Open in Colab

1.Environment Setup

1.1.Install Python Libraries

In [1]:
```
!pip install rasterio -q
!pip install seaborn -q
!pip install xlsxwriter -q
!pip install matplotlib-scalebar -q
!pip install matplotlib-map-utils -q
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 22.2/22.2 MB 38.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 169.4/169.4 kB 4.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 91.0/91.0 kB 5.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.7/11.7 MB 60.3 MB/s eta 0:00:00
```

1.2.Importing Libraries

In [2]:
```
import os
import sys
import glob
import time
import numba
import pickle
import rasterio
import xlsxwriter
import numba as nb
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.ticker as ticker

from rasterio.plot import show
from rasterio.mask import mask
from rasterio.enums import Resampling

from pyproj import Transformer

from matplotlib.ticker import FuncFormatter
from matplotlib_scalebar.scalebar import ScaleBar
from matplotlib_map_utils import scale_bar, north_arrow
from matplotlib.colors import ListedColormap, BoundaryNorm, Normalize, LinearSegmentedColorma
from matplotlib.patches import Patch, Rectangle, FancyArrowPatch

from sklearn.metrics import confusion_matrix
from tqdm import tqdm
```

1.3.Mounting Google Drive in Colab

In [3]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

2.Data Preparation

2.1.Setting Paths to Image Files

The user must include the year in the raster map name.

In [4]:
```
# List of input image paths
image_paths = [
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_1990.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_1995.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_2000.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_2005.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_2010.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_2015.tif',
    '/content/drive/MyDrive/change-components/input/lulc_mapbiomas_2020.tif'
]

# Raster mask
mask_path = '/content/drive/MyDrive/change-components/input/WesternBahiaMask.tif'
```

2.2.Setting Path to Output Files

In [5]:
```
# Setting Output Directory Path for Data Storage on Google Drive
output_path = "/content/drive/MyDrive/change-components/output"

# Check if the folder exists, if not, create the folder
if not os.path.exists(output_path):
    os.makedirs(output_path)
    print("Folder created:", output_path)
else:
    print("Folder already exists.")
```

```
Folder already exists.
```

2.3.Setting Years of the Time Intervals

In [6]:
```
years = [
    1990,
    1995,
    2000,
    2005,
    2010,
    2015,
    2020
    ]
```

2.4.Setting the classes

In [7]:
```
# Defining the ID and Name of the Classes
class_labels_dict = {
     0: {"name": "Backgroud",                 "rename": "Background", "color": "#ffffff"},
     4: {"name": "Savanna Formation",         "rename": "Savanna",    "color": "#ff0000"},
    12: {"name": "Grassland",                 "rename": "Grassland",  "color": "#8b0000"},
     3: {"name": "Forest Formation",          "rename": "Forest",     "color": "#ffffcc"},
    33: {"name": "River, Lake and Ocean",     "rename": "Water",      "color": "#ffff99"},
    46: {"name": "Coffee",                    "rename": "Coffee",     "color": "#ffff66"},
    29: {"name": "Rocky Outcrop",             "rename": "Rocky",      "color": "#ffff00"},
    11: {"name": "Wetland",                   "rename": "Wetland",    "color": "#cc9900"},
    24: {"name": "Urban Area",                "rename": "Urban",      "color": "#ffcccc"},
     9: {"name": "Forest Plantation",         "rename": "Plantation", "color": "#ffcc99"},
    48: {"name": "Other Perennial Crops",     "rename": "Perennial",  "color": "#ffcc66"},
    25: {"name": "Other non Vegetated Areas", "rename": "Barren",     "color": "#ffcc00"},
    21: {"name": "Mosaic of Uses",            "rename": "Mosaic",     "color": "#ff9900"},
```

```
    62: {"name": "Cotton (beta)",              "rename": "Cotton",     "color": "#ff66ff"},
    15: {"name": "Pasture",                    "rename": "Pasture",    "color": "#cc00cc"},
    41: {"name": "Other Temporary Crops",      "rename": "Temporary",  "color": "#0000ff"},
    39: {"name": "Soybean",                    "rename": "Soybean",    "color": "#00bfff"},
}
class_labels = [class_labels_dict[key]["rename"] for key in sorted(class_labels_dict.keys())]
```

2.4.Apply Mask

In [8]:

```
def apply_mask_to_images(image_paths, output_path, mask_path=None):
    """
    Apply a given mask to a series of image files and save the masked images as 8-bit TIFFs.

    Parameters:
    image_paths (list): List of paths to the images to which the mask will be applied.
    output_path (str): Directory to save the masked images.
    mask_path (str, optional): Path to the mask file. If None, no mask is applied.

    Returns:
    list: A list containing the paths to the saved masked images.
    """
    # Create output directory if it doesn't exist
    if not os.path.exists(output_path):
        os.makedirs(output_path)

    saved_paths = []

    # Load mask if provided
    mask_data = None
    if mask_path:
        with rasterio.open(mask_path) as mask_file:
            mask_data = mask_file.read(1)

    # Apply the mask to each image specified in the image_paths
    for path in image_paths:
        with rasterio.open(path) as image:
            meta = image.meta.copy()

            # Ensure the data type is uint8 and set the correct driver for TIFF files
            meta['dtype'] = 'uint8'
            meta['nodata'] = 0
            meta['driver'] = 'GTiff'
            meta['compress'] = 'lzw'

            # Read all bands of the image and apply mask if provided
            image_data = image.read(1)
            if mask_data is not None:
                masked_data = (image_data * (mask_data == 1)).astype('uint8')
            else:
                masked_data = image_data.astype('uint8')

            # Construct the path for the masked image
            base_name = os.path.basename(path).replace('.tif', '_masked.tif')
            masked_path = os.path.join(output_path, base_name)

            # Save the masked image
            with rasterio.open(masked_path, 'w', **meta) as dest:
                dest.write(masked_data, 1)

            saved_paths.append(masked_path)

    return saved_paths

masked_image_paths = apply_mask_to_images(image_paths, output_path, mask_path)
```

```python
# Print the paths of the saved masked images
for path in masked_image_paths:
    print(path)
```

```
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_1990_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_1995_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_2000_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_2005_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_2010_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_2015_masked.tif
/content/drive/MyDrive/change-components/output/lulc_mapbiomas_2020_masked.tif
```

2.5.Display the Maps

The code defines a function to find files with a given suffix and returns their paths. It then defines a function that plots each raster file with a specified color map, and a legend. It reads the raster data with rasterio, reshapes it, applies the color map, and arranges the images in a single row of subplots. It sets each plot's title with the provided year, hides the axes, and includes a custom legend. Finally, it saves the plot as a JPEG and displays it.

