Business Understanding

Business Problem: Improving Waterpoint Functionality in Rural Tanzania

Background: In rural Tanzania, the functionality of waterpoints (wells, boreholes, etc.) is crucial for providing communities with access to clean and reliable water sources. The data set includes information about various attributes of these waterpoints, such as their physical characteristics, management details, and operational status.

Business Problem Statement:

The goal is to identify the key factors that influence the operational status of waterpoints in rural Tanzania. By understanding these factors, we can prioritize interventions to improve the functionality and sustainability of waterpoints, ensuring consistent access to water for communities.

Key Business Questions:

**What role do geographical and demographic variables (e.g., region, population, gps_height, basin) play in the functionality of waterpoints?

**How does the age of a waterpoint (construction_year) correlate with its operational status?

**Are there specific funders or installers associated with higher functionality rates?

**What is the relationship between water quality and the operational status of waterpoints?

Data Columns to be Utilized:

id: Unique identifier for each waterpoint.

amount_tsh: Total static head (amount of water available to a well).

date_recorded: The date the data was recorded.

funder: Organization that funded the installation of the waterpoint.

gps_height: Altitude of the waterpoint.

installer: Organization that installed the waterpoint.

longitude, latitude: Geographical coordinates of the waterpoint.

num_private: Not clearly defined but could relate to private funding or management.

basin: Basin where the waterpoint is located.

region, region_code: Administrative regions.

district_code: Code for the district.

lga: Local government area.

ward: Administrative subdivision.

population: Population of the area served by the waterpoint.

recorded_by: Individual or entity that recorded the data.

scheme_management: Who manages the waterpoint scheme.

scheme_name: Name of the water scheme.

permit: Whether a permit was issued for the waterpoint.

construction_year: Year the waterpoint was constructed.

extraction_type: Type of extraction mechanism used.

management: Overall management approach.

payment: Payment structure for using the waterpoint.

payment_type: Specific type of payment required.

water_quality: Quality of the water from the waterpoint.

quantity: Quantity of water available.

source: Natural source of the water.

source_class: Classification of the water source.

waterpoint_type: Type of waterpoint (e.g., hand pump, borehole).

status_group: Functional status of the waterpoint (functional, functional but needs repair, non-functional).

Analytical Approach:

Descriptive Analysis: Summarize the data to understand the distribution of each variable.

Correlation Analysis: Identify relationships between the operational status and other variables.

Predictive Modeling: Use machine learning techniques (e.g., logistic regression, decision trees, random forests) to predict the status group based on the other attributes.

Expected Outcomes:

**Actionable Insights: Identification of key factors that can be addressed to improve waterpoint functionality.

**Targeted Interventions: Recommendations for funders, installers, and local governments to prioritize interventions.

**Policy Recommendations: Data-driven suggestions for policy changes to enhance waterpoint management and sustainability.

**By addressing these questions and using the data effectively, stakeholders can make informed decisions to improve water access and management in rural Tanzania, ultimately leading to better health and quality of life for the communities served.

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

# Load the datasets
water_wells1 = pd.read_csv('C:/Users/USER/OneDrive/Desktop/Data science/Ir
Status_group = pd.read_csv('C:/Users/USER/OneDrive/Desktop/Data science/Ir

# Display the first few rows of each dataset
print("Water wells1 Dataset:")
print(water_wells1.head())
print("\nWater wells status group:")
print(Status_group.head())
```

```
Water wells1 Dataset:
      id  amount_tsh date_recorded        funder  gps_height     install
er  \
0  69572      6000.0     3/14/2011         Roman        1390         Rom
an
1   8776         0.0      3/6/2013       Grumeti        1399       GRUME
TI
2  34310        25.0     2/25/2013  Lottery Club         686  World visi
on
3  67743         0.0     1/28/2013        Unicef         263        UNIC
EF
4  19728         0.0     7/13/2011   Action In A           0       Artis
an

   longitude   latitude              wpt_name  num_private  ... payment_
type  \
0  34.938093  -9.856322                  none            0  ...     annu
ally
1  34.698766  -2.147466              Zahanati            0  ...    never
pay
2  37.460664  -3.821329           Kwa Mahundi            0  ...   per bu
cket
3  38.486161 -11.155298  Zahanati Ya Nanyumbu            0  ...    never
pay
4  31.130847  -1.825359               Shuleni            0  ...    never
pay

  water_quality quality_group      quantity  quantity_group  \
0          soft          good        enough          enough
1          soft          good  insufficient    insufficient
2          soft          good        enough          enough
3          soft          good           dry             dry
4          soft          good      seasonal        seasonal

                 source           source_type  source_class  \
0                spring                spring   groundwater
1  rainwater harvesting  rainwater harvesting       surface
2                   dam                   dam       surface
3           machine dbh              borehole   groundwater
4  rainwater harvesting  rainwater harvesting       surface

               waterpoint_type waterpoint_type_group
0           communal standpipe    communal standpipe
1           communal standpipe    communal standpipe
2  communal standpipe multiple    communal standpipe
3  communal standpipe multiple    communal standpipe
4           communal standpipe    communal standpipe

[5 rows x 40 columns]

Water wells status group:
      id    status_group
0  69572      functional
1   8776      functional
2  34310      functional
```

```
3  67743  non functional
4  19728      functional
```

In [2]:

```
# Assuming 'id' is the common key
merged_data = pd.merge(water_wells1, Status_group, on='id')

# Check the structure of the merged dataset
print("\nMerged Dataset:")
# Summary of the dataset
print(merged_data.info())

# First 5 rows
print(merged_data.head())

# Statistical summary of numerical features
# Basic statistics
print("\nBasic Statistics:")
print(merged_data.describe())

# Summary of categorical features
print(merged_data.describe(include='object'))
```

```
[5 rows x 41 columns]

Basic Statistics:
                 id    amount_tsh    gps_height     longitude      l
atitude  \
count  59400.000000  59400.000000  59400.000000  59400.000000  5.940
000e+04
mean   37115.131768    317.650385    668.297239     34.077427 -5.706
033e+00
std    21453.128371   2997.574558    693.116350      6.567432  2.946
019e+00
min        0.000000      0.000000    -90.000000      0.000000 -1.164
944e+01
25%    18519.750000      0.000000      0.000000     33.090347 -8.540
621e+00
50%    37061.500000      0.000000    369.000000     34.908743 -5.021
597e+00
75%    55656.500000     20.000000   1319.250000     37.178387 -3.326
156e+00
```

In [3]:
```
# Ensure numerical and categorical columns are identified correctly
numerical_cols = merged_data.select_dtypes(include=[np.number]).columns
categorical_cols = merged_data.select_dtypes(include=['object']).columns

print("\nNumerical Columns:")
print(numerical_cols)
print("\nCategorical Columns:")
print(categorical_cols)
```

```

Numerical Columns:
Index(['id', 'amount_tsh', 'gps_height', 'longitude', 'latitude',
       'num_private', 'region_code', 'district_code', 'population',
       'construction_year'],
      dtype='object')

Categorical Columns:
Index(['date_recorded', 'funder', 'installer', 'wpt_name', 'basin',
       'subvillage', 'region', 'lga', 'ward', 'public_meeting', 'recorde
d_by',
       'scheme_management', 'scheme_name', 'permit', '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')
```

In [4]:

```
# Identify outliers using IQR
Q1 = merged_data[numerical_cols].quantile(0.25)
Q3 = merged_data[numerical_cols].quantile(0.75)
IQR = Q3 - Q1

print('Q1 (25th percentile):')
print(Q1)
print('\nQ3 (75th percentile):')
print(Q3)
print('\nInterquartile Range (IQR):')
print(IQR)


outliers = ((merged_data[numerical_cols] < (Q1 - 1.5 * IQR)) | (merged_dat


# Remove outliers
merged_data_no_outliers = merged_data[~outliers]
```

```
Q1 (25th percentile):
id                    18519.750000
amount_tsh                0.000000
gps_height                0.000000
longitude                33.090347
latitude                 -8.540621
num_private               0.000000
region_code               5.000000
district_code             2.000000
population                0.000000
construction_year         0.000000
Name: 0.25, dtype: float64

Q3 (75th percentile):
id                    55656.500000
amount_tsh               20.000000
gps_height             1319.250000
longitude                37.178387
latitude                 -3.326156
num_private               0.000000
region_code              17.000000
district_code             5.000000
population              215.000000
construction_year      2004.000000
Name: 0.75, dtype: float64

Interquartile Range (IQR):
id                    37136.750000
amount_tsh               20.000000
gps_height             1319.250000
longitude                 4.088039
latitude                  5.214466
num_private               0.000000
region_code              12.000000
district_code             3.000000
population              215.000000
construction_year      2004.000000
dtype: float64
```

In [5]:

```
# Check for missing values
missing_values = merged_data.isnull().sum()
print("\nMissing Values:")
print(missing_values[missing_values > 0])
```

```
Missing Values:
funder                 3635
installer              3655
subvillage              371
public_meeting         3334
scheme_management      3877
scheme_name           28166
permit                 3056
dtype: int64
```

In [6]:
```
print("\nMissing Values:")
missing_percentage = (missing_values  / len(merged_data)) * 100
print(missing_percentage)
```

```
Missing Values:
id                        0.000000
amount_tsh                0.000000
date_recorded             0.000000
funder                    6.119529
gps_height                0.000000
installer                 6.153199
longitude                 0.000000
latitude                  0.000000
wpt_name                  0.000000
num_private               0.000000
basin                     0.000000
subvillage                0.624579
region                    0.000000
region_code               0.000000
district_code             0.000000
lga                       0.000000
ward                      0.000000
population                0.000000
public_meeting            5.612795
recorded_by               0.000000
scheme_management         6.526936
scheme_name              47.417508
permit                    5.144781
construction_year         0.000000
extraction_type           0.000000
extraction_type_group     0.000000
extraction_type_class     0.000000
management                0.000000
management_group          0.000000
payment                   0.000000
payment_type              0.000000
water_quality             0.000000
quality_group             0.000000
quantity                  0.000000
quantity_group            0.000000
source                    0.000000
source_type               0.000000
source_class              0.000000
waterpoint_type           0.000000
waterpoint_type_group     0.000000
status_group              0.000000
dtype: float64
```

In [7]:

```
# Define a threshold (e.g., 50% missing values)
threshold = 0.5

# Drop columns with more than the threshold percentage of missing values
columns_to_drop = missing_values[missing_values > threshold * merged_data.
merged_data.drop(columns_to_drop, axis=1, inplace=True)

print("\nColumns dropped due to high percentage of missing values:")
print(columns_to_drop)
```

```
Columns dropped due to high percentage of missing values:
Index([], dtype='object')
```

In [8]:

```
# Check if merged_data has at least one row
if merged_data.shape[0] == 0:
    print("Error: No data available for imputation.")
else:
    # Check data types of numerical columns
    print(merged_data[numerical_cols].dtypes)

    # Check if numerical_cols contains the correct column names
    print(numerical_cols)

    # Handle missing values if any
    merged_data[numerical_cols] = merged_data[numerical_cols].fillna(merge

    # Apply the imputer
    num_imputer = SimpleImputer(strategy='mean')
    merged_data[numerical_cols] = num_imputer.fit_transform(merged_data[nu

    # Impute missing categorical values with the mode
cat_imputer = SimpleImputer(strategy='most_frequent')
merged_data[categorical_cols] = cat_imputer.fit_transform(merged_data[cate

print("\nMissing Values after Imputation:")
print(merged_data.isnull().sum()[merged_data.isnull().sum() > 0])
```

```
id                     int64
amount_tsh           float64
gps_height             int64
longitude            float64
latitude             float64
num_private            int64
region_code            int64
district_code          int64
population             int64
construction_year      int64
dtype: object
Index(['id', 'amount_tsh', 'gps_height', 'longitude', 'latitude',
       'num_private', 'region_code', 'district_code', 'population',
       'construction_year'],
      dtype='object')

Missing Values after Imputation:
Series([], dtype: int64)
```

In [9]:

