National TeleCritical Care Assignment Tool (NTAAT)

Python-in-Excel Implementation

Workbook template: GPT PY NTAAT 12.29v4.xlsx
Prepared: 2025-12-29

Audience: This document is written for a developer/analyst who will productionize the logic in the VA environment (for example, via SQL-fed census, Power BI reporting, and a Power Apps interface). It also functions as a user manual for charge staff running shift assignments.

Problem statement: NTAAT assigns 85 ICU sites of care to on-shift RNs and MD teams, multiple times per day, with a strong requirement for continuity across reruns and reasonable workload balance.

Key design priorities (in order):
- Continuity: keep sites with the same RN/MD when possible (baseline is the 'Previous' table pasted at shift start).
- Balance: distribute total patient load across staff as evenly as feasible using a deterministic greedy algorithm.
- Special rules: enforce required clusters and conflict pairs; include zero-census sites.

1. Workbook Architecture

The Excel workbook is a template that contains: (a) a census table, (b) RN and MD input tables, and (c) Python-in-Excel scripts that compute detailed assignments and roll-up summaries. The workbook is designed to be the 'logic reference' for a future VA implementation.

Worksheet	Purpose	Key Tables / Named Cells
Census_Data	Daily census input (updated 2x/day; future SQL feed).	Table: CensusTable (A1:D86)
RN_Inputs	RN staffing inputs and continuity baseline.	Names: RN_Count (B1), RN_ContinuityWeight (B2) Table: RN_Previous (A3:B87)
MD_Inputs	MD staffing inputs and continuity baseline.	Name: MD_Count (B1) Table: MD_Previous (A3:B88)
RN_Assignments	Python-in-Excel =PY detailed RN assignments output.	Spill output: StaffID, SiteCode, FacilityName, PatientCount
MD_Assignments	Python-in-Excel =PY detailed MD assignments output.	Spill output: StaffID, SiteCode, FacilityName, PatientCount, Category
Summary	Summary tables for RN and MD teams (patients, sites, category counts).	Python-in-Excel summary scripts; totals row
Notes	Optional documentation and future enhancements.	N/A

2. Data Model and Inputs

2.1 CensusTable schema

CensusTable is the authoritative daily input. It is updated by copy/paste today and is intended to be SQL-fed in the VA production version.

Columns (expected order):
- SiteCode (string): short code (for example, ABQ, HOU, SPO).
- FacilityName (string): human-readable site name.
- TotalCensus (integer): patient count. Zero-census sites are valid and must still be assigned.
- Category (integer): site classification (currently 1-4). Used only for MD category-1 distribution.

2.2 Staffing inputs and continuity baselines

RN and MD staffing counts are provided via named cells, and continuity is provided by staff pasting the prior shift's detailed assignments (two columns: StaffID and SiteCode) into the corresponding 'Previous' table.

RN inputs: RN_Count, RN_ContinuityWeight, RN_Previous table.
MD inputs: MD_Count, MD_Previous table.

3. Operational Workflow (User Guide)

This section is written for charge staff running assignments each shift. RN and MD workflows are intentionally independent; each can be rerun at different times and frequencies.

3.1 Start of shift - establish continuity baseline

At the start of a shift (or whenever leadership decides to reset continuity):
1) Run the RN and/or MD assignment scripts once using current census and staffing counts.
2) Copy two columns from the detailed output (StaffID and SiteCode).
3) Paste **values** into RN_Previous (for nurses) and/or MD_Previous (for physicians).
4) Subsequent reruns during the shift will prioritize keeping those sites with the same staff.

3.2 Mid-shift reruns

During a shift, staffing changes may occur 3-5 times per day. When RN_Count or MD_Count changes, re-run the corresponding script. Do not repaste the Previous tables unless you intend to reset continuity.

3.3 Expected behavior

- Continuity is the dominant criterion for both roles.
- The algorithm still aims to balance patient load while respecting special rules.
- Zero-census sites remain assigned (to preserve coverage and simplify downstream workflows).