In [9]:
```python
def plot_classified_images(image_paths, class_map, years, output_path):
    """
    Plots saved classified images with legends and specified colors,
    inverting the legend order.

    Parameters:
    image_paths (list): Paths to the images to be plotted.
    class_map (dict): Dictionary containing class IDs with names and colors.
    years (list): List of years corresponding to each image.
    output_path (str): Directory where images are stored.
    """

    # Generate paths from years and suffix
    image_paths = [
        os.path.join(output_path, f)
        for year in years
        for f in os.listdir(output_path)
        if f.endswith(f"{year}_masked.tif")
    ]

    n_images = len(image_paths)
    fig, axs = plt.subplots(1,
                            n_images,
                            figsize=(15, 5),
                            sharey=True,
                            constrained_layout=True)

    # Prepare the colormap
    sorted_keys = sorted(class_map.keys())
    colors = [class_map[key]["color"] for key in sorted_keys]
    cmap = ListedColormap(colors)
    class_labels = [class_map[key]["rename"] for key in sorted_keys]
    class_values = sorted_keys

    # BoundaryNorm needs monotonically increasing boundaries
    norm = BoundaryNorm(class_values + [class_values[-1] + 1], cmap.N)

    # Plot each image
    for i, path in enumerate(image_paths):
        ax = axs[i] if n_images > 1 else axs
        with rasterio.open(path) as src:
            data = src.read(1, out_shape=(src.height // 4, src.width // 4),
                            resampling=Resampling.bilinear)
```

```
        ax.imshow(data,
                  cmap=cmap,
                  norm=norm)
        ax.set_title(f"{years[i]}",
                     fontweight='bold')
        ax.axis('off')

    # Custom legend elements
    legend_elements = [
    plt.Rectangle((0, 0), 1, 1,
                  color=class_labels_dict[key]["color"],
                  label=class_labels_dict[key]["rename"])
    for key in class_labels_dict if key != 0
    ]

    fig.legend(handles=legend_elements,
       loc='center right',
       bbox_to_anchor=(1.1, 0.5),
       frameon=False)

    # Get handles and labels from legend elements
    handles = [elem for elem in legend_elements]
    labels = [elem.get_label() for elem in legend_elements]

    fig.legend(
      handles=handles,
      labels=labels,
      loc='center right',
      bbox_to_anchor=(1.1, 0.5),
      frameon=False)

    # Add scale bar
    scalebar = ScaleBar(
        1/10000,
        units='km',
        length_fraction=0.4,
        location='lower right',
        scale_loc='bottom',
        color='black',
        box_alpha=0,
        scale_formatter=lambda value, _: f"{int(value)} km"
    )
    ax.add_artist(scalebar)

    # Add north arrow
    north_arrow(
        ax,
        location="upper right",
        shadow=False,
        rotation={"degrees":0}
    )

    plt.savefig(os.path.join(output_path,
                             "plot_input_maps.jpeg"),
                format='jpeg',
                bbox_inches="tight",
                dpi=300)
    plt.close()
    plt.show()

# Call the function to plot the images
plot_classified_images(masked_image_paths,
                       class_labels_dict,
```

```
                        years,
                        output_path)
```

3.Generate the Transition Matrix

In this section, the computer code will generate four different transition matrices. The first one is related to each time interval. The second one is the transition matrix for the temporal extent, which is represented for the first and last time point of the time extent. The third one is a transition matrix that represents the sum of all time intervals. And the last transition matrix is the alternation matrix, which is the sum matrix minus the extent matrix.

Before generate the transition matrix, the computer code will analyze the presence of 0, NULL and NA value in all maps. If there is a presence of one of these values, the computer code will create a mask with these values and will remove all the pixels in the same position in all maps and years.

All the transition matrix will be salved in the Google Drive in the ".csv" format.

In [10]:
```
# Generate the Transition Matrix
def generate_mask_and_flatten_rasters(output_path, suffix='_masked.tif'):
    """
    Reads rasters with a specific suffix from a directory, applies a mask where
    data is zero, missing, or NaN, and flattens the non-masked values.

    Parameters:
    output_path (str): Directory containing raster files to process.
    suffix (str): File suffix to identify relevant rasters.

    Returns:
    list of numpy arrays: Flattened arrays of non-masked raster data for further analysis.
    """
    image_paths = [os.path.join(output_path, f)
                   for f in os.listdir(output_path) if f.endswith(suffix)]
    image_paths.sort()

    all_data = []
    all_masks = []

    # Read and mask all rasters
    for path in image_paths:
        with rasterio.open(path) as src:
            data = src.read(1)
            mask = (data == 0) | (data == src.nodata) | np.isnan(data)
            all_masks.append(mask)
            all_data.append(data)

    # Create combined mask
    combined_mask = np.any(all_masks, axis=0)

    # Return flattened data
    return [data[~combined_mask].flatten() if np.any(combined_mask)
            else data.flatten() for data in all_data]

transition_matrix = confusion_matrix

def generate_all_matrices(output_path, suffix='_masked.tif'):
    """Generate all required matrices and save to CSV"""
    global years

    # Get processed data
    flattened_data = generate_mask_and_flatten_rasters(output_path, suffix)
```

```python
    # Validate year count
    if len(years) != len(flattened_data):
        raise ValueError(f"Mismatch: {len(years)} years vs {len(flattened_data)} rasters")

    # Get class labels from data
    all_classes = np.unique(np.concatenate(flattened_data)).astype(int)

    # Generate interval matrices
    for i in range(len(flattened_data)-1):
        cm = transition_matrix(flattened_data[i],
                               flattened_data[i+1],
                               labels=all_classes)
        pd.DataFrame(cm,
                     index=all_classes,
                     columns=all_classes
        ).to_csv(os.path.join(output_path,
                               f'transition_matrix_{years[i]}-{years[i+1]}.csv'))

    # Generate extent matrix
    extent_matrix = transition_matrix(flattened_data[0],
                                      flattened_data[-1],
                                      labels=all_classes)
    pd.DataFrame(extent_matrix,
                 index=all_classes,
                 columns=all_classes
    ).to_csv(os.path.join(output_path,
                           f'transition_matrix_extent_{years[0]}-{years[-1]}.csv'))

    # Generate sum matrix
    sum_matrix = np.zeros((len(all_classes), len(all_classes)), dtype=int)
    for i in range(len(flattened_data)-1):
        sum_matrix += transition_matrix(flattened_data[i],
                                        flattened_data[i+1],
                                        labels=all_classes)
    pd.DataFrame(sum_matrix,
                 index=all_classes,
                 columns=all_classes
    ).to_csv(os.path.join(output_path,
                           f'transition_matrix_sum_{years[0]}-{years[-1]}.csv'))

    # Generate alternation matrix
    alternation_matrix = sum_matrix - extent_matrix
    pd.DataFrame(alternation_matrix,
                 index=all_classes,
                 columns=all_classes
    ).to_csv(os.path.join(output_path,
                           f'transition_matrix_alternation_{years[0]}-{years[-1]}.csv'))

    return years, all_classes

def main(output_path):
    """Matrix generation workflow"""
    # Create output directory if needed
    os.makedirs(output_path, exist_ok=True)

    # Generate all matrices
    print("Generating transition matrices...")
    years, all_classes = generate_all_matrices(output_path)

    print(f"Detected classes: {all_classes}")
    print("Matrices saved in:", output_path)

if __name__ == "__main__":
    main(output_path)
```

```
Generating transition matrices...
Detected classes: [ 3  4  9 11 12 15 21 24 25 29 33 39 41 46 48 62]
Matrices saved in: /content/drive/MyDrive/change-components/output
```

4.Components of Change

The code calculates components of change from transition matrices generated in the previous step. It features a ComponentCalculator class that processes matrices to determine the gain and loss of Quantity, Exchange, and Shift. The process_matrix function handles matrices for defined time intervals and the main function systematically processes these matrices for each time interval, aggregates the results, and exports the outcomes to a CSV file.