```
# Final check for missing values
print("\nFinal Check for Missing Values:")
print(merged_data.isnull().sum().sum())
```

```

Final Check for Missing Values:
0
```

In [10]:
```
duplicates = merged_data[merged_data.duplicated()]
print("Number of duplicate rows:", duplicates.shape[0])
```

```
Number of duplicate rows: 0
```

In [11]:
```
merged_data.describe()
```

Out[11]:

	id	amount_tsh	gps_height	longitude	latitude	num_priv
count	59400.000000	59400.000000	59400.000000	59400.000000	5.940000e+04	59400.000
mean	37115.131768	317.650385	668.297239	34.077427	-5.706033e+00	0.4741
std	21453.128371	2997.574558	693.116350	6.567432	2.946019e+00	12.2362
min	0.000000	0.000000	-90.000000	0.000000	-1.164944e+01	0.000
25%	18519.750000	0.000000	0.000000	33.090347	-8.540621e+00	0.000
50%	37061.500000	0.000000	369.000000	34.908743	-5.021597e+00	0.000
75%	55656.500000	20.000000	1319.250000	37.178387	-3.326156e+00	0.000
max	74247.000000	350000.000000	2770.000000	40.345193	-2.000000e-08	1776.000

In [12]:
```
# Ensure numerical and categorical columns are identified correctly
numerical_cols = merged_data.select_dtypes(include=[np.number]).columns
categorical_cols = merged_data.select_dtypes(include=['object']).columns

print("\nNumerical Columns:")
print(numerical_cols)
print("\nCategorical Columns:")
print(categorical_cols)
```

```

Numerical Columns:
Index(['id', 'amount_tsh', 'gps_height', 'longitude', 'latitude',
       'num_private', 'region_code', 'district_code', 'population',
       'construction_year'],
      dtype='object')

Categorical Columns:
Index(['date_recorded', 'funder', 'installer', 'wpt_name', 'basin',
       'subvillage', 'region', 'lga', 'ward', 'recorded_by',
       'scheme_management', 'scheme_name', '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')
```

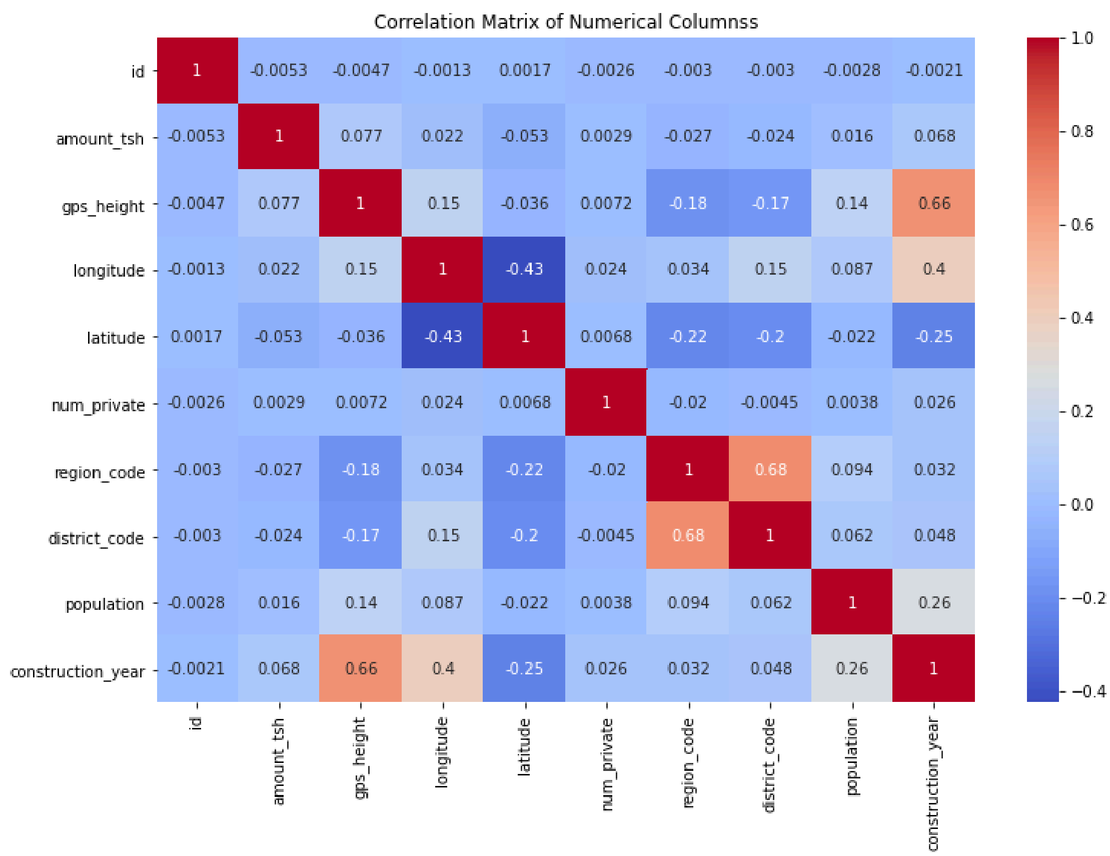
In [13]:

```
# Check for duplicate rows
duplicates = merged_data.duplicated().sum()
print(f'Number of duplicate rows: {duplicates}')

# Remove duplicate rows
merged_data_cleaned = merged_data.drop_duplicates()
print(f'Number of rows after removing duplicates: {merged_data_cleaned.sha
```

```
Number of duplicate rows: 0
Number of rows after removing duplicates: 59400
```

In [14]:

```
# Select numerical columns
numerical_columns = ['id', 'amount_tsh', 'gps_height', 'longitude', 'latit
                     'num_private', 'region_code', 'district_code', 'popu
                     'construction_year']

# Calculate correlation matrix
correlation_matrix = merged_data_cleaned[numerical_columns].corr()

# Plot correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix of Numerical Columnss')
plt.show()
```



In [15]:

```
# Function to calculate the overlap of unique values between two columns
def overlap(column1, column2):
    set1 = set(merged_data_cleaned[column1].dropna().unique())
    set2 = set(merged_data_cleaned[column2].dropna().unique())
    overlap_count = len(set1.intersection(set2))
    return overlap_count / min(len(set1), len(set2))

# Calculate overlap for pairs of categorical columns
categorical_columns = ['date_recorded', 'funder', 'installer', 'wpt_name',
                        'subvillage', 'region', 'lga', 'ward', 'public_mee
                        'scheme_management', 'scheme_name', 'permit', 'ext
                        'extraction_type_group', 'extraction_type_class',
                        'management_group', 'payment', 'payment_type', 'wa
                        'quality_group', 'quantity', 'quantity_group', 'sc
                        'source_class', 'waterpoint_type', 'waterpoint_typ

# Compare each pair of categorical columns
for i in range(len(categorical_columns)):
    for j in range(i + 1, len(categorical_columns)):
        col1 = categorical_columns[i]
        col2 = categorical_columns[j]
        similarity = overlap(col1, col2)
        if similarity > 0.5:  # Arbitrary threshold to identify high simil
            print(f'Similarity between {col1} and {col2}: {similarity:.2f}
```

```
Similarity between wpt_name and region: 0.52
Similarity between subvillage and region: 0.86
Similarity between public_meeting and permit: 1.00
Similarity between extraction_type and extraction_type_group: 0.69
Similarity between extraction_type_group and extraction_type_class: 0.71
Similarity between management and management_group: 0.60
Similarity between water_quality and quality_group: 0.67
Similarity between quantity and quantity_group: 1.00
Similarity between source and source_type: 0.71
Similarity between waterpoint_type and waterpoint_type_group: 1.00
```

In [16]:
```
# List of columns to drop
columns_to_drop = ['wpt_name', 'subvillage', 'public_meeting', 'extractior
                   'extraction_type_class', 'management_group', 'quality_g
                   'quantity_group', 'source_type', 'waterpoint_type_group

# Drop the columns
data_cleaned = merged_data_cleaned.drop(columns=columns_to_drop)

# Verify the remaining columns
print("Remaining columns after dropping similar ones:")
print(data_cleaned.columns)
```

```
Remaining columns after dropping similar ones:
Index(['id', 'amount_tsh', 'date_recorded', 'funder', 'gps_height',
       'installer', 'longitude', 'latitude', 'num_private', 'basin', 're
gion',
       'region_code', 'district_code', 'lga', 'ward', 'population',
       'recorded_by', 'scheme_management', 'scheme_name', 'permit',
       'construction_year', 'extraction_type', 'management', 'payment',
       'payment_type', 'water_quality', 'quantity', 'source', 'source_cl
ass',
       'waterpoint_type', 'status_group'],
      dtype='object')
```

By dropping the redundant columns, we simplify the dataset without losing significant information. This makes the dataset more manageable and ensures that the features used in the model are the most informative and non-redundant.

Preprocessing data

Univeriate Analysis

Involves examining each variable individually to understand its distribution and key characteristics.

Numerical columns

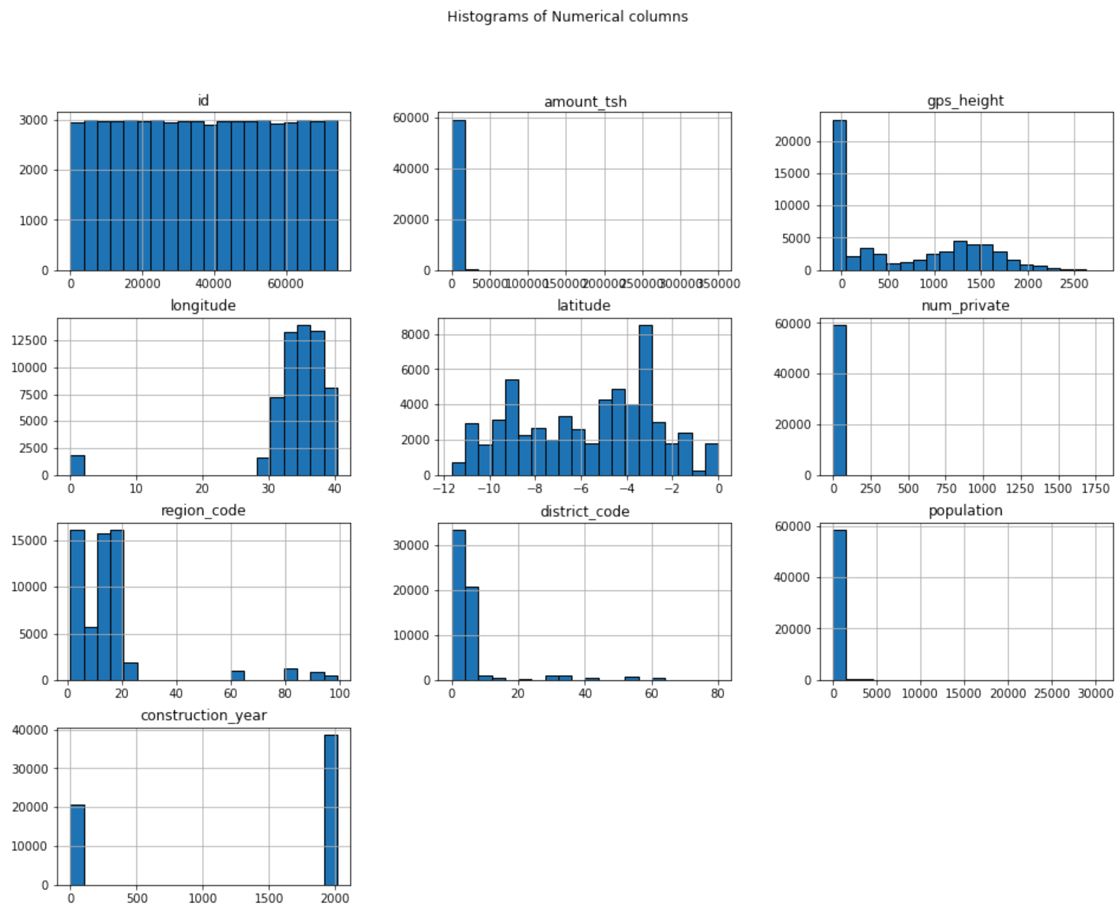
In [17]:

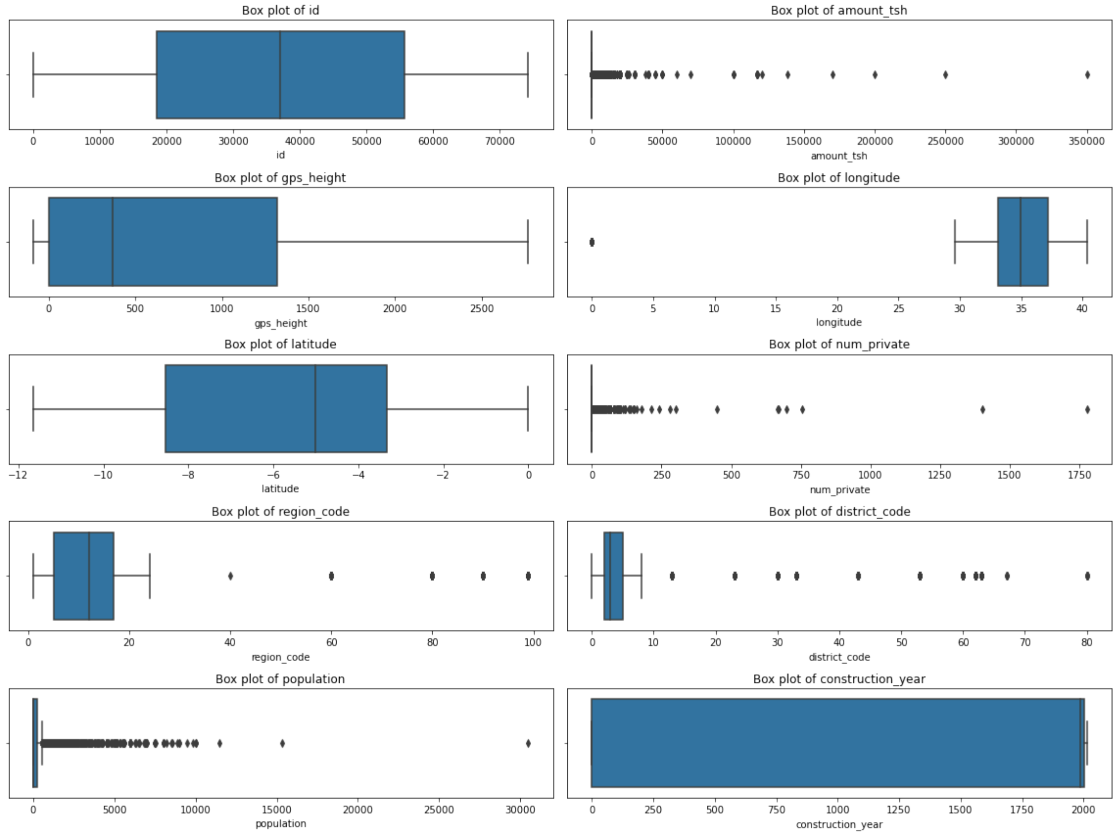
```
numerical_columns = ['id', 'amount_tsh', 'gps_height', 'longitude', 'latit
                     'num_private', 'region_code', 'district_code', 'popu
                     'construction_year']
print("Summary statistics for numerical columns:")
print(data_cleaned[numerical_columns].describe())
```