4. RN Assignment Algorithm

RN assignment uses a deterministic greedy heuristic. Sites are processed from highest to lowest census. Each site is assigned to the RN that minimizes a score that combines projected load and continuity penalty, while enforcing conflict constraints.

4.1 Objective and scoring

Score components:
- Conflict check: if adding the site violates a conflict pair, the score is effectively infinite (disallowed).
- Continuity penalty: if the RN did not previously hold the site, add RN_ContinuityWeight to the projected load.
- Tie-breakers: fewer assigned sites, then lower current load, then lexical StaffID for determinism.

4.2 Special rules (RN)

- Cluster rule: SPO + DOD + HOU must remain together; if HOU is absent, SPO + DOD remain together.
- Conflict pairs: FAV vs FNC and CMO vs CMS cannot be assigned to the same RN.

4.3 Output schema (RN)

The RN detailed output includes:
- StaffID (Nurse 1 .. Nurse N)
- SiteCode
- FacilityName
- PatientCount

5. MD Assignment Algorithm

MD assignment is structurally similar to RN assignment, but adds a category distribution tie-breaker. Continuity is enforced as the primary criterion: continuity always wins, even if it slightly worsens category-1 distribution.

5.1 Objective and scoring

Score tuple (lexicographic): (continuity_penalty, category1_penalty, projected_patient_load, team_id). Because the first element is continuity_penalty, any incumbent assignment will outrank non-incumbent alternatives unless there is a conflict constraint.

5.2 Category-1 distribution (MD only)

When assigning a category-1 site, the algorithm applies a small penalty to teams that already have more category-1 sites than the minimum across teams. This is a tie-breaker only. To increase or decrease the influence, adjust CAT1_STEP_PENALTY. Do not increase CAT1_STEP_PENALTY above the continuity penalty unless leadership explicitly decides to allow category balancing to override continuity.

5.3 Special rules (MD)

MD special rules match RN rules: clustering and conflict pairs are enforced identically.

5.4 Output schema (MD)

The MD detailed output includes:
- StaffID (MD Team 1 .. MD Team M)
- SiteCode
- FacilityName
- PatientCount
- Category

6. Validation and Troubleshooting

6.1 Built-in validation

Both RN and MD scripts validate that:
- Every site in CensusTable appears exactly once in the detailed output (after expanding clusters).
- No unexpected sites appear.
- The sum of PatientCount equals the sum of TotalCensus in CensusTable.
If validation fails, the script raises an explicit error to prevent silent incorrect assignments.

6.2 Common failure modes observed during development

- **ABQ missing**: Python-in-Excel sometimes interpreted the first data row as headers when using headers=True. The fix was to load CensusTable using headers=False and explicitly name columns.
- **#CONNECT!**: usually indicates a transient Python runtime connectivity issue in Excel. Resetting the Python runtime typically resolves it.
- **Names not resolved**: KeyError on RN_Count/MD_Count indicates workbook-scoped names are missing or sheet-scoped. Fix in Name Manager to ensure WORKBOOK scope and a single-cell reference.
- **Circular references**: occur if a Python formula reads from a range that depends on its own spill output. Keep input tables and outputs on separate sheets and avoid reading the output spill as an input.

6.3 Deterministic ordering in outputs

Excel and downstream users often expect Nurse 1, Nurse 2, ... Nurse 10 (numeric order). Default lexical sorts place Nurse 10 before Nurse 2. Use a natural sort key based on the trailing integer to display outputs in true numeric order. See Appendix C for the exact snippet.

7. VA Production Integration Roadmap

The current workbook is a functional template. The recommended VA production implementation is to separate data acquisition, assignment logic, and user experience.

7.1 Census data automation (SQL Server)

Recommended approach:
- Create a SQL view that produces the CensusTable schema: SiteCode, FacilityName, TotalCensus, Category.
- Refresh cadence: at least 2x/day (current operations), with ad hoc refresh allowed.
- In Excel prototype: Power Query can populate CensusTable from SQL; in production: consider a service endpoint that Power Apps calls directly.