In [11]:
```python
class ComponentCalculator:
    def __init__(self, transition_matrix):
        self.matrix = transition_matrix.astype(int)
        self.num_classes = transition_matrix.shape[0]
        self.class_components = []
        self.total_components = {
            'Quantity_Gain': 0, 'Quantity_Loss': 0,
            'Exchange_Gain': 0, 'Exchange_Loss': 0,
            'Shift_Gain': 0, 'Shift_Loss': 0
        }

    def calculate_components(self):
        for class_idx in range(self.num_classes):
            gain_sum = np.sum(self.matrix[:, class_idx])
            loss_sum = np.sum(self.matrix[class_idx, :])

            q_gain = max(0, gain_sum - loss_sum)
            q_loss = max(0, loss_sum - gain_sum)

            mutual = np.sum(np.minimum(
                self.matrix[class_idx, :],
                self.matrix[:, class_idx]
            ))
            exchange = mutual - self.matrix[class_idx, class_idx]

            total_trans = loss_sum - self.matrix[class_idx, class_idx]
            shift = max(0, total_trans - q_loss - exchange)

            self.class_components.append({
                'Quantity_Gain': q_gain,
                'Quantity_Loss': q_loss,
                'Exchange_Gain': exchange,
                'Exchange_Loss': exchange,
                'Shift_Gain': shift,
                'Shift_Loss': shift
            })
        return self

def process_matrix(matrix_type, start_year=None, end_year=None):
    results = []
    fname = None

    try:
        # Determine filename
        if matrix_type == 'interval':
            fname = f'transition_matrix_{start_year}-{end_year}.csv'
        elif matrix_type == 'extent':
            fname = f'transition_matrix_extent_{years[0]}-{years[-1]}.csv'
        elif matrix_type == 'sum':
            fname = f'transition_matrix_sum_{years[0]}-{years[-1]}.csv'
```

```python
        elif matrix_type == 'alternation':
            fname = f'transition_matrix_alternation_{years[0]}-{years[-1]}.csv'

        # Validate file path
        full_path = os.path.join(output_path, fname)
        if not os.path.exists(full_path):
            raise FileNotFoundError(f"File {full_path} does not exist")

        # Process data
        df = pd.read_csv(full_path, index_col=0)
        matrix_classes = [int(c) for c in df.index]
        calc = ComponentCalculator(df.values).calculate_components()

        # Build results
        for idx, class_id in enumerate(matrix_classes):
            cls_name = class_labels_dict.get(class_id, {}).get("rename", f"Unknown_{class_id}
            comp = calc.class_components[idx]

            for component in ['Quantity', 'Exchange', 'Shift']:
                component_name = component
                if matrix_type in ['extent', 'sum'] and component in ['Exchange', 'Shift']:
                    component_name = f'Allocation_{component}'

                results.append({
                    'Time_Interval': f"{start_year}-{end_year}" if matrix_type == 'interval'
                    'Class': cls_name,
                    'Component': component_name,
                    'Gain': comp[f'{component}_Gain'],
                    'Loss': comp[f'{component}_Loss']
                })

    except Exception as e:
        print(f"ERROR in {matrix_type}: {str(e)}")
        return []

    return results

def main(output_path):
    """Main execution flow with enhanced logging"""
    all_results = []
    print("\n=== Starting processing ===")

    # 1. Process interval matrices
    try:
        print("\nProcessing intervals...")
        for i in range(len(years)-1):
            start = years[i]
            end = years[i+1]
            print(f"  - {start}-{end}")
            all_results.extend(process_matrix('interval', start, end))

    except Exception as e:
        print(f"Fatal error in intervals: {str(e)}")
        return

    # 2. Process extent matrix
    try:
        print("\nProcessing extent matrix...")
        all_results.extend(process_matrix('extent', None, None))
    except Exception as e:
        print(f"Fatal error in extent: {str(e)}")
        return

    # 3. Process sum matrix
    try:
        print("\nProcessing sum matrix...")
```

```
        all_results.extend(process_matrix('sum', None, None))
    except Exception as e:
        print(f"Fatal error in sum: {str(e)}")
        return

    # 4. Process alternation matrix
    try:
        print("\nAttempting alternation matrix...")
        alternation_path = os.path.join(output_path,
            f'transition_matrix_alternation_{years[0]}-{years[-1]}.csv')

        if os.path.exists(alternation_path):
            df_alt = pd.read_csv(alternation_path, index_col=0)
            calc = ComponentCalculator(df_alt.values).calculate_components()

            for idx, class_id in enumerate(df_alt.index.astype(int)):
                cls_name = class_labels_dict.get(class_id, {}).get("rename", f"Unknown_{class_
                comp = calc.class_components[idx]

                all_results.extend([{
                    'Time_Interval': 'alternation',
                    'Class': cls_name,
                    'Component': 'Alternation_Exchange',
                    'Gain': comp['Exchange_Gain'],
                    'Loss': comp['Exchange_Loss']
                }, {
                    'Time_Interval': 'alternation',
                    'Class': cls_name,
                    'Component': 'Alternation_Shift',
                    'Gain': comp['Shift_Gain'],
                    'Loss': comp['Shift_Loss']
                }])
        else:
            print("Alternation matrix not found - skipping")

    except Exception as e:
        print(f"Non-fatal alternation error: {str(e)}")

    # 5. Final export
    try:
        output_file = os.path.join(output_path, 'change_components.csv')
        pd.DataFrame(all_results).to_csv(output_file, index=False)
        print(f"\n=== Success! Saved to: {output_file} ===")
        print(f"Total records: {len(all_results):,}")

    except Exception as e:
        print(f"\n!!! FATAL EXPORT ERROR: {str(e)} !!!")

if __name__ == "__main__":
    main(output_path)
```

```
=== Starting processing ===

Processing intervals...
  - 1990-1995
  - 1995-2000
  - 2000-2005
  - 2005-2010
  - 2010-2015
  - 2015-2020

Processing extent matrix...

Processing sum matrix...

Attempting alternation matrix...

=== Success! Saved to: /content/drive/MyDrive/change-components/output/change_components.csv =
==
Total records: 416
```