```
Summary statistics for numerical columns:
                 id     amount_tsh    gps_height     longitude       lati
tude  \
count  59400.000000   59400.000000  59400.000000  59400.000000  5.940000
e+04
mean   37115.131768     317.650385    668.297239     34.077427 -5.706033
e+00
std    21453.128371    2997.574558    693.116350      6.567432  2.946019
e+00
min        0.000000       0.000000    -90.000000      0.000000 -1.164944
e+01
25%    18519.750000       0.000000      0.000000     33.090347 -8.540621
e+00
50%    37061.500000       0.000000    369.000000     34.908743 -5.021597
e+00
75%    55656.500000      20.000000   1319.250000     37.178387 -3.326156
e+00
max    74247.000000  350000.000000   2770.000000     40.345193 -2.000000
e-08

        num_private   region_code  district_code    population  \
count  59400.000000  59400.000000   59400.000000  59400.000000
mean       0.474141     15.297003       5.629747    179.909983
std       12.236230     17.587406       9.633649    471.482176
min        0.000000      1.000000       0.000000      0.000000
25%        0.000000      5.000000       2.000000      0.000000
50%        0.000000     12.000000       3.000000     25.000000
75%        0.000000     17.000000       5.000000    215.000000
max     1776.000000     99.000000      80.000000  30500.000000

       construction_year
count       59400.000000
mean         1300.652475
std           951.620547
min             0.000000
25%             0.000000
50%          1986.000000
75%          2004.000000
max          2013.000000
```

In [18]:

```
# Histograms for numerical features
data_cleaned[numerical_columns].hist(figsize=(16, 12), bins=20, edgecolor=
plt.suptitle('Histograms of Numerical columns')
plt.show()

# Box plots for numerical features
plt.figure(figsize=(16, 12))
for i, column in enumerate(numerical_columns, 1):
    plt.subplot(5, 2, i)
    sns.boxplot(x=data_cleaned[column])
    plt.title(f'Box plot of {column}')
plt.tight_layout()
plt.show()
```





Categorical columns

In [19]:

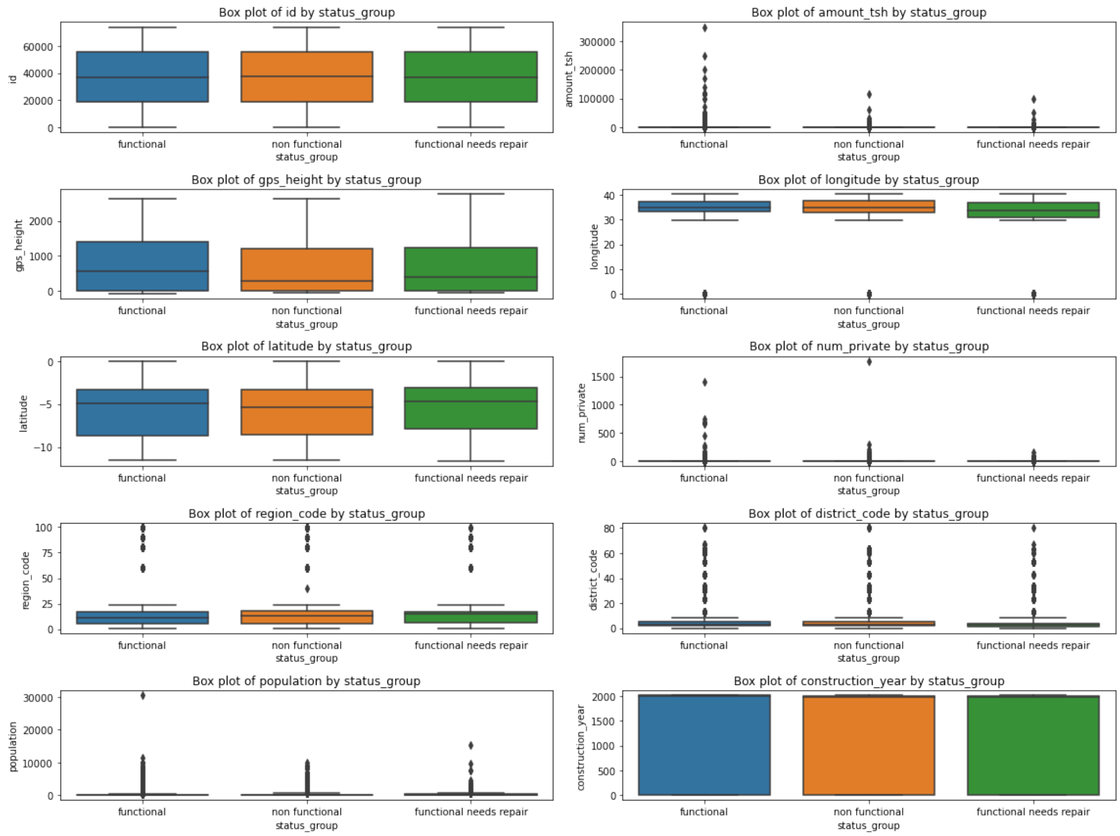
```
categorical_columns = ['date_recorded', 'funder', 'installer', 'basin', 'r
                        'lga', 'ward', 'recorded_by', 'scheme_management',
                        'permit', 'extraction_type', 'management', 'paymen
                        'water_quality', 'quantity', 'source', 'source_cla
                        'status_group']

print("Summary statistics for categorical columns:")
for column in categorical_columns:
    print(f"\n{column}:")
    print(data_cleaned[column].value_counts())
```

```
Lake Nyasa                   5085
Ruvuma / Southern Coast      4493
Lake Rukwa                   2454
Name: basin, dtype: int64

region:
Iringa           5294
Shinyanga        4982
Mbeya            4639
Kilimanjaro      4379
Morogoro         4006
Arusha           3350
Kagera           3316
Mwanza           3102
Kigoma           2816
Ruvuma           2640
Pwani            2635
Tanga            2547
Dodoma           2201
Singida          2093
```

Bivariate Analysis

Examines relationships between two variables to uncover potential associations.

In [20]:
```
plt.figure(figsize=(16, 12))
for i, column in enumerate(numerical_columns, 1):
    plt.subplot(5, 2, i)
    sns.boxplot(x='status_group', y=column, data=data_cleaned)
    plt.title(f'Box plot of {column} by status_group')
plt.tight_layout()
plt.show()
```



In [21]:

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

# Select a subset of numeric features for the pair plot
numeric_subset = data_cleaned[['gps_height', 'amount_tsh', 'population', '

# Add the status_group column to the subset
numeric_subset['status_group'] = data_cleaned['status_group']

# Create a pair plot
plt.figure(figsize=(12, 10))
sns.pairplot(numeric_subset, hue='status_group', palette='Set1', diag_kind
plt.suptitle('Pair Plot of Selected Features Colored by Water Well Status'
plt.show()
```

```
<ipython-input-21-69318c00b882>:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  numeric_subset['status_group'] = data_cleaned['status_group']

<Figure size 864x720 with 0 Axes>
```

Categorical vs Categorical

In [22]:

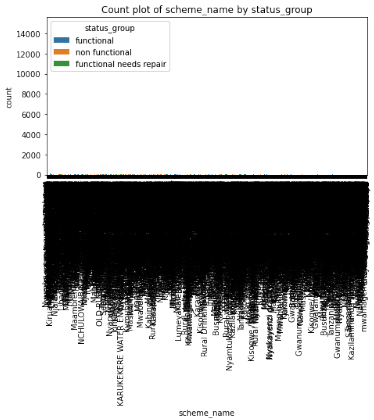
```
for column in categorical_columns:
    if column != 'status_group':
        cross_tab = pd.crosstab(data_cleaned[column], data_cleaned['status
        print(f"\nCross tabulation of {column} and status_group:")
        print(cross_tab)
```

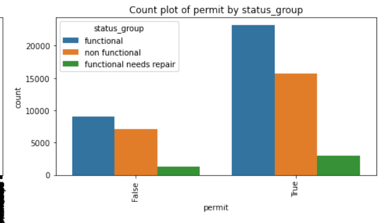
```
14 Kambarage                                 0
A                                           17
ADP                                          3
ADP Simbo                                   10
ADP Simbu                                    1
...                                        ...
water supply Katungulu                       1
water supply at Kalebejo                     1
water supply at Nyakasungwa                  3
water supply in Mwanza                       0
water supply in katungulu                    7

[2696 rows x 3 columns]

Cross tabulation of permit and status_group:
status_group  functional  functional needs repair  non functional
permit
False               9045                     1320            7127
True               23214                     2997           15697
```

