Critical Bug: Features East/South/West Always Empty

Problem Statement

A Python FastAPI backend analyzes forest boundaries and should return nearby features in 4 directions (North, East, South, West). **Only "Features North" populates with data. Features East/South/West are always `None` in the database.**

What We Know FOR CERTAIN

1. The SQL Queries Work Perfectly

When tested directly with the same geometry, the queries return features in ALL directions:

NORTH: 7 features ✅

EAST: 11 features ✅

SOUTH: 8 features ✅

WEST: 14 features ✅

2. The Function Works in Isolation

When `analyze_nearby_features()` is called standalone (outside the upload flow), it returns all 4 directions correctly:

```
result_features = {
  'features_north': 'River/Strem Flow Line, Gothdanda, ...',
  'features_east': 'River/Strem Flow Line, Chaughada, ...',
  'features_south': 'River/Strem Flow Line, Kyampadanda, ...',
  'features_west': 'Karra Khola, River/Strem Flow Line, ...'
}
```

3. Database Shows Only North is Saved

Features North: "River/Strem Flow Line, Gothdanda, Kyampadanda, Phikrepakha, Pipalechok, unclassified"

Features East: None

Features South: None

Features West: None

4. NO Console Logs Appear During Upload

Despite adding print statements with `flush=True`, `PYTHONUNBUFFERED=1`, and `-u` flag, **ZERO debug output appears in the console during file upload**. The console only shows application startup, then nothing when files are uploaded.

Code Structure

Function: `analyze_nearby_features(geometry_wkt: str, db: Session)`

Location: `D:\forest_management\backend\app\services\analysis.py` (lines 1408-1519)

How it works:

1. Loops through 9 different database tables (roads, rivers, POI, etc.)
2. For each direction (north, east, south, west):
  * Queries each table for nearby features within 100m
  * Uses ST_Azimuth to calculate bearing angles
  * Filters by angle ranges (North: 315-45°, East: 45-135°, etc.)
  * Appends feature names to `features_list`
3. Stores result: `result_features[f"features_{direction_name}"] = features_str`
4. Returns dict with all 4 keys

Called from: `analyze_block_geometry()` at line 297

Upload Flow

1. User uploads file → `POST /api/forests/upload` (forests.py:211)
2. Backend calls `analyze_forest_boundary(calc_id, db)` (forests.py:295)
3. For each block, calls `analyze_block_geometry()` (analysis.py:46)
4. Block analysis calls `analyze_nearby_features()` (analysis.py:297)
5. Results merged via `block_results.update(nearby_results)` (analysis.py:298)
6. Saved to database via SQL UPDATE (forests.py:303)

What We've Tried (All Failed)

1. ❌ Unicode Encoding Fix

* **Issue:** Thought print() was crashing on Nepali characters
* **Fix:** Added try/except around print statements
* **Result:** Standalone test works, but real upload still broken

2. ❌ Transaction Rollback Fixes

* **Issue:** Thought failed transactions were contaminating subsequent queries
* **Fix:** Added `db.rollback()` in exception handlers
* **Result:** No change

3. ❌ Python Output Buffering

* **Issue:** Thought print statements were buffered
* **Fixes Applied:**
  * Added `-u` flag to python command
  * Set `PYTHONUNBUFFERED=1` environment variable
  * Added `flush=True` to all print statements
* **Result:** Still NO console output during uploads

4. ❌ GeoJSON to WKT Conversion Error

* **Issue:** Suspected geometry conversion failures

* **Fix:** Added rollback before raising exception

* **Result:** No improvement

Key Mystery

Why does the function work perfectly in isolation but fail during actual uploads?

The standalone test uses the EXACT same:

* Database session

* WKT geometry string

* Function code

* Database credentials

Yet it returns all 4 directions. But during real uploads, only North is saved.

Critical Questions to Answer

1. **Why no console logs?** Even with unbuffered output and flush=True, print statements during upload don't appear. This suggests:

  * The code path isn't executing at all?

  * Output is redirected somewhere else?

  * Uvicorn/FastAPI is capturing stdout?

2. **Where is the data lost?** Between these points:

  * `analyze_nearby_features()` returns dict (works in test)

* `block_results.update(nearby_results)` merges it

* Data saved to database (only North appears)

3. **Is there a silent exception?** Could an exception be raised after North but before East/South/West, and being caught/swallowed somewhere?

Environment Details

* **OS:** Windows 11

* **Python:** 3.14

* **Framework:** FastAPI with Uvicorn (--reload mode)

* **Database:** PostgreSQL 15 with PostGIS

* **ORM:** SQLAlchemy 2.0.45

Files to Examine (appended the last part of this file)

1. `D:\forest_management\backend\app\services\analysis.py` - Main analysis logic

2. `D:\forest_management\backend\app\api\forests.py` - Upload endpoint

3. `D:\forest_management\start_server_8001.bat` - Server startup script

Request for Help

What could cause a function to work perfectly in isolation but only return partial data (1 of 4 dictionary keys) when called during the actual application flow, with NO error messages and NO console output?

Specific help needed:

1. How to debug when print statements don't appear (even with unbuffered output)?

2. What could cause only the FIRST iteration of a loop to save, but not subsequent ones?

3. How to trace where dictionary values are being lost between function return and database save?

Test Commands

Run standalone test (works):

```
cd D:\forest_management
.\venv\Scripts\python.exe test_features_direct.py
```

Check database after upload:

```
cd D:\forest_management
.\venv\Scripts\python.exe check_db.py
```

Expected vs Actual

Expected in database:

```
{
  "features_north": "River/Strem Flow Line, ...",
  "features_east": "River/Strem Flow Line, Chaughada, ...",
  "features_south": "River/Strem Flow Line, ...",
  "features_west": "Karra Khola, ..."
}
```