5.Graphics

5.1 Setting the parameters for the graphics

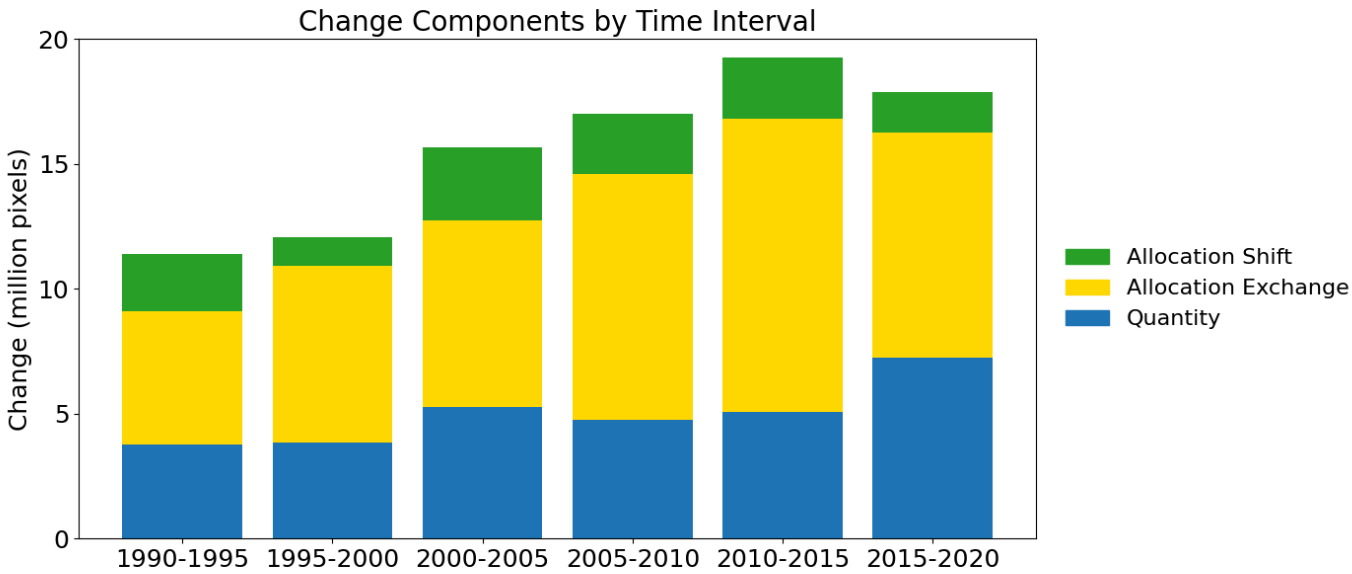
In [12]:
```python
# Read the generated CSV file
csv_path = os.path.join(output_path, 'change_components.csv')
df = pd.read_csv(csv_path)
```

5.2 Change Components by Time Interval

In [13]:
```python
# Filter only time intervals
time_df = df[df['Time_Interval'].str.contains('-')]

# Prepare data structure
totals = time_df.groupby(['Time_Interval', 'Component'])['Gain'].sum().unstack()

# Print the totals by time interval in millions of pixels
print("\nComponent Totals by Time Interval (million pixels):")
for index, row in totals.iterrows():
    print(f"\nTime Interval: {index}")
    for component in row.index:
        print(f"{component}: {row[component] / 1e6:.2f} M")


# Create figure and axes
fig, ax = plt.subplots(figsize=(14, 6))

# Colors for the bars
colors = ['#1f77b4', # Quantity
          '#ffd700', # Exchange
          '#2ca02c'] # Shift

components_color = {
    'Quantity': '#1f77b4',
    'Exchange': '#ffd700',
    'Shift': '#2ca02c'
}

# Plot bars for totals
for idx, comp in enumerate(['Quantity', 'Exchange', 'Shift']):
    bottom_values = totals[['Quantity', 'Exchange', 'Shift']].iloc[:, :idx].sum(axis=1) if id
```

```python
    ax.bar(totals.index,
           totals[comp],
           label=comp,
           color=colors[idx],
           edgecolor='none',
           bottom=bottom_values)

# Formating axes
ax.set_ylabel('Change (million pixels)',
              fontsize=18)
ax.set_title('Change Components by Time Interval',
             fontsize=20)
ax.tick_params(axis='both',
               which='major',
               labelsize=18)
ax.yaxis.set_major_locator(ticker.MaxNLocator(integer=True))

# Configuring y-axis to scale with millions
def millions_formatter(x, pos):
    return '%1.0f' % (x * 1e-6)
ax.set_ylim(0, 2.0e7)
ax.yaxis.set_major_locator(ticker.MultipleLocator(5000000))
ax.yaxis.set_major_formatter(ticker.FuncFormatter(millions_formatter))

# Legend
legend_elements = [
    plt.Rectangle((0,0),1,1, color=components_color['Shift'], label='Allocation Shift'),
    plt.Rectangle((0,0),1,1, color=components_color['Exchange'], label='Allocation Exchange')
    plt.Rectangle((0,0),1,1, color=components_color['Quantity'], label='Quantity')
]

ax.legend(handles=legend_elements,
          loc='center left',
          bbox_to_anchor=(1.01, 0.5),
          fontsize=16,
          frameon=False)

# Save and show plot
plt.tight_layout()
plt.savefig(os.path.join(output_path,
                         "graphic_change_components_time_interval.jpeg"),
            bbox_inches='tight',
            format='jpeg',
            dpi=300)
plt.show()
```

```
Component Totals by Time Interval (million pixels):

Time Interval: 1990-1995
Exchange: 5.37 M
Quantity: 3.75 M
Shift: 2.27 M

Time Interval: 1995-2000
Exchange: 7.07 M
Quantity: 3.86 M
Shift: 1.15 M

Time Interval: 2000-2005
Exchange: 7.47 M
Quantity: 5.28 M
Shift: 2.93 M

Time Interval: 2005-2010
Exchange: 9.86 M
Quantity: 4.74 M
Shift: 2.42 M

Time Interval: 2010-2015
Exchange: 11.75 M
Quantity: 5.07 M
Shift: 2.44 M

Time Interval: 2015-2020
Exchange: 9.02 M
Quantity: 7.24 M
Shift: 1.60 M
```