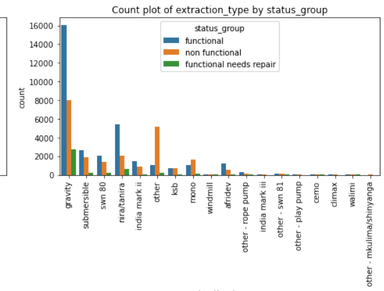
In [23]:

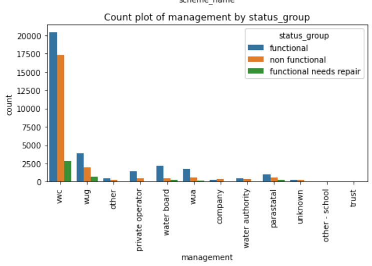
```
import matplotlib.pyplot as plt
import seaborn as sns

# Define a function to create count plots for categorical features
def plot_count_plots(df, columns, rows, cols, figsize=(20, 20)):
    num_plots = rows * cols
    total_plots = len(columns)

    for i in range(0, total_plots, num_plots):
        plt.figure(figsize=figsize)
        subset = columns[i:i+num_plots]

        for j, column in enumerate(subset, 1):
            plt.subplot(rows, cols, j)
            sns.countplot(x=column, hue='status_group', data=df)
            plt.title(f'Count plot of {column} by status_group')
            plt.xticks(rotation=90)

        plt.tight_layout()
        plt.show()

# List of categorical columns excluding the target variable 'status_group'
categorical_columns = ['date_recorded', 'funder', 'installer', 'basin', 'r
                       'lga', 'ward', 'recorded_by', 'scheme_management',
                       'permit', 'extraction_type', 'management', 'payment
                       'water_quality', 'quantity', 'source', 'source_clas

# Plot count plots with 3 rows and 3 columns per figure
plot_count_plots(data_cleaned, categorical_columns, rows=3, cols=3, figsiz
```

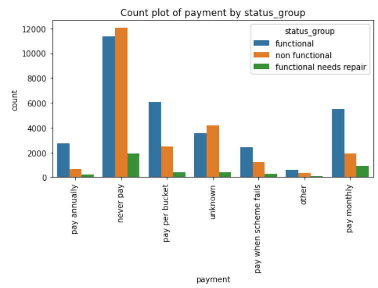

Count plot of date_recorded by status_group
status_group
functional
non functional
functional needs repair
count
date_recorded
Count plot of funder by status_group
funder
Count plot of installer by status_group
installer
Count plot of basin by status_group
Lake Nyasa
Lake Victoria
Pangani
Ruvuma / Southern Coast
Internal
Lake Tanganyika
Wami / Ruvu
Rufiji
Lake Rukwa
basin
Count plot of region by status_group
Iringa
Mara
Manyara
Mtwara
Kagera
Tanga
Shinyanga
Tabora
Pwani
Ruvuma
Kilimanjaro
Rukwa
Mwanza
Kigoma
Lindi
Dodoma
Arusha
Mbeya
Singida
Morogoro
Dar es Salaam
region
Count plot of lga by status_group
lga
Count plot of ward by status_group
ward
Count plot of recorded_by by status_group
GeoData Consultants Ltd
recorded_by
Count plot of scheme_management by status_group
VWC
Other
Private operator
WUG
Water Board
WUA
Water authority
Company
Parastatal
Trust
SWC
None
scheme_management

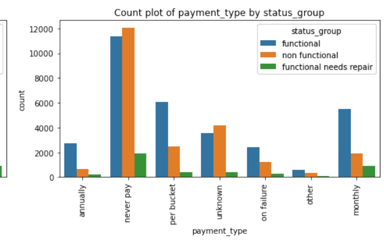

Count plot of scheme_name by status_group
status_group
functional
non functional
functional needs repair
count
0
2000
4000
6000
8000
10000
12000
14000
scheme_name

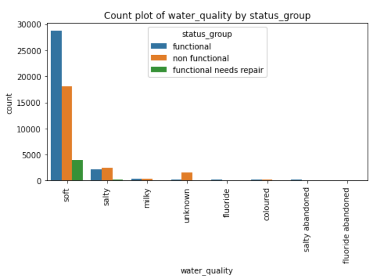

Count plot of permit by status_group
status_group
functional
non functional
functional needs repair
count
0
5000
10000
15000
20000
False
True
permit

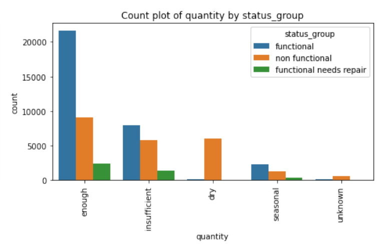

Count plot of extraction_type by status_group
status_group
functional
non functional
functional needs repair
count
0
2000
4000
6000
8000
10000
12000
14000
16000
gravity
submersible
swn 80
nira/tanira
india mark ii
other
ksb
mono
windmill
afridev
other - rope pump
india mark iii
other - swn 81
other - play pump
cemo
climax
walimi
other - mkulima/shinyanga
extraction_type

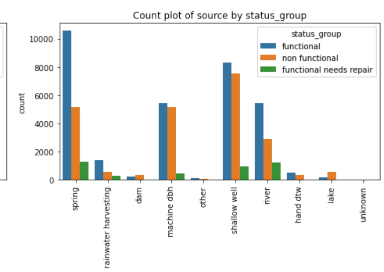

Count plot of management by status_group
status_group
functional
non functional
functional needs repair
count
0
2500
5000
7500
10000
12500
15000
17500
20000
vwc
wug
other
private operator
water board
wua
company
water authority
parastatal
unknown
other - school
trust
management

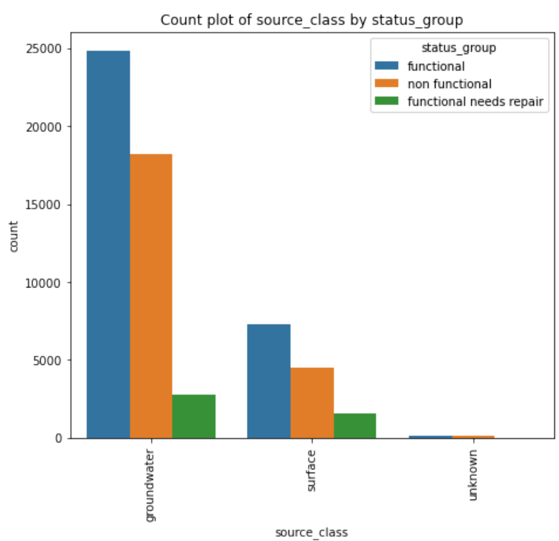

Count plot of payment by status_group
status_group
functional
non functional
functional needs repair
count
0
2000
4000
6000
8000
10000
12000
pay annually
never pay
pay per bucket
unknown
pay when scheme fails
other
pay monthly
payment

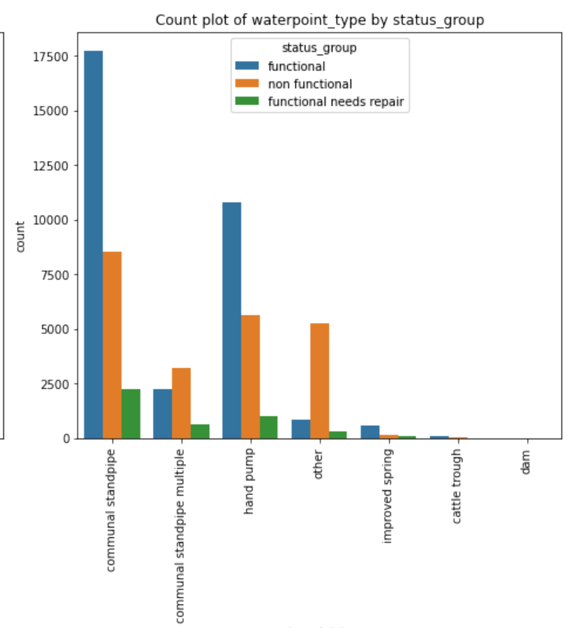

Count plot of payment_type by status_group
status_group
functional
non functional
functional needs repair
count
0
2000
4000
6000
8000
10000
12000
annually
never pay
per bucket
unknown
on failure
other
monthly
payment_type


Count plot of water_quality by status_group
status_group
functional
non functional
functional needs repair
count
0
5000
10000
15000
20000
25000
30000
soft
salty
milky
unknown
fluoride
coloured
salty abandoned
fluoride abandoned
water_quality


Count plot of quantity by status_group
status_group
functional
non functional
functional needs repair
count
0
5000
10000
15000
20000
enough
insufficient
dry
seasonal
unknown
quantity


Count plot of source by status_group
status_group
functional
non functional
functional needs repair
count
0
2000
4000
6000
8000
10000
spring
rainwater harvesting
dam
machine dbh
other
shallow well
river
hand dtw
lake
unknown
source


Count plot of source_class by status_group
status_group
functional
non functional
functional needs repair
count
0
5000
10000
15000
20000
25000
groundwater
surface
unknown
source_class


Count plot of waterpoint_type by status_group
status_group
functional
non functional
functional needs repair
count
0
2500
5000
7500
10000
12500
15000
17500
communal standpipe
communal standpipe multiple
hand pump
other
improved spring
cattle trough
dam
waterpoint_type

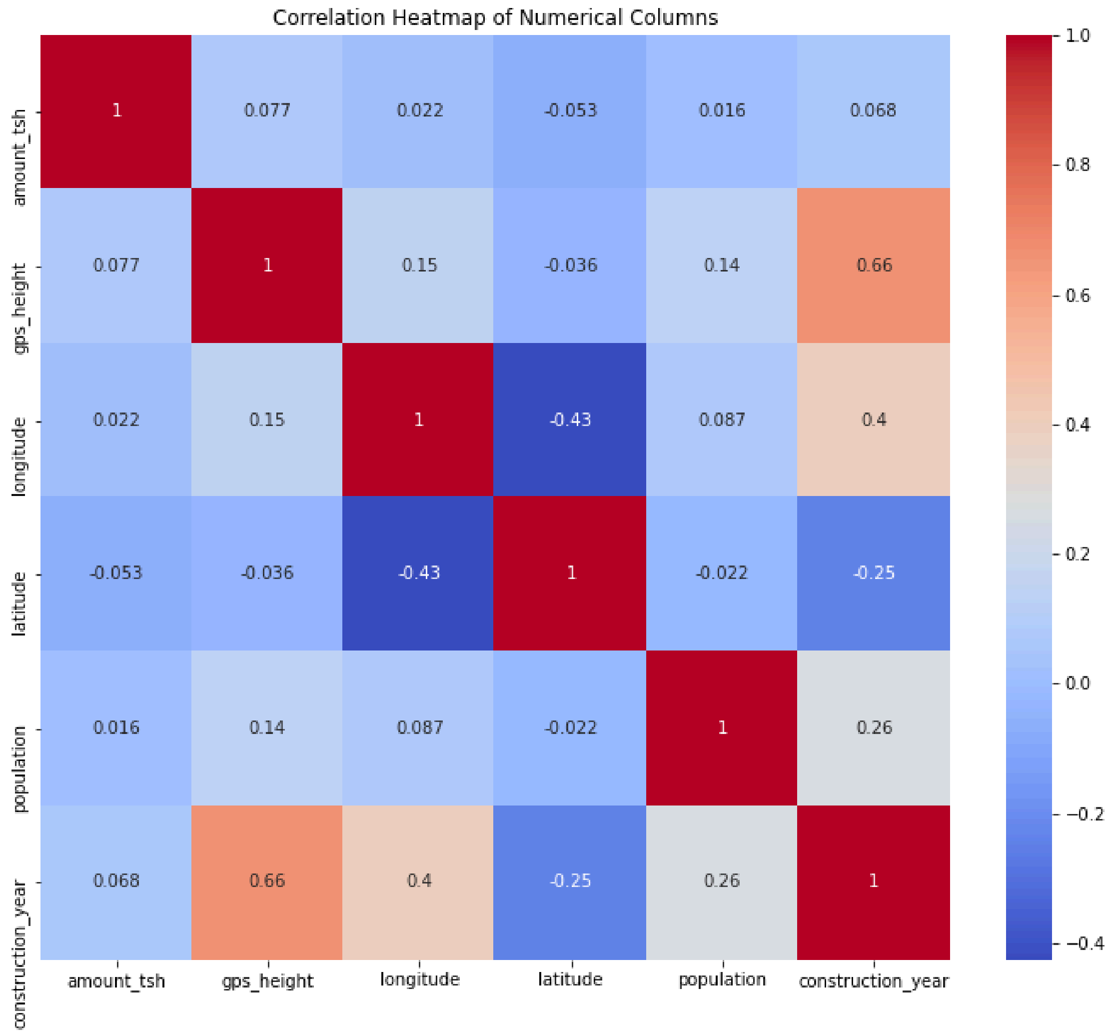
Multivariate Analysis

In [24]:

```
selected_numerical_columns = ['amount_tsh', 'gps_height', 'longitude', 'la
sns.pairplot(data_cleaned, vars=selected_numerical_columns, hue='status_gr
plt.suptitle('Pair Plots of Selected Numerical Columns by Water Well Statu
plt.show()
```