Actual in database:

```
{
  "features_north": "River/Strem Flow Line, ...",
  "features_east": null,
  "features_south": null,
  "features_west": null
```

```
  }
```

```
////////////////////////////////////////////////////////////////////////////
"""
Forest boundary analysis service
Performs raster and vector analysis on uploaded forest boundaries
"""
import time
from typing import Dict, Any, Tuple
from uuid import UUID
from sqlalchemy.orm import Session
from sqlalchemy import func, text
from datetime import datetime

from ..models.calculation import Calculation


async def analyze_forest_boundary(calculation_id: UUID, db: Session) -> Tuple[Dict[str, Any], int]:
  """
  Perform comprehensive analysis on forest boundary
  Now analyzes each block separately and adds analysis results to each block

  Args:
    calculation_id: UUID of calculation record
    db: Database session

  Returns:
    Tuple of (result_data dict, processing_time_seconds)
  """
```

```
    # Per-block analysis enabled
    start_time = time.time()

    # Get calculation record
    calculation = db.query(Calculation).filter(Calculation.id == calculation_id).first()
    if not calculation:
        raise ValueError(f"Calculation {calculation_id} not found")

    results = {}

    # Get existing result_data with blocks
    existing_data = calculation.result_data or {}
    blocks = existing_data.get('blocks', [])

    # Analyze each block separately
    analyzed_blocks = []
    for i, block in enumerate(blocks):
        print(f"Analyzing block {i+1}/{len(blocks)}: {block.get('block_name', f'Block {i+1}')}",
flush=True)

        try:
            block_analysis = await analyze_block_geometry(
                block['geometry'],
                calculation_id,
                db
            )
            # Merge analysis results into block data
            analyzed_block = {**block, **block_analysis}
            print(f"Block {i+1} analysis completed successfully", flush=True)

        except Exception as block_error:
```

```
            print(f"Error analyzing block {i+1}: {block_error}")
            db.rollback()  # Rollback failed block transaction

            # Commit to clear transaction state for next block
            try:
                db.commit()
                print(f"Transaction reset after block {i+1} error")
            except Exception as commit_error:
                print(f"Warning: Could not commit after block error: {commit_error}")

            # Continue with minimal data for this block
            analyzed_block = {**block, "analysis_error": str(block_error)[:200]}

        analyzed_blocks.append(analyzed_block)

        # ALWAYS commit after each block to ensure clean transaction state for next block
        try:
            db.commit()
            print(f"Block {i+1} transaction committed")
        except Exception as commit_error:
            print(f"Warning: Could not commit after block {i+1}: {commit_error}")
            db.rollback()

    # Store analyzed blocks
    results['blocks'] = analyzed_blocks

    # Commit block results to clear any transaction issues
    print(f"Committing block analysis results...")
    try:
        db.commit()
        print("Block analysis committed successfully")
```

```
except Exception as commit_error:
    print(f"Warning: Could not commit block results: {commit_error}")
    db.rollback()

# Also calculate whole-area statistics for summary
# 1. Calculate area (using UTM projection for accuracy)
print("Starting whole-forest analysis...")
area_data = calculate_area(calculation_id, db)
results.update(area_data)

# 1b. Calculate whole forest extent (bounding box)
extent_query = text("""
    SELECT
        ST_YMax(ST_Envelope(boundary_geom)) as north,
        ST_YMin(ST_Envelope(boundary_geom)) as south,
        ST_XMax(ST_Envelope(boundary_geom)) as east,
        ST_XMin(ST_Envelope(boundary_geom)) as west
    FROM calculations WHERE id = :calc_id
""")
whole_extent = db.execute(extent_query, {"calc_id": str(calculation_id)}).first()
if whole_extent:
    results["whole_forest_extent"] = {
        "N": round(float(whole_extent.north), 7),
        "S": round(float(whole_extent.south), 7),
        "E": round(float(whole_extent.east), 7),
        "W": round(float(whole_extent.west), 7)
    }

# 2. Raster analysis on whole boundary
raster_results = await analyze_rasters(calculation_id, db)
results.update(raster_results)
```

```
# 3. Vector analysis
vector_results = await analyze_vectors(calculation_id, db)
results.update(vector_results)

# 3b. Get administrative location for whole forest
whole_geom_query = text("""
  SELECT ST_AsText(boundary_geom) as wkt
  FROM public.calculations
  WHERE id = :calc_id
""")
whole_geom = db.execute(whole_geom_query, {"calc_id": str(calculation_id)}).first()
if whole_geom:
  whole_location = get_administrative_location(whole_geom.wkt, db)
  # Prefix keys with "whole_" to distinguish from block-level data
  results["whole_province"] = whole_location.get("province")
  results["whole_district"] = whole_location.get("district")
  results["whole_municipality"] = whole_location.get("municipality")
  results["whole_ward"] = whole_location.get("ward")
  results["whole_watershed"] = whole_location.get("watershed")
  results["whole_major_river_basin"] = whole_location.get("major_river_basin")

# 3c. Geology analysis for whole forest
if whole_geom:
  whole_geology = analyze_geology_geometry(whole_geom.wkt, db)
  results["whole_geology_percentages"] = whole_geology.get("geology_percentages")

# 3d. Access information for whole forest
if whole_geom:
  whole_access = calculate_access_info(whole_geom.wkt, db)
  results["whole_access_info"] = whole_access.get("access_info")
```

```
    # 3e. Nearby features for whole forest
    if whole_geom:
        whole_features = analyze_nearby_features(whole_geom.wkt, db)
        results["whole_features_north"] = whole_features.get("features_north")
        results["whole_features_east"] = whole_features.get("features_east")
        results["whole_features_south"] = whole_features.get("features_south")
        results["whole_features_west"] = whole_features.get("features_west")

    # 4. Administrative boundaries
    admin_results = await analyze_admin_boundaries(calculation_id, db)
    results.update(admin_results)

    processing_time = int(time.time() - start_time)

    # Commit all analysis results to ensure clean transaction state
    print("Committing final analysis results...")
    try:
        db.commit()
        print("Analysis transaction committed successfully")
    except Exception as commit_error:
        print(f"Warning: Could not commit analysis: {commit_error}")
        db.rollback()

    return results, processing_time


async def analyze_block_geometry(geojson_geometry: Dict, calculation_id: UUID, db: Session)
-> Dict[str, Any]:
    """
    Analyze a single block's geometry
```

```
Args:
    geojson_geometry: Block geometry in GeoJSON format
    calculation_id: UUID of parent calculation (for context)
    db: Database session

Returns:
    Dict with analysis results for this block
"""
import json

# Convert GeoJSON to WKT for PostGIS
geojson_str = json.dumps(geojson_geometry)

try:
    # Create a temporary geometry from GeoJSON
    geom_query = text("""
        SELECT ST_AsText(ST_GeomFromGeoJSON(:geojson)) as wkt
    """)
    wkt_result = db.execute(geom_query, {"geojson": geojson_str}).first()
    block_wkt = wkt_result.wkt
except Exception as e:
    print(f"Error converting GeoJSON to WKT: {e}")
    print(f"GeoJSON string (first 500 chars): {geojson_str[:500]}")
    # CRITICAL: Rollback the failed transaction before raising
    db.rollback()
    raise ValueError(f"Invalid geometry format: {str(e)}")

block_results = {}

# Calculate bounding box extent for this block
```

```
extent_query = text("""
  SELECT
    ST_YMax(ST_Envelope(ST_GeomFromText(:wkt, 4326))) as north,
    ST_YMin(ST_Envelope(ST_GeomFromText(:wkt, 4326))) as south,
    ST_XMax(ST_Envelope(ST_GeomFromText(:wkt, 4326))) as east,
    ST_XMin(ST_Envelope(ST_GeomFromText(:wkt, 4326))) as west
""")
extent_result = db.execute(extent_query, {"wkt": block_wkt}).first()
if extent_result:
  block_results["extent"] = {
    "N": round(float(extent_result.north), 7),
    "S": round(float(extent_result.south), 7),
    "E": round(float(extent_result.east), 7),
    "W": round(float(extent_result.west), 7)
  }

# Run all raster analyses on this block's geometry
# 1. DEM - Elevation
dem_results = analyze_dem_geometry(block_wkt, db)
block_results.update(dem_results)

# 2. Slope
slope_results = analyze_slope_geometry(block_wkt, db)
block_results.update(slope_results)

# 3. Aspect
aspect_results = analyze_aspect_geometry(block_wkt, db)
block_results.update(aspect_results)

# 4. Canopy Height
canopy_results = analyze_canopy_height_geometry(block_wkt, db)
```

```
block_results.update(canopy_results)

# 5. AGB (Biomass)
agb_results = analyze_agb_geometry(block_wkt, db)
block_results.update(agb_results)

# 6. Forest Health
health_results = analyze_forest_health_geometry(block_wkt, db)
block_results.update(health_results)

# 7. Forest Type
forest_type_results = analyze_forest_type_geometry(block_wkt, db)
block_results.update(forest_type_results)

# 8. Land Cover (ESA WorldCover)
landcover_results = analyze_landcover_geometry(block_wkt, db)
block_results.update(landcover_results)

# 9. Forest Loss
loss_results = analyze_forest_loss_geometry(block_wkt, db)
block_results.update(loss_results)

# 10. Forest Gain
gain_results = analyze_forest_gain_geometry(block_wkt, db)
block_results.update(gain_results)

# 11. Fire Loss
fire_results = analyze_fire_loss_geometry(block_wkt, db)
block_results.update(fire_results)

# 12. Temperature
```

```
    temp_results = analyze_temperature_geometry(block_wkt, db)
    block_results.update(temp_results)

    # 13. Precipitation
    precip_results = analyze_precipitation_geometry(block_wkt, db)
    block_results.update(precip_results)

    # 14. Soil Texture
    soil_results = analyze_soil_geometry(block_wkt, db)
    block_results.update(soil_results)

    # 15. Administrative Location (Province, District, Municipality, Ward, Watershed)
    location_results = get_administrative_location(block_wkt, db)
    block_results.update(location_results)

    # 16. Geology Analysis
    geology_results = analyze_geology_geometry(block_wkt, db)
    block_results.update(geology_results)

    # 17. Access Information
    access_results = calculate_access_info(block_wkt, db)
    block_results.update(access_results)

    # 18. Nearby Features (within 100m, by direction)
    print("  Calling analyze_nearby_features...", flush=True)
    try:
        nearby_results = analyze_nearby_features(block_wkt, db)
        print(f"  analyze_nearby_features returned: {list(nearby_results.keys())}", flush=True)
        print(f"  Features: N={nearby_results.get('features_north', 'MISSING')},
E={nearby_results.get('features_east', 'MISSING')}, S={nearby_results.get('features_south',
'MISSING')}, W={nearby_results.get('features_west', 'MISSING')}", flush=True)
        block_results.update(nearby_results)
```

```
        print(f"  After update, block_results has keys: {list(block_results.keys())}", flush=True)
    except Exception as e:
        print(f"  ERROR in analyze_nearby_features: {e}", flush=True)
        import traceback
        traceback.print_exc()
        # Add empty features to avoid crashes
        block_results.update({
            'features_north': None,
            'features_east': None,
            'features_south': None,
            'features_west': None
        })

    return block_results



def calculate_area(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """
    Calculate area using appropriate UTM projection

    Returns area in both square meters and hectares
    """
    # Get geometry centroid to determine UTM zone
    query = text("""
        SELECT
            ST_X(ST_Centroid(boundary_geom)) as lon,
            ST_Y(ST_Centroid(boundary_geom)) as lat
        FROM public.calculations
        WHERE id = :calc_id
    """)
```

```
    result = db.execute(query, {"calc_id": str(calculation_id)}).first()
    lon = result.lon

    # Determine UTM zone (32644 for western Nepal, 32645 for eastern)
    utm_srid = 32645 if lon > 84 else 32644

    # Calculate area in UTM projection
    area_query = text(f"""
      SELECT
        ST_Area(ST_Transform(boundary_geom, {utm_srid})) as area_sqm,
        ST_Area(ST_Transform(boundary_geom, {utm_srid})) / 10000.0 as area_hectares
      FROM public.calculations
      WHERE id = :calc_id
    """)

    area_result = db.execute(area_query, {"calc_id": str(calculation_id)}).first()

    return {
      "area_sqm": round(area_result.area_sqm, 2),
      "area_hectares": round(area_result.area_hectares, 4),
      "utm_zone": utm_srid
    }


async def analyze_rasters(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """
    Analyze all 16 raster datasets from rasters schema

    Performs zonal statistics for each raster layer
    """
    results = {}
```

```
# 1. DEM - Elevation statistics
dem_results = analyze_dem(calculation_id, db)
results.update(dem_results)

# 2. Slope - Classification percentages
slope_results = analyze_slope(calculation_id, db)
results.update(slope_results)