7.2 UI modernization (Power Apps)

Power Apps can reduce user errors by providing:
- Shift context (RN vs MD, staff counts, timestamped runs).
- A single 'Run assignments' action that writes outputs to a durable store (SQL table).
- A controlled 'Set continuity baseline' action that snapshots assignments into a baseline table.

7.3 Analytics and audit (Power BI)

Power BI on top of a SQL-backed assignment history can provide:
- Workload distribution over time (patients per RN/MD).
- Continuity metrics (percentage of sites retained across reruns).
- Category-1 distribution trend (MD only).
- Exceptions reporting (reassignments due to conflicts or staffing changes).

7.4 Recommended production architecture

A pragmatic pattern in VA is:
- SQL Server as the system of record for census, baseline, and assignment history.
- A Python service that runs the assignment algorithm and persists results.
- Power Apps for operational entry points, Power BI for analytics.
- Excel remains the reference implementation and contingency fallback.

8. Future Enhancements (Suggestions)

The current greedy approach is intentionally simple, deterministic, and operationally reliable. Future improvements can be staged without changing the workflow.

8.1 Algorithmic enhancements

- Replace greedy assignment with an optimization approach (integer programming) to minimize maximum load while preserving continuity constraints.
- Add explicit limits (for example, max sites per RN, max patient cap per RN/MD) and apply as hard constraints.
- Add a controlled rule engine for additional site coupling or separation as new requirements emerge.

8.2 Data quality and governance

- Validate SiteCode master list and map to facility identifiers used in VA systems.
- Implement automated checks for missing/duplicated sites or unexpected categories before computing assignments.
- Version control the assignment logic and maintain release notes for changes to business rules.

8.3 Operational resiliency

- Add an explicit shift baseline table and UI action (Power Apps) to capture continuity state without manual paste.
- Store each run with timestamp, user, and reason code (routine rerun, staffing change, census update).
- Provide a single printable report view for handoffs.

Appendix A. RN Assignment Script (current reference)