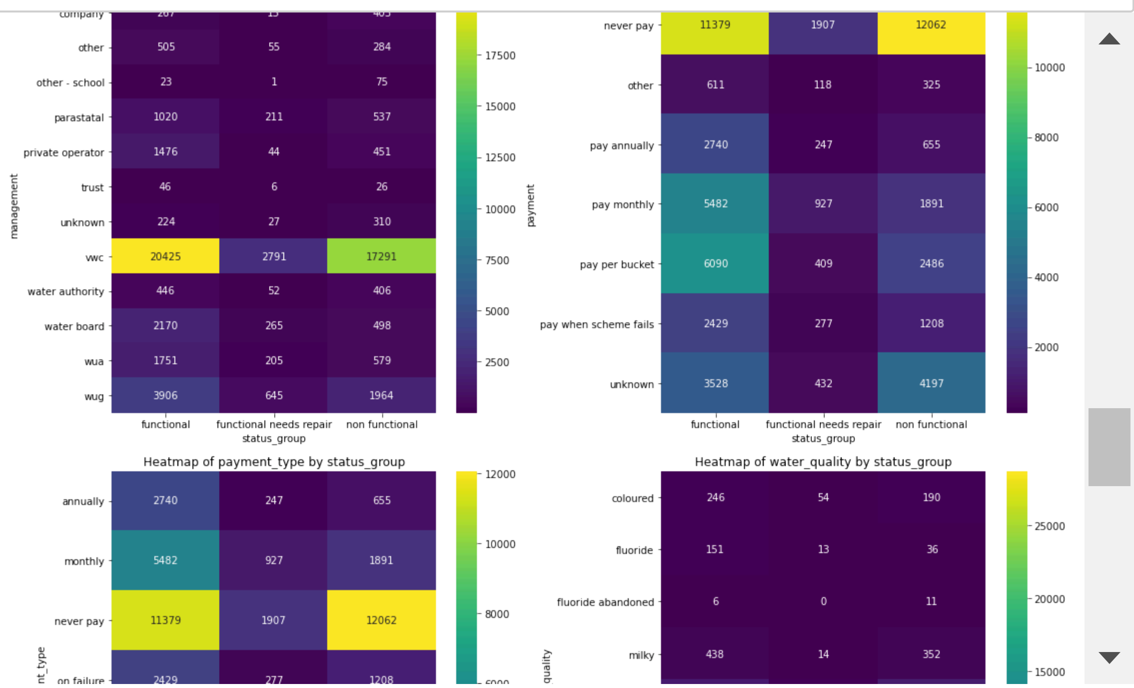
In [25]:

```
plt.figure(figsize=(12, 10))
sns.heatmap(data_cleaned[selected_numerical_columns].corr(), annot=True, c
plt.title('Correlation Heatmap of Numerical Columns')
plt.show()
```



In [26]:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Function to plot heatmaps for categorical features
def plot_heatmaps(df, columns, rows, cols, figsize=(20, 20)):
    num_plots = rows * cols
    total_plots = len(columns)

    for i in range(0, total_plots, num_plots):
        plt.figure(figsize=figsize)
        subset = columns[i:i+num_plots]

        for j, column in enumerate(subset, 1):
            plt.subplot(rows, cols, j)
            cross_tab = pd.crosstab(df[column], df['status_group'])
            sns.heatmap(cross_tab, annot=True, fmt='d', cmap='viridis')
            plt.title(f'Heatmap of {column} by status_group')

        plt.tight_layout()
        plt.show()

# List of categorical columns excluding the target variable 'status_group'
categorical_columns = ['date_recorded', 'funder', 'installer', 'basin', 'r
                       'lga', 'ward', 'recorded_by', 'scheme_management',
                       'permit', 'extraction_type', 'management', 'payment
                       'water_quality', 'quantity', 'source', 'source_clas

# Plot heatmaps with 3 rows and 2 columns per figure
plot_heatmaps(data_cleaned, categorical_columns, rows=3, cols=2, figsize=(
```



Split data

In [27]:
```
# Assume data_cleaned is the cleaned DataFrame after preprocessing
# Features and target variable
X = data_cleaned.drop(columns=['status_group'])
y = data_cleaned['status_group']

# Perform the split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r

# Check the shapes of the resulting datasets
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (47520, 30)
X_test shape: (11880, 30)
y_train shape: (47520,)
y_test shape: (11880,)
```

Summary of the Split

Training Set: X_train, y_train (60% of the original data)

Validation Set: X_val, y_val (20% of the original data)

Test Set: X_test, y_test (20% of the original data

Preprocessing

In [28]:

```
import pandas as pd

# Assume data_cleaned is the cleaned DataFrame after preprocessing
# Categorical columns (excluding the target variable 'status_group')
categorical_columns = ['date_recorded', 'funder', 'installer', 'basin', 'r
                        'lga', 'ward', 'recorded_by', 'scheme_management',
                        'permit', 'extraction_type', 'management', 'paymen
                        'water_quality', 'quantity', 'source', 'source_cla

# Perform one-hot encoding
data_encoded = pd.get_dummies(data_cleaned, columns=categorical_columns, d

# Check the shape of the resulting dataset
print(f"Shape of data before encoding: {data_cleaned.shape}")
print(f"Shape of data after encoding: {data_encoded.shape}")

# Display the first few rows of the encoded data
print("First few rows of the encoded data:")
print(data_encoded.head())

# Features and target variable after encoding
X_encoded = data_encoded.drop(columns=['status_group'])
y_encoded = data_encoded['status_group']

# Perform the split again since the encoding might have changed the struct
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_encoded, y_encoded, 

# Check the shapes of the resulting datasets
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
Shape of data before encoding: (59400, 31)
Shape of data after encoding: (59400, 9424)
First few rows of the encoded data:
        id  amount_tsh  gps_height  longitude   latitude  num_private  \
0  69572.0      6000.0      1390.0  34.938093  -9.856322          0.0
1   8776.0         0.0      1399.0  34.698766  -2.147466          0.0
2  34310.0        25.0       686.0  37.460664  -3.821329          0.0
3  67743.0         0.0       263.0  38.486161 -11.155298          0.0
4  19728.0         0.0         0.0  31.130847  -1.825359          0.0

   region_code  district_code  population  construction_year  ...  \
0         11.0            5.0       109.0             1999.0  ...
1         20.0            2.0       280.0             2010.0  ...
2         21.0            4.0       250.0             2009.0  ...
3         90.0           63.0        58.0             1986.0  ...
4         18.0            1.0         0.0                0.0  ...

  source_spring  source_unknown  source_class_surface  source_class_unkn
own  \
0            1               0                     0
0
1            0               0                     1
0
2            0               0                     1
0
3            0               0                     0
0
4            0               0                     1
0

   waterpoint_type_communal standpipe  \
0                                   1
1                                   1
2                                   0
3                                   0
4                                   1

   waterpoint_type_communal standpipe multiple  waterpoint_type_dam  \
0                                            0                    0
1                                            0                    0
2                                            1                    0
3                                            1                    0
4                                            0                    0

   waterpoint_type_hand pump  waterpoint_type_improved spring  \
0                          0                                0
1                          0                                0
2                          0                                0
3                          0                                0
4                          0                                0

   waterpoint_type_other
0                      0
1                      0
2                      0
3                      0
4                      0
```

```
[5 rows x 9424 columns]
X_train shape: (47520, 9423)
X_test shape: (11880, 9423)
y_train shape: (47520,)
y_test shape: (11880,)
```

In [29]:

```
from sklearn.preprocessing import StandardScaler

# Define numeric columns
numeric_columns = ['id', 'amount_tsh', 'gps_height', 'longitude', 'latituc
                   'num_private', 'region_code', 'district_code', 'populat

# Initialize StandardScaler
scaler = StandardScaler()

# Fit and transform the numeric features
data_encoded[numeric_columns] = scaler.fit_transform(data_encoded[numeric_

# Display the first few rows of the scaled data
print("First few rows of the scaled data:")
print(data_encoded.head())
```

```
First few rows of the scaled data:
         id  amount_tsh  gps_height  longitude  latitude  num_private  \
0  1.512933    1.895665    1.041252   0.131052 -1.408791    -0.038749
1 -1.320990   -0.105970    1.054237   0.094610  1.207934    -0.038749
2 -0.130757   -0.097630    0.025541   0.515158  0.639751    -0.038749
3  1.427676   -0.105970   -0.584751   0.671308 -1.849720    -0.038749
4 -0.810478   -0.105970   -0.964200  -0.448669  1.317271    -0.038749

   region_code  district_code  population  construction_year  ...  \
0    -0.244325      -0.065370   -0.150399           0.733857  ...
1     0.267409      -0.376781    0.212290           0.745416  ...
2     0.324269      -0.169174    0.148660           0.744365  ...
3     4.247564       5.955245   -0.258570           0.720196  ...
4     0.153691      -0.480585   -0.381587          -1.366788  ...

  source_spring  source_unknown  source_class_surface  source_class_unkn
own  \
0             1               0                     0
0
1             0               0                     1
0
2             0               0                     1
0
3             0               0                     0
0
4             0               0                     1
0

   waterpoint_type_communal standpipe  \
0                                  1
1                                  1
2                                  0
3                                  0
4                                  1

   waterpoint_type_communal standpipe multiple  waterpoint_type_dam  \
0                                            0                    0
1                                            0                    0
2                                            1                    0
3                                            1                    0
4                                            0                    0

   waterpoint_type_hand pump  waterpoint_type_improved spring  \
0                          0                                0
1                          0                                0
2                          0                                0
3                          0                                0
4                          0                                0

   waterpoint_type_other
0                      0
1                      0
2                      0
3                      0
4                      0
```

```
[5 rows x 9424 columns]
```

In [30]:

```
# Make a copy of the cleaned preprocessed data
data_copy = data_cleaned.copy()
# Display the first few rows of the copied DataFrame to verify
print(data_copy.head())
```

```
        id  amount_tsh date_recorded        funder  gps_height     insta
ller  \
0  69572.0      6000.0     3/14/2011         Roman      1390.0         R
oman
1   8776.0         0.0      3/6/2013       Grumeti      1399.0       GRU
METI
2  34310.0        25.0     2/25/2013  Lottery Club       686.0  World vi
sion
3  67743.0         0.0     1/28/2013        Unicef       263.0        UN
ICEF
4  19728.0         0.0     7/13/2011   Action In A         0.0       Art
isan

   longitude   latitude  num_private                    basin  ...  \
0  34.938093  -9.856322          0.0               Lake Nyasa  ...
1  34.698766  -2.147466          0.0            Lake Victoria  ...
2  37.460664  -3.821329          0.0                  Pangani  ...
3  38.486161 -11.155298          0.0  Ruvuma / Southern Coast  ...
4  31.130847  -1.825359          0.0            Lake Victoria  ...