# 3. Aspect - Directional percentages
aspect_results = analyze_aspect(calculation_id, db)
results.update(aspect_results)

# 4. Canopy Height - Forest structure
canopy_results = analyze_canopy_height(calculation_id, db)
results.update(canopy_results)

# 5. Above-ground Biomass (AGB)
agb_results = analyze_agb(calculation_id, db)
results.update(agb_results)

# 6. Forest Health
health_results = analyze_forest_health(calculation_id, db)
results.update(health_results)

# 7. Forest Type
forest_type_results = analyze_forest_type(calculation_id, db)
results.update(forest_type_results)

# 8. ESA WorldCover - Land cover
esa_results = analyze_esa_worldcover(calculation_id, db)
```

```
    results.update(esa_results)

    # 9. Climate - Temperature and Precipitation
    climate_results = analyze_climate(calculation_id, db)
    results.update(climate_results)

    # 10. Forest Loss and Gain
    change_results = analyze_forest_change(calculation_id, db)
    results.update(change_results)

    # 11. Soil properties (temporarily disabled - complex multi-band query)
    # soil_results = analyze_soil(calculation_id, db)
    # results.update(soil_results)
    results.update({"soil_texture": None, "soil_properties": {}})

    return results


def analyze_dem(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze DEM raster - min, max, mean elevation

    DEM NoData value is -32768 but not registered in PostGIS metadata.
    We must explicitly exclude it by filtering pixel values.
    """
    try:
        query = text("""
            WITH clipped_raster AS (
                SELECT ST_Clip(rast, boundary_geom) as rast
                FROM rasters.dem, public.calculations
                WHERE calculations.id = :calc_id
                 AND ST_Intersects(rast, boundary_geom)
```

```
            LIMIT 1
        ),
        pixel_values AS (
            SELECT (ST_PixelAsPolygons(rast)).val as elevation
            FROM clipped_raster
        )
        SELECT
            MIN(elevation) as elevation_min,
            MAX(elevation) as elevation_max,
            AVG(elevation) as elevation_mean
        FROM pixel_values
        WHERE elevation > -32000  -- Exclude NoData values (-32768)
          AND elevation >= 0      -- Nepal minimum elevation
          AND elevation <= 9000   -- Nepal maximum elevation
    """)

    result = db.execute(query, {"calc_id": str(calculation_id)}).first()

    if result and result.elevation_mean is not None:
        return {
            "elevation_min_m": round(result.elevation_min, 1) if result.elevation_min else None,
            "elevation_max_m": round(result.elevation_max, 1) if result.elevation_max else None,
            "elevation_mean_m": round(result.elevation_mean, 1) if result.elevation_mean else
None,
        }
  except Exception as e:
    print(f"Error analyzing DEM: {e}")

  return {"elevation_min_m": None, "elevation_max_m": None, "elevation_mean_m": None}
```

```
def analyze_slope(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze slope raster - categorical codes (0-4)

    Slope classes from raster:
    0 = No data / Water (excluded from analysis)
    1 = <10° (Gentle/Flat)
    2 = 10-20° (Moderate)
    3 = 20-30° (Steep)
    4 = >30° (Very Steep)
    """
    try:
        # Mapping from raster categorical codes to class names
        slope_map = {
            1: "gentle",
            2: "moderate",
            3: "steep",
            4: "very_steep"
        }

        query = text("""
            SELECT
                (pvc).value as slope_code,
                SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
                FROM rasters.slope, public.calculations
                WHERE calculations.id = :calc_id
                 AND ST_Intersects(rast, boundary_geom)
            ) as subquery
            WHERE (pvc).value IS NOT NULL
              AND (pvc).value > 0
```

```
        AND (pvc).value <= 4
      GROUP BY slope_code
    """)

    results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

    if not results:
      return {"slope_dominant_class": None, "slope_percentages": {}}

    total_pixels = sum(r.pixel_count for r in results)
    slope_percentages = {}
    dominant_class = None
    max_percentage = 0

    for r in results:
      code = int(r.slope_code)
      if code in slope_map:
        percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
        class_name = slope_map[code]
        slope_percentages[class_name] = round(percentage, 2)

        if percentage > max_percentage:
          max_percentage = percentage
          dominant_class = class_name

    return {
      "slope_dominant_class": dominant_class,
      "slope_percentages": slope_percentages
    }
  except Exception as e:
    print(f"Error analyzing slope: {e}")
```

```
    return {"slope_dominant_class": None, "slope_percentages": {}}


def analyze_aspect(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze aspect raster - categorical codes (0-8)

    Aspect classes from raster:
    0 = Flat (slope < 2°) - excluded from dominant calculation
    1 = N (337.5° - 22.5°)
    2 = NE (22.5° - 67.5°)
    3 = E (67.5° - 112.5°)
    4 = SE (112.5° - 157.5°)
    5 = S (157.5° - 202.5°)
    6 = SW (202.5° - 247.5°)
    7 = W (247.5° - 292.5°)
    8 = NW (292.5° - 337.5°)
    """
    try:
        # Mapping from raster categorical codes to direction names
        aspect_map = {
            0: "Flat",
            1: "N",
            2: "NE",
            3: "E",
            4: "SE",
            5: "S",
            6: "SW",
            7: "W",
            8: "NW"
        }
```

```
query = text("""
  SELECT
    (pvc).value as aspect_code,
    SUM((pvc).count) as pixel_count
  FROM (
    SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
    FROM rasters.aspect, public.calculations
    WHERE calculations.id = :calc_id
     AND ST_Intersects(rast, boundary_geom)
  ) as subquery
  WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 0 AND 8
  GROUP BY aspect_code
""")

results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

if not results:
  return {"aspect_dominant": None, "aspect_percentages": {}}

total_pixels = sum(r.pixel_count for r in results)
aspect_percentages = {}
dominant_aspect = None
max_percentage = 0

for r in results:
  code = int(r.aspect_code)
  if code in aspect_map:
    percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
    direction = aspect_map[code]
    aspect_percentages[direction] = round(percentage, 2)
```

```
            # Exclude "Flat" from dominant calculation
            if code != 0 and percentage > max_percentage:
                max_percentage = percentage
                dominant_aspect = direction

        # If no non-flat aspects found, use Flat
        if dominant_aspect is None and "Flat" in aspect_percentages:
            dominant_aspect = "Flat"

        return {
            "aspect_dominant": dominant_aspect,
            "aspect_percentages": aspect_percentages
        }
    except Exception as e:
        print(f"Error analyzing aspect: {e}")

    return {"aspect_dominant": None, "aspect_percentages": {}}


def analyze_canopy_height(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze canopy height raster - actual height values in meters

    Canopy height raster contains height values 0-41m:
    0m = Non-forest (agricultural land, open areas, water, etc.)
    1-5m = Bush or shrub land
    6-15m = Pole sized forest
    >15m = High forest
    """
    try:
        query = text("""
```

```
        SELECT
          CASE
            WHEN (pvc).value = 0 THEN 'non_forest'
            WHEN (pvc).value > 0 AND (pvc).value <= 5 THEN 'bush_regenerated'
            WHEN (pvc).value > 5 AND (pvc).value <= 15 THEN 'pole_trees'
            WHEN (pvc).value > 15 THEN 'high_forest'
          END as canopy_class,
          SUM((pvc).count) as pixel_count,
          AVG((pvc).value) as avg_height
        FROM (
          SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
          FROM rasters.canopy_height, public.calculations
          WHERE calculations.id = :calc_id
           AND ST_Intersects(rast, boundary_geom)
        ) as subquery
        WHERE (pvc).value IS NOT NULL
         AND (pvc).value >= 0
         AND (pvc).value <= 50
        GROUP BY canopy_class
    """)

    results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

    if not results:
        return {"canopy_dominant_class": None, "canopy_percentages": {}, "canopy_mean_m":
None}

    total_pixels = sum(r.pixel_count for r in results)
    canopy_percentages = {}
    dominant_class = None
    max_percentage = 0
```

```
        for r in results:
            class_name = r.canopy_class
            percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
            canopy_percentages[class_name] = round(percentage, 2)

            if percentage > max_percentage:
                max_percentage = percentage
                dominant_class = class_name

        # Calculate weighted mean height (including all pixels)
        total_weighted_height = sum(r.avg_height * r.pixel_count for r in results)
        canopy_mean_m = round(total_weighted_height / total_pixels, 1) if total_pixels > 0 else
None

        return {
            "canopy_dominant_class": dominant_class,
            "canopy_percentages": canopy_percentages,
            "canopy_mean_m": canopy_mean_m
        }
    except Exception as e:
        print(f"Error analyzing canopy height: {e}")

    return {"canopy_dominant_class": None, "canopy_percentages": {}, "canopy_mean_m": None}



def analyze_agb(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze above-ground biomass"""
    try:
        query = text("""
            SELECT
```

```
            (stats).mean as agb_mean,
            (stats).sum as agb_total
        FROM (
            SELECT ST_SummaryStats(
                ST_Clip(rast, 1, boundary_geom)
            ) as stats
            FROM rasters.agb_2022_nepal, public.calculations
            WHERE calculations.id = :calc_id
              AND ST_Intersects(rast, boundary_geom)
            LIMIT 1
        ) as subquery
    """)

    result = db.execute(query, {"calc_id": str(calculation_id)}).first()

    if result and result.agb_mean and result.agb_mean > 0:
        # Convert AGB to carbon (carbon is approximately 50% of AGB)
        carbon_stock = result.agb_total * 0.5 if result.agb_total and result.agb_total > 0 else 0

        return {
            "agb_mean_mg_ha": round(result.agb_mean, 2) if result.agb_mean > 0 else 0,
            "agb_total_mg": round(result.agb_total, 2) if result.agb_total and result.agb_total > 0
else 0,
            "carbon_stock_mg": round(carbon_stock, 2) if carbon_stock > 0 else 0
        }
  except Exception as e:
    print(f"Error analyzing AGB: {e}")

  return {"agb_mean_mg_ha": None, "agb_total_mg": None, "carbon_stock_mg": None}
```

```
def analyze_forest_health(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze forest health classes - categorical codes (1-5)

    Forest health classes (Sentinel-2 NDVI):
    1 = Stressed (NDVI < 0.2)
    2 = Poor (NDVI 0.2 - 0.4)
    3 = Moderate (NDVI 0.4 - 0.6)
    4 = Healthy (NDVI 0.6 - 0.8)
    5 = Excellent (NDVI > 0.8)
    """
    try:
        # Correct mapping from table comments
        health_labels = {
            1: "stressed",
            2: "poor",
            3: "moderate",
            4: "healthy",
            5: "excellent"
        }

        query = text("""
            SELECT
                (pvc).value as health_class,
                SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
                FROM rasters.nepal_forest_health, public.calculations
                WHERE calculations.id = :calc_id
                 AND ST_Intersects(rast, boundary_geom)
            ) as subquery
            WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 5
```

```
        GROUP BY health_class
    """)

    results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

    if not results:
        return {"forest_health_dominant": None, "forest_health_percentages": {}}

    total_pixels = sum(r.pixel_count for r in results)
    health_percentages = {}
    dominant_class = None
    max_percentage = 0

    for r in results:
        percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
        class_label = health_labels.get(int(r.health_class), "unknown")
        health_percentages[class_label] = round(percentage, 2)

        if percentage > max_percentage:
            max_percentage = percentage
            dominant_class = class_label

    return {
        "forest_health_dominant": dominant_class,
        "forest_health_percentages": health_percentages
    }
except Exception as e:
    print(f"Error analyzing forest health: {e}")

return {"forest_health_dominant": None, "forest_health_percentages": {}}
```

```
def analyze_forest_type(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze forest type distribution - categorical codes (1-26)

    Forest type classes (FRTC, Kathmandu):
    1 = Shorea robusta Forest
    2 = Tropical Mixed Broadleaved Forest
    3 = Subtropical Mixed Broadleaved Forest
    ... (26 total classes)
    26 = Non forest
    """
    try:
        # Mapping from codes to forest type names
        forest_type_map = {
            1: "Shorea robusta",
            2: "Tropical Mixed Broadleaved",
            3: "Subtropical Mixed Broadleaved",