Reference script used in Python-in-Excel for RN detailed assignments.

```
import pandas as pd

# ----------------------------
# Config (RN)
# ----------------------------
PREFIX = "Nurse"

# Load RN count
try:
    rn_count = xl("RN_Count")
    rn_count = int(rn_count.iloc[0, 0]) if hasattr(rn_count, "iloc") else int(rn_count)
except Exception:
    rn_count = 18

# Continuity weight (penalty if moving a site off incumbent)
try:
    w = xl("RN_ContinuityWeight")
    continuity_weight = float(w.iloc[0, 0]) if hasattr(w, "iloc") else float(w)
except Exception:
    continuity_weight = 5.0

# ----------------------------
# Load and clean Census (ABQ-safe)
# ----------------------------
# Load without headers to avoid the "first data row becomes headers" issue.
census_raw = xl("CensusTable", headers=False)

# Expect columns: SiteCode, FacilityName, TotalCensus, Category (Category not used for RN logic)
census_raw = census_raw.iloc[:, :4].copy()
census_raw.columns = ["SiteCode", "FacilityName", "TotalCensus", "Category"]

census_raw["SiteCode"] = census_raw["SiteCode"].astype(str).str.strip()
census_raw["FacilityName"] = census_raw["FacilityName"].astype(str).str.strip()

# Keep zero-census sites; coerce blanks/text to 0
census_raw["TotalCensus"] = pd.to_numeric(census_raw["TotalCensus"], errors="coerce").fillna(0).astype(int)

# Drop rows with blank SiteCode
census = census_raw[census_raw["SiteCode"].ne("")].copy()

expected_sites = set(census["SiteCode"])
expected_total = int(census["TotalCensus"].sum())

site_census = dict(zip(census["SiteCode"], census["TotalCensus"]))
site_name = dict(zip(census["SiteCode"], census["FacilityName"]))

# ----------------------------
# Load previous assignments (RN_Previous)
# ----------------------------
prev_map = {}  # StaffID -> set(SiteCode)
try:
    prev = xl("RN_Previous", headers=True)
except Exception:
    prev = None

if prev is not None and isinstance(prev, pd.DataFrame) and not prev.empty:
    p = prev.copy()
    p.columns = [str(c).strip() for c in p.columns]
```

```
    # Normalize to StaffID, SiteCode
    if "StaffID" in p.columns and "SiteCode" in p.columns:
        p = p[["StaffID", "SiteCode"]]
    else:
        p = p.iloc[:, :2].copy()
        p.columns = ["StaffID", "SiteCode"]

    p["StaffID"] = p["StaffID"].astype(str).str.strip()
    p["SiteCode"] = p["SiteCode"].astype(str).str.strip()
    p = p[(p["StaffID"].ne("")) & (p["SiteCode"].ne(""))]
    prev_map = p.groupby("StaffID")["SiteCode"].apply(lambda x: set(x)).to_dict()

# ----------------------------
# Special rules
# ----------------------------
# Cluster: SPO + DOD + HOU together (fallback: SPO + DOD)
CLUSTER3_KEY = "SPO_DOD_HOU_CLUSTER"
CLUSTER2_KEY = "SPO_DOD_CLUSTER"
cluster_key = None
cluster_members = []

working = site_census.copy()

if ("SPO" in working) and ("DOD" in working) and ("HOU" in working):
    cluster_key = CLUSTER3_KEY
    cluster_members = ["SPO", "DOD", "HOU"]
    working[cluster_key] = working.pop("SPO") + working.pop("DOD") + working.pop("HOU")
elif ("SPO" in working) and ("DOD" in working):
    cluster_key = CLUSTER2_KEY
    cluster_members = ["SPO", "DOD"]
    working[cluster_key] = working.pop("SPO") + working.pop("DOD")

# Conflict pairs: cannot be assigned to same RN
conflict_sets = [
    {"FAV", "FNC"},
    {"CMO", "CMS"},
]

def violates_conflict(existing_sites, new_site):
    ss = set(existing_sites)
    for s in conflict_sets:
        if new_site in s and len(s.intersection(ss)) > 0:
            return True
    return False

def is_incumbent(staff_id, site):
    incumbents = prev_map.get(staff_id, set())
    if (cluster_key is not None) and (site == cluster_key):
        # incumbent if they previously had ANY member of the cluster
        return any(m in incumbents for m in cluster_members)
    return site in incumbents

# ----------------------------
# Assignment engine (greedy + deterministic tie-breaks)
# ----------------------------
staff_ids = [f"{PREFIX} {i+1}" for i in range(rn_count)]
staff_state = {s: {"load": 0, "sites": []} for s in staff_ids}

sorted_sites = sorted(working.items(), key=lambda x: x[1], reverse=True)

for site, load in sorted_sites:
    def score(staff_id):
        cur_load = staff_state[staff_id]["load"]
        cur_sites = staff_state[staff_id]["sites"]
```

```
        if violates_conflict(cur_sites, site):
            return (10**9, 10**9, 10**9, staff_id)

        penalty = 0.0 if is_incumbent(staff_id, site) else continuity_weight
        projected = cur_load + load + penalty

        # Tie-breakers: fewer sites, lower current load, lexical StaffID
        return (projected, len(cur_sites), cur_load, staff_id)

    best = min(staff_ids, key=score)
    staff_state[best]["load"] += int(load)
    staff_state[best]["sites"].append(site)

# ----------------------------
# Expand cluster and produce output
# ----------------------------
rows = []
for staff_id, d in staff_state.items():
    for assigned_site in d["sites"]:
        if (cluster_key is not None) and (assigned_site == cluster_key):
            for sub in cluster_members:
                rows.append({
                    "StaffID": staff_id,
                    "SiteCode": sub,
                    "FacilityName": site_name.get(sub, ""),
                    "PatientCount": int(site_census.get(sub, 0))
                })
        else:
            rows.append({
                "StaffID": staff_id,
                "SiteCode": assigned_site,
                "FacilityName": site_name.get(assigned_site, ""),
                "PatientCount": int(site_census.get(assigned_site, 0))
            })

out = pd.DataFrame(rows)

# ----------------------------
# Validation (fail loud)
# ----------------------------
assigned_sites = set(out["SiteCode"])
missing = expected_sites - assigned_sites
extra = assigned_sites - expected_sites
assigned_total = int(out["PatientCount"].sum())

if missing:
    raise ValueError(f"RN output missing sites: {sorted(list(missing))}")
if extra:
    raise ValueError(f"RN output has unexpected sites: {sorted(list(extra))}")
if assigned_total != expected_total:
    raise ValueError(f"RN total mismatch: assigned={assigned_total}, expected={expected_total}")

out.sort_values(["StaffID", "SiteCode"]).reset_index(drop=True)
```