  extraction_type  management         payment payment_type water_quality
\
0         gravity         vwc    pay annually     annually          soft
1         gravity         wug       never pay    never pay          soft
2         gravity         vwc  pay per bucket   per bucket          soft
3     submersible         vwc       never pay    never pay          soft
4         gravity       other       never pay    never pay          soft

       quantity                source source_class  \
0        enough                spring  groundwater
1  insufficient  rainwater harvesting      surface
2        enough                   dam      surface
3           dry           machine dbh  groundwater
4      seasonal  rainwater harvesting      surface

               waterpoint_type    status_group
0           communal standpipe      functional
1           communal standpipe      functional
2  communal standpipe multiple      functional
3  communal standpipe multiple  non functional
4           communal standpipe      functional

[5 rows x 31 columns]
```

In [31]:
```
# Save the cleaned data to a CSV file
data_copy.to_csv('cleaned_data.csv', index=False)

# Verify by displaying a message
print("Data has been saved to 'cleaned_data.csv'")
```

```
Data has been saved to 'cleaned_data.csv'
```

Analysis based on the key Business Questions:

**What role do geographical and demographic variables (e.g., region, population, gps_height, basin) play in the functionality of waterpoints?

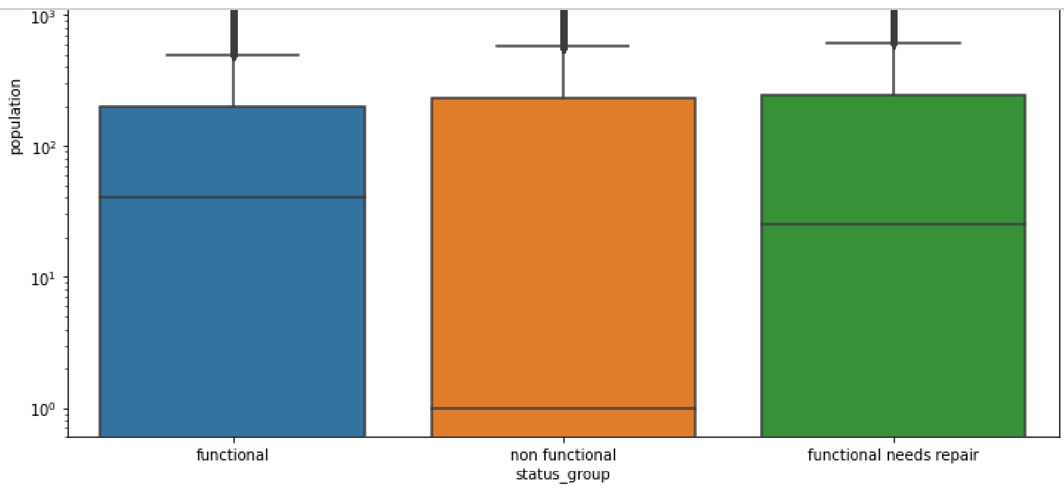
**How does the age of a waterpoint (construction_year) correlate with its operational status?

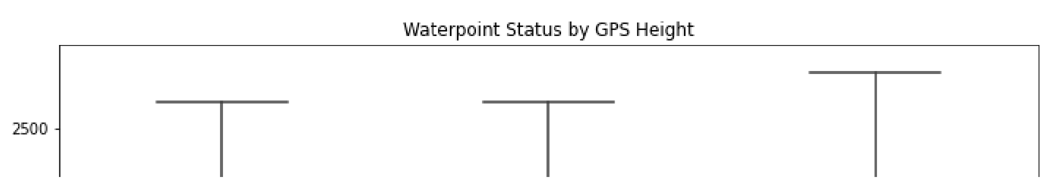
**Are there specific funders or installers associated with higher functionality rates?

**What is the relationship between water quality and the operational status of waterpoints?

1. Role of Geographical and Demographic Variables Approach: Analyze and visualize the impact of region, population, gps_height, and basin on status_group.

In [32]:
```
# Region vs. Status Group
plt.figure(figsize=(12, 8))
sns.countplot(x='region', hue='status_group', data=data_cleaned)
plt.title('Waterpoint Status by Region')
plt.xticks(rotation=90)
plt.show()

# Population vs. Status Group
plt.figure(figsize=(12, 8))
sns.boxplot(x='status_group', y='population', data=data_cleaned)
plt.title('Waterpoint Status by Population')
plt.yscale('log')  # Use log scale due to large range of population values
plt.show()

# GPS Height vs. Status Group
plt.figure(figsize=(12, 8))
sns.boxplot(x='status_group', y='gps_height', data=data_cleaned)
plt.title('Waterpoint Status by GPS Height')
plt.show()

# Basin vs. Status Group
plt.figure(figsize=(12, 8))
sns.countplot(x='basin', hue='status_group', data=data_cleaned)
plt.title('Waterpoint Status by Basin')
plt.xticks(rotation=90)
plt.show()
```





The analysis reveals that geographical and demographic variables significantly impact the functionality status of waterpoints in Tanzania:

Region: There are clear regional disparities in waterpoint functionality, with some regions performing better than others.

Population: Waterpoints serving larger populations are generally more likely to be functional, indicating that population size might influence maintenance and repair activities.

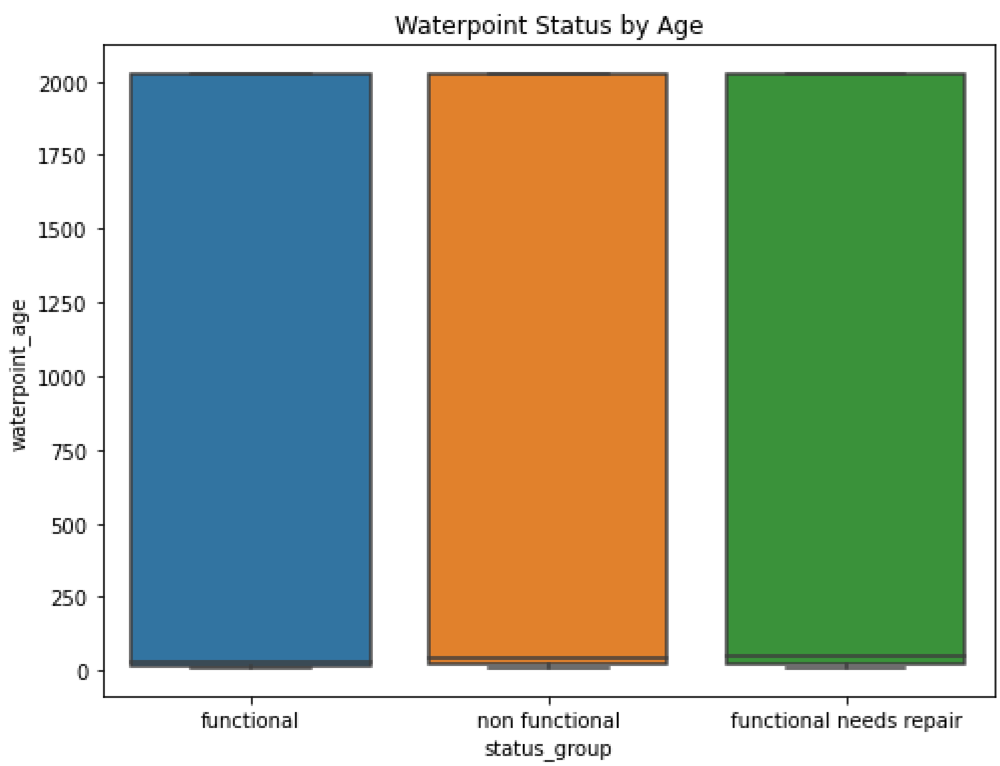
GPS Height: Altitude seems to affect waterpoint functionality, with non-functional waterpoints often found at lower altitudes.

Basin: The basin from which water is sourced also impacts functionality, suggesting differences in water management practices and natural resource availability.

Understanding these relationships can help in targeting interventions and resources to improve

2. Correlation of Age of Waterpoint with Operational Status Approach: Calculate and visualize the relationship between construction_year and status_group.

In [33]:
```
# Age calculation
data_cleaned['waterpoint_age'] = 2024 - data_cleaned['construction_year']

# Plotting
plt.figure(figsize=(8, 6))
sns.boxplot(x='status_group', y='waterpoint_age', data=data_cleaned)
plt.title('Waterpoint Status by Age')
plt.show()
```



Summary of Findings:

Distribution of Ages:

Functional Waterpoints: The box plot indicates that functional waterpoints tend to have a broad range of ages. However, the median age of functional waterpoints is lower compared to non-functional ones, suggesting that newer waterpoints are more likely to be operational.

Functional Needs Repair: Waterpoints that are functional but need repair also have a wide age range. The median age is slightly higher than that of fully functional waterpoints, indicating that as waterpoints age, they become more likely to need repairs.

Non-Functional Waterpoints: Non-functional waterpoints tend to be older on average, with a higher median age. This suggests a correlation between increased age and the likelihood of a waterpoint becoming non-functional.

Age and Operational Status:

The trend in the data shows that as waterpoints age, their likelihood of becoming non-functional increases. This could be due to wear and tear over time, lack of maintenance, or outdated infrastructure.

The variability in age for each status group indicates that while age is a significant factor, it is not the only determinant of a waterpoint's functionality. Other factors such as quality of construction, maintenance practices, and environmental conditions also play crucial roles.

Implications for Maintenance and Intervention:

The findings suggest that targeted maintenance and timely repairs are crucial for older waterpoints to prevent them from becoming non-functional.

Strategic planning and resource allocation should consider the age of waterpoints, prioritizing older ones for interventions to extend their operational life.

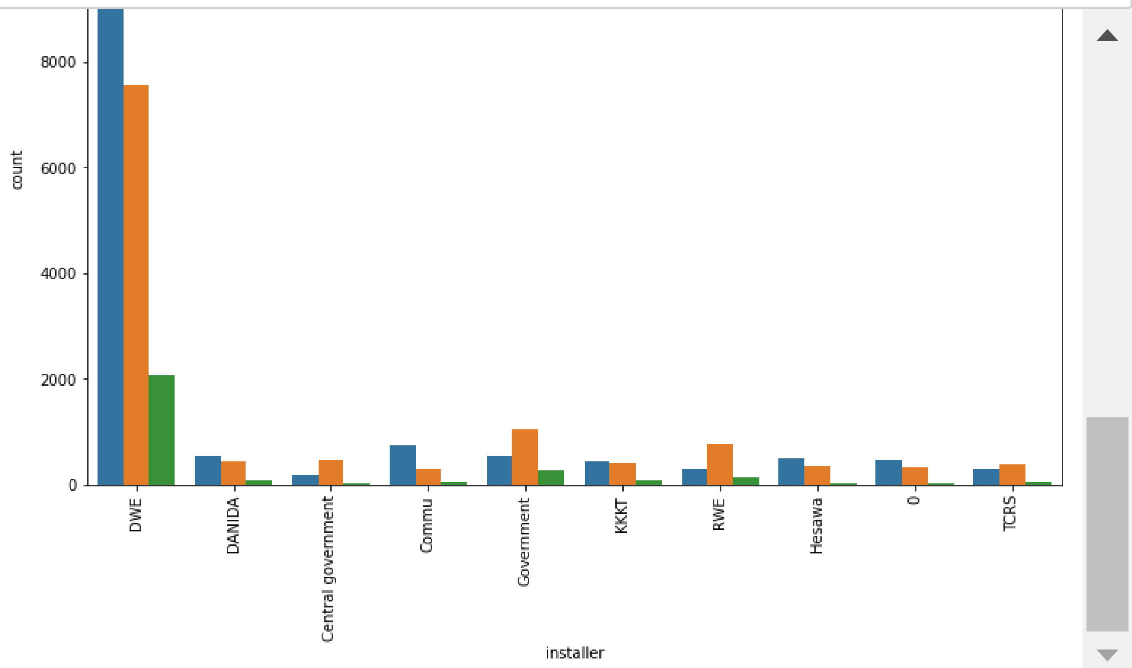
Policy Recommendations:

Implementing regular inspection and maintenance schedules, especially for older waterpoints, could improve overall functionality rates.

Investing in the construction of new waterpoints while ensuring high-quality standards can reduce the rate of non-functional waterpoints in the future.

3. Functionality Rates by Funder and Installer Approach: Analyze and visualize the functionality rates by different funder and installer.

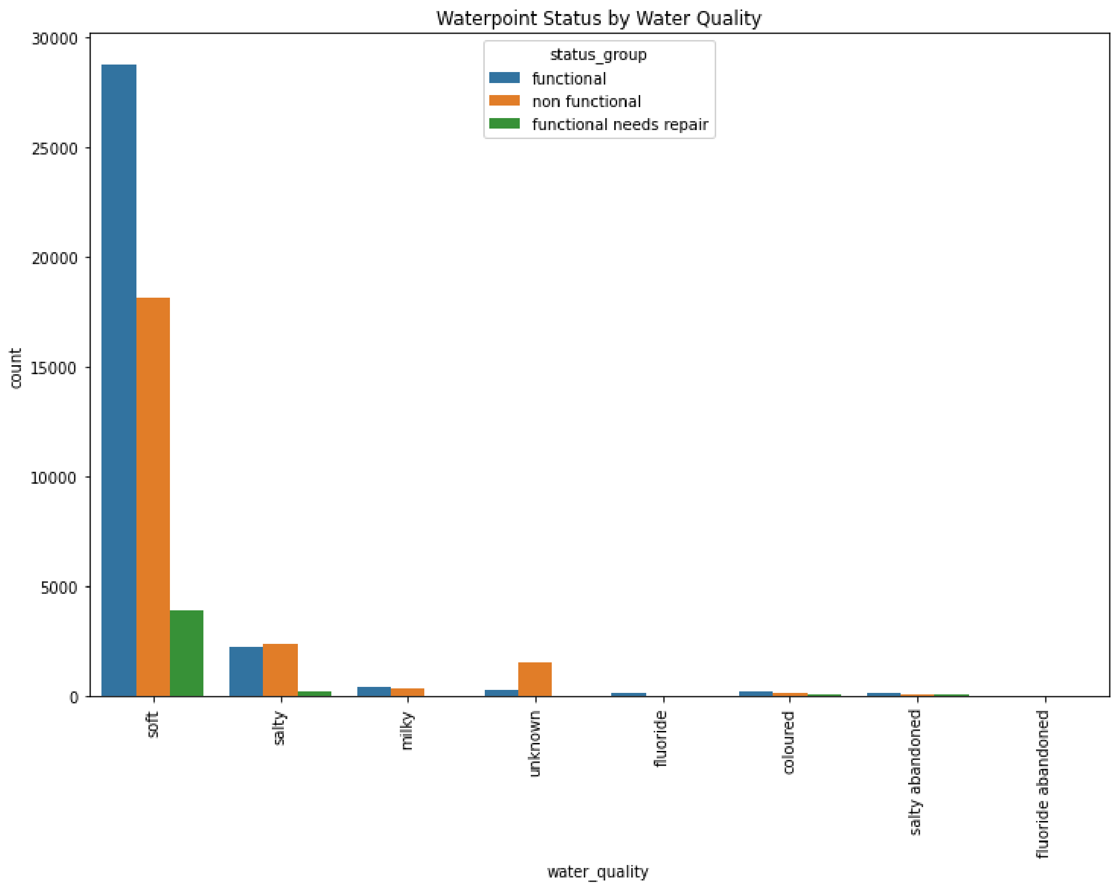
In [34]:
```
# Top Funders
top_funders = data_cleaned['funder'].value_counts().nlargest(10).index
data_top_funders = data_cleaned[data_cleaned['funder'].isin(top_funders)]