            4: "Shorea robusta-Mixed Broadleaved",
            5: "Abies Mixed",
            6: "Upper Temperate Coniferous",
            7: "Cool Temperate Mixed Broadleaved",
            8: "Castanopsis Lower Temperate Mixed Broadleaved",
            9: "Pinus roxburghii",
            10: "Alnus",
            11: "Schima",
            12: "Pinus roxburghii-Mixed Broadleaved",
            13: "Pinus wallichiana",
            14: "Warm Temperate Mixed Broadleaved",
            15: "Upper Temperate Quercus",
            16: "Rhododendron arboreum",
            17: "Temperate Rhododendron Mixed Broadleaved",
```

```
    18: "Dalbergia sissoo-Senegalia catechu",
    19: "Terminalia-Tropical Mixed Broadleaved",
    20: "Temperate Mixed Broadleaved",
    21: "Tropical Deciduous Indigenous Riverine",
    22: "Tropical Riverine",
    23: "Lower Temperate Mixed robusta",
    24: "Pinus roxburghii-Shorea robusta",
    25: "Lower Temperate Pinus roxburghii-Quercus",
    26: "Non forest"
}

query = text("""
    SELECT
        (pvc).value as forest_type_code,
        SUM((pvc).count) as pixel_count
    FROM (
        SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
        FROM rasters.forest_type, public.calculations
        WHERE calculations.id = :calc_id
          AND ST_Intersects(rast, boundary_geom)
    ) as subquery
    WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 26
    GROUP BY forest_type_code
    ORDER BY pixel_count DESC
""")

results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

if not results:
    return {"forest_type_dominant": None, "forest_type_percentages": {}}
```

```
        total_pixels = sum(r.pixel_count for r in results)
        forest_type_percentages = {}
        dominant_type = None
        max_percentage = 0

        for r in results:
            code = int(r.forest_type_code)
            if code in forest_type_map:
                percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
                type_name = forest_type_map[code]
                forest_type_percentages[type_name] = round(percentage, 2)

                # Find dominant (excluding "Non forest" if possible)
                if code != 26 and percentage > max_percentage:
                    max_percentage = percentage
                    dominant_type = type_name
                elif code == 26 and dominant_type is None:
                    dominant_type = type_name

        return {
            "forest_type_dominant": dominant_type,
            "forest_type_percentages": forest_type_percentages
        }
    except Exception as e:
        print(f"Error analyzing forest type: {e}")

    return {"forest_type_dominant": None, "forest_type_percentages": {}}


def analyze_esa_worldcover(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze ESA WorldCover land cover - categorical codes
```

```
Land cover classes (ESA/WorldCover/v200):
10 = Tree cover
20 = Shrubland
30 = Grassland
40 = Cropland
50 = Built-up
60 = Bare/sparse vegetation
70 = Snow and ice
80 = Permanent water bodies
90 = Herbaceous wetland
95 = Mangroves
100 = Moss and lichen
"""
try:
    # Mapping from codes to land cover names
    landcover_map = {
        10: "Tree cover",
        20: "Shrubland",
        30: "Grassland",
        40: "Cropland",
        50: "Built-up",
        60: "Bare/sparse vegetation",
        70: "Snow and ice",
        80: "Permanent water bodies",
        90: "Herbaceous wetland",
        95: "Mangroves",
        100: "Moss and lichen"
    }

    query = text("""
```

```
    SELECT
      (pvc).value as landcover_code,
      SUM((pvc).count) as pixel_count
    FROM (
      SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
      FROM rasters.esa_world_cover, public.calculations
      WHERE calculations.id = :calc_id
       AND ST_Intersects(rast, boundary_geom)
    ) as subquery
    WHERE (pvc).value IS NOT NULL
    GROUP BY landcover_code
    ORDER BY pixel_count DESC
""")

results = db.execute(query, {"calc_id": str(calculation_id)}).fetchall()

if not results:
    return {"landcover_dominant": None, "landcover_percentages": {}}

total_pixels = sum(r.pixel_count for r in results)
landcover_percentages = {}
dominant_cover = None
max_percentage = 0

for r in results:
    code = int(r.landcover_code)
    if code in landcover_map:
        percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
        cover_name = landcover_map[code]
        landcover_percentages[cover_name] = round(percentage, 2)
```

```
            if percentage > max_percentage:
                max_percentage = percentage
                dominant_cover = cover_name

        return {
            "landcover_dominant": dominant_cover,
            "landcover_percentages": landcover_percentages
        }
    except Exception as e:
        print(f"Error analyzing ESA WorldCover: {e}")

    return {"landcover_dominant": None, "landcover_percentages": {}}


def analyze_climate(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze climate data (temperature and precipitation)

    WorldClim data:
    - annual_mean_temperature: Bio01, scale = 0.1, unit = deg C
    - min_temp_coldest_month: Bio06, scale = 0.1, unit = deg C
    - annual_precipitation: Bio12, unit = mm
    """
    try:
        # Temperature analysis (annual mean and min)
        temp_query = text("""
            SELECT
                (amt_stats).mean * 0.1 as temp_mean_c,
                (mint_stats).mean * 0.1 as temp_min_c
            FROM (
                SELECT
                    ST_SummaryStats(ST_Clip(amt.rast, calc.boundary_geom), 1, true) as amt_stats,
```

```
        ST_SummaryStats(ST_Clip(mint.rast, calc.boundary_geom), 1, true) as mint_stats
      FROM rasters.annual_mean_temperature amt,
        rasters.min_temp_coldest_month mint,
        public.calculations calc
      WHERE calc.id = :calc_id
       AND ST_Intersects(amt.rast, calc.boundary_geom)
       AND ST_Intersects(mint.rast, calc.boundary_geom)
      LIMIT 1
    ) as subquery
""")

temp_result = db.execute(temp_query, {"calc_id": str(calculation_id)}).first()

# Precipitation analysis
precip_query = text("""
  SELECT
    (stats).mean as precip_mean_mm
  FROM (
    SELECT ST_SummaryStats(
      ST_Clip(rast, boundary_geom),
      1,
      true
    ) as stats
    FROM rasters.annual_precipitation, public.calculations
    WHERE calculations.id = :calc_id
     AND ST_Intersects(rast, boundary_geom)
    LIMIT 1
  ) as subquery
""")

precip_result = db.execute(precip_query, {"calc_id": str(calculation_id)}).first()
```

```
        # Filter out invalid values (None, NaN, Infinity)
        temp_mean = None
        temp_min = None
        precip_mean = None

        if temp_result and temp_result.temp_mean_c is not None:
            import math
            if not math.isinf(temp_result.temp_mean_c) and not
math.isnan(temp_result.temp_mean_c):
                temp_mean = round(temp_result.temp_mean_c, 1)

        if temp_result and temp_result.temp_min_c is not None:
            import math
            if not math.isinf(temp_result.temp_min_c) and not math.isnan(temp_result.temp_min_c):
                temp_min = round(temp_result.temp_min_c, 1)

        if precip_result and precip_result.precip_mean_mm is not None:
            import math
            if not math.isinf(precip_result.precip_mean_mm) and not
math.isnan(precip_result.precip_mean_mm):
                precip_mean = round(precip_result.precip_mean_mm, 1)

        return {
            "temperature_mean_c": temp_mean,
            "temperature_min_c": temp_min,
            "precipitation_mean_mm": precip_mean
        }
    except Exception as e:
        print(f"Error analyzing climate: {e}")

    return {
```

```
        "temperature_mean_c": None,
        "temperature_min_c": None,
        "precipitation_mean_mm": None
    }


def analyze_forest_change(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze forest loss and gain (Hansen Global Forest Change)

    nepal_lossyear: 0 = no loss, 1-24 = year of loss (2001-2024)
    nepal_gain: 0 = no gain, 1 = forest gain (2000-2012)
    forest_loss_fire: 0 = no fire loss, 1-24 = year of fire loss (2001-2024)
    """
    try:
        # Get area calculation for converting pixels to hectares
        area_query = text("""
            SELECT ST_Area(ST_Transform(boundary_geom, 32645)) / 10000.0 as area_hectares
            FROM public.calculations
            WHERE id = :calc_id
        """)
        area_result = db.execute(area_query, {"calc_id": str(calculation_id)}).first()
        total_area_ha = area_result.area_hectares if area_result else 0

        # Forest loss by year
        loss_query = text("""
            SELECT
                (pvc).value as loss_year_code,
                SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
                FROM rasters.nepal_lossyear, public.calculations
```

```
      WHERE calculations.id = :calc_id
        AND ST_Intersects(rast, boundary_geom)
    ) as subquery
    WHERE (pvc).value IS NOT NULL AND (pvc).value > 0
    GROUP BY loss_year_code
    ORDER BY loss_year_code
""")

loss_results = db.execute(loss_query, {"calc_id": str(calculation_id)}).fetchall()

# Forest gain
gain_query = text("""
    SELECT
      SUM((pvc).count) as gain_pixels
    FROM (
      SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
      FROM rasters.nepal_gain, public.calculations
      WHERE calculations.id = :calc_id
        AND ST_Intersects(rast, boundary_geom)
    ) as subquery
    WHERE (pvc).value = 1
""")

gain_result = db.execute(gain_query, {"calc_id": str(calculation_id)}).first()

# Forest loss from fire
fire_query = text("""
    SELECT
      (pvc).value as fire_year_code,
      SUM((pvc).count) as pixel_count
    FROM (
```

```
      SELECT ST_ValueCount(ST_Clip(rast, boundary_geom)) as pvc
      FROM rasters.forest_loss_fire, public.calculations
      WHERE calculations.id = :calc_id
       AND ST_Intersects(rast, boundary_geom)
   ) as subquery
   WHERE (pvc).value IS NOT NULL AND (pvc).value > 0
   GROUP BY fire_year_code
""")

fire_results = db.execute(fire_query, {"calc_id": str(calculation_id)}).fetchall()

# Calculate total pixels and percentages
total_loss_pixels = sum(r.pixel_count for r in loss_results) if loss_results else 0
gain_pixels = gain_result.gain_pixels if gain_result and gain_result.gain_pixels else 0
total_fire_pixels = sum(r.pixel_count for r in fire_results) if fire_results else 0

# Estimate hectares (30m pixel = 900 sqm = 0.09 ha)
pixel_area_ha = 0.09
loss_hectares = total_loss_pixels * pixel_area_ha
gain_hectares = gain_pixels * pixel_area_ha
fire_loss_hectares = total_fire_pixels * pixel_area_ha

# Build year-by-year loss data
forest_loss_by_year = {}
for r in loss_results:
   year = 2000 + int(r.loss_year_code)
   forest_loss_by_year[str(year)] = round(r.pixel_count * pixel_area_ha, 2)

# Build year-by-year fire loss data
fire_loss_by_year = {}
for r in fire_results:
```

```
            year = 2000 + int(r.fire_year_code)
            fire_loss_by_year[str(year)] = round(r.pixel_count * pixel_area_ha, 2)

        return {
            "forest_loss_hectares": round(loss_hectares, 2) if loss_hectares > 0 else 0,
            "forest_gain_hectares": round(gain_hectares, 2) if gain_hectares > 0 else 0,
            "fire_loss_hectares": round(fire_loss_hectares, 2) if fire_loss_hectares > 0 else 0,
            "forest_loss_by_year": forest_loss_by_year,
            "fire_loss_by_year": fire_loss_by_year
        }
    except Exception as e:
        print(f"Error analyzing forest change: {e}")

    return {
        "forest_loss_hectares": 0,
        "forest_gain_hectares": 0,
        "fire_loss_hectares": 0,
        "forest_loss_by_year": {},
        "fire_loss_by_year": {}
    }


def analyze_soil(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """Analyze soil properties (ISRIC SoilGrids)

    8 bands from soilgrids_isric:
    Band 1 = clay_g_kg (Clay content in g/kg)
    Band 2 = sand_g_kg (Sand content in g/kg)
    Band 3 = silt_g_kg (Silt content in g/kg)
    Band 4 = ph_h2o (Soil pH in H2O)
    Band 5 = soc_dg_kg (Soil organic carbon in dg/kg)
```

```
Band 6 = nitrogen_cg_kg (Nitrogen content in cg/kg)
Band 7 = bdod_cg_cm3 (Bulk density in cg/cm3)
Band 8 = cec_mmol_kg (Cation exchange capacity in mmol/kg)
"""
try:
  query = text("""
    SELECT
      (clay_stats).mean as clay_mean,
      (sand_stats).mean as sand_mean,
      (silt_stats).mean as silt_mean,
      (ph_stats).mean as ph_mean,
      (soc_stats).mean as soc_mean,
      (nitrogen_stats).mean as nitrogen_mean,
      (bdod_stats).mean as bdod_mean,
      (cec_stats).mean as cec_mean
    FROM (
      SELECT
        ST_SummaryStats(ST_Clip(rast, 1, boundary_geom), 1, true) as clay_stats,
        ST_SummaryStats(ST_Clip(rast, 2, boundary_geom), 1, true) as sand_stats,
        ST_SummaryStats(ST_Clip(rast, 3, boundary_geom), 1, true) as silt_stats,
        ST_SummaryStats(ST_Clip(rast, 4, boundary_geom), 1, true) as ph_stats,
        ST_SummaryStats(ST_Clip(rast, 5, boundary_geom), 1, true) as soc_stats,
        ST_SummaryStats(ST_Clip(rast, 6, boundary_geom), 1, true) as nitrogen_stats,
        ST_SummaryStats(ST_Clip(rast, 7, boundary_geom), 1, true) as bdod_stats,
        ST_SummaryStats(ST_Clip(rast, 8, boundary_geom), 1, true) as cec_stats
      FROM rasters.soilgrids_isric, public.calculations
      WHERE calculations.id = :calc_id
       AND ST_Intersects(rast, boundary_geom)
      LIMIT 1
    ) as subquery
  """)
```

```
result = db.execute(query, {"calc_id": str(calculation_id)}).first()

if result:
    # Determine soil texture class based on clay/sand/silt percentages
    clay_pct = result.clay_mean / 10 if result.clay_mean else 0  # g/kg to %
    sand_pct = result.sand_mean / 10 if result.sand_mean else 0
    silt_pct = result.silt_mean / 10 if result.silt_mean else 0

    # Simple texture classification
    soil_texture = "Unknown"
    if clay_pct > 40:
        soil_texture = "Clay"
    elif sand_pct > 50:
        soil_texture = "Sandy"
    elif silt_pct > 40:
        soil_texture = "Silty"
    else:
        soil_texture = "Loam"

    return {
        "soil_texture": soil_texture,
        "soil_properties": {
            "clay_g_kg": round(result.clay_mean, 1) if result.clay_mean else None,
            "sand_g_kg": round(result.sand_mean, 1) if result.sand_mean else None,
            "silt_g_kg": round(result.silt_mean, 1) if result.silt_mean else None,
            "ph_h2o": round(result.ph_mean, 2) if result.ph_mean else None,
            "soc_dg_kg": round(result.soc_mean, 1) if result.soc_mean else None,
            "nitrogen_cg_kg": round(result.nitrogen_mean, 1) if result.nitrogen_mean else None,
            "bulk_density_cg_cm3": round(result.bdod_mean, 1) if result.bdod_mean else None,
            "cec_mmol_kg": round(result.cec_mean, 1) if result.cec_mean else None
```

```
            }
        }
    except Exception as e:
        print(f"Error analyzing soil: {e}")

    return {"soil_texture": None, "soil_properties": {}}


async def analyze_vectors(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """
    Analyze vector datasets - proximity and intersection analysis
    """
    results = {}

    # Placeholder for vector analysis
    results["buildings_within_1km"] = 0
    results["nearest_settlement"] = None
    results["nearest_road"] = None

    return results


def get_administrative_location(geometry_wkt: str, db: Session) -> Dict[str, Any]:
    """
    Get administrative location by intersecting geometry centroid with admin boundaries

    Args:
        geometry_wkt: WKT string of geometry
        db: Database session
```

```
    Returns:
        Dict with: province, district, municipality, ward, watershed, major_river_basin
    """
    location = {}

    # Query for province
    province_query = text("""
        SELECT p.province
        FROM admin.province p
        WHERE ST_Intersects(
            p.shape,
            ST_Centroid(ST_GeomFromText(:wkt, 4326))
        )
        LIMIT 1
    """)
    province_result = db.execute(province_query, {"wkt": geometry_wkt}).first()
    location["province"] = province_result.province if province_result else None

    # Query for municipality (includes district)
    municipality_query = text("""
        SELECT m.district, m.gapa_napa as municipality
        FROM admin.municipality m
        WHERE ST_Intersects(
            m.geom,
            ST_Centroid(ST_GeomFromText(:wkt, 4326))
        )
        LIMIT 1
    """)
    municipality_result = db.execute(municipality_query, {"wkt": geometry_wkt}).first()
    if municipality_result:
```

```
        location["district"] = municipality_result.district
        location["municipality"] = municipality_result.municipality
    else:
        location["district"] = None
        location["municipality"] = None

    # Query for ward
    ward_query = text("""
        SELECT w.ward
        FROM admin.ward w
        WHERE ST_Intersects(
            w.geom,
            ST_Centroid(ST_GeomFromText(:wkt, 4326))
        )
        LIMIT 1
    """)
    ward_result = db.execute(ward_query, {"wkt": geometry_wkt}).first()
    location["ward"] = str(ward_result.ward) if ward_result else None

    # Query for watershed
    watershed_query = text("""
        SELECT w."watershed name" as watershed_name, w."major river basin" as
major_river_basin
        FROM admin."watershed_Nepal" w
        WHERE ST_Intersects(
            w.geom,
            ST_Centroid(ST_GeomFromText(:wkt, 4326))
        )
        LIMIT 1
    """)
    watershed_result = db.execute(watershed_query, {"wkt": geometry_wkt}).first()
```

```
    if watershed_result:
        location["watershed"] = watershed_result.watershed_name
        location["major_river_basin"] = watershed_result.major_river_basin
    else:
        location["watershed"] = None
        location["major_river_basin"] = None

    return location




def analyze_geology_geometry(geometry_wkt: str, db: Session) -> Dict[str, Any]:
    """
    Analyze geology classes that intersect with the geometry
    Returns percentage coverage for each geology class

    Args:
        geometry_wkt: WKT string of geometry
        db: Database session

    Returns:
        Dict with geology_percentages: {class_name: percentage}
    """
    geology_query = text("""
        WITH input_geom AS (
            SELECT ST_GeomFromText(:wkt, 4326) as geom
        ),
        total_area AS (
            SELECT ST_Area(ST_Transform(geom, 32645)) as area
            FROM input_geom
```

```
  ),
  geology_intersections AS (
    SELECT
      g."geology class" as geology_class,
      ST_Area(
        ST_Transform(
          ST_Intersection(g.geom, i.geom),
          32645
        )
      ) as intersection_area
    FROM geology.geology g, input_geom i
    WHERE ST_Intersects(g.geom, i.geom)
    AND g."geology class" IS NOT NULL
    AND g."geology class" != 'No Data'
  )
  SELECT
    geology_class,
    (intersection_area / (SELECT area FROM total_area)) * 100 as percentage
  FROM geology_intersections
  WHERE intersection_area > 0
  ORDER BY percentage DESC
""")

try:
  result = db.execute(geology_query, {"wkt": geometry_wkt})
  geology_data = {}

  for row in result:
    if row.geology_class and row.percentage:
      geology_data[row.geology_class] = round(float(row.percentage), 2)
```

```
        return {
            "geology_percentages": geology_data if geology_data else None
        }
    except Exception as e:
        print(f"Geology analysis error: {e}")
        return {"geology_percentages": None}


def calculate_access_info(geometry_wkt: str, db: Session) -> Dict[str, Any]:
    """
    Calculate access information: distance and direction to nearest district headquarters

    Args:
        geometry_wkt: WKT string of geometry
        db: Database session

    Returns:
        Dict with access_info: "Location Direction (degrees°) distance km"
    """
    access_query = text("""
        WITH input_geom AS (
            SELECT
                ST_GeomFromText(:wkt, 4326) as geom,
                ST_Centroid(ST_GeomFromText(:wkt, 4326)) as centroid
        ),
        nearest_hq AS (
            SELECT
                dh."head quarter" as name,
                ST_Distance(
                    ST_Transform(i.centroid, 32645),
                    ST_Transform(dh.geom, 32645)
```

```
        ) / 1000.0 as distance_km,
        degrees(
          ST_Azimuth(
            i.centroid,
            dh.geom
          )
        ) as azimuth_degrees
      FROM admin."district Headquarter" dh, input_geom i
      WHERE dh."head quarter" IS NOT NULL
      ORDER BY ST_Distance(i.centroid, dh.geom)
      LIMIT 1
    )
    SELECT
      name,
      distance_km,
      azimuth_degrees,
      CASE
        WHEN azimuth_degrees >= 337.5 OR azimuth_degrees < 22.5 THEN 'North'
        WHEN azimuth_degrees >= 22.5 AND azimuth_degrees < 67.5 THEN 'Northeast'
        WHEN azimuth_degrees >= 67.5 AND azimuth_degrees < 112.5 THEN 'East'
        WHEN azimuth_degrees >= 112.5 AND azimuth_degrees < 157.5 THEN 'Southeast'
        WHEN azimuth_degrees >= 157.5 AND azimuth_degrees < 202.5 THEN 'South'
        WHEN azimuth_degrees >= 202.5 AND azimuth_degrees < 247.5 THEN 'Southwest'
        WHEN azimuth_degrees >= 247.5 AND azimuth_degrees < 292.5 THEN 'West'
        ELSE 'Northwest'
      END as direction
    FROM nearest_hq
"""")

try:
    result = db.execute(access_query, {"wkt": geometry_wkt}).first()
```

```
        if result and result.name:
            access_str = f"{result.name} {result.direction} ({int(result.azimuth_degrees)}°) {result.distance_km:.1f} km"
            return {"access_info": access_str}
        else:
            return {"access_info": None}
    except Exception as e:
        print(f"Access calculation error: {e}")
        return {"access_info": None}


def analyze_nearby_features(geometry_wkt: str, db: Session) -> Dict[str, Any]:
    """
    Find natural and infrastructure features within 100m of boundary
    Reports features by direction: North, East, South, West

    Args:
        geometry_wkt: WKT string of geometry
        db: Database session

    Returns:
        Dict with features_north, features_east, features_south, features_west
    """

    # Define feature tables and their name columns
    feature_tables = [
        ('river.river_line', ['river_name', 'features']),
        ('river.ridge', ['ridge_name']),
        ('infrastructure.road', ['name', 'name_en', 'highway']),
        ('infrastructure.poi', ['name', 'name_en', 'amenity', 'shop', 'tourism']),
```

```
    ('infrastructure.health_facilities', ['hf_type', 'vdc_name1']),
    ('infrastructure.education_facilities', ['name', 'name_en', 'amenity']),
    ('buildings.building', ['Building']),  # Generic name
    ('admin.settlement', ['vil_name']),
    ('admin."esa_forest_Boundary"', ['description', '"boundary of"']),
  ]

  directions = {
    'north': (315, 45),     # 315° to 45°
    'east': (45, 135),      # 45° to 135°
    'south': (135, 225),    # 135° to 225°
    'west': (225, 315)      # 225° to 315° (wraps to 360/0)
  }

  result_features = {}

  for direction_name, (start_angle, end_angle) in directions.items():
    features_list = []
    print(f"  Querying features in {direction_name} direction ({start_angle}° to {end_angle}°)...")

    # Build query for this direction
    for table_name, name_columns in feature_tables:
      # Build COALESCE for name columns
      name_coalesce = "COALESCE(" + ", ".join([f"f.{col}" if '"' not in col else f'f.{col}' for col in
name_columns]) + ")"

      # Handle angle wrapping for North (315-45)
      if direction_name == 'north':
        angle_condition = f"(azimuth_deg >= {start_angle} OR azimuth_deg < {end_angle})"
      else:
        angle_condition = f"(azimuth_deg >= {start_angle} AND azimuth_deg < {end_angle})"
```

```
        direction_query = text(f"""
            WITH input_geom AS (
                SELECT ST_GeomFromText(:wkt, 4326) as geom
            ),
            nearby AS (
                SELECT
                    {name_coalesce} as feature_name,
                    ST_Distance(ST_Transform(i.geom, 32645), ST_Transform(f.geom, 32645)) as
distance_m,
                    degrees(ST_Azimuth(ST_Centroid(i.geom), ST_ClosestPoint(f.geom,
ST_Centroid(i.geom)))) as azimuth_deg
                FROM {table_name} f, input_geom i
                WHERE ST_DWithin(ST_Transform(i.geom, 32645), ST_Transform(f.geom, 32645), 100)
            )
            SELECT DISTINCT feature_name
            FROM nearby
            WHERE feature_name IS NOT NULL
            AND feature_name != ''
            AND {angle_condition}
            LIMIT 10
        """)

        try:
            result = db.execute(direction_query, {"wkt": geometry_wkt})
            row_count = 0
            for row in result:
                row_count += 1
                if row.feature_name and row.feature_name.strip():
                    features_list.append(row.feature_name.strip())
            # Debug: Log if no results found for this table in this direction
            if row_count == 0:
```

```
                print(f"  No features from {table_name} in {direction_name} direction")
            except Exception as e:
                # Skip tables that might have issues
                print(f"  Error querying {table_name} for {direction_name}: {str(e)[:100]}")
                # CRITICAL: Rollback transaction to prevent failed state from affecting subsequent queries
                db.rollback()
                continue

        # Store comma-separated list or None
        features_str = ", ".join(features_list) if features_list else None
        result_features[f"features_{direction_name}"] = features_str
        # Use safe printing to handle Unicode characters
        try:
            print(f"Direction {direction_name}: Found {len(features_list)} features - {features_str if features_str else 'None'}")
        except UnicodeEncodeError:
            print(f"Direction {direction_name}: Found {len(features_list)} features (contains non-ASCII characters)")

        # Debug: Show what's in result_features after each direction
        print(f"  result_features now has {len(result_features)} keys: {list(result_features.keys())}")

    # Final debug before return
    print(f"  FINAL result_features has {len(result_features)} keys: {list(result_features.keys())}")
    for key, val in result_features.items():
        val_preview = val[:50] if val else None
        print(f"    {key}: {val_preview}")

    return result_features
```

```
async def analyze_admin_boundaries(calculation_id: UUID, db: Session) -> Dict[str, Any]:
    """