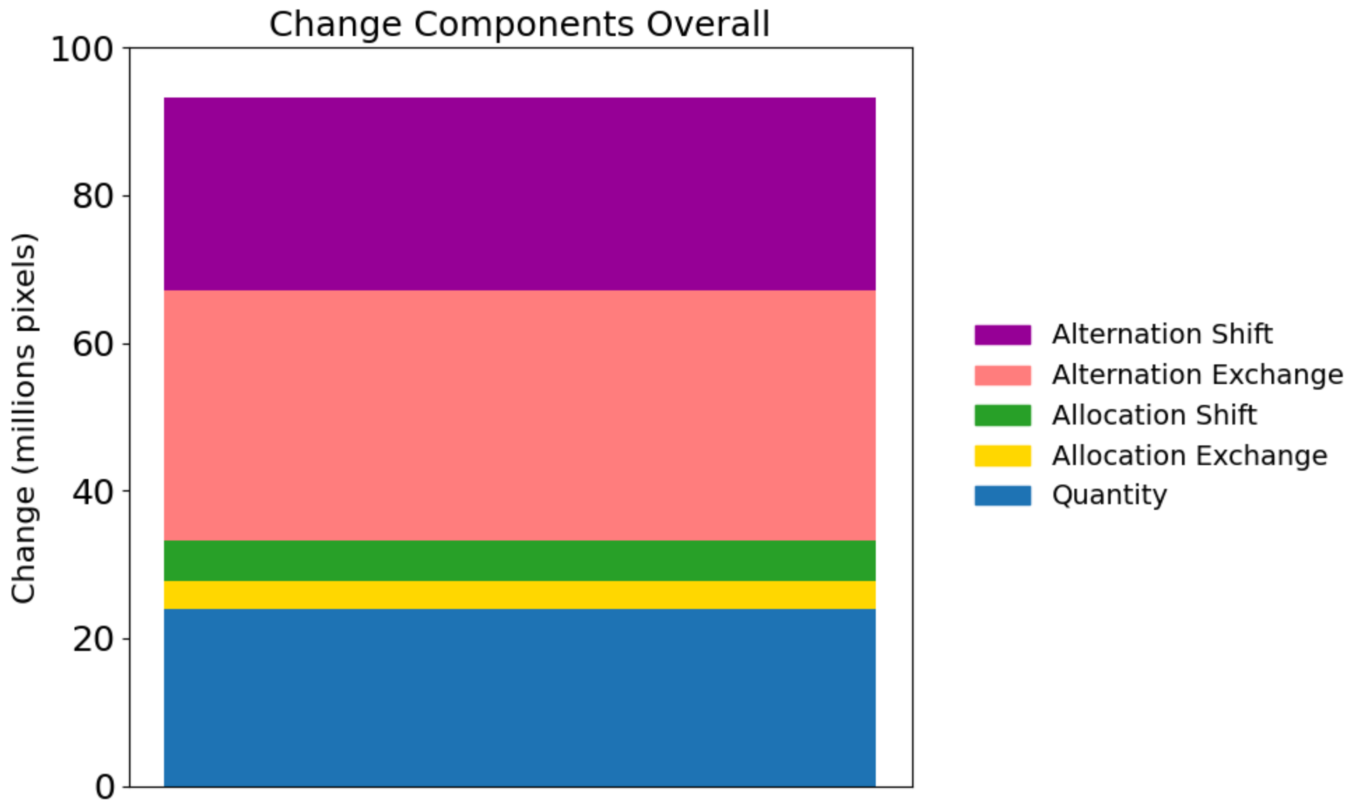
5.3 Change Components Overall

In [14]:
```python
def plot_components_with_alternation(csv_path, output_path):
    """
    Generates stacked bar chart showing:
    - Quantity (from extent matrix)
    - Exchange + Alternation Exchange
    - Shift + Alternation Shift

    Args:
        csv_path: Path to change_components.csv
        output_path: Directory to save the graphic
    """
    # Load data
    df = pd.read_csv(csv_path)
```

```python
# Define colors and component order
components_color = {
    'Quantity': '#1f77b4',
    'Allocation_Exchange': '#ffd700',
    'Alternation_Exchange': '#ff8080',
    'Allocation_Shift': '#2ca02c',
    'Alternation_Shift': '#990099'
}

component_order = [
    'Quantity',
    'Allocation_Exchange',
    'Allocation_Shift',
    'Alternation_Exchange',
    'Alternation_Shift'
]

# Calculate component totals
component_totals = {
    'Quantity': df[(df['Component'] == 'Quantity') &
                  (df['Time_Interval'] == 'extent')]['Gain'].sum(),

    'Allocation_Exchange': df[(df['Component'] == 'Allocation_Exchange') &
                  (df['Time_Interval'] == 'extent')]['Gain'].sum(),

    'Allocation_Shift': df[(df['Component'] == 'Allocation_Shift') &
               (df['Time_Interval'] == 'extent')]['Gain'].sum(),

    'Alternation_Exchange': df[df['Component'] == 'Alternation_Exchange']['Gain'].sum(),


    'Alternation_Shift': df[df['Component'] == 'Alternation_Shift']['Gain'].sum()
}

# Print dos valores em milhões de pixels
print("\nComponent Totals (million pixels):")
for component, value in component_totals.items():
    print(f"{component.replace('_', ' ')}: {value / 1e6:.2f} M")

# Create figure with original styling
fig, ax = plt.subplots(figsize=(10, 6))

# Plot stacked bars
bottom = 0
bars = []
labels = []

for component in component_order:
    value = component_totals.get(component, 0)
    bar = ax.bar(
        x=0,
        height=value,
        bottom=bottom,
        color=components_color[component],
        edgecolor='none',
        width=1
    )
    bars.append(bar[0])
    labels.append(component.replace('_', ' '))
    bottom += value

# Axis formatting
ax.set_ylabel('Change (millions pixels)',
              fontsize=16)
ax.set_title('Change Components Overall',
```

```python
                   fontsize=18)
    ax.xaxis.set_visible(False)
    ax.tick_params(axis='both', which='major',
                     labelsize=18)
    ax.yaxis.set_major_locator(ticker.MaxNLocator(integer=True))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    # Remove chart borders
    for spine in ['top', 'right', 'left', 'bottom']:
        ax.spines[spine].set_visible(True)

    # Configuring y-axis to scale with millions
    def millions_formatter(x, pos):
        return '%1.0f' % (x * 1e-6)
    ax.set_ylim(0, 10e7)
    ax.yaxis.set_major_locator(ticker.MultipleLocator(20000000))
    ax.yaxis.set_major_formatter(ticker.FuncFormatter(millions_formatter))

    # Legend
    legend_elements = [
        plt.Rectangle((0,0),1,1, color=components_color['Alternation_Shift'], label='Alternat
        plt.Rectangle((0,0),1,1, color=components_color['Alternation_Exchange'], label='Alter
        plt.Rectangle((0,0),1,1, color=components_color['Allocation_Shift'], label='Allocatio
        plt.Rectangle((0,0),1,1, color=components_color['Allocation_Exchange'], label='Alloca
        plt.Rectangle((0,0),1,1, color=components_color['Quantity'], label='Quantity')
    ]
    ax.legend(handles=legend_elements,
            loc='center left',
            bbox_to_anchor=(1.05, 0.5),
            fontsize=14,
            frameon=False)

    # Save and display
    plt.tight_layout()
    plt.savefig(
        os.path.join(output_path,
                     "graphic_change_components_overall.jpeg"),
        bbox_inches='tight',
        format='jpeg',
        dpi=300,
    )
    plt.show()

if __name__ == "__main__":
    csv_file = os.path.join(output_path,
                            "change_components.csv")

    # Generate visualization
    plot_components_with_alternation(csv_file, output_path)
```

```
Component Totals (million pixels):
Quantity: 23.95 M
Allocation Exchange: 3.79 M
Allocation Shift: 5.39 M
Alternation Exchange: 33.89 M
Alternation Shift: 26.26 M
```