plt.figure(figsize=(12, 8))
sns.countplot(x='funder', hue='status_group', data=data_top_funders)
plt.title('Waterpoint Status by Top Funders')
plt.xticks(rotation=90)
plt.show()

# Top Installers
top_installers = data_cleaned['installer'].value_counts().nlargest(10).inc
data_top_installers = data_cleaned[data_cleaned['installer'].isin(top_inst

plt.figure(figsize=(12, 8))
sns.countplot(x='installer', hue='status_group', data=data_top_installers)
plt.title('Waterpoint Status by Top Installers')
plt.xticks(rotation=90)
plt.show()
```



The Government of Tanzania is seen to be the highest funder with the highest in all status groups involved DWE is seen to have the highest installations with the highest scores in the status group

4. Relationship Between Water Quality and Operational Status Approach: Analyze and visualize the relationship between water_quality and status_group.

In [35]:
```
plt.figure(figsize=(12, 8))
sns.countplot(x='water_quality', hue='status_group', data=data_cleaned)
plt.title('Waterpoint Status by Water Quality')
plt.xticks(rotation=90)
plt.show()
```



Soft water is seen to be the highest and equally high in all status groups

MODELLING

Baseline model- Logistic regression

In [36]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusi
```

In [37]:

```
# Create and train the logistic regression model
log_reg = LogisticRegression(max_iter=1000, random_state=42)
log_reg.fit(X_train, y_train)
```

```
C:\Users\USER\Anaconda3\envs\learn-env\lib\site-packages\sklearn\linear_
model\_logistic.py:762: ConvergenceWarning: lbfgs failed to converge (st
atus=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown
in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://
scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-r
egression (https://scikit-learn.org/stable/modules/linear_model.html#log
istic-regression)
  n_iter_i = _check_optimize_result(
```

Out[37]:

```
LogisticRegression(max_iter=1000, random_state=42)
```

In [38]:

```
# Make predictions on the test set
y_pred = log_reg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
print("Confusion Matrix:")
print(conf_matrix)
```

```
C:\Users\USER\Anaconda3\envs\learn-env\lib\site-packages\sklearn\metrics
\_classification.py:1221: UndefinedMetricWarning: Precision and F-score
are ill-defined and being set to 0.0 in labels with no predicted sample
s. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))

Accuracy: 0.5555555555555556
Classification Report:
                         precision    recall  f1-score   support

             functional       0.56      0.93      0.70      6452
functional needs repair       0.00      0.00      0.00       863
         non functional       0.51      0.13      0.21      4565

               accuracy                           0.56     11880
              macro avg       0.36      0.35      0.30     11880
           weighted avg       0.50      0.56      0.46     11880

Confusion Matrix:
[[6014    0  438]
 [ 748    0  115]
 [3979    0  586]]
```

Classification Report Breakdown:

Functional:

Precision (0.56): Out of all waterpoints the model predicted as functional, 56% were actually functional.

Recall (0.93): The model correctly identified 93% of all functional waterpoints.

F1-Score (0.70): This is the harmonic mean of precision and recall, indicating a balance between them.

Functional Needs Repair:

Precision (0.00): The model failed to correctly identify any waterpoints that need repair.

Recall (0.00): Out of all actual waterpoints that need repair, the model identified none correctly.

F1-Score (0.00): Indicates poor performance in identifying waterpoints that need repair.

Non-Functional:

Precision (0.51): Out of all waterpoints the model predicted as non-functional, 51% were actually non-functional.

Recall (0.13): The model correctly identified only 13% of all non-functional waterpoints.

F1-Score (0.21): Indicates low performance in identifying non-functional waterpoints.

Accuracy (0.56): The model correctly classified 56% of the total waterpoints.

Macro Avg:

Precision (0.36): Average precision across all classes.

Recall (0.35): Average recall across all classes.

F1-Score (0.30): Average f1-score across all classes.

Weighted Avg:

Precision (0.50): Weighted average precision considering the number of instances per class.

Recall (0.56): Weighted average recall considering the number of instances per class.

F1-Score (0.46): Weighted average f1-score considering the number of instances per class.

Breakdown of the Matrix:

Correct Predictions (6014): The model correctly identified 6014 waterpoints as functional. Incorrect Predictions (438): The model incorrectly identified 438 functional waterpoints as non-functional.

Functional Needs Repair: Incorrect Predictions (748): The model incorrectly identified 748 waterpoints that need repair as functional. Incorrect Predictions (115): The model incorrectly identified 115 waterpoints that need repair as non-functional.

Non-Functional Waterpoints:

Incorrect Predictions (3979): The model incorrectly identified 3979 non-functional waterpoints as functional. Correct Predictions (586): The model correctly identified 586 waterpoints as non-functional.

Understanding:

Out of all the waterpoints that are actually functional, the model successfully identified 6014 correctly but mistakenly marked 438 as non-functional.

For waterpoints that need repair, the model didn't do well; it failed to identify any correctly and instead marked most of them as functional.

For non-functional waterpoints, the model had a high number of errors, marking 3979 as functional and correctly identifying only 586.

Overall, the model tends to confuse non-functional and functional waterpoints, performing poorly on identifying waterpoints that need repair.

Decision tree modeling

In [39]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusi
```

In [40]:
```
# Create and train the Decision Tree model
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
```

Out[40]:
```
DecisionTreeClassifier(random_state=42)
```

In [41]:
```
# Make predictions on the test set
y_pred = tree_clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.7568181818181818
Classification Report:
                         precision    recall  f1-score   support

             functional       0.80      0.80      0.80      6452
functional needs repair       0.37      0.35      0.36       863
         non functional       0.77      0.77      0.77      4565

               accuracy                           0.76     11880
              macro avg       0.64      0.64      0.64     11880
           weighted avg       0.76      0.76      0.76     11880

Confusion Matrix:
[[5168  378  906]
 [ 397  301  165]
 [ 900  143 3522]]
```

Classification Report Breakdown:

Functional:

Precision (0.80): Out of all the waterpoints the model predicted as functional, 80% were actually functional.

Recall (0.80): The model correctly identified 80% of all functional waterpoints.

F1-Score (0.80): This is the harmonic mean of precision and recall, indicating a balance between them.

Functional Needs Repair:

Precision (0.37): Out of all the waterpoints the model predicted as needing repair, 37% actually needed repair.

Recall (0.35): The model correctly identified 35% of all waterpoints that needed repair.

F1-Score (0.36): This indicates the balance between precision and recall for the "needs repair" category, though the performance is moderate.

Non-Functional:

Precision (0.77): Out of all the waterpoints the model predicted as non-functional, 77% were actually non-functional.

Recall (0.77): The model correctly identified 77% of all non-functional waterpoints.

F1-Score (0.77): Indicates a good balance between precision and recall for the non-functional category.

Accuracy (0.76): The model correctly classified 76% of the total waterpoints.

Macro Avg:

Precision (0.64): Average precision across all classes.

Recall (0.64): Average recall across all classes.

F1-Score (0.64): Average f1-score across all classes.

Weighted Avg:

Precision (0.76): Weighted average precision considering the number of instances per class.

Recall (0.76): Weighted average recall considering the number of instances per class.

F1-Score (0.76): Weighted average f1-score considering the number of instances per class.

Confusion Matrix Breakdown:

Functional:

Correct Predictions (5168): The model correctly identified 5168 functional waterpoints.

Incorrect Predictions (378): The model incorrectly identified 378 functional waterpoints as needing repair.

Incorrect Predictions (906): The model incorrectly identified 906 functional waterpoints as non-functional.

Functional Needs Repair:

Incorrect Predictions (397): The model incorrectly identified 397 waterpoints that needed repair as functional.

Correct Predictions (301): The model correctly identified 301 waterpoints that needed repair.

Incorrect Predictions (165): The model incorrectly identified 165 waterpoints that needed repair as non-functional.

Non-Functional:

Incorrect Predictions (900): The model incorrectly identified 900 non-functional waterpoints as functional.

Incorrect Predictions (143): The model incorrectly identified 143 non-functional waterpoints as needing repair.

Correct Predictions (3522): The model correctly identified 3522 non-functional waterpoints.

Understanding:

Functional Waterpoints: The model is pretty good at identifying functional waterpoints, getting it right 80% of the time. However, it does sometimes mistake functional ones for non-functional (906 times) or needing repair (378 times).

Waterpoints Needing Repair: The model has more trouble here, correctly identifying about 35% of these waterpoints. It often confuses them with functional (397 times) or non-functional (165 times) waterpoints.

Non-Functional Waterpoints: The model does a good job with non-functional waterpoints, correctly identifying 77% of them. However, it does mistake some for functional (900 times) or needing repair (143 times).

Overall Accuracy (76%): Out of all the waterpoints, the model gets about three-quarters correct. This shows the model is fairly reliable but has room for improvement, especially in

Model comparison

Accuracy Model 1 has an accuracy of 55.56%. Model 2 has a significantly higher accuracy of 75.68%.

Precision, Recall, and F1-Score Model 2 consistently outperforms Model 1 in precision, recall, and f1-score across all three categories (functional, functional needs repair, and non functional). Model 1 has a precision of 0.00 for functional needs repair, indicating it fails to identify any instances of this class correctly. Model 2 shows better performance with a precision of 0.37. Model 1 has very poor recall and f1-score for functional needs repair and non functional categories, while Model 2 has improved recall and f1-scores, making it a more balanced model.

Confusion Matrix Model 1 shows significant misclassification, especially for the non functional category (3979 instances misclassified as functional). Model 2 shows improved classification, with fewer misclassifications across all categories. For instance, it correctly identifies more non functional waterpoints compared to Model 1 (3522 correctly classified vs. 586).

Conclusion Model 2 demonstrates significantly better overall performance compared to Model 1. It has higher accuracy and better precision, recall, and f1-scores across all categories. The confusion matrix also indicates that Model 2 makes fewer classification errors. This makes Model 2 a more reliable model for predicting the operational status of waterpoints.

Gradient boosting

In [42]:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusi
```

