [53]:
```
# Import confusion_matrix
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay

# Print confusion matrix
cnf_matrix = confusion_matrix(y_val_enc, y_val_pred)
print('Confusion Matrix:\n', cnf_matrix)
```

```
Confusion Matrix:
 [[3162  427]
 [ 614 1598]]
```

In [54]:
```
# Visualize your confusion matrix
import matplotlib.pyplot as plt
disp = ConfusionMatrixDisplay(confusion_matrix=cnf_matrix, display_labels=best_
disp.plot(cmap=plt.cm.Blues)
```

Out[54]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1d74dd321
60>
```




---

### FINDINGS AND RESULTS

---

### Modeling

> - After preprocessing the data and running it through a Logistic Regression and Decision Tree model, the desired accuracy and f1 score was not achieved on either model
> - Further modeling out was carried out using a Random Forest Classifier and XGBoost classifier. These results were slightly better but carried some slight overfitting bias to them.
> - Using XGBoost classifier the model underwent some parameter tuning and achieved the target accuracy of 80% and target f1 score of 75%

---

## CONCLUSIONS

1. Age of the pumps at the water points are a key indicator for failure with the majority of pumps failing between 15-25 years.
2. Gravity waterpoints give the best performance compared to other forms of technology with motor pumps on the other hand providing the most water per waterpoint i.e. total static head
3. Heavy investment in the past 7-10 years has been on rain water harvesting and has also been focused on the regions of Pwani, Shinyanga and Manyara.
4. Population main source of water are from dams and water harvesting with dams having few failures
5. Areas of high altitude have more functional water points and mainly use rope pumps but lower altitude areas favor submersible pumps

## RECOMMENDATIONS

- Water Point Pumps require replacement every 10-15 years to ensure failure doesnt affect the population as well as a premise for predictive maintenance.
- Focus needs to be meet in regions such as Lindi, Mwanza, Mara and Rukwa which have high population accessing fewer water points.
- Leverage more reliable extraction type technology such as motor pumps which give more water output per water point. As well as seek to encorporate green technology for sustainability eg solor powered or wind powered technologies
- Using the predictive algorithm, you can predict with upto 80% accuracy to prevent water point downtimes.
- New data required as the dataset is missing significant data points and was recorded over 11 years so feature elemts might have changed

```
In [ ]:
```