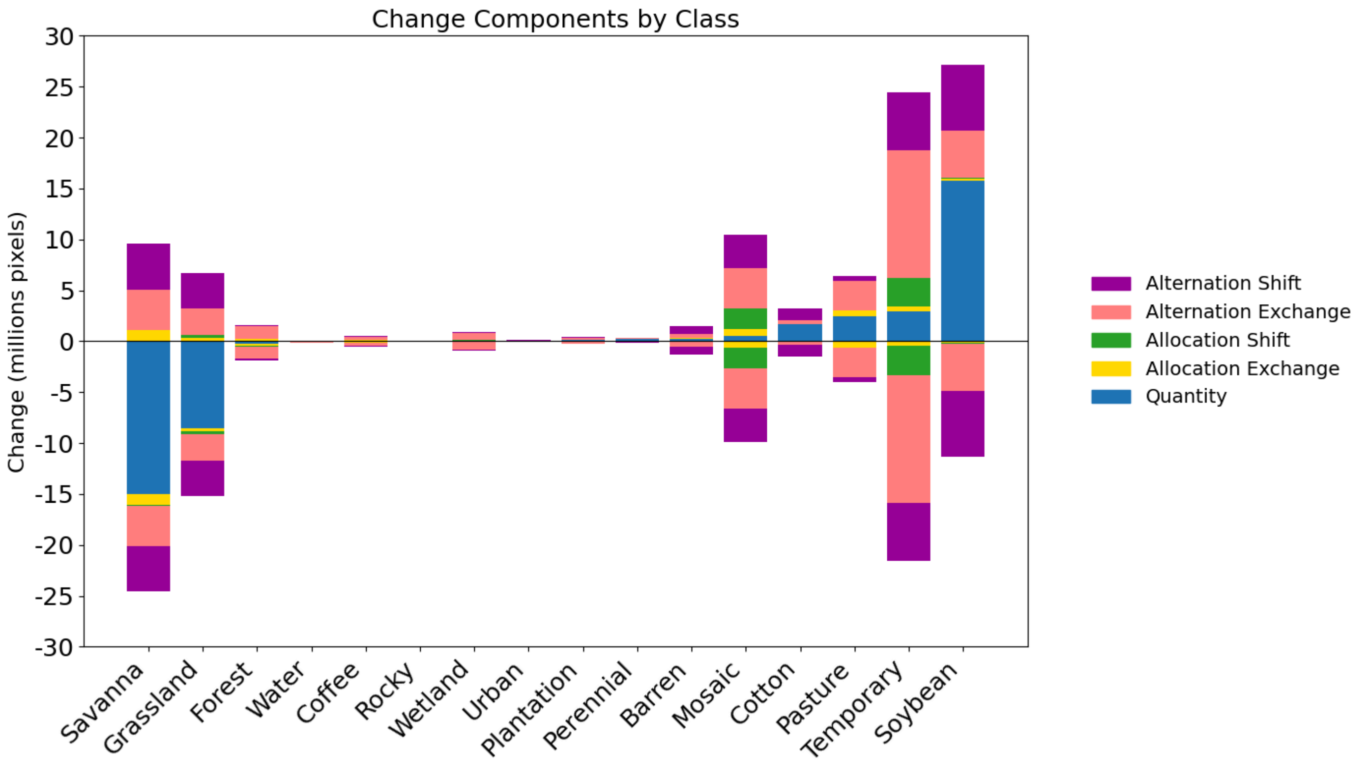


5.4 Change Componentes by Class

In [16]:
```
class ComponentVisualizer:
    """Class to visualize components including alternation"""

    @staticmethod
    def plot_gain_loss_stacked(class_labels_dict, title, output_path):
        """Plot gains and losses with alternation components"""
        # Access predefined parameters from global scope
        global df, components_color, component_order
        global figsize, title_fontsize, label_fontsize, tick_labelsize, legend_fontsize

        # Filter data using original criteria
        filtered_df = df[df['Time_Interval'].isin(['extent', 'alternation'])]

        # Class sorting logic
        existing_classes = [cls for cls in filtered_df['Class'].unique() if cls != 0]
        class_totals = []

        # Define colors and component order
        components_color = {
            'Quantity': '#1f77b4',
            'Allocation_Exchange': '#ffd700',
            'Alternation_Exchange': '#ff8080',
            'Allocation_Shift': '#2ca02c',
            'Alternation_Shift': '#990099'
        }

        component_order = [
            'Quantity',
            'Allocation_Exchange',
            'Allocation_Shift',
            'Alternation_Exchange',
            'Alternation_Shift'
        ]

        for cls in existing_classes:
            class_data = filtered_df[filtered_df['Class'] == cls]
            quantity_gain = class_data[class_data['Component'] == 'Quantity']['Gain'].sum()
```

```
        quantity_loss = class_data[class_data['Component'] == 'Quantity']['Loss'].sum()
        class_totals.append((cls, quantity_gain - quantity_loss))

    sorted_classes = sorted(class_totals, key=lambda x: x[1])
    ordered_classes = [cls for cls, _ in sorted_classes]

    # Create figure using predefined dimensions
    fig, ax = plt.subplots(figsize=(14, 8))
    fig.subplots_adjust(left=0.1, right=0.75)
    x_positions = np.arange(len(ordered_classes))
    width = 0.8

    # Plot each class using predefined component order
    for idx, cls in enumerate(ordered_classes):
        class_data = filtered_df[filtered_df['Class'] == cls]

        # Initialize accumulators
        gain_bottom = loss_bottom = 0

        # Plot components in original + alternation order
        for comp in ['Quantity', 'Allocation_Exchange', 'Allocation_Shift', 'Alternation_
            # Gains
            gains = class_data[class_data['Component'] == comp]['Gain'].sum()
            ax.bar(x_positions[idx], gains, width,
                   bottom=gain_bottom,
                   color=components_color[comp],
                   edgecolor='none')
            gain_bottom += gains

            # Losses
            losses = -class_data[class_data['Component'] == comp]['Loss'].sum()
            ax.bar(x_positions[idx], losses, width,
                   bottom=loss_bottom,
                   color=components_color[comp],
                   edgecolor='none')
            loss_bottom += losses

    # Axis formatting with predefined parameters
    class_names = [class_labels_dict.get(cls, {}).get("rename", f"{cls}") for cls in orde
    ax.set_xticks(x_positions)
    ax.set_xticklabels(class_names,
                       rotation=45,
                       ha='right',
                       fontsize=18)
    ax.axhline(0,
               color='black',
               linewidth=0.8)
    ax.set_ylabel('Change (millions pixels)',
                  fontsize=16)
    ax.set_title(title,
                 fontsize=18)
    ax.tick_params(axis='both',
                   which='major',
                   labelsize=18)
    ax.yaxis.set_major_locator(ticker.MaxNLocator(integer=True))

    # Configuring y-axis to scale with millions
    def millions_formatter(x, pos):
        return '%1.0f' % (x * 1e-6)
    ax.set_ylim(-3e7, 3e7)
    ax.yaxis.set_major_locator(ticker.MultipleLocator(5000000))
    ax.yaxis.set_major_formatter(ticker.FuncFormatter(millions_formatter))

    # Legend
    legend_elements = [
        plt.Rectangle((0,0),1,1, color=components_color['Alternation_Shift'], label='Alte
```

```
            plt.Rectangle((0,0),1,1, color=components_color['Alternation_Exchange'], label='A
            plt.Rectangle((0,0),1,1, color=components_color['Allocation_Shift'], label='Alloc
            plt.Rectangle((0,0),1,1, color=components_color['Allocation_Exchange'], label='Al
            plt.Rectangle((0,0),1,1, color=components_color['Quantity'], label='Quantity')
        ]

        ax.legend(handles=legend_elements,
                loc='center left',
                bbox_to_anchor=(1.05, 0.5),
                fontsize=14,
                frameon=False)

        # Save and show plot
        plt.tight_layout()
        plt.savefig(
            os.path.join(output_path,
                         "graphic_change_component_change_class.jpeg"),
            format='jpeg',
            dpi=300,
            bbox_inches='tight'
        )
        plt.show()

ComponentVisualizer.plot_gain_loss_stacked(
    class_labels_dict,
    "Change Components by Class",
    output_path
)
```



6. Trajectory Classification

Overview

This section provides a framework for processing and classifying pixel trajectories in raster datasets.