Appendix B. MD Assignment Script (current reference)

Reference script used in Python-in-Excel for MD detailed assignments.

```
import pandas as pd

# ============================================================
# MD DETAILED ASSIGNMENTS
# Priorities:
#   1) Continuity ALWAYS wins (incumbent sites are strongly preferred)
#   2) Category-1 distribution is a tie-breaker only (does not override continuity)
#   3) Balance total patient load
# Special rules:
#   - Cluster SPO + DOD + HOU together (fallback: SPO + DOD)
#   - Conflict pairs cannot be assigned to the same MD: CMO/CMS and FAV/FNC
# ============================================================

# --- Inputs ---
md_count_raw = xl("MD_Count")
md_count_raw = md_count_raw.iloc[0,0] if hasattr(md_count_raw,"iloc") else md_count_raw
md_count = int(pd.to_numeric(md_count_raw, errors="raise"))

# Tuning parameters
CONTINUITY_PENALTY = 50          # Larger = more "sticky" continuity
CAT1_STEP_PENALTY = 10           # Tie-breaker only (keep small vs continuity)

teams = [f"MD Team {i+1}" for i in range(md_count)]

# --- Census (robust, ABQ-safe) ---
c = xl("CensusTable", headers=False)
c = c.iloc[:, :4].copy()
c.columns = ["SiteCode","FacilityName","TotalCensus","Category"]

c["SiteCode"] = c["SiteCode"].astype(str).str.strip()
c["FacilityName"] = c["FacilityName"].astype(str).str.strip()
c["TotalCensus"] = pd.to_numeric(c["TotalCensus"], errors="coerce").fillna(0).astype(int)
c["Category"] = pd.to_numeric(c["Category"], errors="coerce").fillna(0).astype(int)
c = c[c["SiteCode"].ne("")].copy()

site_census = dict(zip(c["SiteCode"], c["TotalCensus"]))
site_name   = dict(zip(c["SiteCode"], c["FacilityName"]))
site_cat    = dict(zip(c["SiteCode"], c["Category"]))

# --- Previous assignments for continuity (OK if empty) ---
prev_map = {}
try:
    prev = xl("MD_Previous", headers=True)
except Exception:
    prev = None

if isinstance(prev, pd.DataFrame) and not prev.empty:
    p = prev.iloc[:, :2].copy()
    p.columns = ["StaffID","SiteCode"]
    p["StaffID"] = p["StaffID"].astype(str).str.strip()
    p["SiteCode"] = p["SiteCode"].astype(str).str.strip()
    p = p[(p["StaffID"] != "") & (p["SiteCode"] != "")]
    prev_map = p.groupby("StaffID")["SiteCode"].apply(set).to_dict()

# --- Special rules (cluster) ---
CLUSTER3_KEY = "SPO_DOD_HOU_CLUSTER"
CLUSTER2_KEY = "SPO_DOD_CLUSTER"
cluster_key = None
cluster_members = []
```

```
working = site_census.copy()

if ("SPO" in working) and ("DOD" in working) and ("HOU" in working):
    cluster_key = CLUSTER3_KEY
    cluster_members = ["SPO", "DOD", "HOU"]
    working[cluster_key] = working.pop("SPO") + working.pop("DOD") + working.pop("HOU")
elif ("SPO" in working) and ("DOD" in working):
    cluster_key = CLUSTER2_KEY
    cluster_members = ["SPO", "DOD"]
    working[cluster_key] = working.pop("SPO") + working.pop("DOD")

# Conflict sets
CONFLICT_SETS = [{"CMO","CMS"}, {"FAV","FNC"}]

def violates_conflict(existing_sites, new_site):
    s = set(existing_sites)
    for cs in CONFLICT_SETS:
        if new_site in cs and len(cs.intersection(s)) > 0:
            return True