    Analyze administrative boundaries - province, municipality, ward
    """
    return {
        "province": None,
        "municipality": None,
        "ward": None
    }


#
# ============================================================================
# GEOMETRY-BASED ANALYSIS FUNCTIONS (for individual blocks)
#
# ============================================================================

def analyze_dem_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze DEM raster for a specific geometry (WKT)

    DEM contains 16-bit signed integer values in meters.
    NoData value is -32768 but PostGIS doesn't know this (NoData=NULL in raster metadata).
    We must explicitly exclude -32768 values by filtering in the query.
    """
    try:
        # Use ST_PixelAsPolygons to get all pixel values, then filter and aggregate
        query = text("""
            WITH clipped_raster AS (
                SELECT ST_Clip(rast, ST_GeomFromText(:wkt, 4326)) as rast
                FROM rasters.dem
                WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
                LIMIT 1
```

```
        ),
        pixel_values AS (
          SELECT (ST_PixelAsPolygons(rast)).val as elevation
          FROM clipped_raster
        )
        SELECT
          MIN(elevation) as elevation_min,
          MAX(elevation) as elevation_max,
          AVG(elevation) as elevation_mean
        FROM pixel_values
        WHERE elevation > -32000  -- Exclude NoData values (-32768)
          AND elevation >= 0      -- Nepal minimum elevation
          AND elevation <= 9000   -- Nepal maximum elevation
      """)

      result = db.execute(query, {"wkt": wkt}).first()

      if result and result.elevation_mean is not None:
        return {
          "elevation_min_m": round(result.elevation_min, 1) if result.elevation_min else None,
          "elevation_max_m": round(result.elevation_max, 1) if result.elevation_max else None,
          "elevation_mean_m": round(result.elevation_mean, 1) if result.elevation_mean else
None
        }
    except Exception as e:
      print(f"Error analyzing DEM: {e}")

    return {"elevation_min_m": None, "elevation_max_m": None, "elevation_mean_m": None}


def analyze_slope_geometry(wkt: str, db: Session) -> Dict[str, Any]:
```

```
"""Analyze slope for a specific geometry

Slope raster contains categorical codes (8-bit unsigned):
0 = No data / Water (excluded)
1 = Gentle (<10°)
2 = Moderate (10-20°)
3 = Steep (20-30°)
4 = Very Steep (>30°)
"""
try:
  # Mapping from raster values to class names (excluding 0)
  slope_map = {
    1: "gentle",
    2: "moderate",
    3: "steep",
    4: "very_steep"
  }

  query = text("""
    SELECT
      (pvc).value as slope_code,
      SUM((pvc).count) as pixel_count
    FROM (
      SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
      FROM rasters.slope
      WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
    ) as subquery
    WHERE (pvc).value IS NOT NULL
     AND (pvc).value > 0
     AND (pvc).value <= 4
    GROUP BY slope_code
```

```
        """)

        results = db.execute(query, {"wkt": wkt}).fetchall()

        if results:
            total_pixels = sum(r.pixel_count for r in results)
            percentages = {}
            for r in results:
                code = int(r.slope_code)
                if code in slope_map:
                    percentages[slope_map[code]] = round((r.pixel_count / total_pixels) * 100, 2)

            if percentages:
                dominant = max(percentages.items(), key=lambda x: x[1])[0]
                return {
                    "slope_dominant_class": dominant,
                    "slope_percentages": percentages
                }
    except Exception as e:
        print(f"Error analyzing slope: {e}")

    return {"slope_dominant_class": None, "slope_percentages": {}}


def analyze_aspect_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze aspect for a specific geometry

    Aspect raster contains categorical codes (8-bit unsigned):
    0 = Flat (excluded from dominant), 1 = N, 2 = NE, 3 = E, 4 = SE, 5 = S, 6 = SW, 7 = W, 8 = NW
    """
    try:
```

```
# Mapping from raster values to direction names
aspect_map = {
    0: "Flat",
    1: "N",
    2: "NE",
    3: "E",
    4: "SE",
    5: "S",
    6: "SW",
    7: "W",
    8: "NW"
}

query = text("""
    SELECT
        (pvc).value as aspect_code,
        SUM((pvc).count) as pixel_count
    FROM (
        SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
        FROM rasters.aspect
        WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
    ) as subquery
    WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 0 AND 8
    GROUP BY aspect_code
""")

results = db.execute(query, {"wkt": wkt}).fetchall()

if results:
    total_pixels = sum(r.pixel_count for r in results)
    percentages = {}
```

```
        for r in results:
            code = int(r.aspect_code)
            if code in aspect_map:
                percentages[aspect_map[code]] = round((r.pixel_count / total_pixels) * 100, 2)

        if percentages:
            # Find dominant (excluding Flat if possible)
            non_flat = {k: v for k, v in percentages.items() if k != "Flat"}
            dominant = max(non_flat.items(), key=lambda x: x[1])[0] if non_flat else "Flat"

            return {
                "aspect_dominant": dominant,
                "aspect_percentages": percentages
            }
    except Exception as e:
        print(f"Error analyzing aspect: {e}")

    return {"aspect_dominant": None, "aspect_percentages": {}}


def analyze_canopy_height_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze canopy height for a specific geometry

    Canopy height raster contains actual height values in meters (0-41m):
    0m = Non-forest (agricultural land, open areas, water, etc.)
    1-5m = Bush or shrub land
    6-15m = Pole sized forest
    >15m = High forest (tree sized)

    Uses conservative pixel containment: only pixels whose center point
    is FULLY CONTAINED within the polygon boundary are counted.
```

```
This matches QGIS zonal statistics behavior more closely.
"""
try:
    # Use ST_PixelAsPolygons with geom parameter to get pixel center points
    # Filter to only include pixels where center is CONTAINED (not just intersects)
    query = text("""
        WITH boundary AS (
            SELECT ST_GeomFromText(:wkt, 4326) as geom
        ),
        all_pixels AS (
            SELECT
                (pix).val as height,
                (pix).geom as pixel_center
            FROM (
                SELECT ST_PixelAsPolygons(rast) as pix
                FROM rasters.canopy_height, boundary
                WHERE ST_Intersects(rast, geom)
            ) as subq
        )
        SELECT
            CASE
                WHEN height = 0 THEN 'non_forest'
                WHEN height > 0 AND height <= 5 THEN 'bush_regenerated'
                WHEN height > 5 AND height <= 15 THEN 'pole_trees'
                WHEN height > 15 THEN 'high_forest'
            END as canopy_class,
            COUNT(*) as pixel_count,
            AVG(height) as avg_height
        FROM all_pixels, boundary
        WHERE ST_Contains(boundary.geom, ST_Centroid(pixel_center))
          AND height IS NOT NULL
```

```
            AND height >= 0
            AND height <= 50
          GROUP BY canopy_class
        """)

        results = db.execute(query, {"wkt": wkt}).fetchall()

        if results:
          total_pixels = sum(r.pixel_count for r in results)
          percentages = {r.canopy_class: round((r.pixel_count / total_pixels) * 100, 2) for r in results}
          dominant = max(percentages.items(), key=lambda x: x[1])[0]

          # Calculate weighted mean height (including all pixels)
          total_weighted_height = sum(r.avg_height * r.pixel_count for r in results)
          canopy_mean_m = round(total_weighted_height / total_pixels, 1) if total_pixels > 0 else
None

          return {
            "canopy_mean_m": canopy_mean_m,
            "canopy_dominant_class": dominant,
            "canopy_percentages": percentages
          }
      except Exception as e:
        print(f"Error analyzing canopy height: {e}")

      return {"canopy_mean_m": None, "canopy_dominant_class": None, "canopy_percentages": {}}


def analyze_agb_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze above-ground biomass for a specific geometry
```

```
AGB raster (16-bit unsigned) has 2 bands:
Band 1 = AGB estimate in Mg/ha, Band 2 = standard deviation
"""
try:
    query = text("""
        SELECT
            (stats).mean as agb_mean,
            (stats).sum as agb_total
        FROM (
            SELECT ST_SummaryStats(
                ST_Clip(rast, 1, ST_GeomFromText(:wkt, 4326)),
                1,  -- band 1
                true  -- exclude nodata
            ) as stats
            FROM rasters.agb_2022_nepal
            WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
            LIMIT 1
        ) as subquery
    """)

    result = db.execute(query, {"wkt": wkt}).first()

    if result and result.agb_total:
        return {
            "agb_mean_mg_ha": round(result.agb_mean, 2) if result.agb_mean else None,
            "agb_total_mg": round(result.agb_total, 2) if result.agb_total else None,
            "carbon_stock_mg": round(result.agb_total * 0.5, 2) if result.agb_total else None
        }
except Exception as e:
    print(f"Error analyzing AGB: {e}")
```

```
    return {"agb_mean_mg_ha": None, "agb_total_mg": None, "carbon_stock_mg": None}


def analyze_forest_health_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze forest health for a specific geometry

    Forest health classes (8-bit unsigned):
    1=Stressed, 2=Poor, 3=Moderate, 4=Healthy, 5=Excellent
    """
    try:
        # Correct mapping according to table comment
        health_map = {
            1: "stressed",
            2: "poor",
            3: "moderate",
            4: "healthy",
            5: "excellent"
        }

        query = text("""
            SELECT
                (pvc).value as health_class,
                SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
                FROM rasters.nepal_forest_health
                WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
            ) as subquery
            WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 5
            GROUP BY health_class
        """)
```

```
        results = db.execute(query, {"wkt": wkt}).fetchall()

        if results:
            total_pixels = sum(r.pixel_count for r in results)
            percentages = {}
            for r in results:
                class_val = int(r.health_class)
                if class_val in health_map:
                    percentages[health_map[class_val]] = round((r.pixel_count / total_pixels) * 100, 2)

            if percentages:
                dominant = max(percentages.items(), key=lambda x: x[1])[0]
                return {
                    "forest_health_dominant": dominant,
                    "forest_health_percentages": percentages
                }
    except Exception as e:
        print(f"Error analyzing forest health: {e}")

    return {"forest_health_dominant": None, "forest_health_percentages": {}}


def analyze_forest_type_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze forest type for a specific geometry (WKT)"""
    try:
        forest_type_map = {
            1: "Shorea robusta", 2: "Alnus nepalensis", 3: "Schima-Castanopsis",
            4: "Quercus semecarpifolia", 5: "Larix/Abies spectabilis",
            6: "Pinus wallichiana-Tsuga dumosa", 7: "Plantation (Pinus-Eucalyptus)",
            8: "Ficus-Other Tropical Riverine", 9: "Tropical Mixed Broadleaved",
```

```
    10: "Quercus-Pinus", 11: "Abies spectabilis",
    12: "Pinus roxburghii-Mixed Broadleaved", 13: "Pinus wallichiana",
    14: "Warm Temperate Mixed Broadleaved", 15: "Upper Temperate Quercus",
    16: "Rhododendron arboreum", 17: "Temperate Rhododendron Mixed Broadleaved",
    18: "Dalbergia sissoo-Senegalia catechu", 19: "Terminalia-Tropical Mixed Broadleaved",
    20: "Temperate Mixed Broadleaved", 21: "Tropical Deciduous Indigenous Riverine",
    22: "Tropical Riverine", 23: "Lower Temperate Mixed robusta",
    24: "Pinus roxburghii-Shorea robusta", 25: "Lower Temperate Pinus roxburghii-Quercus",
    26: "Non forest"
}
query = text("""
    SELECT (pvc).value as forest_type_code, SUM((pvc).count) as pixel_count
    FROM (
        SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
        FROM rasters.forest_type WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
    ) as subquery
    WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 26
    GROUP BY forest_type_code ORDER BY pixel_count DESC
""")
results = db.execute(query, {"wkt": wkt}).fetchall()
if not results:
    return {"forest_type_dominant": None, "forest_type_percentages": {}}
total_pixels = sum(r.pixel_count for r in results)
forest_type_percentages = {}
dominant_type = None
max_percentage = 0
for r in results:
    code = int(r.forest_type_code)
    if code in forest_type_map:
        percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
        type_name = forest_type_map[code]
```

```
            forest_type_percentages[type_name] = round(percentage, 2)
            if code != 26 and percentage > max_percentage:
                max_percentage = percentage
                dominant_type = type_name
            elif code == 26 and dominant_type is None:
                dominant_type = type_name
        return {"forest_type_dominant": dominant_type, "forest_type_percentages":
forest_type_percentages}
    except Exception as e:
        print(f"Error analyzing forest type for geometry: {e}")
    return {"forest_type_dominant": None, "forest_type_percentages": {}}



def analyze_landcover_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze ESA WorldCover land cover for a specific geometry (WKT)"""
    try:
        landcover_map = {
            10: "Tree cover", 20: "Shrubland", 30: "Grassland", 40: "Cropland",
            50: "Built-up", 60: "Bare/sparse vegetation", 70: "Snow and ice",
            80: "Permanent water bodies", 90: "Herbaceous wetland", 95: "Mangroves",
            100: "Moss and lichen"
        }
        query = text("""
            SELECT (pvc).value as landcover_code, SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
                FROM rasters.esa_world_cover WHERE ST_Intersects(rast, ST_GeomFromText(:wkt,
4326))
            ) as subquery
            WHERE (pvc).value IS NOT NULL
            GROUP BY landcover_code ORDER BY pixel_count DESC
        """)
```

```
        results = db.execute(query, {"wkt": wkt}).fetchall()
        if not results:
            return {"landcover_dominant": None, "landcover_percentages": {}}
        total_pixels = sum(r.pixel_count for r in results)
        landcover_percentages = {}
        dominant_cover = None
        max_percentage = 0
        for r in results:
            code = int(r.landcover_code)
            if code in landcover_map:
                percentage = (r.pixel_count / total_pixels * 100) if total_pixels > 0 else 0
                cover_name = landcover_map[code]
                landcover_percentages[cover_name] = round(percentage, 2)
                if percentage > max_percentage:
                    max_percentage = percentage
                    dominant_cover = cover_name
        return {"landcover_dominant": dominant_cover, "landcover_percentages":
landcover_percentages}
    except Exception as e:
        print(f"Error analyzing land cover for geometry: {e}")
    return {"landcover_dominant": None, "landcover_percentages": {}}



def analyze_forest_loss_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze Hansen forest loss (2001-2023) for a specific geometry"""
    try:
        query = text("""
            SELECT (pvc).value as loss_year, SUM((pvc).count) as pixel_count
            FROM (
                SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
                FROM rasters.nepal_lossyear WHERE ST_Intersects(rast, ST_GeomFromText(:wkt,
4326))
```

```
        ) as subquery
        WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 23
        GROUP BY loss_year ORDER BY loss_year
    """)
    results = db.execute(query, {"wkt": wkt}).fetchall()
    if not results:
        return {"forest_loss_hectares": 0, "forest_loss_by_year": {}}
    pixel_area_ha = 0.0009
    loss_by_year = {}
    total_loss_pixels = 0
    for r in results:
        year_code = int(r.loss_year)
        actual_year = 2000 + year_code
        loss_ha = r.pixel_count * pixel_area_ha
        loss_by_year[str(actual_year)] = round(loss_ha, 4)
        total_loss_pixels += r.pixel_count
    total_loss_ha = total_loss_pixels * pixel_area_ha
    return {"forest_loss_hectares": round(total_loss_ha, 4), "forest_loss_by_year": loss_by_year}
  except Exception as e:
    print(f"Error analyzing forest loss for geometry: {e}")
  return {"forest_loss_hectares": None, "forest_loss_by_year": {}}


def analyze_forest_gain_geometry(wkt: str, db: Session) -> Dict[str, Any]:
  """Analyze Hansen forest gain (2000-2012) for a specific geometry"""
  try:
    query = text("""
      SELECT SUM((pvc).count) as gain_pixels
      FROM (
        SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
        FROM rasters.nepal_gain WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
```

```
        ) as subquery
        WHERE (pvc).value = 1
    """)
    result = db.execute(query, {"wkt": wkt}).first()
    if result and result.gain_pixels:
        pixel_area_ha = 0.0009
        gain_ha = result.gain_pixels * pixel_area_ha
        return {"forest_gain_hectares": round(gain_ha, 4)}
  except Exception as e:
    print(f"Error analyzing forest gain for geometry: {e}")
  return {"forest_gain_hectares": None}



def analyze_fire_loss_geometry(wkt: str, db: Session) -> Dict[str, Any]:
  """Analyze fire-related forest loss for a specific geometry"""
  try:
    query = text("""
      SELECT (pvc).value as fire_year, SUM((pvc).count) as pixel_count
      FROM (
        SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
        FROM rasters.forest_loss_fire WHERE ST_Intersects(rast, ST_GeomFromText(:wkt,
4326))
      ) as subquery
      WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 23
      GROUP BY fire_year ORDER BY fire_year
    """)
    results = db.execute(query, {"wkt": wkt}).fetchall()
    if not results:
      return {"fire_loss_hectares": 0, "fire_loss_by_year": {}}
    pixel_area_ha = 0.0009
    fire_by_year = {}
```

```
        total_fire_pixels = 0
        for r in results:
            year_code = int(r.fire_year)
            actual_year = 2000 + year_code
            fire_ha = r.pixel_count * pixel_area_ha
            fire_by_year[str(actual_year)] = round(fire_ha, 4)
            total_fire_pixels += r.pixel_count
        total_fire_ha = total_fire_pixels * pixel_area_ha
        return {"fire_loss_hectares": round(total_fire_ha, 4), "fire_loss_by_year": fire_by_year}
    except Exception as e:
        print(f"Error analyzing fire loss for geometry: {e}")
    return {"fire_loss_hectares": None, "fire_loss_by_year": {}}



def analyze_temperature_geometry(wkt: str, db: Session) -> Dict[str, Any]:
    """Analyze temperature data for a specific geometry"""
    try:
        query = text("""
            WITH clipped AS (
                SELECT ST_Clip(rast, ST_GeomFromText(:wkt, 4326)) as rast
                FROM rasters.annual_mean_temperature WHERE ST_Intersects(rast,
ST_GeomFromText(:wkt, 4326))
                LIMIT 1
            ), pixel_values AS (
                SELECT (ST_PixelAsPolygons(rast)).val as temp FROM clipped
            )
            SELECT AVG(temp) as temp_mean FROM pixel_values WHERE temp IS NOT NULL AND
temp > -100 AND temp < 100
        """)
        result = db.execute(query, {"wkt": wkt}).first()
        temp_mean = None
        if result and result.temp_mean is not None:
```

```
        temp_mean = round(result.temp_mean, 2)


    query_min = text("""
        WITH clipped AS (
            SELECT ST_Clip(rast, ST_GeomFromText(:wkt, 4326)) as rast
            FROM rasters.min_temp_coldest_month WHERE ST_Intersects(rast,
ST_GeomFromText(:wkt, 4326))
            LIMIT 1
        ), pixel_values AS (
            SELECT (ST_PixelAsPolygons(rast)).val as temp FROM clipped
        )
        SELECT AVG(temp) as temp_min FROM pixel_values WHERE temp IS NOT NULL AND
temp > -100 AND temp < 100
    """)
    result_min = db.execute(query_min, {"wkt": wkt}).first()
    temp_min = None
    if result_min and result_min.temp_min is not None:
        temp_min = round(result_min.temp_min, 2)


    return {"temperature_mean_c": temp_mean, "temperature_min_c": temp_min}
  except Exception as e:
    print(f"Error analyzing temperature for geometry: {e}")
  return {"temperature_mean_c": None, "temperature_min_c": None}



def analyze_precipitation_geometry(wkt: str, db: Session) -> Dict[str, Any]:
  """Analyze precipitation data for a specific geometry"""
  try:
    query = text("""
      WITH clipped AS (
        SELECT ST_Clip(rast, ST_GeomFromText(:wkt, 4326)) as rast
```

```
            FROM rasters.annual_precipitation WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
            LIMIT 1
        ), pixel_values AS (
            SELECT (ST_PixelAsPolygons(rast)).val as precip FROM clipped
        )
        SELECT AVG(precip) as precip_mean FROM pixel_values WHERE precip IS NOT NULL AND precip >= 0
    """)
    result = db.execute(query, {"wkt": wkt}).first()
    if result and result.precip_mean is not None:
      return {"precipitation_mean_mm": round(result.precip_mean, 1)}
  except Exception as e:
    print(f"Error analyzing precipitation for geometry: {e}")
  return {"precipitation_mean_mm": None}



def analyze_soil_geometry(wkt: str, db: Session) -> Dict[str, Any]:
  """Analyze soil texture for a specific geometry"""
  try:
    texture_map = {
      1: "Clay", 2: "Silty Clay", 3: "Sandy Clay", 4: "Clay Loam",
      5: "Silty Clay Loam", 6: "Sandy Clay Loam", 7: "Loam",
      8: "Silty Loam", 9: "Sandy Loam", 10: "Silt", 11: "Loamy Sand", 12: "Sand"
    }
    query = text("""
      SELECT (pvc).value as texture_code, SUM((pvc).count) as pixel_count
      FROM (
        SELECT ST_ValueCount(ST_Clip(rast, ST_GeomFromText(:wkt, 4326))) as pvc
        FROM rasters.soilgrids_isric WHERE ST_Intersects(rast, ST_GeomFromText(:wkt, 4326))
      ) as subquery
      WHERE (pvc).value IS NOT NULL AND (pvc).value BETWEEN 1 AND 12
```

```
        GROUP BY texture_code ORDER BY pixel_count DESC LIMIT 1
      """)
      result = db.execute(query, {"wkt": wkt}).first()
      if result:
        code = int(result.texture_code)
        if code in texture_map:
          return {"soil_texture": texture_map[code]}
    except Exception as e:
      print(f"Error analyzing soil for geometry: {e}")
    return {"soil_texture": None}
////////////////////////////////////////////////////////////////////////////////////////////
"""
Forest management API endpoints
"""
from fastapi import APIRouter, Depends, HTTPException, status, UploadFile, File, Form
from sqlalchemy.orm import Session
from sqlalchemy import func, text
from sqlalchemy.orm.attributes import flag_modified
from typing import Optional, List
from uuid import UUID
import json

from ..core.database import get_db
from ..models.user import User
from ..models.community_forest import CommunityForest
from ..models.forest_manager import ForestManager
from ..models.calculation import Calculation, CalculationStatus
from ..schemas.forest import (
    CommunityForestResponse,
    ForestManagerCreate,
    ForestManagerResponse,
```

```
    CalculationResponse,
    MyForestsResponse,
)
from ..utils.auth import get_current_active_user, require_super_admin
from ..services.file_processor import process_uploaded_file
from ..services.analysis import analyze_forest_boundary
from shapely.geometry import mapping


router = APIRouter()


@router.get("/community-forests", response_model=List[CommunityForestResponse])
async def list_community_forests(
    search: Optional[str] = None,
    regime: Optional[str] = None,
    limit: int = 100,
    offset: int = 0,
    db: Session = Depends(get_db)
):
    """
    List community forests from the database

    - **search**: Search by name (case-insensitive)
    - **regime**: Filter by regime type (CF, CFM, etc.)
    - **limit**: Number of results to return (max 1000)
    - **offset**: Number of results to skip for pagination
    """
    query = db.query(CommunityForest)

    # Apply filters
```

```
    if search:
        query = query.filter(CommunityForest.name.ilike(f"%{search}%"))
    if regime:
        query = query.filter(CommunityForest.regime == regime)

    # Apply pagination
    query = query.limit(min(limit, 1000)).offset(offset)

    forests = query.all()

    # Convert to response format
    results = []
    for forest in forests:
        results.append({
            "id": forest.id,
            "name": forest.name,
            "code": forest.code,
            "regime": forest.regime,
            "area_hectares": forest.area_hectares,
            "geometry": None  # Don't include full geometry in list view
        })

    return results


@router.get("/community-forests/{forest_id}", response_model=CommunityForestResponse)
async def get_community_forest(forest_id: int, db: Session = Depends(get_db)):
    """
    Get detailed information about a specific community forest

    Returns forest metadata and boundary geometry as GeoJSON
```

```
    """
    forest = db.query(CommunityForest).filter(CommunityForest.id == forest_id).first()

    if not forest:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Community forest with ID {forest_id} not found"
        )

    # Get geometry as GeoJSON
    geojson_query = db.query(
        func.ST_AsGeoJSON(CommunityForest.geom).label("geojson")
    ).filter(CommunityForest.id == forest_id).first()

    geometry = json.loads(geojson_query.geojson) if geojson_query else None

    return {
        "id": forest.id,
        "name": forest.name,
        "code": forest.code,
        "regime": forest.regime,
        "area_hectares": forest.area_hectares,
        "geometry": geometry
    }


@router.get("/my-forests", response_model=MyForestsResponse)
async def get_my_forests(
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
```

```
"""
Get forests assigned to current user

Returns list of community forests the user manages
"""
# Query forests assigned to user
query = db.query(
    CommunityForest,
    ForestManager.role
).join(
    ForestManager,
    ForestManager.community_forest_id == CommunityForest.id
).filter(
    ForestManager.user_id == current_user.id,
    ForestManager.is_active == True
)

results = query.all()

forests = []
total_area = 0.0

for forest, role in results:
    forests.append({
        "id": forest.id,
        "name": forest.name,
        "code": forest.code,
        "regime": forest.regime,
        "area_hectares": forest.area_hectares,
        "role": role
    })
```

```
        total_area += forest.area_hectares

    return {
        "forests": forests,
        "total_count": len(forests),
        "total_area_hectares": total_area
    }


@router.post("/assign-manager", response_model=ForestManagerResponse)
async def assign_forest_manager(
    assignment: ForestManagerCreate,
    current_user: User = Depends(require_super_admin),
    db: Session = Depends(get_db)
):
    """
    Assign a user to manage a community forest

    Requires super admin privileges
    """
    # Verify user exists
    user = db.query(User).filter(User.id == assignment.user_id).first()
    if not user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User not found"
        )

    # Verify forest exists
    forest = db.query(CommunityForest).filter(
        CommunityForest.id == assignment.community_forest_id
```

```
).first()
if not forest:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Community forest not found"
    )

# Check if assignment already exists
existing = db.query(ForestManager).filter(
    ForestManager.user_id == assignment.user_id,
    ForestManager.community_forest_id == assignment.community_forest_id
).first()

if existing:
    # Update existing assignment
    existing.role = assignment.role
    existing.is_active = True
    db.commit()
    db.refresh(existing)
    return existing

# Create new assignment
new_assignment = ForestManager(
    user_id=assignment.user_id,
    community_forest_id=assignment.community_forest_id,
    role=assignment.role
)

db.add(new_assignment)
db.commit()
db.refresh(new_assignment)
```

```
    return new_assignment


@router.post("/upload", response_model=CalculationResponse,
status_code=status.HTTP_201_CREATED)
async def upload_forest_boundary(
    file: UploadFile = File(...),
    forest_name: str = Form(...),
    block_name: Optional[str] = Form(None),
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    Upload forest boundary file for analysis

    Supported formats: Shapefile (.shp/.zip), KML, GeoJSON

    The file will be processed to extract geometry and prepare for analysis

    - **forest_name**: Required - Name of the forest (mandatory)
    - **block_name**: Optional - Name of the block
    """
    # Process uploaded file
    try:
        wkt, metadata = await process_uploaded_file(file)
    except ValueError as e:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=str(e)
        )
```

```
except Exception as e:
    raise HTTPException(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        detail=f"Error processing file: {str(e)}"
    )

# Prepare blocks data for JSONB storage
blocks_data = []
if 'blocks' in metadata:
    for block in metadata['blocks']:
        blocks_data.append({
            'block_index': block['index'],
            'block_name': block['name'],
            'area_sqm': block['area_sqm'],
            'area_hectares': block['area_hectares'],
            'geometry': mapping(block['geometry']),  # Convert to GeoJSON
            'centroid': {
                'lon': block['centroid'].x,
                'lat': block['centroid'].y
            }
        })

# Prepare result_data with blocks information
result_data = {
    'total_blocks': metadata.get('block_count', 1),
    'blocks': blocks_data,
    'processing_info': {
        'partitioned': metadata.get('partitioned', False),
        'partition_info': metadata.get('partition_info', {})
    }
}
```

```
# Create calculation record with WKT geometry
calculation = Calculation(
    user_id=current_user.id,
    uploaded_filename=file.filename,
    boundary_geom=func.ST_GeomFromText(wkt, 4326),
    forest_name=forest_name,  # Now mandatory from form
    block_name=block_name or (blocks_data[0]['block_name'] if blocks_data else "Block 1"),
    status=CalculationStatus.PROCESSING,
    result_data=result_data
)

db.add(calculation)
db.commit()
db.refresh(calculation)

# Get the calculation ID before analysis
calc_id = calculation.id

# Start analysis in background
try:
    # Run raster analysis on the uploaded boundary
    print(f"Starting analysis for calculation {calc_id}")

    # Get a fresh calculation reference with eager loading to avoid detached instance issues
    db.expire_all()  # Expire cached objects

    analysis_results, processing_time = await analyze_forest_boundary(calc_id, db)
    print(f"Analysis completed with {len(analysis_results)} keys")

    # Merge analysis results with existing block data using SQL JSONB operators
```

```
    # Use CAST syntax instead of :: to avoid parameter binding conflict
    update_query = text("""
        UPDATE public.calculations
        SET
            result_data = result_data || CAST(:analysis_data AS jsonb),
            processing_time_seconds = :processing_time,
            status = :status,
            completed_at = NOW()
        WHERE id = :calc_id
    """)

    print(f"Executing UPDATE with {len(json.dumps(analysis_results))} bytes of data")
    result = db.execute(update_query, {
        "analysis_data": json.dumps(analysis_results),
        "processing_time": processing_time,
        "status": "COMPLETED",
        "calc_id": str(calc_id)  # Use calc_id instead of calculation.id
    })
    print(f"UPDATE affected {result.rowcount} rows")

    db.commit()
    print("Commit successful")

    # Refresh calculation object after commit
    calculation = db.query(Calculation).filter(Calculation.id == calc_id).first()
    if calculation and calculation.result_data:
        print(f"Refreshed calculation, result_data has {len(calculation.result_data)} keys")
    else:
        print(f"Warning: Could not refresh calculation or result_data is empty")

except Exception as e:
```

```
    db.rollback()  # Rollback failed transaction first
    print(f"Analysis failed: {str(e)}")

    # Update status in a new transaction
    try:
        calculation.status = CalculationStatus.FAILED
        calculation.error_message = str(e)[:500]  # Limit error message length
        db.commit()
    except Exception as commit_error:
        print(f"Failed to update error status: {commit_error}")
        db.rollback()

    raise HTTPException(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        detail=f"Analysis failed: {str(e)}"
    )

# Re-query calculation to ensure we have fresh data
calculation = db.query(Calculation).filter(Calculation.id == calc_id).first()
if not calculation:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Calculation not found after processing"
    )

# Get geometry as GeoJSON
geojson_query = db.query(
    func.ST_AsGeoJSON(Calculation.boundary_geom).label("geojson")
).filter(Calculation.id == calc_id).first()

geometry_json = json.loads(geojson_query.geojson) if geojson_query else None
```

```
    return CalculationResponse(
        id=calculation.id,
        user_id=calculation.user_id,
        uploaded_filename=calculation.uploaded_filename,
        forest_name=calculation.forest_name,
        block_name=calculation.block_name,
        status=calculation.status,
        processing_time_seconds=calculation.processing_time_seconds,
        error_message=calculation.error_message,
        created_at=calculation.created_at,
        completed_at=calculation.completed_at,
        geometry=geometry_json,
        result_data=calculation.result_data
    )


@router.get("/calculations/{calculation_id}", response_model=CalculationResponse)
async def get_calculation(
    calculation_id: UUID,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    Get calculation results by ID

    Users can only access their own calculations unless they are super admin
    """
    calculation = db.query(Calculation).filter(Calculation.id == calculation_id).first()

    if not calculation:
```

```
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Calculation not found"
    )

# Check permissions
from ..models.user import UserRole
if calculation.user_id != current_user.id and current_user.role != UserRole.SUPER_ADMIN:
    raise HTTPException(
        status_code=status.HTTP_403_FORBIDDEN,
        detail="You don't have permission to access this calculation"
    )

# Get geometry as GeoJSON
geojson_query = db.query(
    func.ST_AsGeoJSON(Calculation.boundary_geom).label("geojson")
).filter(Calculation.id == calculation_id).first()

geometry_json = json.loads(geojson_query.geojson) if geojson_query else None

return CalculationResponse(
    id=calculation.id,
    user_id=calculation.user_id,
    uploaded_filename=calculation.uploaded_filename,
    forest_name=calculation.forest_name,
    block_name=calculation.block_name,
    status=calculation.status,
    processing_time_seconds=calculation.processing_time_seconds,
    error_message=calculation.error_message,
    created_at=calculation.created_at,
    completed_at=calculation.completed_at,
```

```
        geometry=geometry_json,
        result_data=calculation.result_data
    )


@router.get("/calculations", response_model=List[CalculationResponse])
async def list_calculations(
    limit: int = 50,
    offset: int = 0,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    List user's calculations

    Returns all calculations for the current user
    """
    query = db.query(Calculation).filter(Calculation.user_id == current_user.id)
    query = query.order_by(Calculation.created_at.desc())
    query = query.limit(limit).offset(offset)

    calculations = query.all()

    results = []
    for calc in calculations:
        results.append(CalculationResponse(
            id=calc.id,
            user_id=calc.user_id,
            uploaded_filename=calc.uploaded_filename,
            forest_name=calc.forest_name,
            block_name=calc.block_name,
```

```
            status=calc.status,
            processing_time_seconds=calc.processing_time_seconds,
            error_message=calc.error_message,
            created_at=calc.created_at,
            completed_at=calc.completed_at,
            geometry=None,  # Don't include geometry in list view
            result_data=None  # Don't include full results in list view
        ))

    return results


@router.delete("/calculations/{calculation_id}", status_code=status.HTTP_204_NO_CONTENT)
async def delete_calculation(
    calculation_id: UUID,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    Delete a calculation

    Users can only delete their own calculations
    """
    calculation = db.query(Calculation).filter(Calculation.id == calculation_id).first()

    if not calculation:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Calculation not found"
        )
```

```
    # Check permissions
    if calculation.user_id != current_user.id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="You don't have permission to delete this calculation"
        )

    db.delete(calculation)
    db.commit()

    return None

@router.patch("/calculations/{calculation_id}/result-data")
async def update_result_data(
    calculation_id: UUID,
    update_data: dict,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    Update result_data fields for a calculation (field verification edits)

    Users can only update their own calculations.
    Accepts a JSON body with fields to update - merges into existing result_data.
    """
    calculation = db.query(Calculation).filter(Calculation.id == calculation_id).first()

    if not calculation:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Calculation not found"
```

```
        )

    # Check permissions
    from ..models.user import UserRole
    if calculation.user_id != current_user.id and current_user.role != UserRole.SUPER_ADMIN:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="You don't have permission to update this calculation"
        )

    # Merge update_data into existing result_data
    existing = calculation.result_data or {}
    existing.update(update_data)
    calculation.result_data = existing
    flag_modified(calculation, "result_data")

    db.commit()
    db.refresh(calculation)

    return {"status": "updated", "result_data": calculation.result_data}
////////////////////////////////////////////////////////////////////////////////
@echo off
REM Start Community Forest Management System on port 8001

echo ========================================
echo Community Forest Management System
echo ========================================
echo.

REM Activate virtual environment
echo Activating virtual environment...
```

```
call venv\Scripts\activate
echo.

REM Disable Python output buffering
set PYTHONUNBUFFERED=1

REM Start the server on port 8001
echo ========================================
echo Starting FastAPI server on port 8001...
echo API Documentation: http://localhost:8001/docs
echo Health Check: http://localhost:8001/health
echo ========================================
echo.

cd backend
..\venv\Scripts\python -u -m uvicorn app.main:app --reload --host 0.0.0.0 --port 8001
cd ..
////////////////////////////////////////////////////////////////////////////////////////
```