In [17]:
```
@nb.njit(nogil=True, cache=True)
def classify_pixel(pixel_series):
    """Numba-optimized trajectory classification (20-50x faster)"""
```

```python
    if pixel_series[0] == 0:
        return 0

    start = pixel_series[0]
    end = pixel_series[-1]
    has_variation = False
    direct_transition = False

    # First pass: check all conditions in single loop
    for i in range(len(pixel_series)-1):
        current = pixel_series[i]
        next_val = pixel_series[i+1]

        # Check for any variation
        if not has_variation and current != next_val:
            has_variation = True

        # Check for direct transition
        if not direct_transition and current == start and next_val == end:
            direct_transition = True

    # Trajectory 1:
    if not has_variation:
        return 1

    # Trajectory 2:
    if start == end:
        return 2

    # Trajectory 3:
    if direct_transition:
        return 3

    # Trajectory 4:
    return 4

@nb.njit(nogil=True, parallel=True)
def process_stack_parallel(stack, height, width):
    """Parallel processing of raster stack"""
    result = np.zeros((height, width), dtype=np.uint8)

    for y in nb.prange(height):
        for x in range(width):
            result[y, x] = classify_pixel(stack[:, y, x])

    return result

class TrajectoryAnalyzer:
    @staticmethod
    def process_rasters(output_path, suffix='_masked.tif'):
        """Optimized raster processing with chunked loading"""
        # Validate directory
        os.makedirs(output_path, exist_ok=True)
        if not os.path.isdir(output_path):
            raise ValueError(f"Path must be a directory: {output_path}")

        # Find input files
        raster_files = sorted([
            os.path.join(output_path, f)
            for f in os.listdir(output_path)
            if f.endswith(suffix)
        ])
        if not raster_files:
            raise ValueError(f"No files found with suffix '{suffix}'")

        # Load metadata
```

```
        with rasterio.open(raster_files[0]) as src:
            meta = src.meta
            height, width = src.shape

        # Process in memory-friendly chunks
        chunk_size = 500
        result = np.zeros((height, width),
                          dtype=np.uint8)

        for y_start in range(0, height, chunk_size):
            y_end = min(y_start + chunk_size, height)
            chunk_height = y_end - y_start

            # Load chunk data
            stack = np.zeros((len(raster_files),
                              chunk_height,
                              width),
                             dtype=np.uint8)
            for i, f in enumerate(raster_files):
                with rasterio.open(f) as src:
                    stack[i] = src.read(1, window=((y_start, y_end), (0, width)))

            # Process chunk
            result[y_start:y_end] = process_stack_parallel(stack,
                                                           chunk_height,
                                                           width)

        # Save results
        meta.update({
            'dtype': 'uint8',
            'nodata': 0,
            'count': 1,
            'compress': 'lzw'
        })
        output_file = os.path.join(output_path, 'trajectory.tif')
        with rasterio.open(output_file, 'w', **meta) as dst:
            dst.write(result, 1)

        return output_file

if __name__ == "__main__":
    TrajectoryAnalyzer.process_rasters(output_path)
    print(f"Processing complete! Results saved to: {output_path}")
```

```
Processing complete! Results saved to: /content/drive/MyDrive/change-components/output
```

6.1 Map visualization

This section processes and visualizes raster data by scaling, applying a color map, and adding graphical elements like legends, scale bars, and north arrows. The output is a high-resolution image of the classified raster data.

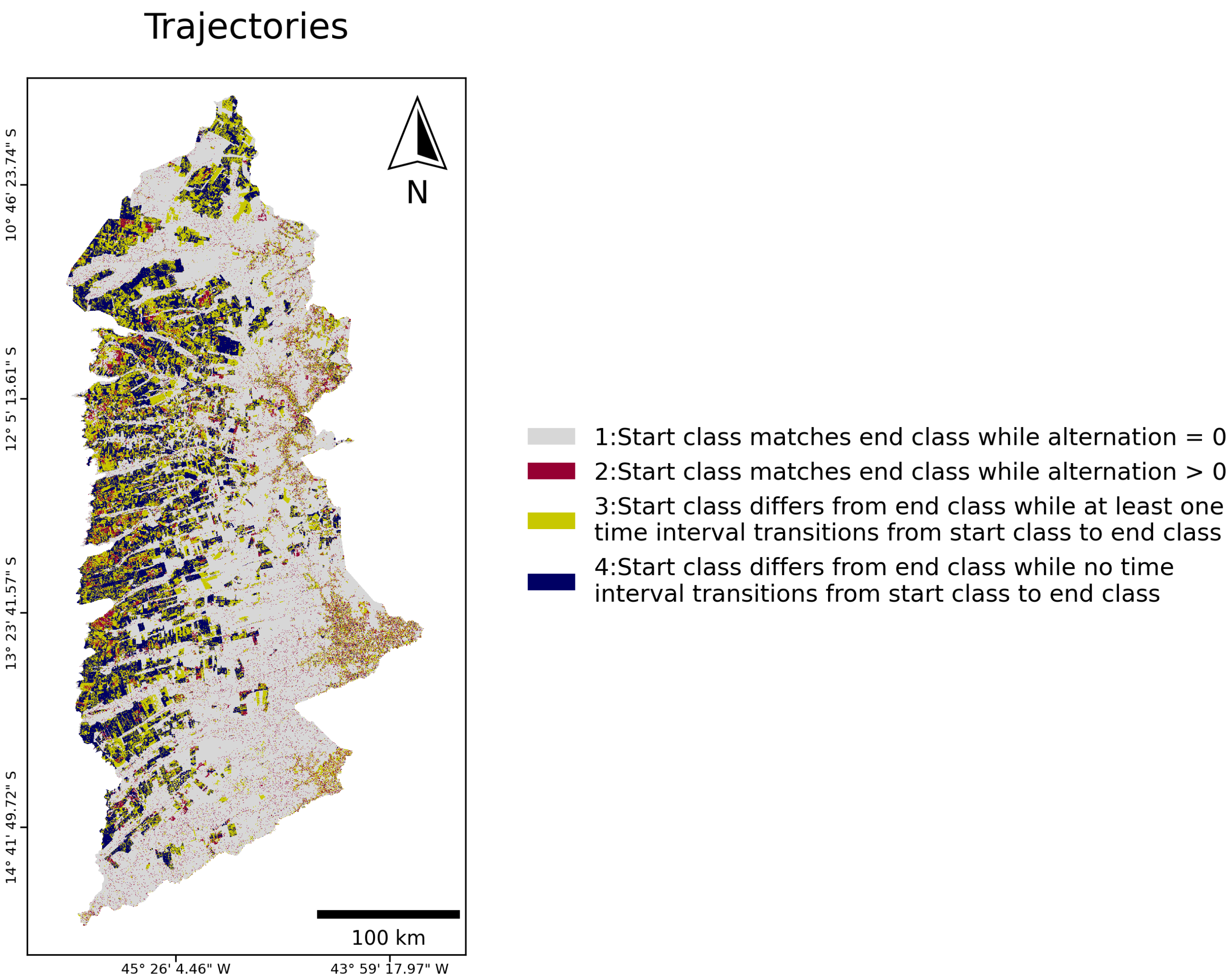
In [18]:

```
# Set the path for the input raster file
raster_path = os.path.join(output_path, 'trajectory.tif')

# Set the resolution quality of the output image
dpi = 300

# Define a scale factor to resize the raster data
scale_factor = 0.15

# Legend elements
legend_elements = [
    Rectangle((0, 0), 1, 1, facecolor='#d9d9d9', label='1:Start class matches end class while
```

```python
    Rectangle((0, 0), 1, 1, facecolor='#990033', label='2:Start class matches end class while
    Rectangle((0, 0), 1, 1, facecolor='#cccc00', label='3:Start class differs from end class w
    Rectangle((0, 0), 1, 1, facecolor='#000066', label='4:Start class differs from end class w
]

cmap = ListedColormap([
    '#ffffff',
    '#d9d9d9',
    '#990033',
    '#cccc00',
    '#000066',
])

with rasterio.open(raster_path) as src:
    # Read and scale data
    data = src.read(
        1,
        out_shape=(int(src.height * scale_factor), int(src.width * scale_factor)),
        resampling=rasterio.enums.Resampling.nearest
    )

    # Get bounds
    left, bottom, right, top = src.bounds

    # Create coordinate transformer
    src_crs = src.crs.to_string()
    transformer = Transformer.from_crs(src_crs,
                                       "EPSG:4326",
                                       always_xy=True)

    # Create figure
    fig, ax = plt.subplots(figsize=(14, 8), dpi=dpi)

    # Plot raster data
    img = ax.imshow(
        data,
        cmap=cmap,
        extent=[left, right, bottom, top],
        interpolation='none'
    )

    # Custom tick formatters for degrees
    def format_x_ticks(x, pos):
        lon, _ = transformer.transform(x, bottom)
        deg = int(abs(lon))
        min = int((abs(lon) - deg) * 60)
        sec = ((abs(lon) - deg) * 60 - min) * 60
        return f"{deg}° {min}' {sec:.2f}\"" + (" E" if lon >= 0 else " W")

    def format_y_ticks(y, pos):
        _, lat = transformer.transform(left, y)
        deg = int(abs(lat))
        min = int((abs(lat) - deg) * 60)
        sec = ((abs(lat) - deg) * 60 - min) * 60
        return f"{deg}° {min}' {sec:.2f}\"" + (" N" if lat >= 0 else " S")

    ax.xaxis.set_major_formatter(FuncFormatter(format_x_ticks))
    ax.yaxis.set_major_formatter(FuncFormatter(format_y_ticks))

    # Add scale bar
    def km_to_degrees(value, dimension):
        approx_deg = value / 111  # Approximate conversion: 1° ≈ 111km
        return f"{approx_deg:.1f}°"

    scalebar = ScaleBar(
        1/1000,
```

```python
    units='km',
    length_fraction=0.4,
    location='lower right',
    scale_formatter=lambda value, _: f"{int(value)} km"
)
ax.add_artist(scalebar)

# Add north arrow
north_arrow(
    ax,
    location="upper right",
    shadow=False,
    rotation={"degrees":0}
)

# Style adjustments
ax.set_aspect('equal')

ax.tick_params(axis='x',
               which='major',
               labelsize=7,
               pad=4)

ax.tick_params(axis='y',
           which='major',
           labelsize=7,
           pad=4)

ax.xaxis.set_major_locator(plt.MaxNLocator(3))

ax.yaxis.set_major_locator(plt.MaxNLocator(6))

plt.setp(ax.get_yticklabels(),
 rotation=90,
 va='center',
 ha='center')

plt.title("Trajectories",
          fontsize=18,
          pad=20)

# Position all tick labels centrally relative to their ticks
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_horizontalalignment('center')
    label.set_verticalalignment('center')

# Add legend
legend = ax.legend(
    handles=legend_elements,
    loc='center left',
    bbox_to_anchor=(1.05, 0.5),
    frameon=False,
    fontsize=12,
    borderpad=1.2,
    handletextpad=0.8,
    columnspacing=2,
)

# Save the plot as a JPEG file
plt.savefig(
    os.path.join(output_path, 'map_trajectories_degree.jpeg'),
    dpi=dpi,
    bbox_inches='tight',
    pad_inches=0.5,
    pil_kwargs={'optimize': True,
                'quality': 95}
```

```python
    )

plt.show()
plt.close()

print("Map saved successfully in the folder:", output_path)
```




```
Map saved successfully in the folder: /content/drive/MyDrive/change-components/output
```

In [19]:
```python
def plot_trajectory_distribution(output_path):
    """
    Generates trajectory distribution bar chart from raster data
    with consistent styling and proper frame
    """
    # Path to trajectory raster
    raster_path = os.path.join(output_path, 'trajectory.tif')

    # Read raster data
    with rasterio.open(raster_path) as src:
        traj_data = src.read(1)
        nodata = src.nodata

    # Filter nodata values and count trajectories
    masked_traj = np.ma.masked_where(traj_data == nodata, traj_data)
    unique, counts = np.unique(masked_traj.compressed(), return_counts=True)
    total_pixels = counts.sum()

    # Calculate percentages
    percentages = {k: (v/total_pixels)*100 for k, v in zip(unique, counts)}

    # Print trajectory percentages
    print("\nTrajectory Percentages:")
    for traj, percentage in percentages.items():
        print(f"Trajectory {traj}: {percentage:.2f}%")
```

```python
# Define trajectories to show (4, 3, 2) and colors
ordered_trajs = [4, 3, 2]
colors = {
    4: '#000066',  # Dark blue
    3: '#cccc00',  # Gold
    2: '#990033'   # Dark red
}

# Calculate maximum Y value (round up to nearest 10)
max_percentage = sum(percentages.get(traj, 0) for traj in ordered_trajs)
y_max = np.ceil(max_percentage / 10) * 10  # Round up to nearest 10

# Create figure with consistent styling
fig, ax = plt.subplots(figsize=(8, 6))

# Plot stacked bars
bottom = 0
for traj in ordered_trajs:
    if traj in percentages:
        ax.bar(0, percentages[traj],
              bottom=bottom,
              color=colors[traj],
              width=0.4,
              edgecolor='none')
        bottom += percentages[traj]

# Formatting labels
ax.set_ylabel('Trajectory Area (% of western Bahia)',
             fontsize=16)

# Configure frame (box) - all spines visible
for spine in ['top', 'right', 'bottom', 'left']:
    ax.spines[spine].set_visible(True)
    ax.spines[spine].set_color('black')
    ax.spines[spine].set_linewidth(0.5)

# Remove minor ticks and configure major ticks
ax.tick_params(
    axis='y',
    which='minor',
    length=0
    )
ax.tick_params(
    axis='y',
    which='major',
    labelsize=18,
    length=6,
    width=1
    )

# Set dynamic Y-axis limits
ax.set_ylim(0, y_max)
ax.yaxis.set_major_locator(ticker.MultipleLocator(np.floor(y_max/5)))

ax.xaxis.set_visible(False)

# Remove grid lines
ax.grid(False)

# Legend
legend_elements = [
    Patch(facecolor='#000066', label='Trajectory 4'),
    Patch(facecolor='#cccc00', label='Trajectory 3'),
    Patch(facecolor='#990033', label='Trajectory 2')
]
ax.legend(handles=legend_elements,
```

```python
                loc='center left',
                bbox_to_anchor=(1.05, 0.5),
                fontsize=14,
                frameon=False)

    # Save and show
    plt.tight_layout()
    plt.savefig(os.path.join(
        output_path,
        'trajectory_distribution.jpeg'),
                dpi=300,
                bbox_inches='tight')
    plt.show()

# Usage
if __name__ == "__main__":
    plot_trajectory_distribution(output_path)
```

```
Trajectory Percentages:
Trajectory 1: 58.01%
Trajectory 2: 5.81%
Trajectory 3: 19.35%
Trajectory 4: 16.82%
```