    return False

def was_incumbent(team, site):
    incumbents = prev_map.get(team, set())
    if (cluster_key is not None) and (site == cluster_key):
        return any(m in incumbents for m in cluster_members)
    return site in incumbents

def is_cat1_site(site):
    if (cluster_key is not None) and (site == cluster_key):
        # Cat1 if any member is Cat1 (tunable)
        return any(site_cat.get(m, 0) == 1 for m in cluster_members)
    return site_cat.get(site, 0) == 1

# --- State ---
state = {t: {"patients": 0, "sites": [], "cat1_count": 0} for t in teams}

for site, load in sorted(working.items(), key=lambda x: x[1], reverse=True):
    site_is_cat1 = is_cat1_site(site)
    min_cat1 = min(state[t]["cat1_count"] for t in teams) if teams else 0

    def score(team):
        if violates_conflict(state[team]["sites"], site):
            return (10**9, 10**9, 10**9, team)

        # 1) Continuity (dominant)
        cont_penalty = 0 if was_incumbent(team, site) else CONTINUITY_PENALTY

        # 2) Category-1 balance (tie-breaker only)
        cat1_penalty = 0
        if site_is_cat1:
            lead = state[team]["cat1_count"] - min_cat1
            if lead > 0:
                cat1_penalty = lead * CAT1_STEP_PENALTY

        # 3) Patient load balance
        projected_patients = state[team]["patients"] + int(load)

        return (cont_penalty, cat1_penalty, projected_patients, team)

    best = min(teams, key=score)
    state[best]["patients"] += int(load)
    state[best]["sites"].append(site)
    if site_is_cat1:
        state[best]["cat1_count"] += 1
```

```
# --- Output: expand cluster back into member rows ---
rows = []
for team, st in state.items():
    for assigned_site in st["sites"]:
        if (cluster_key is not None) and (assigned_site == cluster_key):
            for sub in cluster_members:
                rows.append({
                    "StaffID": team,
                    "SiteCode": sub,
                    "FacilityName": site_name.get(sub, ""),
                    "PatientCount": int(site_census.get(sub, 0)),
                    "Category": int(site_cat.get(sub, 0)),
                })
        else:
            rows.append({
                "StaffID": team,
                "SiteCode": assigned_site,
                "FacilityName": site_name.get(assigned_site, ""),
                "PatientCount": int(site_census.get(assigned_site, 0)),
                "Category": int(site_cat.get(assigned_site, 0)),
            })

out = pd.DataFrame(rows).sort_values(["StaffID","SiteCode"]).reset_index(drop=True)
out
```

Appendix C. Natural numeric ordering for StaffID

Use this snippet to sort outputs and summaries as Nurse 1, Nurse 2, ... Nurse 10 (instead of lexical ordering). Apply to both detailed outputs and summary tables.

```
import re

def staff_sort_key(x: str) -> int:
    m = re.search(r"(\d+)\s*$", str(x))
    return int(m.group(1)) if m else 10**9

df["_staff_sort"] = df["StaffID"].apply(staff_sort_key)
df = df.sort_values(["_staff_sort", "StaffID", "SiteCode"]).drop(columns=["_staff_sort"]).reset_index(drop=
```