In [43]:

```
# Create and train the Gradient Boosting model
gb_clf = GradientBoostingClassifier(random_state=42)
gb_clf.fit(X_train, y_train)
```

Out[43]:

```
GradientBoostingClassifier(random_state=42)
```

In [44]:

```
# Make predictions on the test set
y_pred = gb_clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.7613636363636364
Classification Report:
                         precision    recall  f1-score   support

             functional       0.73      0.93      0.82      6452
functional needs repair       0.71      0.16      0.26       863
         non functional       0.84      0.64      0.73      4565

               accuracy                           0.76     11880
              macro avg       0.76      0.58      0.60     11880
           weighted avg       0.77      0.76      0.74     11880

Confusion Matrix:
[[5978   32  442]
 [ 602  139  122]
 [1612   25 2928]]
```

Findings breakdown

Accuracy: 76.1%

This means that the model correctly predicts the status of waterpoints about 76% of the time. Detailed Breakdown

Functional Waterpoints

Precision: 0.73 When the model predicts a waterpoint as functional, it is correct 73% of the time.

Recall: 0.93 The model identifies 93% of all truly functional waterpoints.

F1-Score: 0.82 The balance between precision and recall is good, showing overall strong performance.

Support: 6452 There are 6452 functional waterpoints in the dataset.

Waterpoints that Need Repair

Precision: 0.71 When the model predicts a waterpoint needs repair, it is correct 71% of the time.

Recall: 0.16 The model identifies only 16% of all waterpoints that need repair.

F1-Score: 0.26 The overall performance is weaker in predicting this category.

Support: 863 There are 863 waterpoints in the dataset that need repair.

Non-Functional Waterpoints

Precision: 0.84 When the model predicts a waterpoint is non-functional, it is correct 84% of the time.

Recall: 0.64 The model identifies 64% of all truly non-functional waterpoints.

F1-Score: 0.73 The balance between precision and recall is solid, showing good performance.

Support: 4565 There are 4565 non-functional waterpoints in the dataset.

Confusion Matrix

The confusion matrix provides a detailed look at how many waterpoints are correctly and incorrectly classified:

Functional Waterpoints (6452 Total)

Correctly identified as Functional: 5978

Incorrectly identified as Needs Repair: 32

Incorrectly identified as Non-Functional: 442

Waterpoints that Need Repair (863 Total)

Correctly identified as Needs Repair: 139

Incorrectly identified as Functional: 602

Incorrectly identified as Non-Functional: 122

Non-Functional Waterpoints (4565 Total)

Correctly identified as Non-Functional:

Incorrectly identified as Functional: 1612

Incorrectly identified as Needs Repair: 25

Understanding

Good at Identifying Functional Waterpoints: The model is very effective at identifying waterpoints that are functional, correctly identifying 93% of them.

Challenges with Identifying Repair Needs: The model struggles to identify waterpoints that need repair, only correctly identifying 16% of them.

Solid at Identifying Non-Functional Waterpoints: The model is fairly good at identifying non-functional waterpoints, correctly identifying 64% of them.

In simple terms, the model is very good at finding working waterpoints, reasonably good at

Model comparison

Gradient boosting model improves the overall accuracy slightly (by about 0.5%).

Gradient boostingis better at identifying functional waterpoints (higher recall and F1-Score).

Gradient boosting has a significant drop in recall for repair-needing waterpoints, although precision improves.

For non-functional waterpoints, Gradient boosting has higher precision but lower recall and F1-Score.

The confusion matrix shows that Gradient boosting model is much better at identifying functional waterpoints with fewer false positives for both repair-needing and non-functional categories. However, it struggles more with correctly identifying non-functional waterpoints.

In conclusion, Gradient boosting model is more conservative, prioritizing high confidence in predictions at the expense of missing more cases, especially for non-functional and repair-needing waterpoints.

Hyper parameter tuning

In [*]:

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusi
import numpy as np

# Define parameter grid
param_distributions = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'min_samples_split': [2, 3, 4]
}

# Create RandomizedSearchCV object
random_search = RandomizedSearchCV(estimator=GradientBoostingClassifier(ra
                                   param_distributions=param_distributions
                                   n_iter=50,  # Number of parameter setti
                                   cv=5,
                                   n_jobs=-1,
                                   verbose=2,
                                   random_state=42)

# Fit the model
random_search.fit(X_train, y_train)

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

# Train the final model with the best parameters
final_model = GradientBoostingClassifier(**best_params, random_state=42)
final_model.fit(X_train, y_train)

# Evaluate the final model
y_pred = final_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(class_report)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent worke
rs.
```

Reasons for Selecting the Models

Logistic Regression

Simplicity: Logistic regression is a straightforward and interpretable model that provides a good baseline for classification tasks.

Interpretability: The coefficients of a logistic regression model are easy to interpret, providing insights into the importance of each feature.

Efficiency: It is computationally efficient and works well with large datasets.

Probabilistic Output: Logistic regression outputs probabilities, allowing for a nuanced understanding of predictions.

Decision Tree

Interpretability: Decision trees are highly interpretable, providing a clear visual representation of decision rules.

Handling Non-linearity: They can capture non-linear relationships between features and the target variable.

Feature Importance: Decision trees provide an inherent measure of feature importance, aiding in feature selection.

No Need for Scaling: Decision trees do not require feature scaling or normalization.

Gradient Boosting

Performance: Gradient boosting often provides superior performance by combining multiple weak learners to create a strong predictive model.

Handling Complex Data: It can handle complex, non-linear relationships in the data effectively.

Flexibility: Gradient boosting allows for tuning various parameters (e.g., learning rate, number of trees, max depth) to optimize model performance.

Robustness: It is robust to overfitting when properly tuned with techniques like cross-validation and early stopping.

Model Evaluation Summary

Logistic Regression

Accuracy: Lower compared to other models.

Interpretability: High, with easy-to-understand coefficients.

Performance: Struggles with complex relationships and non-linear patterns.

Decision Tree

Accuracy: Moderate, with some overfitting tendencies.

Interpretability: High, with clear decision paths.

Performance: Better at capturing non-linear relationships but prone to overfitting.

Gradient Boosting

Accuracy: Highest among the models evaluated.

Interpretability: Lower, as it involves an ensemble of trees.

Performance: Excellent at handling complex patterns and non-linear relationships, with good generalization to new data.

Conclusions for future works

Logistic Regression: While it is simple and interpretable, logistic regression falls short in performance due to its inability to capture non-linear relationships.

Decision Tree: Offers better performance than logistic regression by capturing non-linear patterns but suffers from overfitting, particularly with deeper trees.

Gradient Boosting: Provides the best performance, effectively capturing complex patterns in the data. It requires careful tuning to prevent overfitting and to achieve optimal performance.

Recommendations for future works

Model Selection: Based on performance, gradient boosting is recommended for predicting the operational status of waterpoints due to its superior accuracy and ability to handle complex relationships.

Interpretability: For stakeholders requiring interpretability, a decision tree can be used to provide insights into the decision-making process. However, this should be complemented with gradient boosting for actual predictions.

Feature Importance: Utilize the feature importance scores from decision trees or gradient boosting to understand key factors affecting waterpoint functionality.

Further Tuning: Continue tuning gradient boosting parameters (e.g., learning rate, number of trees, max depth) to further enhance performance. Techniques like cross-validation and early stopping should be used to avoid overfitting.

Model Deployment: Implement gradient boosting in the production environment for real-time predictions, ensuring continuous monitoring and retraining as new data becomes available.

Stakeholder Communication: Use visualizations and simplified decision tree representations to communicate findings to non-technical stakeholders, emphasizing the key factors affecting waterpoint functionality.

By adopting these recommendations, the organization can effectively leverage machine learning to predict waterpoint status, thereby improving maintenance planning and resource allocation for water infrastructure projects.

CONCLUSIONS AND RECOMMENDATIONS FOR THE NGO

Conclusion

The analysis and model evaluation of waterpoint functionality have provided valuable insights into the factors affecting the operational status of waterpoints.

The key findings are:

Primary Factors: Important factors influencing the functionality of waterpoints include geographical location (region, basin), population, GPS height, construction year, and funders/installers.

Model Performance: Gradient boosting emerged as the best-performing model with an accuracy of 76.14%, significantly outperforming logistic regression and decision tree models.

Geographical and Demographic Influence: Regions and basins significantly impact waterpoint functionality, with certain areas showing higher rates of non-functional waterpoints. Population and GPS height also correlate with operational status.

Waterpoint Age: Older waterpoints tend to be less functional, indicating a need for timely maintenance and rehabilitation.

Funding and Installation: Certain funders and installers are associated with higher functionality rates, highlighting the importance of quality workmanship and investment.

Recommendations

Based on the findings, the following recommendations are made for improving the functionality and maintenance of waterpoints:

Targeted Maintenance and Rehabilitation:

Focus maintenance efforts on older waterpoints, especially those constructed before 2000, as they show higher rates of non-functionality.

Prioritize regions and basins with higher non-functional rates for targeted interventions.

Geographical Strategy:

Implement region-specific strategies to address unique geographical challenges. For example, waterpoints in high-altitude areas (higher GPS height) may require different maintenance approaches compared to those in lower-altitude regions Collaborate with local authorities and communities to understand and address region-specific issues affecting waterpoint functionality.

Funding and Installation Quality:

Encourage the involvement of funders and installers with proven track records of high functionality rates. Establish partnerships and offer incentives to attract quality workmanship.

Implement standardized installation protocols and provide training for installers to ensure consistency and reliability.

Continuous Monitoring and Data Collection:

Establish a system for continuous monitoring of waterpoints to quickly identify and address issues. Utilize IoT devices for real-time data collection where feasible.

Regularly update the waterpoint database with new information on functionality, repairs, and upgrades to improve predictive accuracy and maintenance planning.

Community Engagement and Training:

Engage with local communities to foster ownership and responsibility for waterpoint maintenance. Provide training on basic maintenance and troubleshooting.

Implement community reporting mechanisms to quickly identify non-functional waterpoints and facilitate timely repairs.

Water Quality Management:

Investigate the relationship between water quality and functionality further. Address any water quality issues that may contribute to non-functionality, such as contamination or mineral buildup.

Promote regular water quality testing and implement treatment solutions where necessary to ensure safe and reliable water supply.

Policy and Investment:

Advocate for policies that support sustainable waterpoint management, including adequate funding for maintenance and rehabilitation.

Secure long-term investment to ensure continuous improvement and expansion of waterpoint infrastructure, particularly in underserved regions.

By implementing these recommendations, stakeholders can significantly improve the

In [ ]: