Talent Acquisition Analytics & Intelligence Platform (TAAIP)

Representative Source Code Deposit – Light Redaction

Author: Amber Nakeya Mooney
Copyright Claimant: Maroon Moon, LLC

Explanatory Statement – Computer Program Deposit

This document forms part of the copyright registration deposit for the unpublished computer program titled "Talent Acquisition Analytics & Intelligence Platform (TAAIP)." The work is an original software system authored by Amber Nakeya Mooney in 2025 and owned by Maroon Moon, LLC. The program implements a FastAPI-based backend service that supports recruiting analytics, event and funnel management, targeting logic, and data-ingestion pipelines.

The source code excerpts included in this deposit are taken directly from the primary backend module "taaip_service.py." These excerpts are representative of the structure, sequence, organization, and logic of the work. They show how the service initializes the application, defines data models, sets up RESTful endpoints, implements lead-scoring logic, and processes analytics and targeting operations for the platform.

Consistent with 37 C.F.R. § 202.20(c)(2)(vii), a small number of lines have been lightly redacted to remove confidential elements such as environment-variable values, API tokens, and other sensitive operational details. Where such content has been removed, the text "[REDACTED_FOR_DEPOSIT]" appears in place of the sensitive value. These limited redactions do not affect the Office's ability to examine the work's original expression, nor do they alter the overall structure or readability of the program.

The program was developed on personal equipment and personal time and is not a work made for hire by any government agency or employer. All rights in the code excerpts contained herein, and in the complete underlying program, are held by Maroon Moon, LLC as copyright claimant. A complete, unredacted version of the source code is maintained by the claimant in the ordinary course of business.

```
from fastapi import FastAPI, HTTPException, Request
from fastapi.responses import JSONResponse, StreamingResponse
import io
import csv
from pathlib import Path
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field
import uvicorn
import logging
import random
import os
import json
import sqlite3
import secrets
from datetime import datetime
from typing import Optional, Dict, Any
import threading


# --- Configuration & Initialization ---
app = FastAPI(
    title="TAAIP - Talent Acquisition Analytics and Intelligence Platform",
    description="Provides real-time lead scoring, targeting recommendations, and intelligence analytics.",
    version="2.0.0",
)
logging.basicConfig(level=logging.INFO)

# Allow CORS for local development (adjust origins for production)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Data file locations
DATA_DIR = os.path.join(os.path.dirname(__file__), "data")
os.makedirs(DATA_DIR, exist_ok=True)
LEADS_FILE = os.path.join(DATA_DIR, "leads.json")
PILOT_FILE = os.path.join(DATA_DIR, "pilot_state.json")

# Use project-root DB aligned with all migration/populate scripts
DB_FILE = os.path.join(os.path.dirname(__file__), "recruiting.db")


# --- SQLite helpers ---
def get_db_conn():
    conn = sqlite3.connect(DB_FILE, check_same_thread=False)
    conn.row_factory = sqlite3.Row
    return conn


def model_to_dict(m):
    """Compatibility helper for Pydantic v1/v2: prefer model_dump(), fall back to dict()."""
    if hasattr(m, "model_dump"):
        return m.model_dump()
    if hasattr(m, "dict"):
        return m.dict()
    try:
        return dict(m)
    except Exception:
        return {}


def init_db():
    conn = get_db_conn()
    cur = conn.cursor()

    # --- Original tables ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS leads (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            lead_id TEXT,
            age INTEGER,
            education_level TEXT,
            cbsa_code TEXT,
            campaign_source TEXT,
            received_at TEXT,
            predicted_probability REAL,
            score INTEGER,
            recommendation TEXT,
            converted INTEGER DEFAULT 0,
            raw_json TEXT
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS pilot_state (
            id INTEGER PRIMARY KEY,
            started_at TEXT,
            config TEXT,
            status TEXT
        )
        """
    )

    # --- Extended: Events & ROI Tracking ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS events (
            event_id TEXT PRIMARY KEY,
            name TEXT NOT NULL,
            type TEXT,
            location TEXT,
            start_date TEXT,
            end_date TEXT,
            budget REAL,
            team_size INTEGER,
            targeting_principles TEXT,
            status TEXT DEFAULT 'planned',
            created_at TEXT,
            updated_at TEXT
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS event_metrics (
            metric_id INTEGER PRIMARY KEY AUTOINCREMENT,
            event_id TEXT NOT NULL,
            date TEXT,
            leads_generated INTEGER DEFAULT 0,
            leads_qualified INTEGER DEFAULT 0,
            conversion_count INTEGER DEFAULT 0,
            cost_per_lead REAL,
            roi REAL,
            engagement_rate REAL,
            created_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS capture_survey (
            survey_id TEXT PRIMARY KEY,
            event_id TEXT NOT NULL,
            lead_id TEXT,
            timestamp TEXT,
            technician_id TEXT,
            effectiveness_rating INTEGER,
            feedback TEXT,
            data_quality_flag INTEGER DEFAULT 0,
            created_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )

    # --- Extended: Recruiting Funnel ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS funnel_stages (
            stage_id TEXT PRIMARY KEY,
            stage_name TEXT NOT NULL,
            sequence_order INTEGER,
            description TEXT
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS funnel_transitions (
            transition_id INTEGER PRIMARY KEY AUTOINCREMENT,
            lead_id TEXT NOT NULL,
            from_stage TEXT,
            to_stage TEXT,
            transition_date TEXT,
            transition_reason TEXT,
            technician_id TEXT,
            created_at TEXT,
            FOREIGN KEY(to_stage) REFERENCES funnel_stages(stage_id)
        )
        """
    )

    # --- Extended: Project Management ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS projects (
            project_id TEXT PRIMARY KEY,
            name TEXT NOT NULL,
            event_id TEXT,
            start_date TEXT,
            target_date TEXT,
            owner_id TEXT,
            status TEXT DEFAULT 'planning',
            objectives TEXT,
            success_criteria TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS tasks (
            task_id TEXT PRIMARY KEY,
            project_id TEXT NOT NULL,
            title TEXT NOT NULL,
            description TEXT,
            assigned_to TEXT,
            due_date TEXT,
            status TEXT DEFAULT 'open',
            priority TEXT,
            completion_date TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(project_id) REFERENCES projects(project_id)
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS milestones (
            milestone_id TEXT PRIMARY KEY,
            project_id TEXT NOT NULL,
            name TEXT NOT NULL,
            target_date TEXT,
            actual_date TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(project_id) REFERENCES projects(project_id)
        )
        """
    )

    # --- Extended: M-IPOE ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS mipoe (
            mipoe_id TEXT PRIMARY KEY,
            event_id TEXT NOT NULL,
            phase TEXT NOT NULL,
            content TEXT,
            owner_id TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )

    # --- Extended: Targeting Profiles (D3AE/F3A) ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS targeting_profiles (
            profile_id TEXT PRIMARY KEY,
            event_id TEXT NOT NULL,
            target_age_min INTEGER,
            target_age_max INTEGER,
            target_education_level TEXT,
            target_locations TEXT,
            message_themes TEXT,
            contact_frequency INTEGER,
            conversion_target REAL,
            cost_per_lead_target REAL,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )

    # --- Extended: Forecasting ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS forecasts (
            forecast_id TEXT PRIMARY KEY,
            quarter INTEGER,
            year INTEGER,
            projected_leads INTEGER,
            projected_conversions INTEGER,
            projected_roi REAL,
            confidence_level REAL,
            methodology TEXT,
            created_at TEXT,
            updated_at TEXT
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS analytics_snapshots (
            snapshot_id TEXT PRIMARY KEY,
            quarter INTEGER,
            year INTEGER,
            total_events INTEGER,
            total_leads INTEGER,
            conversion_rate REAL,
            avg_cost_per_lead REAL,
            total_roi REAL,
            by_event TEXT,
            created_at TEXT
        )
        """
    )

    # NEW: Marketing Activity Tracking (USAREC-specific)
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS marketing_activities (
            activity_id TEXT PRIMARY KEY,
            event_id TEXT,
            activity_type TEXT,
            campaign_name TEXT,
            channel TEXT,
            data_source TEXT,
            impressions INTEGER DEFAULT 0,
            engagement_count INTEGER DEFAULT 0,
            awareness_metric REAL DEFAULT 0.0,
            activation_conversions INTEGER DEFAULT 0,
            reporting_date TEXT,
            metadata TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )
    # Ensure cost column exists for activity-level costing (backwards-safe)
    try:
        cur.execute("ALTER TABLE marketing_activities ADD COLUMN cost REAL DEFAULT 0.0")
    except Exception:
        # Column probably already exists or SQLite cannot alter; ignore
        pass
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS data_source_mappings (
            mapping_id TEXT PRIMARY KEY,
            source_system TEXT,
            source_name TEXT,
            description TEXT,
            api_endpoint TEXT,
            last_sync TEXT,
            sync_status TEXT,
            created_at TEXT,
            updated_at TEXT
        )
        """
    )

    # Budgets and cost allocations
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS budgets (
            budget_id TEXT PRIMARY KEY,
            event_id TEXT,
            campaign_name TEXT,
            allocated_amount REAL DEFAULT 0.0,
            currency TEXT DEFAULT 'USD',
            start_date TEXT,
            end_date TEXT,
            created_at TEXT,
            updated_at TEXT,
            FOREIGN KEY(event_id) REFERENCES events(event_id)
        )
        """
    )

    # --- Segmentation: Profiles and History ---
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS segment_profiles (
            profile_id TEXT PRIMARY KEY,
            lead_id TEXT,
            segments TEXT,
            attributes TEXT,
            last_updated TEXT,
            created_at TEXT,
            FOREIGN KEY(lead_id) REFERENCES leads(lead_id)
        )
        """
    )
    cur.execute(
        """
        CREATE TABLE IF NOT EXISTS segment_history (
            history_id INTEGER PRIMARY KEY AUTOINCREMENT,
            profile_id TEXT,
            lead_id TEXT,
            segments TEXT,
            attributes TEXT,
            changed_at TEXT,
            source TEXT,
            notes TEXT,
            FOREIGN KEY(profile_id) REFERENCES segment_profiles(profile_id)
        )
        """
    )
    # Initialize USAREC data source mappings (one-time)
    try:
        cur.execute("SELECT COUNT(*) FROM data_source_mappings")
```

```
        if cur.fetchone()[0] == 0:
            data_sources = [
                ("emm", "EMM", "Enterprise Marketing Manager - USAREC lead management"),
                ("ikrome", "iKrome", "Advanced analytics and attribution platform"),
                ("vantage", "Vantage", "Marketing performance and channel analysis"),
                ("g2_report_zone", "G2 Report Zone", "Competitive intelligence and market analysis"),
                ("aiem", "AIEM", "Army Integrated Enlisted Marketing system"),
                ("usarec_systems", "USAREC Systems", "Army Recruiting Command databases"),
            ]
            for idx, (sys, name, desc) in enumerate(data_sources, 1):
                cur.execute(
                    "INSERT INTO data_source_mappings (mapping_id, source_system, source_name, description, last_sync, sync_status, created_at) VALUES (?, ?, ?, ?, ?, ?, ?)
                    (f"map_{idx}", sys, name, desc, None, "pending", datetime.now().isoformat()),
                )
    except Exception as e:
        logging.warning(f"Data source mappings already initialized: {e}")

    # Initialize USAREC recruiting funnel stages (one-time)
    try:
        cur.execute("SELECT COUNT(*) FROM funnel_stages")
        if cur.fetchone()[0] == 0:
            stages = [
                ("lead", "Lead", 1, "Raw prospect, initial capture from marketing channels"),
                ("prospect", "Prospect", 2, "Qualified demographic match, engaged with content"),
                ("appointment_made", "Appointment Made", 3, "Scheduled appointment with recruiter"),
                ("appointment_conducted", "Appointment Conducted", 4, "Met with recruiter, initial discussion completed"),
                ("test", "Test", 5, "ASVAB or qualification test administered"),
                ("test_pass", "Test Pass", 6, "Passed ASVAB with qualifying score"),
                ("physical", "Physical", 7, "Medical examination and physical qualification completed"),
                ("enlist", "Enlist", 8, "Contract signed, enlisted into service"),
            ]
            for stage_id, name, order, desc in stages:
                cur.execute(
                    "INSERT INTO funnel_stages (stage_id, stage_name, sequence_order, description) VALUES (?, ?, ?, ?)",
                    (stage_id, name, order, desc),
                )
    except Exception as e:
        logging.warning(f"USAREC funnel stages already initialized: {e}")

    conn.commit()
    conn.close()


def migrate_json_to_db():
    # Migrate leads
    if os.path.exists(LEADS_FILE):
        try:
            with open(LEADS_FILE, "r", encoding="utf-8") as f:
                leads = json.load(f)
        except Exception:
            leads = []
        if leads:
            conn = get_db_conn()
            cur = conn.cursor()
            for l in leads:
                cur.execute(
                    "SELECT COUNT(1) as c FROM leads WHERE lead_id = ? AND received_at = ?",
                    (l.get("lead_id"), l.get("received_at")),
                )
                if cur.fetchone()[0] > 0:
                    continue
                cur.execute(
                    """
                    INSERT INTO leads (lead_id, age, education_level, cbsa_code, campaign_source, received_at, predicted_probability, score, recommendation, converted, raw_j
                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                    """,
                    (
                    l.get("lead_id"),
                    l.get("age"),
                    l.get("education_level"),
                    l.get("cbsa_code"),
                    l.get("campaign_source"),
                    l.get("received_at"),
                    l.get("predicted_probability"),
                    l.get("score"),
                    l.get("recommendation"),
                    1 if l.get("converted") else 0,
                    json.dumps(l),
                ),
            )
        conn.commit()
        conn.close()
    # Migrate pilot state
    if os.path.exists(PILOT_FILE):
        try:
            with open(PILOT_FILE, "r", encoding="utf-8") as f:
                state = json.load(f)
        except Exception:
            state = None
        if state:
            conn = get_db_conn()
            cur = conn.cursor()
            cur.execute(
                "REPLACE INTO pilot_state (id, started_at, config, status) VALUES (1, ?, ?, ?)",
                (state.get("started_at"), json.dumps(state.get("config")), state.get("status")),
            )
            conn.commit()
            conn.close()


# Ensure DB exists and migrate any JSON demo data
init_db()
# migrate_json_to_db()  # DEPRECATED: Old leads table schema, now using PRID-based recruiting funnel


# --- ML Model Loader & Scoring ---
def load_ml_model():
    logging.info("Loading Lead Scoring Model from storage...")
    model_path = os.path.join(DATA_DIR, "model.joblib")
    if os.path.exists(model_path):
        try:
            from joblib import load

            mdl = load(model_path)
            logging.info(f"Loaded model from {model_path}")
            return {"status": "ready", "model": mdl, "model_version": getattr(mdl, 'version', 'unknown')}
        except Exception as e:
            logging.warning(f"Failed to load model.joblib: {e}; falling back to simulated model")
    return {"status": "simulated", "model": None, "model_version": "simulated-v1"}


ML_MODEL = load_ml_model()
logging.info(f"ML Model initialized. Status: {ML_MODEL['status']}")


# --- Simple token auth (optional) ---
API_TOKEN = "[REDACTED_FOR_DEPOSIT]"
# Permanently disable auth checks for this deployment (temporary, but persistent)
# NOTE: This change removes authentication checks at the application level.
# Revert by setting DISABLE_AUTH back to False or restoring the original logic.
DISABLE_AUTH = True
# [REDACTED_FOR_DEPOSIT] if API_TOKEN:
# [REDACTED_FOR_DEPOSIT] logging.warning("TAAIP_API_TOKEN present but auth checks are permanently disabled in this build")
@app.middleware("http")
```

```
def auth_middleware(request: Request, call_next):
    # Authentication is disabled in this deployment. Allow all requests.
    return call_next(request)


def compute_score_from_dict(d: Dict[str, Any]) -> Dict[str, Any]:
    """Use real model if available, otherwise fall back to the simple simulator."""
    if ML_MODEL.get("model") is not None:
        try:
            model = ML_MODEL["model"]
            features = [
                float(d.get("age", 30)),
                1.0 if d.get("education_level") in ("Bachelors", "Masters") else 0.0,
                1.0 if d.get("campaign_source") == "High-Impact-Targeting-Campaign" else 0.0,
            ]
            prob = float(model.predict_proba([features])[0][1])
            score_int = int(min(100, max(1, round(prob * 100))))
            rec = "High Priority: Immediate Recruiter Engagement Required" if score_int >= 85 else (
                "Medium Priority: Add to Nurture Campaign Queue" if score_int >= 60 else "Low Priority: Monitor and Re-evaluate"
            )
            return {"lead_id": d.get("lead_id"), "predicted_probability": round(prob, 4), "score": score_int, "recommendation": rec}
        except Exception as e:
            logging.warning(f"Model scoring failed, falling back to simulated logic: {e}")

    base_score = random.randint(30, 85)
    education = d.get("education_level", "")
    campaign = d.get("campaign_source", "")
    if education in ["Bachelors", "Masters"]:
        base_score += 5
    if campaign == "High-Impact-Targeting-Campaign":
        base_score += 10
    final_score = min(100, base_score)
    probability = final_score / 100.0
    if final_score >= 85:
        recommendation = "High Priority: Immediate Recruiter Engagement Required"
    elif final_score >= 60:
        recommendation = "Medium Priority: Add to Nurture Campaign Queue"
    else:
        recommendation = "Low Priority: Monitor and Re-evaluate"
    return {
        "lead_id": d.get("lead_id"),
        "predicted_probability": round(probability, 4),
        "score": final_score,
        "recommendation": recommendation,
    }
@app.get("/api/v2/market/potential")
def get_market_potential():
    """Get market segmentation and potential analysis."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Inspect leads columns
    cur.execute("PRAGMA table_info(leads)")
    lead_cols = {row[1] for row in cur.fetchall()}
    # Choose a source-like column if available
    source_col = None
    for cand in ("source", "campaign_source", "lead_source", "channel"):
        if cand in lead_cols:
            source_col = cand
            break
    stage_col = 'current_stage' if 'current_stage' in lead_cols else ('stage' if 'stage' in lead_cols else None)

    # Get lead source statistics
    sources = []
    if source_col:
        cur.execute(f"""
            SELECT {source_col}, COUNT(*) as count
            FROM leads
            WHERE {source_col} IS NOT NULL
            GROUP BY {source_col}
            ORDER BY count DESC
        """)
        sources = [{"source": row[0], "count": row[1]} for row in cur.fetchall()]

    # Get demographics if available
    if stage_col:
        cur.execute(f"""
            SELECT
                COUNT(*) as total,
                SUM(CASE WHEN {stage_col} IN ('enlistment', 'ship') THEN 1 ELSE 0 END) as converted
            FROM leads
        """)
    else:
        cur.execute("SELECT COUNT(*) as total, 0 as converted FROM leads")
    market_row = cur.fetchone()
    total = market_row[0] or 0
    converted = market_row[1] or 0
    conversion_rate = round((converted / total * 100), 2) if total > 0 else 0

    conn.close()

    return {
        "status": "ok",
        "market": {
            "total_addressable_market": total,
            "conversion_rate": conversion_rate,
            "top_sources": sources[:5],
            "last_updated": datetime.now().isoformat()
        }
    }


@app.get("/api/v2/targeting/recommendations")
def get_targeting_recommendations():
    """Get AI-powered targeting recommendations for recruiting."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Inspect leads columns to adjust query
    cur.execute("PRAGMA table_info(leads)")
    lead_cols = {row[1] for row in cur.fetchall()}
    stage_col = 'current_stage' if 'current_stage' in lead_cols else ('stage' if 'stage' in lead_cols else None)
    score_col = 'propensity_score' if 'propensity_score' in lead_cols else ('score' if 'score' in lead_cols else None)
    name_first = 'first_name' if 'first_name' in lead_cols else None
    name_last = 'last_name' if 'last_name' in lead_cols else None
    source_col = None
    for cand in ("source", "campaign_source", "lead_source", "channel"):
        if cand in lead_cols:
            source_col = cand
            break
    days_col = 'days_in_stage' if 'days_in_stage' in lead_cols else None

    # Get high-potential leads (high propensity score, early stage)
    high_potential = []
    if stage_col and score_col:
        select_name = (
            f", {name_first}, {name_last}" if name_first and name_last else ""
        )
        select_source = f", {source_col}" if source_col else ""
        cur.execute(f"""
            SELECT lead_id{select_name}, {score_col}, {stage_col}{select_source}
            FROM leads
            WHERE {stage_col} IN ('lead', 'prospect', 'appointment_made')
            AND {score_col} >= 70
            ORDER BY {score_col} DESC
            LIMIT 10
        """)
        rows = cur.fetchall()
        for r in rows:
            idx = 0
            lead_id = r[idx]; idx += 1
            if name_first and name_last:
                first = r[idx]; idx += 1
                last = r[idx]; idx += 1
                name = f"{first or ''} {last or ''}".strip()
            else:
                name = None
            score = r[idx]; idx += 1
            stage = r[idx]; idx += 1
            src = r[idx] if source_col else None
            high_potential.append({
                "lead_id": lead_id,
                "name": name or lead_id,
                "score": score,
                "stage": stage,
                "source": src,
                "recommendation": "High priority - strong conversion potential"
            })

    # Get stagnant leads (long time in stage, needs attention)
    stagnant = []
    if stage_col and days_col:
        select_name = (
            f", {name_first}, {name_last}" if name_first and name_last else ""
        )
        cur.execute(f"""
            SELECT lead_id{select_name}, {stage_col}, {days_col}
            FROM leads
            WHERE {stage_col} IN ('prospect', 'appointment_made')
            AND {days_col} > 30
            ORDER BY {days_col} DESC
            LIMIT 10
        """)
        rows = cur.fetchall()
        for r in rows:
            idx = 0
            lead_id = r[idx]; idx += 1
            if name_first and name_last:
                first = r[idx]; idx += 1
                last = r[idx]; idx += 1
                name = f"{first or ''} {last or ''}".strip()
            else:
                name = None
            stage = r[idx]; idx += 1
            days = r[idx]
            stagnant.append({
                "lead_id": lead_id,
                "name": name or lead_id,
                "stage": stage,
                "days_in_stage": days,
                "recommendation": f"Follow up needed - {days} days in stage"
            })

    conn.close()

    return {
        "status": "ok",
        "recommendations": {
            "high_potential_leads": high_potential,
            "stagnant_leads": stagnant,
            "last_updated": datetime.now().isoformat()
        }
    }




class LeadData(BaseModel):
    """Schema for the input data required for lead scoring."""
    lead_id: str = Field(..., description="Unique identifier for the recruitment lead.")
    age: int = Field(..., ge=18, description="Age of the prospective recruit.")
    education_level: str = Field(..., description="Highest level of education completed (e.g., 'High School', 'Some College', 'Bachelors').")
    cbsa_code: str = Field(..., description="Core Based Statistical Area (CBSA) code for geographic targeting.")
    campaign_source: str = Field(..., description="Marketing channel/campaign that generated the lead.")


class ScoringResult(BaseModel):
    """Schema for the output data returned by the scoring engine."""
    lead_id: str
    predicted_probability: float = Field(..., ge=0.0, le=1.0, description="The probability (0.0 to 1.0) the lead will convert.")
    score: int = Field(..., ge=1, le=100, description="Lead score scaled from 1 to 100.")
    recommendation: str = Field(..., description="Actionable recommendation for the recruiter (e.g., High Priority Engagement).")


def get_metrics() -> Dict[str, Any]:
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT COUNT(1) as total, AVG(score) as avg_score, SUM(converted) as converted_sum FROM leads")
    row = cur.fetchone()
    total = row[0] or 0
    avg_score = float(row[1]) if row[1] is not None else 0.0
    converted = row[2] or 0
    conversion_rate = (converted / total) if total > 0 else 0.0

    by_cbsa = {}
    cur.execute("SELECT cbsa_code, COUNT(1) as cnt, AVG(score) as avg_score FROM leads GROUP BY cbsa_code")
    for r in cur.fetchall():
        cbsa = r[0] or "unknown"
        by_cbsa[cbsa] = {"count": r[1], "average_score": float(r[2]) if r[2] is not None else 0.0}

    conn.close()
    return {"total_leads": total, "average_score": round(avg_score, 2), "conversion_rate": round(conversion_rate, 4), "by_cbsa": by_cbsa}


@app.post("/api/v1/scoreLead", response_model=ScoringResult)
def score_lead(data: LeadData):
    try:
        logging.info(f"Scoring lead {data.lead_id} from CBSA {data.cbsa_code}...")
        result = compute_score_from_dict(model_to_dict(data))
        logging.info(f"Lead {data.lead_id} scored {result['score']}/100.")
        return ScoringResult(**result)
    except Exception as e:
        logging.error(f"Error during lead scoring for {data.lead_id}: {e}")
        raise HTTPException(status_code=500, detail="Internal processing error in ML service.")


@app.post("/api/v1/ingestLead")
def ingest_lead(data: LeadData):
    """Score the lead and persist it to the SQLite store."""
    result = compute_score_from_dict(model_to_dict(data))
    received_at = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO leads (lead_id, age, education_level, cbsa_code, campaign_source, received_at, predicted_probability, score, recommendation, converted, raw_json)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (
            data.lead_id,
            data.age,
            data.education_level,
            data.cbsa_code,
            data.campaign_source,
            received_at,
            result["predicted_probability"],
            result["score"],
            result["recommendation"],
            0,
            json.dumps(model_to_dict(data)),
        ),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "lead": {**result, "received_at": received_at}}


@app.get("/api/v1/metrics")
def metrics_endpoint():
    return get_metrics()


@app.post("/api/v1/startPilot")
def start_pilot(payload: Optional[Dict[str, Any]] = None):
    payload = payload or {}
    started_at = datetime.utcnow().isoformat()
    config = payload.get("config", {})
    status = payload.get("status", "running")
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        "REPLACE INTO pilot_state (id, started_at, config, status) VALUES (1, ?, ?, ?)",
        (started_at, json.dumps(config), status),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "started_at": started_at, "config": config, "pilot_status": status}


@app.get("/api/v1/pilotStatus")
def pilot_status():
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT started_at, config, status FROM pilot_state WHERE id = 1")
    row = cur.fetchone()
    conn.close()
    if not row:
        return {"status": "not_started"}
    return {"started_at": row[0], "config": json.loads(row[1]) if row[1] else {}, "status": row[2]}


@app.get("/health")
def health_check():
    """Returns the status of the service and the loaded ML model."""
    return {"status": "ok", "service": "TAAIP - Talent Acquisition Analytics and Intelligence Platform", "model_status": ML_MODEL.get("status", "unknown")}


# ========== EXTENDED API (v2): ROI, Funnel, Project Management, M-IPOE, Targeting, Forecasting ==========

# --- Pydantic Models ---

class EventCreate(BaseModel):
    name: str
    type: Optional[str] = None
    location: Optional[str] = None
    start_date: Optional[str] = None
    end_date: Optional[str] = None
    budget: Optional[float] = None
    team_size: Optional[int] = None
    targeting_principles: Optional[str] = None


class EventMetricsCreate(BaseModel):
    event_id: str
    date: str
    leads_generated: int = 0
    leads_qualified: int = 0
    conversion_count: int = 0
    cost_per_lead: Optional[float] = None
    roi: Optional[float] = None
    engagement_rate: Optional[float] = None


class CaptureSurveyCreate(BaseModel):
    event_id: str
    lead_id: Optional[str] = None
    technician_id: str
    effectiveness_rating: int
    feedback: str


class FunnelTransitionCreate(BaseModel):
    lead_id: str
    from_stage: Optional[str] = None
    to_stage: str
    transition_reason: Optional[str] = None
    technician_id: Optional[str] = None


class ProjectCreate(BaseModel):
    name: str
    event_id: Optional[str] = None
    start_date: str
    target_date: str
    owner_id: str
    objectives: Optional[str] = None
    success_criteria: Optional[str] = None


class TaskCreate(BaseModel):
    project_id: str
    title: str
    description: Optional[str] = None
    assigned_to: Optional[str] = None
    due_date: str
    priority: Optional[str] = None


class MIPOECreate(BaseModel):
    event_id: str
    phase: str  # intent, plan, order, execute, evaluate
    content: Dict[str, Any]
    owner_id: Optional[str] = None


class TargetingProfileCreate(BaseModel):
    event_id: str
    target_age_min: Optional[int] = None
    target_age_max: Optional[int] = None
    target_education_level: Optional[str] = None
    target_locations: Optional[str] = None  # comma-separated CBSA codes
    message_themes: Optional[str] = None  # comma-separated themes
    contact_frequency: Optional[int] = None
    conversion_target: Optional[float] = None
    cost_per_lead_target: Optional[float] = None


# NEW: Marketing Activity Models (USAREC-specific)
class MarketingActivityCreate(BaseModel):
    event_id: Optional[str] = None
    activity_type: str  # 'social_media', 'email', 'display_ad', 'event', 'referral', 'organic'
    campaign_name: str
    channel: str  # 'Facebook', 'Instagram', 'Email', 'Google Ads', 'TikTok', 'In-Person', 'YouTube'
    data_source: str  # 'emm', 'ikrome', 'vantage', 'g2_report_zone', 'aiem', 'usarec_systems'
    impressions: int = 0
    engagement_count: int = 0
    awareness_metric: float = 0.0  # 0.0-1.0 scale
    activation_conversions: int = 0
    reporting_date: str
    metadata: Optional[str] = None


class DataSourceSync(BaseModel):
    source_system: str  # 'emm', 'ikrome', 'vantage', 'g2_report_zone', 'aiem', 'usarec_systems'
    sync_data: Dict[str, Any]  # Flexible JSON for source-specific data


# --- Segmentation & Ingest Models ---
class SegmentProfileCreate(BaseModel):
    lead_id: str
    segments: Optional[Dict[str, Any]] = None  # e.g., {"age_group":"18-24","interests":[...]}
    attributes: Optional[Dict[str, Any]] = None  # free-form attributes


class SurveyIngest(BaseModel):
    lead_id: Optional[str] = None
    survey_id: str
    responses: Dict[str, Any]
    source: Optional[str] = "survey"
    received_at: Optional[str] = None


class CensusIngest(BaseModel):
    geography_code: str
    attributes: Dict[str, Any]
    source: Optional[str] = "census"
    received_at: Optional[str] = None


class SocialSignalIngest(BaseModel):
    external_id: str
    handle: Optional[str] = None
    signals: Dict[str, Any]
    source: Optional[str] = "social"
    received_at: Optional[str] = None


class EngagementIngest(BaseModel):
    event_id: Optional[str] = None
    activity_id: Optional[str] = None
    impressions: Optional[int] = 0
    engagement_count: Optional[int] = 0
    data_source: Optional[str] = None
    reporting_date: Optional[str] = None


# --- Events & ROI Tracking Endpoints ---

@app.post("/api/v2/events")
```

```
def create_event(event: EventCreate):
    """Create a new recruiting event."""
    import uuid
    event_id = f"evt_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO events (event_id, name, type, location, start_date, end_date, budget, team_size, targeting_principles, status, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, 'planned', ?, ?)
        """,
        (event_id, event.name, event.type, event.location, event.start_date, event.end_date, event.budget, event.team_size, event.targeting_principles, now, now),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "event_id": event_id}


@app.get("/api/v2/events")
def list_events(event_type: Optional[str] = None, rsid: Optional[str] = None, limit: int = 100):
    """List events with predicted fields for dashboards."""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cur = conn.cursor()
        query = (
            "SELECT event_id, name, COALESCE(event_type_category, type) AS event_type_category, "
            "location, start_date, end_date, budget, team_size, status, "
            "predicted_leads, predicted_conversions, predicted_roi, predicted_cost_per_lead, prediction_confidence "
            "FROM events WHERE 1=1"
        )
        params = []
        if event_type:
            query += " AND (event_type_category = ? OR type = ?)"
            params.extend([event_type, event_type])
        if rsid:
            # Include if schema has rsid column; ignore if not
            try:
                cur.execute("PRAGMA table_info(events)")
                cols = [r[1] for r in cur.fetchall()]
                if "rsid" in cols:
                    query += " AND rsid = ?"
                    params.append(rsid)
            except Exception:
                pass
        query += " ORDER BY start_date DESC LIMIT ?"
        params.append(max(1, min(limit, 500)))
        cur.execute(query, params)
        rows = [dict(r) for r in cur.fetchall()]
        conn.close()
        return {"status": "ok", "data": rows, "count": len(rows)}
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})


@app.get("/api/v2/events/{event_id}/metrics")
def get_event_metrics(event_id: str):
    """Get real-time ROI metrics for an event."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT date, leads_generated, leads_qualified, conversion_count, cost_per_lead, roi, engagement_rate FROM event_metrics WHERE event_id = ? ORDE
    metrics = [dict(row) for row in cur.fetchall()]
    conn.close()
    return {"event_id": event_id, "metrics": metrics}
def _now_iso():
    return datetime.utcnow().isoformat()
```

```
def update_segment_profile(lead_id: Optional[str], segments: Optional[Dict[str, Any]], attributes: Optional[Dict[str, Any]], source: str = "ingest", notes: Optional[str] = Nor
    """Merge incoming segment/attribute data into segment_profiles and record history."""
    conn = get_db_conn()
    cur = conn.cursor()
    now = _now_iso()

    profile_id = None
    if lead_id:
        profile_id = f"profile_{lead_id}"
    else:
        import uuid
        profile_id = f"profile_{uuid.uuid4().hex[:12]}"

    # Fetch existing
    cur.execute("SELECT segments, attributes FROM segment_profiles WHERE profile_id = ?", (profile_id,))
    row = cur.fetchone()
    existing_segments = {}
    existing_attrs = {}
    if row:
        try:
            existing_segments = json.loads(row[0]) if row[0] else {}
        except Exception:
            existing_segments = {}
        try:
            existing_attrs = json.loads(row[1]) if row[1] else {}
        except Exception:
            existing_attrs = {}

    # Merge (simple overwrite semantics for keys)
    merged_segments = existing_segments.copy()
    if segments:
        for k, v in segments.items():
            merged_segments[k] = v

    merged_attrs = existing_attrs.copy()
    if attributes:
        for k, v in attributes.items():
            merged_attrs[k] = v

    # Upsert profile
    cur.execute(
        "REPLACE INTO segment_profiles (profile_id, lead_id, segments, attributes, last_updated, created_at) VALUES (?, ?, ?, ?, ?, ?)",
        (profile_id, lead_id, json.dumps(merged_segments), json.dumps(merged_attrs), now, now),
    )

    # Insert history
    cur.execute(
        "INSERT INTO segment_history (profile_id, lead_id, segments, attributes, changed_at, source, notes) VALUES (?, ?, ?, ?, ?, ?, ?)",
        (profile_id, lead_id, json.dumps(merged_segments), json.dumps(merged_attrs), now, source, notes),
    )
    conn.commit()
    conn.close()
    return {"profile_id": profile_id, "segments": merged_segments, "attributes": merged_attrs}


@app.post("/api/v2/ingest/survey")
def ingest_survey(payload: SurveyIngest):
    """Ingest survey responses and update segmentation for the lead (if provided)."""
    received_at = payload.received_at or _now_iso()
    # Basic rule: convert some survey answers into segments
    segments = {}
    attributes = {"survey_id": payload.survey_id, "responses": payload.responses}
    # Example mapping: if age question present
    if payload.responses.get("age"):
        age = payload.responses.get("age")
        try:
            age = int(age)
            if age < 25:
                segments["age_group"] = "18-24"
            elif age < 35:
                segments["age_group"] = "25-34"
            else:
                segments["age_group"] = "35_plus"
        except Exception:
            pass

    result = update_segment_profile(payload.lead_id, segments, attributes, source=payload.source, notes=f"survey:{payload.survey_id}")
    return {"status": "ok", "result": result}


@app.post("/api/v2/ingest/census")
def ingest_census(payload: CensusIngest):
    """Ingest census attributes for a geography and update segment profiles of matching leads (basic behavior: store as attributes keyed by geography)."""
    received_at = payload.received_at or _now_iso()
    # For this prototype, we will store the census attributes as a standalone segment profile under geography code
    profile_id = f"census_{payload.geography_code}"
    conn = get_db_conn()
    cur = conn.cursor()
    now = _now_iso()
    cur.execute(
        "REPLACE INTO segment_profiles (profile_id, lead_id, segments, attributes, last_updated, created_at) VALUES (?, ?, ?, ?, ?, ?)",
        (profile_id, None, json.dumps({}), json.dumps(payload.attributes), now, now),
    )
    cur.execute(
        "INSERT INTO segment_history (profile_id, lead_id, segments, attributes, changed_at, source, notes) VALUES (?, ?, ?, ?, ?, ?, ?)",
        (profile_id, None, json.dumps({}), json.dumps(payload.attributes), now, payload.source, "census_import"),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "profile_id": profile_id}


@app.post("/api/v2/ingest/social")
def ingest_social(payload: SocialSignalIngest):
    """Ingest social signals and create/update segment profile mapped to external handle."""
    received_at = payload.received_at or _now_iso()
    # Map external_id/handle to a profile
    profile_id = f"social_{payload.external_id}"
    conn = get_db_conn()
    cur = conn.cursor()
    now = _now_iso()
    cur.execute(
        "REPLACE INTO segment_profiles (profile_id, lead_id, segments, attributes, last_updated, created_at) VALUES (?, ?, ?, ?, ?, ?)",
        (profile_id, None, json.dumps({}), json.dumps(payload.signals), now, now),
    )
    cur.execute(
        "INSERT INTO segment_history (profile_id, lead_id, segments, attributes, changed_at, source, notes) VALUES (?, ?, ?, ?, ?, ?, ?)",
        (profile_id, None, json.dumps({}), json.dumps(payload.signals), now, payload.source, "social_import"),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "profile_id": profile_id}


@app.post("/api/v2/ingest/engagements")
def ingest_engagements(payload: EngagementIngest):
    """Ingest bulk engagement/impression updates and optionally create marketing activity entries or update existing ones."""
    conn = get_db_conn()
    cur = conn.cursor()
    created = 0
    now = _now_iso()
    # If activity_id provided, update that activity
    if payload.activity_id:
        cur.execute("SELECT activity_id FROM marketing_activities WHERE activity_id = ?", (payload.activity_id,))
        if cur.fetchone():
            cur.execute(
                "UPDATE marketing_activities SET impressions = impressions + ?, engagement_count = engagement_count + ?, updated_at = ? WHERE activity_id = ?",
                (payload.impressions or 0, payload.engagement_count or 0, now, payload.activity_id),
            )
            conn.commit()
            conn.close()
            return {"status": "ok", "updated": payload.activity_id}

    # Otherwise, create a lightweight activity record
    import uuid
    activity_id = f"mkt_{uuid.uuid4().hex[:12]}"
    cur.execute(
        "INSERT INTO marketing_activities (activity_id, event_id, activity_type, campaign_name, channel, data_source, impressions, engagement_count, awareness_metric, 
        (activity_id, payload.event_id, 'engagement_batch', None, None, payload.data_source, payload.impressions or 0, payload.engagement_count or 0, 0.0, 0, payload.repor
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "activity_id": activity_id}


@app.get("/api/v2/segments/{lead_id}")
def get_segment_profile(lead_id: str):
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT profile_id, segments, attributes, last_updated FROM segment_profiles WHERE lead_id = ?", (lead_id,))
    row = cur.fetchone()
    conn.close()
    if not row:
        return {"status": "not_found"}
    try:
        segments = json.loads(row[1]) if row[1] else {}
    except Exception:
        segments = {}
    try:
        attrs = json.loads(row[2]) if row[2] else {}
    except Exception:
        attrs = {}
    return {"status": "ok", "profile_id": row[0], "segments": segments, "attributes": attrs, "last_updated": row[3]}


@app.post("/api/v2/events/{event_id}/metrics")
def add_event_metrics(event_id: str, metrics: EventMetricsCreate):
    """Record event metrics (live update from TA technician)."""
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO event_metrics (event_id, date, leads_generated, leads_qualified, conversion_count, cost_per_lead, roi, engagement_rate, created_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (event_id, metrics.date, metrics.leads_generated, metrics.leads_qualified, metrics.conversion_count, metrics.cost_per_lead, metrics.roi, metrics.engagement_rate, now)
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "message": "Metrics recorded"}


@app.post("/api/v2/events/{event_id}/survey")
def capture_survey(event_id: str, survey: CaptureSurveyCreate):
    """Capture real-time survey feedback from TA technician."""
    import uuid
    survey_id = f"sur_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO capture_survey (survey_id, event_id, lead_id, timestamp, technician_id, effectiveness_rating, feedback, created_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (survey_id, event_id, survey.lead_id, now, survey.technician_id, survey.effectiveness_rating, survey.feedback, now),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "survey_id": survey_id}


@app.get("/api/v2/events/{event_id}/feedback")
def get_event_feedback(event_id: str):
    """Get aggregated survey feedback for an event."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT technician_id, effectiveness_rating, feedback FROM capture_survey WHERE event_id = ? ORDER BY created_at DESC", (event_id,))
    feedback = [dict(row) for row in cur.fetchall()]
    conn.close()
    return {"event_id": event_id, "feedback": feedback}


# --- Funnel Endpoints ---

@app.get("/api/v2/funnel/stages")
def get_funnel_stages():
    """List all recruiting funnel stages."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT stage_id, stage_name, sequence_order, description FROM funnel_stages ORDER BY sequence_order")
    stages = [dict(row) for row in cur.fetchall()]
    conn.close()
    return {"stages": stages}


@app.post("/api/v2/funnel/transition")
def record_funnel_transition(transition: FunnelTransitionCreate):
    """Move a lead between funnel stages."""
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    # Insert using whichever identifier column exists in the DB (`lead_id` or `prid`).
    cur.execute("PRAGMA table_info(funnel_transitions)")
    existing_cols = [r[1] for r in cur.fetchall()]
    if "prid" in existing_cols:
        cur.execute(
            """
            INSERT INTO funnel_transitions (prid, from_stage, to_stage, transition_date, transition_reason, technician_id, created_at)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """,
            (transition.lead_id, transition.from_stage, transition.to_stage, now, transition.transition_reason, transition.technician_id, now),
        )
    else:
        cur.execute(
            """
            INSERT INTO funnel_transitions (lead_id, from_stage, to_stage, transition_date, transition_reason, technician_id, created_at)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """,
            (transition.lead_id, transition.from_stage, transition.to_stage, now, transition.transition_reason, transition.technician_id, now),
        )
    conn.commit()
    conn.close()
    return {"status": "ok", "message": f"Lead {transition.lead_id} transitioned to {transition.to_stage}"}


@app.get("/api/v2/funnel/metrics")
```

```
def get_funnel_metrics():
    """Get conversion metrics across all funnel stages."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Count leads in each stage (via most recent transition).
    # The DB schema may use `lead_id` (older) or `prid` (migrated). Attempt
    # the `lead_id`-based query first, and fall back to a `prid`-based
    # equivalent if the column doesn't exist.
    try:
        cur.execute("""
            SELECT on_stage, COUNT(*) as count
            FROM (
                SELECT lead_id, to_stage as on_stage
                FROM funnel_transitions
                WHERE (lead_id, created_at) IN (
                    SELECT lead_id, MAX(created_at)
                    FROM funnel_transitions
                    GROUP BY lead_id
                )
            ) latest_stage
            GROUP BY on_stage
            ORDER BY on_stage
        """)
        stage_counts = {row[0]: row[1] for row in cur.fetchall()}
    except sqlite3.OperationalError:
        # Likely no `lead_id` column — try using `prid` instead.
        try:
            cur.execute("""
                SELECT on_stage, COUNT(*) as count
                FROM (
                    SELECT prid, to_stage as on_stage
                    FROM funnel_transitions
                    WHERE (prid, created_at) IN (
                        SELECT prid, MAX(created_at)
                        FROM funnel_transitions
                        GROUP BY prid
                    )
                ) latest_stage
                GROUP BY on_stage
                ORDER BY on_stage
            """)
            stage_counts = {row[0]: row[1] for row in cur.fetchall()}
        except Exception:
            # Final fallback: count distinct identifiers per stage.
            cur.execute("""
                SELECT to_stage, COUNT(DISTINCT prid) as count
                FROM funnel_transitions
                GROUP BY to_stage
            """)
            stage_counts = {row[0]: row[1] for row in cur.fetchall()}
    except Exception:
        # Fallback for DBs that don't support tuple-comparison in WHERE
        try:
            cur.execute("""
                SELECT to_stage, COUNT(DISTINCT lead_id) as count
                FROM funnel_transitions
                GROUP BY to_stage
            """)
            stage_counts = {row[0]: row[1] for row in cur.fetchall()}
        except sqlite3.OperationalError:
            cur.execute("""
                SELECT to_stage, COUNT(DISTINCT prid) as count
                FROM funnel_transitions
                GROUP BY to_stage
            """)
            stage_counts = {row[0]: row[1] for row in cur.fetchall()}
    conn.close()
    return {"stage_distribution": stage_counts}


# --- Project Management Endpoints ---

@app.post("/api/v2/projects")
def create_project(project: ProjectCreate):
    """Create an event planning project."""
    import uuid
    project_id = f"prj_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO projects (project_id, name, event_id, start_date, target_date, owner_id, objectives, success_criteria, status, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, 'planning', ?, ?)
        """,
        (project_id, project.name, project.event_id, project.start_date, project.target_date, project.owner_id, project.objectives, project.success_criteria, now, now),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "project_id": project_id}


@app.post("/api/v2/projects/{project_id}/tasks")
def create_task(project_id: str, task: TaskCreate):
    """Create a task within a project."""
    import uuid
    task_id = f"tsk_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO tasks (task_id, project_id, title, description, assigned_to, due_date, status, priority, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, 'open', ?, ?, ?)
        """,
        (task_id, project_id, task.title, task.description, task.assigned_to, task.due_date, task.priority, now, now),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "task_id": task_id}


@app.put("/api/v2/projects/{project_id}/tasks/{task_id}")
def update_task(project_id: str, task_id: str, updates: Dict[str, Any]):
    """Update task status, due date, etc."""
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()

    set_clause = ", ".join([f"{k} = ?" for k in updates.keys()])
    set_clause += ", updated_at = ?"
    values = list(updates.values()) + [now, task_id]

    cur.execute(f"UPDATE tasks SET {set_clause} WHERE task_id = ?", values)
    conn.commit()
    conn.close()
    return {"status": "ok", "message": "Task updated"}


@app.get("/api/v2/projects/{project_id}/timeline")
def get_project_timeline(project_id: str):
    """Get project milestones and timeline."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT milestone_id, name, target_date, actual_date FROM milestones WHERE project_id = ? ORDER BY target_date", (project_id,))
    milestones = [dict(row) for row in cur.fetchall()]
    conn.close()
    return {"project_id": project_id, "milestones": milestones}


# --- M-IPOE Endpoints ---

@app.post("/api/v2/mipoe")
def create_mipoe(mipoe: MIPOECreate):
    """Create/document M-IPOE phase (Intent, Plan, Order, Execute, Evaluate)."""
    import uuid
    mipoe_id = f"mip_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO mipoe (mipoe_id, event_id, phase, content, owner_id, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, ?)
        """,
        (mipoe_id, mipoe.event_id, mipoe.phase, json.dumps(mipoe.content), mipoe.owner_id, now, now),
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "mipoe_id": mipoe_id}


@app.get("/api/v2/mipoe/{mipoe_id}")
def get_mipoe(mipoe_id: str):
    """Retrieve M-IPOE record."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT event_id, phase, content, owner_id, created_at, updated_at FROM mipoe WHERE mipoe_id = ?", (mipoe_id,))
    row = cur.fetchone()
    conn.close()
    if not row:
        raise HTTPException(status_code=404, detail="M-IPOE not found")
    return {
        "mipoe_id": mipoe_id,
        "event_id": row[0],
        "phase": row[1],
        "content": json.loads(row[2]),
        "owner_id": row[3],
        "created_at": row[4],
        "updated_at": row[5],
    }


# --- Targeting Profile (D3AE/F3A) Endpoints ---

@app.post("/api/v2/targeting-profiles")
def create_targeting_profile(profile: TargetingProfileCreate):
    """Create targeting profile with D3AE/F3A principles."""
    import uuid
    profile_id = f"tgt_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute(
        """
        INSERT INTO targeting_profiles (profile_id, event_id, target_age_min, target_age_max, target_education_level, target_locations, message_themes, contact_frequency
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (profile_id, profile.event_id, profile.target_age_min, profile.target_age_max, profile.target_education_level, profile.target_locations, profile.message_themes, profile.c
    )
    conn.commit()
    conn.close()
    return {"status": "ok", "profile_id": profile_id}


@app.get("/api/v2/targeting-profiles/{profile_id}")
def get_targeting_profile(profile_id: str):
    """Retrieve targeting profile."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("""
        SELECT event_id, target_age_min, target_age_max, target_education_level, target_locations, message_themes, contact_frequency, conversion_target, cost_per_lead_t
        FROM targeting_profiles WHERE profile_id = ?
    """, (profile_id,))
    row = cur.fetchone()
    conn.close()
    if not row:
        raise HTTPException(status_code=404, detail="Targeting profile not found")
    return {
        "profile_id": profile_id,
        "event_id": row[0],
        "target_age_min": row[1],
        "target_age_max": row[2],
        "target_education_level": row[3],
        "target_locations": row[4],
        "message_themes": row[5],
        "contact_frequency": row[6],
        "conversion_target": row[7],
        "cost_per_lead_target": row[8],
        "created_at": row[9],
        "updated_at": row[10],
    }


# --- Forecasting & Analytics Endpoints ---

@app.get("/api/v2/forecasts/{quarter}/{year}")
def get_forecast(quarter: int, year: int):
    """Get quarterly forecast."""
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT forecast_id, projected_leads, projected_conversions, projected_roi, confidence_level, methodology, created_at FROM forecasts WHERE quarter =
    row = cur.fetchone()
    conn.close()
    if not row:
        return {"quarter": quarter, "year": year, "message": "No forecast available"}
    return {
        "forecast_id": row[0],
        "quarter": quarter,
        "year": year,
        "projected_leads": row[1],
        "projected_conversions": row[2],
        "projected_roi": row[3],
        "confidence_level": row[4],
        "methodology": row[5],
        "created_at": row[6],
    }


@app.post("/api/v2/forecasts/generate")
def generate_forecast(quarter: int, year: int):
    """Trigger forecast generation (can use historical data or ML model)."""
    import uuid
    forecast_id = f"fct_{uuid.uuid4().hex[:12]}"
    now = datetime.utcnow().isoformat()

    # Simple heuristic: use average metrics from historical data
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT COUNT(*), AVG(conversion_count), AVG(roi) FROM event_metrics")
    row = cur.fetchone()

    total_events = row[0] or 1
    avg_conversions = row[1] or 5
    avg_roi = row[2] or 1.5

    # Project forward
    projected_leads = int(total_events * 10 * (quarter / 4))
    projected_conversions = int(projected_leads * (avg_conversions / 100))
    projected_roi = avg_roi
    confidence = 0.75

    cur.execute(
        """
        INSERT OR REPLACE INTO forecasts (forecast_id, quarter, year, projected_leads, projected_conversions, projected_roi, confidence_level, methodology, created_at,
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (forecast_id, quarter, year, projected_leads, projected_conversions, projected_roi, confidence, "historical_average", now, now),
    )
    conn.commit()
    conn.close()
    return {
        "status": "ok",
        "forecast_id": forecast_id,
        "quarter": quarter,
        "year": year,
        "projected_leads": projected_leads,
        "projected_conversions": projected_conversions,
        "projected_roi": projected_roi,
        "confidence_level": confidence,
    }


@app.get("/api/v2/analytics/dashboard")
def get_dashboard_snapshot():
    """Get comprehensive dashboard snapshot (all metrics, by quarter)."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Aggregate current quarter metrics
    cur.execute("SELECT COUNT(*), SUM(leads_generated), SUM(conversion_count), AVG(cost_per_lead), AVG(roi) FROM event_metrics")
    row = cur.fetchone()

    total_events = row[0] or 0
    total_leads = row[1] or 0
    total_conversions = row[2] or 0
    avg_cost = row[3] or 0
    avg_roi = row[4] or 0
    conversion_rate = (total_conversions / total_leads) if total_leads > 0 else 0

    conn.close()
    return {
        "dashboard": {
            "total_events": total_events,
            "total_leads": total_leads,
            "total_conversions": total_conversions,
            "conversion_rate": round(conversion_rate, 4),
            "avg_cost_per_lead": round(avg_cost, 2),
            "avg_roi": round(avg_roi, 2),
        }
    }


# --- NEW: Marketing Activity Tracking (USAREC Integration) ---
```

```
@app.post("/api/v2/marketing/activities")
def record_marketing_activity(data: MarketingActivityCreate):
    """Record marketing activity metrics (impressions, engagement, awareness, activation)."""
    import uuid
    conn = get_db_conn()
    cur = conn.cursor()

    activity_id = f"mkt_{uuid.uuid4().hex[:12]}"
    now = datetime.now().isoformat()

    cur.execute(
        """
        INSERT INTO marketing_activities
        (activity_id, event_id, activity_type, campaign_name, channel, data_source,
         impressions, engagement_count, awareness_metric, activation_conversions,
         reporting_date, metadata, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (
            activity_id, data.event_id, data.activity_type, data.campaign_name,
            data.channel, data.data_source, data.impressions, data.engagement_count,
            data.awareness_metric, data.activation_conversions, data.reporting_date,
            data.metadata, now, now
        )
    )
    conn.commit()
    conn.close()

    return {"status": "ok", "activity_id": activity_id}


@app.get("/api/v2/marketing/activities")
def get_marketing_activities(event_id: Optional[str] = None, data_source: Optional[str] = None):
    """Get marketing activities (filtered by event or data source)."""
    conn = get_db_conn()
    cur = conn.cursor()

    query = "SELECT * FROM marketing_activities WHERE 1=1"
    params = []

    if event_id:
        query += " AND event_id = ?"
        params.append(event_id)
    if data_source:
        query += " AND data_source = ?"
        params.append(data_source)

    query += " ORDER BY reporting_date DESC"

    cur.execute(query, params)
    rows = cur.fetchall()
    conn.close()

    activities = [dict(row) for row in rows]
    return {"status": "ok", "count": len(activities), "activities": activities}


@app.get("/api/v2/marketing/analytics")
def get_marketing_analytics(event_id: Optional[str] = None):
    """Get aggregated marketing performance metrics."""
    conn = get_db_conn()
    cur = conn.cursor()

    if event_id:
        cur.execute(
            """
            SELECT
```

```
                SUM(impressions) as total_impressions,
                SUM(engagement_count) as total_engagement,
                AVG(awareness_metric) as avg_awareness,
                SUM(activation_conversions) as total_activations,
                COUNT(DISTINCT data_source) as sources_count,
                COUNT(DISTINCT channel) as channels_count
            FROM marketing_activities
            WHERE event_id = ?
            """,
            (event_id,)
        )
    else:
        cur.execute(
            """
            SELECT
                SUM(impressions) as total_impressions,
                SUM(engagement_count) as total_engagement,
                AVG(awareness_metric) as avg_awareness,
                SUM(activation_conversions) as total_activations,
                COUNT(DISTINCT data_source) as sources_count,
                COUNT(DISTINCT channel) as channels_count
            FROM marketing_activities
            """
        )

    row = cur.fetchone()
    conn.close()

    if not row:
        return {
            "status": "ok",
            "total_impressions": 0,
            "total_engagement": 0,
            "avg_awareness": 0.0,
            "total_activations": 0,
            "sources_count": 0,
            "channels_count": 0
        }

    return {
        "status": "ok",
        "total_impressions": row[0] or 0,
        "total_engagement": row[1] or 0,
        "avg_awareness": round(row[2] or 0.0, 2),
        "total_activations": row[3] or 0,
        "sources_count": row[4] or 0,
        "channels_count": row[5] or 0
    }


@app.get("/api/v2/kpis")
def get_kpis(event_id: Optional[str] = None, start_date: Optional[str] = None, end_date: Optional[str] = None, data_source: Optional[str] = None, segment_key: Optional[st
    """Compute derived KPIs: CPL, CPE, CPC, total event/campaign cost.

    - If `event_id` provided, scope to that event.
    - Optionally filter by `start_date`/`end_date` (reporting_date on activities).
    - `segment_key` and `segment_value` perform a simple substring match against serialized segment JSON in `segment_profiles`.
    """
    conn = get_db_conn()
    cur = conn.cursor()

    params = []
    where_clauses = []

    if event_id:
        where_clauses.append("ma.event_id = ?")
        params.append(event_id)
```

```
if data_source:
    where_clauses.append("ma.data_source = ?")
    params.append(data_source)

if start_date:
    where_clauses.append("ma.reporting_date >= ?")
    params.append(start_date)
if end_date:
    where_clauses.append("ma.reporting_date <= ?")
    params.append(end_date)

base_where = ""
if where_clauses:
    base_where = "WHERE " + " AND ".join(where_clauses)

# Aggregate activity-level sums
query = f"SELECT SUM(ma.cost) as total_cost, SUM(ma.impressions) as impressions, SUM(ma.engagement_count) as engagements, SUM(ma.activation_conversions)
cur.execute(query, params)
row = cur.fetchone()
total_cost = row[0] or 0.0
total_impressions = row[1] or 0
total_engagements = row[2] or 0
total_activations = row[3] or 0

# Include budgets for event-level if event_id provided
budget_total = 0.0
if event_id:
    cur.execute("SELECT SUM(allocated_amount) FROM budgets WHERE event_id = ?", (event_id,))
    brow = cur.fetchone()
    if brow and brow[0]:
        budget_total = brow[0]

# Combine costs (activity-level cost + budgets)
combined_cost = float(total_cost or 0.0) + float(budget_total or 0.0)

# Compute derived KPIs with safe guards
cpl = (combined_cost / total_activations) if total_activations > 0 else None
cpe = (combined_cost / total_engagements) if total_engagements > 0 else None
cpc = (combined_cost / total_impressions) if total_impressions > 0 else None

result = {
    "status": "ok",
    "total_cost": combined_cost,
    "budget_total": budget_total,
    "activity_cost": total_cost,
    "total_impressions": total_impressions,
    "total_engagements": total_engagements,
    "total_activations": total_activations,
    "cpl": round(cpl, 2) if cpl is not None else None,
    "cpe": round(cpe, 2) if cpe is not None else None,
    "cpc": round(cpc, 4) if cpc is not None else None,
}

# If segment filter provided, compute segment-level KPIs by joining segment_profiles
if segment_key and segment_value:
    seg_clause = f"AND sp.segments LIKE ?"
    seg_param = f'%"{segment_key}": "{segment_value}"%'
    # Need to run a join query
    seg_query = f"SELECT SUM(ma.cost) as total_cost, SUM(ma.impressions) as impressions, SUM(ma.engagement_count) as engagements, SUM(ma.activation_conver
    # Build params for seg_query
    seg_params = params.copy()
    seg_params.append(seg_param)
    try:
        cur.execute(seg_query, seg_params)
        srow = cur.fetchone()
        s_total_cost = srow[0] or 0.0
        s_impressions = srow[1] or 0
```

```
            s_engagements = srow[2] or 0
            s_activations = srow[3] or 0
            scpl = (s_total_cost / s_activations) if s_activations > 0 else None
            scpe = (s_total_cost / s_engagements) if s_engagements > 0 else None
            scpc = (s_total_cost / s_impressions) if s_impressions > 0 else None
            result["segment"] = {
                "key": segment_key,
                "value": segment_value,
                "total_cost": s_total_cost,
                "impressions": s_impressions,
                "engagements": s_engagements,
                "activations": s_activations,
                "cpl": round(scpl, 2) if scpl is not None else None,
                "cpe": round(scpe, 2) if scpe is not None else None,
                "cpc": round(scpc, 4) if scpc is not None else None,
            }
        except Exception:
            # If join fails (data shapes), skip segment breakdown
            result["segment"] = {"error": "segment breakdown unavailable"}

    conn.close()
    return result


def _stream_csv(rows, headers):
    """Helper to stream CSV from rows (iterable of dict) and headers list."""
    buffer = io.StringIO()
    writer = csv.DictWriter(buffer, fieldnames=headers)
    writer.writeheader()
    for r in rows:
        writer.writerow({k: r.get(k, "") for k in headers})
    buffer.seek(0)
    return StreamingResponse(buffer, media_type="text/csv")


# --- Simple token-based auth for export endpoints ---
EXPORT_API_TOKEN = "[REDACTED_FOR_DEPOSIT]"


def _verify_export_token(request: Request):
    # Look for X-API-KEY or Bearer token
    key = request.headers.get("X-API-KEY") or None
    if not key:
        auth = request.headers.get("Authorization")
        if auth and auth.lower().startswith("bearer "):
            key = auth.split(None, 1)[1].strip()
    if not key or key ! = "[REDACTED_FOR_DEPOSIT]"
        raise HTTPException(status_code=401, detail="Invalid or missing API token")


# --- Export scheduler (background thread) ---
_export_scheduler = {"thread": None, "stop_event": None, "interval": None}


def _export_worker(interval: int, stop_event: "threading.Event"):
    import time
    while not stop_event.is_set():
        try:
            # call internal export runner
            run_exports()
        except Exception:
            logging.exception("Scheduled export failed")
        # wait for interval or stop
        stop_event.wait(interval)
@app.get("/api/v2/exports/activities.csv")
def export_activities_csv(event_id: Optional[str] = None, data_source: Optional[str] = None, request: Request = None):
    """Return a CSV of marketing activities optionally filtered by event_id or data_source."""
    if request is not None:
        _verify_export_token(request)
    conn = get_db_conn()
    cur = conn.cursor()
    params = []
    where = []
    if event_id:
        where.append("event_id = ?")
        params.append(event_id)
    if data_source:
        where.append("data_source = ?")
        params.append(data_source)
    # include segment JSON columns where lead_id matches event_id (best-effort)
    q = "SELECT ma.activity_id, ma.event_id, ma.activity_type, ma.campaign_name, ma.channel, ma.data_source, ma.impressions, ma.engagement_count, ma.awareness_m
    if where:
        q += " WHERE " + " AND ".join(where)
    cur.execute(q, params)
    rows = [dict(row) for row in cur.fetchall()]
    conn.close()
    headers = ["activity_id", "event_id", "activity_type", "campaign_name", "channel", "data_source", "impressions", "engagement_count", "awareness_metric", "activation_
    return _stream_csv(rows, headers)


@app.get("/api/v2/exports/kpis.csv")
def export_kpis_csv(event_id: Optional[str] = None, request: Request = None):
    """Return a CSV with KPI rows (per event or overall)."""
    if request is not None:
        _verify_export_token(request)
    conn = get_db_conn()
    cur = conn.cursor()
    if event_id:
        cur.execute("SELECT event_id FROM events WHERE event_id = ?", (event_id,))
        if not cur.fetchone():
            conn.close()
            raise HTTPException(status_code=404, detail="event not found")
        # return a single-row CSV for the event
        kpi = get_kpis(event_id=event_id)
        row = {
            "event_id": event_id,
            "total_cost": kpi.get("total_cost"),
            "activity_cost": kpi.get("activity_cost"),
            "budget_total": kpi.get("budget_total"),
            "total_impressions": kpi.get("total_impressions"),
            "total_engagements": kpi.get("total_engagements"),
            "total_activations": kpi.get("total_activations"),
            "cpl": kpi.get("cpl"),
            "cpe": kpi.get("cpe"),
            "cpc": kpi.get("cpc"),
        }
        conn.close()
        return _stream_csv([row], list(row.keys()))
    # otherwise, return KPIs for all events
    cur.execute("SELECT event_id FROM events")
    events = [r[0] for r in cur.fetchall()]
    rows = []
    for ev in events:
        kpi = get_kpis(event_id=ev)
        rows.append({
            "event_id": ev,
            "total_cost": kpi.get("total_cost"),
            "activity_cost": kpi.get("activity_cost"),
            "budget_total": kpi.get("budget_total"),
            "total_impressions": kpi.get("total_impressions"),
            "total_engagements": kpi.get("total_engagements"),
            "total_activations": kpi.get("total_activations"),
            "cpl": kpi.get("cpl"),
            "cpe": kpi.get("cpe"),
            "cpc": kpi.get("cpc"),
        })
    conn.close()
    headers = ["event_id", "total_cost", "activity_cost", "budget_total", "total_impressions", "total_engagements", "total_activations", "cpl", "cpe", "cpc"]
    return _stream_csv(rows, headers)


@app.post("/api/v2/exports/run")
def run_exports(request: Request = None):
    """Run exports and write CSV files to the `exports/` folder inside project root."""
    if request is not None:
        _verify_export_token(request)
    exports_dir = Path(os.path.join(os.path.dirname(__file__), "exports"))
    exports_dir.mkdir(parents=True, exist_ok=True)
    # Write activities.csv
    act_resp = export_activities_csv()
    # act_resp is a StreamingResponse backed by StringIO; read its body
    act_body = act_resp.body_iterator
    # To write, call export endpoint logic directly to obtain rows
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT activity_id, event_id, activity_type, campaign_name, channel, data_source, impressions, engagement_count, awareness_metric, activation_convers
    rows = [dict(r) for r in cur.fetchall()]
    conn.close()
    activities_path = exports_dir / "activities.csv"
    with activities_path.open("w", newline='') as fh:
        writer = csv.DictWriter(fh, fieldnames=["activity_id", "event_id", "activity_type", "campaign_name", "channel", "data_source", "impressions", "engagement_count", "
        writer.writeheader()
        for r in rows:
            writer.writerow(r)

    # Write kpis.csv
    conn = get_db_conn()
    cur = conn.cursor()
    cur.execute("SELECT event_id FROM events")
    events = [r[0] for r in cur.fetchall()]
    kpis_path = exports_dir / "kpis.csv"
    with kpis_path.open("w", newline='') as fh:
        headers = ["event_id", "total_cost", "activity_cost", "budget_total", "total_impressions", "total_engagements", "total_activations", "cpl", "cpe", "cpc"]
        writer = csv.DictWriter(fh, fieldnames=headers)
        writer.writeheader()
        for ev in events:
            kpi = get_kpis(event_id=ev)
            writer.writerow({
                "event_id": ev,
                "total_cost": kpi.get("total_cost"),
                "activity_cost": kpi.get("activity_cost"),
                "budget_total": kpi.get("budget_total"),
                "total_impressions": kpi.get("total_impressions"),
                "total_engagements": kpi.get("total_engagements"),
                "total_activations": kpi.get("total_activations"),
                "cpl": kpi.get("cpl"),
                "cpe": kpi.get("cpe"),
                "cpc": kpi.get("cpc"),
            })

    return {"status": "ok", "exports": [str(activities_path), str(kpis_path)]}


@app.post("/api/v2/exports/schedule")
def schedule_exports(interval_seconds: int = 300, request: Request = None):
    """Start a background export scheduler that runs every `interval_seconds` seconds."""
    if request is not None:
        _verify_export_token(request)
    import threading
    if _export_scheduler.get("thread") and _export_scheduler.get("thread").is_alive():
        return {"status": "ok", "message": "scheduler already running", "interval": _export_scheduler.get("interval")}
    stop_event = threading.Event()
    t = threading.Thread(target=_export_worker, args=(interval_seconds, stop_event), daemon=True)
    _export_scheduler["thread"] = t
    _export_scheduler["stop_event"] = stop_event
    _export_scheduler["interval"] = interval_seconds
    t.start()
    return {"status": "ok", "message": "scheduler started", "interval": interval_seconds}


@app.post("/api/v2/exports/schedule/stop")
def stop_export_scheduler(request: Request = None):
    if request is not None:
        _verify_export_token(request)
    import threading
    ev = _export_scheduler.get("stop_event")
    th = _export_scheduler.get("thread")
    if ev:
        ev.set()
    if th and th.is_alive():
        th.join(timeout=2)
    _export_scheduler["thread"] = None
    _export_scheduler["stop_event"] = None
    _export_scheduler["interval"] = None
    return {"status": "ok", "message": "scheduler stopped"}


@app.get("/api/v2/odata/activities")
def odata_activities(select: Optional[str] = None, filter: Optional[str] = None, top: Optional[int] = None, skip: Optional[int] = None, request: Request = None):
    """Simple OData-like endpoint for marketing activities supporting select, filter (single equality), top, skip.

    Example: /api/v2/odata/activities?select=activity_id,activity_type&filter=activity_type eq 'social_media'&top=10
    """
    if request is not None:
        _verify_export_token(request)
    conn = get_db_conn()
    cur = conn.cursor()

    allowed_cols = {"activity_id", "event_id", "activity_type", "campaign_name", "channel", "data_source", "impressions", "engagement_count", "reporting_date", "cost"}

    if select:
        cols = [c.strip() for c in select.split(',') if c.strip() and c.strip() in allowed_cols]
        if not cols:
            cols = ["activity_id", "event_id", "activity_type"]
    else:
        cols = ["activity_id", "event_id", "activity_type", "campaign_name", "channel", "data_source", "impressions", "engagement_count", "reporting_date"]

    params = []
    where_clauses = []
    if filter:
        parts = filter.split(" eq ")
        if len(parts) == 2:
            field = parts[0].strip()
            val = parts[1].strip().strip("'\"")
            if field in allowed_cols:
                where_clauses.append(f"{field} = ?")
                params.append(val)

    q = f"SELECT {', '.join(cols)} FROM marketing_activities"
    if where_clauses:
        q += " WHERE " + " AND ".join(where_clauses)

    if top is not None:
        q += f" LIMIT {top}"
    if skip is not None:
        if "LIMIT" in q:
            q += f" OFFSET {skip}"
        else:
            q += f" LIMIT -1 OFFSET {skip}"

    cur.execute(q, params)
    rows = [dict(r) for r in cur.fetchall()]
    conn.close()
    return {"count": len(rows), "items": rows}


@app.get("/api/v2/marketing/sources")
def get_data_sources():
    """Get list of available USAREC data sources (EMM, iKrome, Vantage, G2, AIEM, USAREC Systems)."""
    conn = get_db_conn()
    cur = conn.cursor()

    cur.execute("SELECT mapping_id, source_system, source_name, description, last_sync, sync_status FROM data_source_mappings ORDER BY source_system")
    rows = cur.fetchall()
    conn.close()

    sources = [dict(row) for row in rows]
    return {"status": "ok", "sources": sources}


@app.post("/api/v2/marketing/sync")
def sync_data_source(data: DataSourceSync):
    """Sync data from a USAREC data source (EMM, iKrome, Vantage, G2, AIEM, USAREC Systems)."""
    conn = get_db_conn()
    cur = conn.cursor()

    now = datetime.now().isoformat()

    # Validate data source
    cur.execute("SELECT mapping_id FROM data_source_mappings WHERE source_system = ?", (data.source_system,))
    if not cur.fetchone():
        conn.close()
        return {"status": "error", "message": f"Unknown data source: {data.source_system}"}

    # Update sync status
    cur.execute(
        "UPDATE data_source_mappings SET last_sync = ?, sync_status = ?, updated_at = ? WHERE source_system = ?",
        (now, "synced", now, data.source_system)
    )

    # Parse incoming data and create marketing activities
    activities_created = 0
    if isinstance(data.sync_data, dict):
        for key, value in data.sync_data.items():
            if isinstance(value, dict) and all(k in value for k in ['campaign', 'impressions', 'engagement']):
                import uuid
                activity_id = f"mkt_{uuid.uuid4().hex[:12]}"

                cur.execute(
                    """
                    INSERT INTO marketing_activities
                    (activity_id, activity_type, campaign_name, channel, data_source,
                     impressions, engagement_count, awareness_metric, activation_conversions,
                     reporting_date, metadata, created_at, updated_at)
                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                    """,
                    (
                        activity_id, value.get('type', 'sync'), value.get('campaign', key),
                        value.get('channel', key), data.source_system,
                        int(value.get('impressions', 0)), int(value.get('engagement', 0)),
                        float(value.get('awareness', 0.0)), int(value.get('activation', 0)),
                        datetime.now().date().isoformat(), json.dumps(value),
                        now, now
                    )
                )
                activities_created += 1

    conn.commit()
    conn.close()

    return {
        "status": "ok",
        "source": data.source_system,
        "activities_created": activities_created,
        "sync_timestamp": now
    }


@app.get("/api/v2/marketing/funnel-attribution")
def get_funnel_attribution(data_source: Optional[str] = None):
    """Get marketing attribution by recruiting funnel stage."""
    conn = get_db_conn()
    cur = conn.cursor()

    if data_source:
        cur.execute(
            """
            SELECT
                fs.stage_name,
                COUNT(DISTINCT ft.lead_id) as leads_in_stage,
                SUM(ma.impressions) as total_impressions,
                SUM(ma.engagement_count) as total_engagement,
                AVG(ma.awareness_metric) as avg_awareness,
                SUM(ma.activation_conversions) as activations
            FROM funnel_stages fs
            LEFT JOIN funnel_transitions ft ON fs.stage_id = ft.to_stage
            LEFT JOIN marketing_activities ma ON ma.data_source = ?
            GROUP BY fs.stage_id, fs.stage_name
            ORDER BY fs.sequence_order
            """,
            (data_source,)
        )
    else:
        cur.execute(
            """
            SELECT
                fs.stage_name,
                COUNT(DISTINCT ft.lead_id) as leads_in_stage,
                SUM(ma.impressions) as total_impressions,
                SUM(ma.engagement_count) as total_engagement,
                AVG(ma.awareness_metric) as avg_awareness,
                SUM(ma.activation_conversions) as activations
            FROM funnel_stages fs
            LEFT JOIN funnel_transitions ft ON fs.stage_id = ft.to_stage
            LEFT JOIN marketing_activities ma ON 1=1
            GROUP BY fs.stage_id, fs.stage_name
            ORDER BY fs.sequence_order
            """
        )

    rows = cur.fetchall()
    conn.close()

    attribution = [
        {
            "stage": row[0],
            "leads_in_stage": row[1] or 0,
            "impressions": row[2] or 0,
            "engagement": row[3] or 0,
            "awareness": round(row[4] or 0.0, 2),
            "activations": row[5] or 0
        }
        for row in rows
    ]

    return {"status": "ok", "attribution": attribution}


# === AI PIPELINE ENDPOINTS ===

@app.post("/api/v2/ai/train")
async def train_ai_model(request: Request):
    """Train lead propensity model on historical leads from database."""
    try:
        from taaip_ai_pipeline import train_lead_propensity_model
        result = train_lead_propensity_model(DB_FILE)
        return {
            "status": "ok",
            "model": "lead_propensity",
            "accuracy": result.get("accuracy", 0),
            "training_samples": result.get("samples", 0),
            "message": "Model trained successfully"
        }
    except ImportError:
        return {
            "status": "error",
            "message": "scikit-learn not installed. Install with: pip install scikit-learn"
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.post("/api/v2/ai/predict")
async def predict_leads(request: Request):
    """Batch predict lead propensity scores."""
    try:
        from taaip_ai_pipeline import predict_lead_propensity
        body = await request.json()
        leads = body.get("leads", [])

        if not leads:
            return {"status": "error", "message": "No leads provided"}

        predictions = predict_lead_propensity(leads)
        return {
            "status": "ok",
            "predictions": predictions,
            "count": len(predictions)
        }
    except ImportError:
        return {
            "status": "error",
            "message": "scikit-learn not installed. Using mock predictions."
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.get("/api/v2/ai/model-status")
async def get_model_status():
    """Get current AI model status and metadata."""
    try:
        from taaip_ai_pipeline import get_model_status
        status = get_model_status()
        return {
            "status": "ok",
            "model": status
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e),
            "model": {
                "accuracy": 0,
                "training_samples": 0,
                "model_path": "models/lead_propensity_model.pkl",
                "last_updated": None
            }
        }


# === LMS ENDPOINTS ===

@app.post("/api/v2/lms/enroll")
async def enroll_user_lms(request: Request):
    """Enroll a user in an LMS course."""
    try:
        from taaip_lms import get_lms_manager
        body = await request.json()
        user_id = body.get("user_id")
        course_id = body.get("course_id")

        if not user_id or not course_id:
            return {"status": "error", "message": "user_id and course_id required"}

        lms = get_lms_manager(DB_FILE)
        enrollment_id = lms.enroll_user(user_id, course_id)

        return {
            "status": "ok",
            "enrollment_id": enrollment_id,
            "user_id": user_id,
            "course_id": course_id,
            "message": "User enrolled successfully"
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.put("/api/v2/lms/progress")
async def update_lms_progress(request: Request):
    """Update user progress in a course."""
    try:
        from taaip_lms import get_lms_manager
        body = await request.json()
        enrollment_id = body.get("enrollment_id")
        progress_percent = body.get("progress_percent", 0)

        if not enrollment_id:
            return {"status": "error", "message": "enrollment_id required"}

        if not (0 <= progress_percent <= 100):
            return {"status": "error", "message": "progress_percent must be 0-100"}

        lms = get_lms_manager(DB_FILE)
        lms.update_progress(enrollment_id, progress_percent)
        return {
            "status": "ok",
            "enrollment_id": enrollment_id,
            "progress_percent": progress_percent,
            "message": "Progress updated successfully"
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.get("/api/v2/lms/enrollments/{user_id}")
async def get_user_enrollments(user_id: str):
    """Get all courses enrolled by a user."""
    try:
        from taaip_lms import get_lms_manager
        lms = get_lms_manager(DB_FILE)
        enrollments = lms.get_user_enrollments(user_id)

        return {
            "status": "ok",
            "user_id": user_id,
            "enrollments": enrollments,
            "count": len(enrollments)
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.get("/api/v2/lms/stats")
async def get_lms_stats():
    """Get overall LMS statistics."""
    try:
        from taaip_lms import get_lms_manager
        lms = get_lms_manager(DB_FILE)
        stats = lms.get_course_stats()

        return {
            "status": "ok",
            "stats": stats
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


@app.get("/api/v2/lms/courses")
async def get_all_courses():
    """Get all available courses."""
    try:
        from taaip_lms import get_lms_manager
        lms = get_lms_manager(DB_FILE)
        conn = sqlite3.connect(DB_FILE)
        cur = conn.cursor()
        cur.execute("SELECT course_id, title, description FROM courses ORDER BY created_at DESC")
        courses = [{"course_id": row[0], "title": row[1], "description": row[2]} for row in cur.fetchall()]
        conn.close()

        return {
            "status": "ok",
            "courses": courses,
            "count": len(courses)
        }
    except Exception as e:
        return {
            "status": "error",
            "message": str(e)
        }


# --- ANALYTICS & VISUALIZATION ENDPOINTS ---

@app.get("/api/v2/analytics/cbsa")
def get_top_cbsas(limit: int = 10):
    """Get top CBSAs by lead volume, conversion rate, and potential."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Get CBSA data from leads table
    cur.execute("""
        SELECT
            cbsa_code,
            COUNT(*) as lead_count,
            AVG(score) as avg_score,
            SUM(CASE WHEN recommendation LIKE 'Tier 1%' OR recommendation LIKE 'Tier 2%' THEN 1 ELSE 0 END) as high_quality_leads,
            COUNT(*) * 100.0 / (SELECT COUNT(*) FROM leads) as market_share
        FROM leads
        WHERE cbsa_code IS NOT NULL
        GROUP BY cbsa_code
        ORDER BY lead_count DESC
        LIMIT ?
    """, (limit,))

    rows = cur.fetchall()
    conn.close()

    # Mock CBSA names (in production, join with CBSA reference table)
    cbsa_names = {
        "35620": "New York-Newark-Jersey City, NY-NJ-PA",
        "31080": "Los Angeles-Long Beach-Anaheim, CA",
        "16980": "Chicago-Naperville-Elgin, IL-IN-WI",
        "19100": "Dallas-Fort Worth-Arlington, TX",
        "26420": "Houston-The Woodlands-Sugar Land, TX",
        "47900": "Washington-Arlington-Alexandria, DC-VA-MD-WV",
        "33100": "Miami-Fort Lauderdale-West Palm Beach, FL",
        "37980": "Philadelphia-Camden-Wilmington, PA-NJ-DE-MD",
        "12060": "Atlanta-Sandy Springs-Roswell, GA",
        "14460": "Boston-Cambridge-Newton, MA-NH",
    }

    cbsas = []
    for row in rows:
        cbsa_code = row[0]
        cbsas.append({
            "cbsa_code": cbsa_code,
            "cbsa_name": cbsa_names.get(cbsa_code, f"CBSA {cbsa_code}"),
            "lead_count": row[1],
            "avg_score": round(row[2], 2) if row[2] else 0,
            "high_quality_count": row[3],
            "market_share": round(row[4], 2) if row[4] else 0,
            "conversion_potential": round((row[2] or 0) * 0.15, 2)  # Mock calculation
        })

    return {
        "status": "ok",
        "count": len(cbsas),
        "cbsas": cbsas
    }
```

```
@app.get("/api/v2/analytics/schools")
def get_targeted_schools(limit: int = 20):
    """Get targeted schools with recruitment performance metrics."""
    # Mock data (in production, this would come from school_targeting table)
    schools = [
        {"name": "University of Texas at Austin", "city": "Austin, TX", "type": "4-Year", "leads": 245, "conversions": 38, "events": 12, "priority": "Must Win"},
        {"name": "Arizona State University", "city": "Tempe, AZ", "type": "4-Year", "leads": 198, "conversions": 31, "events": 9, "priority": "Must Keep"},
        {"name": "Penn State University", "city": "University Park, PA", "type": "4-Year", "leads": 187, "conversions": 29, "events": 8, "priority": "Must Keep"},
        {"name": "Ohio State University", "city": "Columbus, OH", "type": "4-Year", "leads": 176, "conversions": 24, "events": 7, "priority": "Must Win"},
        {"name": "Florida State University", "city": "Tallahassee, FL", "type": "4-Year", "leads": 165, "conversions": 27, "events": 10, "priority": "Must Keep"},
        {"name": "Georgia Institute of Technology", "city": "Atlanta, GA", "type": "4-Year", "leads": 154, "conversions": 25, "events": 6, "priority": "Must Win"},
        {"name": "University of Florida", "city": "Gainesville, FL", "type": "4-Year", "leads": 143, "conversions": 22, "events": 8, "priority": "Must Keep"},
        {"name": "Texas A&M University", "city": "College Station, TX", "type": "4-Year", "leads": 139, "conversions": 28, "events": 11, "priority": "Must Keep"},
        {"name": "University of Georgia", "city": "Athens, GA", "type": "4-Year", "leads": 128, "conversions": 19, "events": 7, "priority": "Opportunity"},
        {"name": "Clemson University", "city": "Clemson, SC", "type": "4-Year", "leads": 117, "conversions": 20, "events": 9, "priority": "Must Keep"},
        {"name": "San Diego State University", "city": "San Diego, CA", "type": "4-Year", "leads": 112, "conversions": 16, "events": 5, "priority": "Opportunity"},
        {"name": "Virginia Tech", "city": "Blacksburg, VA", "type": "4-Year", "leads": 108, "conversions": 18, "events": 6, "priority": "Must Keep"},
        {"name": "University of South Carolina", "city": "Columbia, SC", "type": "4-Year", "leads": 98, "conversions": 15, "events": 7, "priority": "Opportunity"},
        {"name": "Auburn University", "city": "Auburn, AL", "type": "4-Year", "leads": 94, "conversions": 17, "events": 8, "priority": "Must Keep"},
        {"name": "Oklahoma State University", "city": "Stillwater, OK", "type": "4-Year", "leads": 87, "conversions": 14, "events": 6, "priority": "Opportunity"},
    ]

    # Add conversion rates
    for school in schools:
        school["conversion_rate"] = round((school["conversions"] / school["leads"]) * 100, 1)
        school["cost_per_lead"] = round(random.uniform(45, 125), 2)

    return {
        "status": "ok",
        "count": len(schools),
        "schools": schools[:limit]
    }


@app.get("/api/v2/analytics/segments")
def get_segment_performance():
    """Get performance metrics by segment (D3AE, F3A, demographics)."""
    # Mock segment data (in production, aggregate from segment_profiles + leads)
    segments = [
        {
            "segment_name": "High Propensity Males 18-24",
            "segment_code": "HP_M_18_24",
            "size": 12500,
            "leads_generated": 3450,
            "penetration_rate": 27.6,
            "avg_propensity": 8.2,
            "conversions": 518,
            "priority": "Must Win"
        },
        {
            "segment_name": "College-Bound Females 18-21",
            "segment_code": "CB_F_18_21",
            "size": 8900,
            "leads_generated": 2180,
            "penetration_rate": 24.5,
            "avg_propensity": 7.5,
            "conversions": 327,
            "priority": "Must Keep"
        },
        {
            "segment_name": "Working Adults 25-29",
            "segment_code": "WA_MF_25_29",
            "size": 15200,
            "leads_generated": 2890,
            "penetration_rate": 19.0,
            "avg_propensity": 6.8,
            "conversions": 376,
            "priority": "Opportunity"
        },
        {
            "segment_name": "High School Seniors",
            "segment_code": "HS_SENIOR",
            "size": 11000,
            "leads_generated": 2650,
            "penetration_rate": 24.1,
            "avg_propensity": 7.2,
            "conversions": 398,
            "priority": "Must Keep"
        },
        {
            "segment_name": "Military Family Influencers",
            "segment_code": "MIL_FAM",
            "size": 6400,
            "leads_generated": 1890,
            "penetration_rate": 29.5,
            "avg_propensity": 8.5,
            "conversions": 312,
            "priority": "Must Win"
        },
        {
            "segment_name": "STEM Interest Males 18-24",
            "segment_code": "STEM_M_18_24",
            "size": 9800,
            "leads_generated": 2320,
            "penetration_rate": 23.7,
            "avg_propensity": 7.8,
            "conversions": 348,
            "priority": "Must Keep"
        },
        {
            "segment_name": "Career Explorers 22-26",
            "segment_code": "CE_MF_22_26",
            "size": 13500,
            "leads_generated": 2450,
            "penetration_rate": 18.1,
            "avg_propensity": 6.5,
            "conversions": 294,
            "priority": "Opportunity"
        },
    ]

    # Calculate remaining potential
    for seg in segments:
        seg["remaining_potential"] = seg["size"] - seg["leads_generated"]
        seg["conversion_rate"] = round((seg["conversions"] / seg["leads_generated"]) * 100, 1)

    return {
        "status": "ok",
        "count": len(segments),
        "segments": segments
    }


@app.get("/api/v2/analytics/contracts")
def get_contract_metrics():
    """Get contract achievement metrics vs. mission goals."""
    # Mock contract data (in production, from contracts table + mission goals)
    current_fy = 2025

    metrics = {
        "fiscal_year": current_fy,
        "mission_goal": 62500,
        "contracts_achieved": 48930,
        "remaining": 13570,
        "percent_complete": 78.3,
        "days_remaining": 314,
        "daily_rate_needed": 43.2,
        "current_daily_rate": 41.8,
        "on_track": True,
        "by_month": [
            {"month": "Oct 2024", "goal": 5000, "achieved": 4875, "variance": -125},
            {"month": "Nov 2024", "goal": 5200, "achieved": 5340, "variance": 140},
            {"month": "Dec 2024", "goal": 4800, "achieved": 4650, "variance": -150},
            {"month": "Jan 2025", "goal": 5500, "achieved": 5720, "variance": 220},
            {"month": "Feb 2025", "goal": 5300, "achieved": 5180, "variance": -120},
            {"month": "Mar 2025", "goal": 5400, "achieved": 5560, "variance": 160},
            {"month": "Apr 2025", "goal": 5600, "achieved": 5780, "variance": 180},
            {"month": "May 2025", "goal": 5700, "achieved": 5920, "variance": 220},
            {"month": "Jun 2025", "goal": 5800, "achieved": 5905, "variance": 105},
            {"month": "Jul 2025", "goal": 6000, "achieved": 0, "variance": -6000},
            {"month": "Aug 2025", "goal": 6100, "achieved": 0, "variance": -6100},
            {"month": "Sep 2025", "goal": 6100, "achieved": 0, "variance": -6100},
        ],
        "by_component": [
            {"component": "Regular Army (RA)", "goal": 50000, "achieved": 39144, "percent": 78.3},
            {"component": "Army Reserve (AR)", "goal": 7500, "achieved": 5896, "percent": 78.6},
            {"component": "Army National Guard (ARNG)", "goal": 5000, "achieved": 3890, "percent": 77.8},
        ]
    }

    return {
        "status": "ok",
        "metrics": metrics
    }


@app.get("/api/v2/analytics/overview")
def get_analytics_overview():
    """Get comprehensive analytics overview for dashboard."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Inspect columns to choose score field dynamically
    cur.execute("PRAGMA table_info(leads)")
    lead_cols = {row[1] for row in cur.fetchall()}
    score_col = 'propensity_score' if 'propensity_score' in lead_cols else ('score' if 'score' in lead_cols else None)

    # Get lead statistics
    if score_col:
        cur.execute(f"SELECT COUNT(*), AVG({score_col}) FROM leads")
    else:
        cur.execute("SELECT COUNT(*), NULL FROM leads")
    lead_row = cur.fetchone()
    total_leads = lead_row[0] or 0
    avg_lead_score = (lead_row[1] or 0)

    # Get event statistics
    cur.execute("SELECT COUNT(*) FROM events")
    total_events = cur.fetchone()[0] or 0

    # Get project statistics
    cur.execute("SELECT COUNT(*) FROM projects WHERE status IN ('in_progress', 'at_risk')")
    active_projects = cur.fetchone()[0] or 0

    conn.close()

    return {
        "status": "ok",
        "overview": {
            "total_leads": total_leads,
            "avg_lead_score": round(avg_lead_score, 2),
            "total_events": total_events,
            "active_projects": active_projects,
            "last_updated": datetime.now().isoformat()
        }
    }


# --- PROJECT MANAGEMENT ENDPOINTS ---

@app.get("/api/v2/projects")
def get_all_projects(status: Optional[str] = None):
    """Get all projects with optional status filter."""
    conn = get_db_conn()
    cur = conn.cursor()

    query = "SELECT * FROM projects WHERE is_archived = 0"
    params = []

    if status:
        query += " AND status = ?"
        params.append(status)

    query += " ORDER BY created_at DESC"

    cur.execute(query, params)
    rows = cur.fetchall()
    conn.close()

    projects = []
    for row in rows:
        projects.append(dict(row))

    return {
        "status": "ok",
        "count": len(projects),
        "projects": projects
    }


@app.get("/api/v2/projects/{project_id}")
def get_project_detail(project_id: str):
    """Get detailed project information including tasks, milestones, and budget."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Get project details
    cur.execute("SELECT * FROM projects WHERE project_id = ?", (project_id,))
    project_row = cur.fetchone()

    if not project_row:
        conn.close()
        raise HTTPException(status_code=404, detail="Project not found")

    project = dict(project_row)

    # Get tasks
    cur.execute("SELECT * FROM tasks WHERE project_id = ? ORDER BY due_date", (project_id,))
    tasks = [dict(row) for row in cur.fetchall()]

    # Get milestones
    cur.execute("SELECT * FROM milestones WHERE project_id = ? ORDER BY target_date", (project_id,))
    milestones = [dict(row) for row in cur.fetchall()]

    # Calculate task statistics
    total_tasks = len(tasks)
    completed_tasks = len([t for t in tasks if t['status'] == 'completed'])
    in_progress_tasks = len([t for t in tasks if t['status'] == 'in_progress'])
    blocked_tasks = len([t for t in tasks if t['status'] == 'blocked'])

    # Calculate budget statistics
    funding_amount = project.get('funding_amount', 0) or 0
    spent_amount = project.get('spent_amount', 0) or 0
    remaining_budget = funding_amount - spent_amount
    budget_utilized = (spent_amount / funding_amount * 100) if funding_amount > 0 else 0

    conn.close()

    return {
        "status": "ok",
        "project": project,
        "tasks": tasks,
        "milestones": milestones,
        "statistics": {
            "total_tasks": total_tasks,
            "completed_tasks": completed_tasks,
            "in_progress_tasks": in_progress_tasks,
            "blocked_tasks": blocked_tasks,
            "completion_rate": round((completed_tasks / total_tasks * 100) if total_tasks > 0 else 0, 1),
            "funding_amount": funding_amount,
            "spent_amount": spent_amount,
            "remaining_budget": remaining_budget,
            "budget_utilized": round(budget_utilized, 1)
        }
    }


@app.put("/api/v2/projects/{project_id}")
def update_project(project_id: str, updates: Dict[str, Any]):
    """Update project details including status, budget, progress."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Build dynamic update query
    set_parts = []
    values = []

    for key, value in updates.items():
        set_parts.append(f"{key} = ?")
        values.append(value)

    set_parts.append("updated_at = ?")
    values.append(datetime.now().isoformat())
    values.append(project_id)

    query = f"UPDATE projects SET {', '.join(set_parts)} WHERE project_id = ?"
    cur.execute(query, values)
    conn.commit()
    conn.close()

    return {"status": "ok", "message": "Project updated successfully"}


@app.post("/api/v2/projects/{project_id}/milestones")
def create_milestone(project_id: str, milestone: Dict[str, Any]):
    """Create a project milestone."""
    import uuid
    conn = get_db_conn()
    cur = conn.cursor()

    milestone_id = f"ms_{uuid.uuid4().hex[:12]}"
    now = datetime.now().isoformat()

    cur.execute(
        """
        INSERT INTO milestones (milestone_id, project_id, name, target_date, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?)
        """,
        (milestone_id, project_id, milestone.get('name'), milestone.get('target_date'), now, now)
    )
    conn.commit()
    conn.close()

    return {"status": "ok", "milestone_id": milestone_id}


@app.put("/api/v2/projects/{project_id}/milestones/{milestone_id}")
def update_milestone(project_id: str, milestone_id: str, updates: Dict[str, Any]):
    """Update milestone (e.g., mark as completed)."""
    conn = get_db_conn()
    cur = conn.cursor()

    set_parts = []
    values = []

    for key, value in updates.items():
        set_parts.append(f"{key} = ?")
        values.append(value)

    set_parts.append("updated_at = ?")
    values.append(datetime.now().isoformat())
    values.append(milestone_id)

    query = f"UPDATE milestones SET {', '.join(set_parts)} WHERE milestone_id = ?"
    cur.execute(query, values)
    conn.commit()
    conn.close()

    return {"status": "ok", "message": "Milestone updated"}


@app.get("/api/v2/projects/{project_id}/tasks")
def get_project_tasks(project_id: str, status: Optional[str] = None):
    """Get tasks for a specific project."""
    conn = get_db_conn()
    cur = conn.cursor()

    query = "SELECT * FROM tasks WHERE project_id = ?"
    params = [project_id]

    if status:
        query += " AND status = ?"
        params.append(status)

    query += " ORDER BY due_date"

    cur.execute(query, params)
    tasks = [dict(row) for row in cur.fetchall()]
    conn.close()

    return {
        "status": "ok",
        "project_id": project_id,
        "count": len(tasks),
        "tasks": tasks
    }


@app.post("/api/v2/projects/{project_id}/budget")
def update_project_budget(project_id: str, budget_update: Dict[str, Any]):
    """Update project budget/spending."""
    conn = get_db_conn()
    cur = conn.cursor()

    spent_amount = budget_update.get('spent_amount')
    funding_amount = budget_update.get('funding_amount')

    updates = []
    values = []

    if spent_amount is not None:
        updates.append("spent_amount = ?")
        values.append(spent_amount)

    if funding_amount is not None:
        updates.append("funding_amount = ?")
        values.append(funding_amount)

    if updates:
        updates.append("updated_at = ?")
        values.append(datetime.now().isoformat())
        values.append(project_id)

        query = f"UPDATE projects SET {', '.join(updates)} WHERE project_id = ?"
        cur.execute(query, values)
        conn.commit()

    conn.close()

    return {"status": "ok", "message": "Budget updated"}


@app.get("/api/v2/projects/dashboard/summary")
def get_project_dashboard_summary():
    """Get project management dashboard summary with KPIs."""
    conn = get_db_conn()
    cur = conn.cursor()

    # Overall statistics
    cur.execute("SELECT COUNT(*) FROM projects WHERE is_archived = 0")
    total_projects = cur.fetchone()[0] or 0

    cur.execute("SELECT COUNT(*) FROM projects WHERE status = 'in_progress' AND is_archived = 0")
    active_projects = cur.fetchone()[0] or 0

    cur.execute("SELECT COUNT(*) FROM projects WHERE status = 'completed' AND is_archived = 0")
    completed_projects = cur.fetchone()[0] or 0

    cur.execute("SELECT COUNT(*) FROM projects WHERE status = 'at_risk' AND is_archived = 0")
    at_risk_projects = cur.fetchone()[0] or 0

    # Task statistics
    cur.execute("SELECT COUNT(*) FROM tasks")
    total_tasks = cur.fetchone()[0] or 0

    cur.execute("SELECT COUNT(*) FROM tasks WHERE status = 'completed'")
    completed_tasks = cur.fetchone()[0] or 0

    cur.execute("SELECT COUNT(*) FROM tasks WHERE status = 'blocked'")
    blocked_tasks = cur.fetchone()[0] or 0

    # Budget statistics
    cur.execute("SELECT SUM(funding_amount), SUM(spent_amount) FROM projects WHERE is_archived = 0")
    budget_row = cur.fetchone()
    total_budget = budget_row[0] or 0
    total_spent = budget_row[1] or 0

    # Recent projects
    cur.execute("""
        SELECT project_id, name, status, percent_complete, funding_amount, spent_amount, start_date, target_date
        FROM projects
        WHERE is_archived = 0
        ORDER BY created_at DESC
        LIMIT 5
    """)
    recent_projects = [dict(row) for row in cur.fetchall()]

    # Projects by status
    cur.execute("""
        SELECT status, COUNT(*) as count
        FROM projects
        WHERE is_archived = 0
        GROUP BY status
    """)
    status_distribution = [{"status": row[0], "count": row[1]} for row in cur.fetchall()]

    conn.close()

    return {
        "status": "ok",
        "summary": {
            "total_projects": total_projects,
            "active_projects": active_projects,
            "completed_projects": completed_projects,
            "at_risk_projects": at_risk_projects,
            "total_tasks": total_tasks,
            "completed_tasks": completed_tasks,
            "blocked_tasks": blocked_tasks,
            "task_completion_rate": round((completed_tasks / total_tasks * 100) if total_tasks > 0 else 0, 1),
            "total_budget": total_budget,
            "total_spent": total_spent,
            "budget_remaining": total_budget - total_spent,
            "budget_utilization": round((total_spent / total_budget * 100) if total_budget > 0 else 0, 1)
        },
        "recent_projects": recent_projects,
        "status_distribution": status_distribution
    }


# ============================================================================
# EXPORT ENDPOINTS
# ============================================================================

@app.get("/api/v2/export/projects")
def export_projects(rsid: str = None, status: str = None, format: str = "csv"):
    """Export projects data as CSV or JSON"""
    from utils.data_export import DataExporter
    import io
    from fastapi.responses import StreamingResponse

    exporter = DataExporter(DB_FILE)

    if format == "json":
        data = exporter.export_projects(rsid=rsid, status=status, format='json')
        return JSONResponse(content={"data": data})
    else:
        # CSV export
        csv_data = exporter.export_projects(rsid=rsid, status=status, format='csv')

        # Create filename
        filename = f"taaip_projects"
        if rsid:
            filename += f"_{rsid}"
        if status:
            filename += f"_{status}"
        filename += f"_{datetime.now().strftime('%Y%m%d')}.csv"
        return StreamingResponse(
            io.StringIO(csv_data),
            media_type="text/csv",
            headers={"Content-Disposition": f"attachment; filename={filename}"}
        )


@app.get("/api/v2/export/tasks")
def export_tasks(project_id: str = None, status: str = None, assigned_to: str = None, format: str = "csv"):
    """Export tasks data as CSV or JSON"""
    from utils.data_export import DataExporter
    import io
    from fastapi.responses import StreamingResponse

    exporter = DataExporter(DB_FILE)

    if format == "json":
        data = exporter.export_tasks(project_id=project_id, status=status, assigned_to=assigned_to, format='json')
        return JSONResponse(content={"data": data})
    else:
        csv_data = exporter.export_tasks(project_id=project_id, status=status, assigned_to=assigned_to, format='csv')

        filename = f"taaip_tasks_{datetime.now().strftime('%Y%m%d')}.csv"

        return StreamingResponse(
            io.StringIO(csv_data),
            media_type="text/csv",
            headers={"Content-Disposition": f"attachment; filename={filename}"}
        )


@app.get("/api/v2/export/budget-analysis")
def export_budget_analysis(rsid: str = None, format: str = "csv"):
    """Export budget analysis as CSV or JSON"""
    from utils.data_export import DataExporter
    import io
    from fastapi.responses import StreamingResponse

    exporter = DataExporter(DB_FILE)

    if format == "json":
        data = exporter.export_budget_analysis(rsid=rsid, format='json')
        return JSONResponse(content={"data": data})
    else:
        csv_data = exporter.export_budget_analysis(rsid=rsid, format='csv')

        filename = f"taaip_budget_analysis"
        if rsid:
            filename += f"_{rsid}"
        filename += f"_{datetime.now().strftime('%Y%m%d')}.csv"

        return StreamingResponse(
            io.StringIO(csv_data),
            media_type="text/csv",
            headers={"Content-Disposition": f"attachment; filename={filename}"}
        )


@app.get("/api/v2/export/dashboard-summary")
def export_dashboard_summary(rsid: str = None, format: str = "csv"):
    """Export complete dashboard summary"""
    from utils.data_export import DataExporter
    import io
    from fastapi.responses import StreamingResponse

    exporter = DataExporter(DB_FILE)
    if format == "json":
        data = exporter.export_dashboard_summary(rsid=rsid, format='json')
        return JSONResponse(content={"data": data})
    else:
        csv_data = exporter.export_dashboard_summary(rsid=rsid, format='csv')

        filename = f"taaip_dashboard_summary"
        if rsid:
            filename += f"_{rsid}"
        filename += f"_{datetime.now().strftime('%Y%m%d')}.csv"

        return StreamingResponse(
            io.StringIO(csv_data),
            media_type="text/csv",
            headers={"Content-Disposition": f"attachment; filename={filename}"}
        )


# ============================================================================
# RECRUITING FUNNEL ENDPOINTS
# ============================================================================

@app.get("/api/v2/recruiting-funnel/metrics")
async def get_recruiting_funnel_metrics(fiscal_year: Optional[int] = None):
    """
    Get recruiting funnel metrics with conversion rates and flash-to-bang data.
    Army Recruiting Process: Lead → Prospect → Appointment Made → Appointment Conducted → Test → Test Pass → Enlistment → Ship

    Query Parameters:
    - fiscal_year: Optional fiscal year filter (e.g., 2025 for FY2025)

    Returns:
    - Funnel stage counts for all stages
    - Conversion rates between stages
    - Flash-to-bang metrics (avg days between stages)
    - Test pass rates and appointment show rates
    - Loss analysis (loss rate, top loss reason)
    """
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        # Inspect leads table columns to adapt to schema differences
        cursor.execute("PRAGMA table_info(leads)")
        lead_cols = {row[1] for row in cursor.fetchall()}
        def has(col: str) -> bool:
            return col in lead_cols

        # Choose stage column dynamically
        # Choose stage column dynamically (include 'status' as last resort)
        stage_col = 'current_stage' if has('current_stage') else ('stage' if has('stage') else ('status' if has('status') else None))
        if not stage_col:
            # Return empty metrics rather than error to avoid blank UI
            empty = {
                "funnel_counts": {"leads":0,"prospects":0,"appointments_made":0,"appointments_conducted":0,"tests":0,"test_passes":0,"enlistments":0,"ships":0,"losses":0,"tota
                "conversion_rates": {"lead_to_prospect":0,"prospect_to_appointment":0,"appointment_made_to_conducted":0,"appointment_to_test":0,"test_to_pass":0,"test_pas
                "flash_to_bang": {"avg_lead_to_prospect_days":0,"avg_prospect_to_appointment_days":0,"avg_appointment_to_test_days":0,"avg_test_to_enlistment_days":0,"a
                "appointment_metrics": {"no_show_rate":0},
                "loss_analysis": {"total_losses":0,"loss_rate":0,"top_loss_reason":"None"}
            }
            return JSONResponse(content={"status":"ok","metrics":empty})

        # Build WHERE clause for fiscal year filtering
        where_clause = ""
        params: list = []
        if fiscal_year and has('fiscal_year'):
            where_clause = " WHERE fiscal_year = ?"
            params.append(fiscal_year)

        # Get stage counts
        query = f"""
            SELECT
                {stage_col} as stage,
                COUNT(*) as count
            FROM leads
            {where_clause}
            GROUP BY {stage_col}
        """
        cursor.execute(query, params)
        stage_counts = {row['stage']: row['count'] for row in cursor.fetchall()}

        # Extract counts for each stage (default to 0 if stage doesn't exist)
        leads_count = stage_counts.get('lead', 0)
        prospects_count = stage_counts.get('prospect', 0)
        appointments_made_count = stage_counts.get('appointment_made', 0)
        appointments_conducted_count = stage_counts.get('appointment_conducted', 0)
        tests_count = stage_counts.get('test', 0)
        test_passes_count = stage_counts.get('test_pass', 0)
        enlistments_count = stage_counts.get('enlistment', 0)
        ships_count = stage_counts.get('ship', 0)
        losses_count = stage_counts.get('loss', 0)

        # Calculate total active leads (not including losses)
        total_active = (leads_count + prospects_count + appointments_made_count +
                       appointments_conducted_count + tests_count + test_passes_count +
                       enlistments_count + ships_count)
        total_leads = total_active + losses_count

        # Calculate conversion rates (avoid division by zero)
        def safe_rate(numerator, denominator):
            return round((numerator / denominator) * 100, 2) if denominator > 0 else 0

        lead_to_prospect_rate = safe_rate(prospects_count, leads_count)
        prospect_to_appointment_rate = safe_rate(appointments_made_count, prospects_count)
        appointment_made_to_conducted_rate = safe_rate(appointments_conducted_count, appointments_made_count)
        appointment_to_test_rate = safe_rate(tests_count, appointments_conducted_count)
        test_to_pass_rate = safe_rate(test_passes_count, tests_count)
        test_pass_to_enlistment_rate = safe_rate(enlistments_count, test_passes_count)
        enlistment_to_ship_rate = safe_rate(ships_count, enlistments_count)
        overall_conversion_rate = safe_rate(enlistments_count, total_leads)

        # Calculate flash-to-bang metrics (average days between stages)
        # Flash-to-bang supports only if required columns exist
        flash_supported = all(has(c) for c in [
            'prospect_date','lead_date','appointment_made_date','appointment_conducted_date',
            'test_date','test_pass_date','enlistment_date','ship_date','dep_length_days'
        ])
        avg_lead_to_prospect_days = avg_prospect_to_appointment_days = 0
        avg_appointment_to_test_days = avg_test_to_enlistment_days = 0
        avg_lead_to_enlistment_days = avg_enlistment_to_ship_days = 0
        avg_dep_length_days = 0
        if flash_supported:
            flash_to_bang_query = f"""
                SELECT
                    AVG(JULIANDAY(prospect_date) - JULIANDAY(lead_date)) as avg_lead_to_prospect,
                    AVG(JULIANDAY(appointment_made_date) - JULIANDAY(prospect_date)) as avg_prospect_to_appointment,
                    AVG(JULIANDAY(test_date) - JULIANDAY(appointment_conducted_date)) as avg_appointment_to_test,
                    AVG(JULIANDAY(enlistment_date) - JULIANDAY(test_pass_date)) as avg_test_to_enlistment,
                    AVG(JULIANDAY(enlistment_date) - JULIANDAY(lead_date)) as avg_lead_to_enlistment,
                    AVG(JULIANDAY(ship_date) - JULIANDAY(enlistment_date)) as avg_enlistment_to_ship,
                    AVG(dep_length_days) as avg_dep_length
                FROM leads
                {where_clause}
                {"AND" if where_clause else "WHERE"} enlistment_date IS NOT NULL
            """
            cursor.execute(flash_to_bang_query, params)
            flash_row = cursor.fetchone()
            if flash_row:
                avg_lead_to_prospect_days = round(flash_row['avg_lead_to_prospect'], 1) if flash_row['avg_lead_to_prospect'] else 0
                avg_prospect_to_appointment_days = round(flash_row['avg_prospect_to_appointment'], 1) if flash_row['avg_prospect_to_appointment'] else 0
                avg_appointment_to_test_days = round(flash_row['avg_appointment_to_test'], 1) if flash_row['avg_appointment_to_test'] else 0
                avg_test_to_enlistment_days = round(flash_row['avg_test_to_enlistment'], 1) if flash_row['avg_test_to_enlistment'] else 0
                avg_lead_to_enlistment_days = round(flash_row['avg_lead_to_enlistment'], 1) if flash_row['avg_lead_to_enlistment'] else 0
                avg_enlistment_to_ship_days = round(flash_row['avg_enlistment_to_ship'], 1) if flash_row['avg_enlistment_to_ship'] else 0
                avg_dep_length_days = round(flash_row['avg_dep_length'], 1) if flash_row['avg_dep_length'] else 0

        # Calculate appointment no-show rate
        # Appointment no-show rate only if columns exist
        appointment_no_show_rate = 0
        if has('appointment_made_date') and has('appointment_no_show'):
            no_show_query = f"""
                SELECT
                    COUNT(*) as total_appointments,
                    SUM(CASE WHEN appointment_no_show = 1 THEN 1 ELSE 0 END) as no_shows
                FROM leads
                {where_clause}
                {"AND" if where_clause else "WHERE"} appointment_made_date IS NOT NULL
            """
            cursor.execute(no_show_query, params)
            no_show_row = cursor.fetchone()
            appointment_no_show_rate = safe_rate(no_show_row['no_shows'] if no_show_row else 0,
                                          no_show_row['total_appointments'] if no_show_row else 0)

        # Calculate loss metrics
        loss_rate = safe_rate(losses_count, total_leads)

        # Get top loss reason
        top_loss_reason = "None"
        if has('loss_reason'):
            loss_reason_query = f"""
                SELECT
                    loss_reason,
                    COUNT(*) as count
                FROM leads
                {where_clause}
                {"AND" if where_clause else "WHERE"} loss_reason IS NOT NULL
                GROUP BY loss_reason
                ORDER BY count DESC
                LIMIT 1
            """
            cursor.execute(loss_reason_query, params)
            loss_reason_row = cursor.fetchone()
            top_loss_reason = loss_reason_row['loss_reason'] if loss_reason_row else "None"

        conn.close()

        # Build response with new Army recruiting funnel stages
        metrics = {
            "funnel_counts": {
                "leads": leads_count,
                "prospects": prospects_count,
                "appointments_made": appointments_made_count,
                "appointments_conducted": appointments_conducted_count,
                "tests": tests_count,
                "test_passes": test_passes_count,
                "enlistments": enlistments_count,
                "ships": ships_count,
                "losses": losses_count,
                "total_active": total_active,
                "total_leads": total_leads
            },
            "conversion_rates": {
                "lead_to_prospect": lead_to_prospect_rate,
                "prospect_to_appointment": prospect_to_appointment_rate,
                "appointment_made_to_conducted": appointment_made_to_conducted_rate,
                "appointment_to_test": appointment_to_test_rate,
                "test_to_pass": test_to_pass_rate,
                "test_pass_to_enlistment": test_pass_to_enlistment_rate,
                "enlistment_to_ship": enlistment_to_ship_rate,
                "overall_conversion": overall_conversion_rate
            },
            "flash_to_bang": {
                "avg_lead_to_prospect_days": avg_lead_to_prospect_days,
                "avg_prospect_to_appointment_days": avg_prospect_to_appointment_days,
                "avg_appointment_to_test_days": avg_appointment_to_test_days,
                "avg_test_to_enlistment_days": avg_test_to_enlistment_days,
                "avg_lead_to_enlistment_days": avg_lead_to_enlistment_days,
                "avg_enlistment_to_ship_days": avg_enlistment_to_ship_days,
                "avg_dep_length_days": avg_dep_length_days
            },
            "appointment_metrics": {
                "no_show_rate": appointment_no_show_rate
            },
            "loss_analysis": {
                "total_losses": losses_count,
                "loss_rate": loss_rate,
                "top_loss_reason": top_loss_reason
            }
        }

        return JSONResponse(content={"status": "ok", "metrics": metrics})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


# ============================================================================
# MARKET POTENTIAL & DOD COMPARISON ENDPOINTS
# ============================================================================

@app.get("/api/v2/market-potential")
async def get_market_potential(
    geographic_level: Optional[str] = None,
    geographic_id: Optional[str] = None,
    fiscal_year: Optional[int] = None,
    quarter: Optional[str] = None,
    rsid: Optional[str] = None
):
    """
    Get market potential data with Army vs DOD branch comparisons.

    Query Parameters:
    - geographic_level: Filter by level (zipcode, cbsa, rsid)
    - geographic_id: Specific geographic area ID
    - fiscal_year: Fiscal year filter
    - quarter: Quarter filter (Q1, Q2, Q3, Q4)
    - rsid: RSID filter for station-level data

    Returns:
    - Market potential by geography
    - Army contacted vs remaining
    - DOD branch comparisons
    - Market share analysis
    """
    try:
```

```
conn = sqlite3.connect(DB_FILE)
conn.row_factory = sqlite3.Row
cursor = conn.cursor()

# Build WHERE clause
where_conditions = []
params = []

if geographic_level:
    where_conditions.append("geographic_level = ?")
    params.append(geographic_level)

if geographic_id:
    where_conditions.append("geographic_id = ?")
    params.append(geographic_id)

if fiscal_year:
    where_conditions.append("fiscal_year = ?")
    params.append(fiscal_year)

if quarter:
    where_conditions.append("quarter = ?")
    params.append(quarter)

if rsid:
    # Match by brigade, battalion, or full RSID
    where_conditions.append("(brigade = ? OR battalion = ? OR station = ?)")
    params.extend([rsid, rsid, rsid])

where_clause = " WHERE " + " AND ".join(where_conditions) if where_conditions else ""

query = f"""
    SELECT
        geographic_level,
        geographic_id,
        geographic_name,
        brigade,
        battalion,
        qualified_population,
        army_contacted,
        army_remaining_potential,
        army_market_share,
        navy_contacted,
        navy_remaining_potential,
        navy_market_share,
        air_force_contacted,
        air_force_remaining_potential,
        air_force_market_share,
        marines_contacted,
        marines_remaining_potential,
        marines_market_share,
        space_force_contacted,
        space_force_remaining_potential,
        space_force_market_share,
        coast_guard_contacted,
        coast_guard_remaining_potential,
        coast_guard_market_share,
        total_dod_contacted,
        total_dod_remaining,
        fiscal_year,
        quarter
    FROM market_potential
    {where_clause}
    ORDER BY fiscal_year DESC, quarter DESC, geographic_name ASC
    LIMIT 100
"""
cursor.execute(query, params)
```

```
        results = []

        for row in cursor.fetchall():
            results.append({
                "geographic_level": row["geographic_level"],
                "geographic_id": row["geographic_id"],
                "geographic_name": row["geographic_name"],
                "brigade": row["brigade"],
                "battalion": row["battalion"],
                "qualified_population": row["qualified_population"],
                "army": {
                    "contacted": row["army_contacted"],
                    "remaining": row["army_remaining_potential"],
                    "market_share": round(row["army_market_share"], 2) if row["army_market_share"] else 0
                },
                "navy": {
                    "contacted": row["navy_contacted"],
                    "remaining": row["navy_remaining_potential"],
                    "market_share": round(row["navy_market_share"], 2) if row["navy_market_share"] else 0
                },
                "air_force": {
                    "contacted": row["air_force_contacted"],
                    "remaining": row["air_force_remaining_potential"],
                    "market_share": round(row["air_force_market_share"], 2) if row["air_force_market_share"] else 0
                },
                "marines": {
                    "contacted": row["marines_contacted"],
                    "remaining": row["marines_remaining_potential"],
                    "market_share": round(row["marines_market_share"], 2) if row["marines_market_share"] else 0
                },
                "space_force": {
                    "contacted": row["space_force_contacted"],
                    "remaining": row["space_force_remaining_potential"],
                    "market_share": round(row["space_force_market_share"], 2) if row["space_force_market_share"] else 0
                },
                "coast_guard": {
                    "contacted": row["coast_guard_contacted"],
                    "remaining": row["coast_guard_remaining_potential"],
                    "market_share": round(row["coast_guard_market_share"], 2) if row["coast_guard_market_share"] else 0
                },
                "total_dod": {
                    "contacted": row["total_dod_contacted"],
                    "remaining": row["total_dod_remaining"]
                },
                "fiscal_year": row["fiscal_year"],
                "quarter": row["quarter"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/dod-comparison")
async def get_dod_branch_comparison(
    branch: Optional[str] = None,
    geographic_level: Optional[str] = None,
    geographic_id: Optional[str] = None,
    fiscal_year: Optional[int] = None,
    quarter: Optional[str] = None
):
```

```
"""
Get DOD branch comparison data for competitive analysis.

Query Parameters:
- branch: Filter by specific branch (Army, Navy, Air Force, Marines, Space Force, Coast Guard)
- geographic_level: Filter by level (national, state, cbsa, zipcode)
- geographic_id: Specific geographic area ID
- fiscal_year: Fiscal year filter
- quarter: Quarter filter

Returns:
- Branch performance metrics
- Recruiter productivity
- Conversion rates
- Market penetration
"""
try:
    conn = sqlite3.connect(DB_FILE)
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    where_conditions = []
    params = []

    if branch:
        where_conditions.append("branch = ?")
        params.append(branch)

    if geographic_level:
        where_conditions.append("geographic_level = ?")
        params.append(geographic_level)

    if geographic_id:
        where_conditions.append("geographic_id = ?")
        params.append(geographic_id)

    if fiscal_year:
        where_conditions.append("fiscal_year = ?")
        params.append(fiscal_year)

    if quarter:
        where_conditions.append("quarter = ?")
        params.append(quarter)

    where_clause = " WHERE " + " AND ".join(where_conditions) if where_conditions else ""

    query = f"""
        SELECT
            branch,
            geographic_level,
            geographic_id,
            geographic_name,
            total_recruiters,
            total_leads,
            total_contracts,
            total_ships,
            lead_to_contract_rate,
            contract_to_ship_rate,
            overall_efficiency_score,
            contracts_per_recruiter,
            fiscal_year,
            quarter
        FROM dod_branch_comparison
        {where_clause}
        ORDER BY fiscal_year DESC, quarter DESC, total_contracts DESC
        LIMIT 200
    """
```

```
        cursor.execute(query, params)
        results = []

        for row in cursor.fetchall():
            results.append({
                "branch": row["branch"],
                "geographic_level": row["geographic_level"],
                "geographic_id": row["geographic_id"],
                "geographic_name": row["geographic_name"],
                "recruiters": row["total_recruiters"],
                "leads": row["total_leads"],
                "contracts": row["total_contracts"],
                "ships": row["total_ships"],
                "conversion_rates": {
                    "lead_to_contract": round(row["lead_to_contract_rate"] * 100, 2) if row["lead_to_contract_rate"] else 0,
                    "contract_to_ship": round(row["contract_to_ship_rate"] * 100, 2) if row["contract_to_ship_rate"] else 0
                },
                "efficiency_score": round(row["overall_efficiency_score"] * 100, 2) if row["overall_efficiency_score"] else 0,
                "productivity": {
                    "contracts_per_recruiter": round(row["contracts_per_recruiter"], 2) if row["contracts_per_recruiter"] else 0
                },
                "fiscal_year": row["fiscal_year"],
                "quarter": row["quarter"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/mission-analysis")
async def get_mission_analysis(
    analysis_level: Optional[str] = None,
    brigade: Optional[str] = None,
    battalion: Optional[str] = None,
    company: Optional[str] = None,
    station: Optional[str] = None,
    fiscal_year: Optional[int] = None,
    quarter: Optional[str] = None
):
    """
    Get mission analysis data by USAREC organizational hierarchy.

    Query Parameters:
    - analysis_level: Level of analysis (usarec, brigade, battalion, company, station)
    - brigade: Filter by brigade (e.g., 1BDE, 2BDE)
    - battalion: Filter by battalion (e.g., 1BDE-1BN)
    - company: Filter by company (e.g., 1BDE-1BN-1)
    - station: Filter by station (e.g., 1BDE-1BN-1-1)
    - fiscal_year: Fiscal year filter
    - quarter: Quarter filter

    Returns:
    - Mission goals vs actuals
    - Variance analysis
    - Production metrics
    - Efficiency metrics
    """
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
```

```
cursor = conn.cursor()

where_conditions = []
params = []

if analysis_level:
    where_conditions.append("analysis_level = ?")
    params.append(analysis_level)

if brigade:
    where_conditions.append("brigade = ?")
    params.append(brigade)

if battalion:
    where_conditions.append("battalion = ?")
    params.append(battalion)

if company:
    where_conditions.append("company = ?")
    params.append(company)

if station:
    where_conditions.append("station = ?")
    params.append(station)

if fiscal_year:
    where_conditions.append("fiscal_year = ?")
    params.append(fiscal_year)

if quarter:
    where_conditions.append("quarter = ?")
    params.append(quarter)

where_clause = " WHERE " + " AND ".join(where_conditions) if where_conditions else ""

query = f"""
    SELECT
        analysis_level,
        usarec_region,
        brigade,
        battalion,
        company,
        station,
        mission_goal,
        contracts_actual,
        contracts_variance,
        goal_attainment_pct,
        leads_generated,
        appointments_made,
        appointments_conducted,
        tests_administered,
        tests_passed,
        enlistments,
        ships,
        lead_to_enlistment_rate,
        appointment_show_rate,
        test_pass_rate,
        fiscal_year,
        quarter
    FROM mission_analysis
    {where_clause}
    ORDER BY fiscal_year DESC, quarter DESC, analysis_level, brigade, battalion
    LIMIT 100
"""

cursor.execute(query, params)
results = []
```

```
        for row in cursor.fetchall():
            results.append({
                "level": row["analysis_level"],
                "hierarchy": {
                    "usarec": row["usarec_region"],
                    "brigade": row["brigade"],
                    "battalion": row["battalion"],
                    "company": row["company"],
                    "station": row["station"]
                },
                "mission": {
                    "goal": row["mission_goal"],
                    "actual": row["contracts_actual"],
                    "variance": row["contracts_variance"],
                    "attainment_pct": round(row["goal_attainment_pct"], 2) if row["goal_attainment_pct"] else 0
                },
                "production": {
                    "leads": row["leads_generated"],
                    "appointments_made": row["appointments_made"],
                    "appointments_conducted": row["appointments_conducted"],
                    "tests_administered": row["tests_administered"],
                    "tests_passed": row["tests_passed"],
                    "enlistments": row["enlistments"],
                    "ships": row["ships"]
                },
                "efficiency": {
                    "lead_to_enlistment_rate": round(row["lead_to_enlistment_rate"] * 100, 2) if row["lead_to_enlistment_rate"] else 0,
                    "appointment_show_rate": round(row["appointment_show_rate"] * 100, 2) if row["appointment_show_rate"] else 0,
                    "test_pass_rate": round(row["test_pass_rate"] * 100, 2) if row["test_pass_rate"] else 0
                },
                "fiscal_year": row["fiscal_year"],
                "quarter": row["quarter"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


# ====================
# TWG (Targeting Working Group) ENDPOINTS
# ====================
def _ensure_twg_tables(conn):
        cur = conn.cursor()
        cur.execute(
                """
                CREATE TABLE IF NOT EXISTS twg_events (
                    event_id TEXT PRIMARY KEY,
                    name TEXT,
                    date TEXT,
                    location TEXT,
                    type TEXT,
                    target_audience TEXT,
                    expected_leads INTEGER,
                    budget INTEGER,
                    status TEXT,
                    priority TEXT
                )
                """
        )
        cur.execute(
                """
                CREATE TABLE IF NOT EXISTS twg_agenda_items (
                    id TEXT PRIMARY KEY,
                    meeting_id TEXT,
                    section TEXT,
                    presenter TEXT,
                    status TEXT,
                    notes TEXT,
                    order_index INTEGER
                )
                """
        )
        cur.execute(
                """
                CREATE TABLE IF NOT EXISTS twg_aar_reports (
                    event_id TEXT PRIMARY KEY,
                    event_name TEXT,
                    date TEXT,
                    due_date TEXT,
                    hours_since_event INTEGER,
                    status TEXT,
                    submitted_by TEXT,
                    content TEXT
                )
                """
        )
        cur.execute(
                """
                CREATE TABLE IF NOT EXISTS twg_budget (
                    fy INTEGER PRIMARY KEY,
                    total_budget INTEGER,
                    allocated INTEGER,
                    spent INTEGER,
                    remaining INTEGER,
                    q1 INTEGER,
                    q2 INTEGER,
                    q3 INTEGER,
                    q4 INTEGER
                )
                """
        )
        conn.commit()

def _get_conn_with_twg():
        conn = sqlite3.connect(DB_FILE)
        _ensure_twg_tables(conn)
        conn.row_factory = sqlite3.Row
        return conn

@app.get("/api/v2/twg/boards")
async def get_twg_boards(
    status: Optional[str] = None,
    review_type: Optional[str] = None,
    rsid: Optional[str] = None
):
    """Get all TWG review boards with optional filters"""
    try:
        conn = _get_conn_with_twg()
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        query = "SELECT * FROM twg_review_boards WHERE 1=1"
        params = []

        if status:
            query += " AND status = ?"
            params.append(status)
        if review_type:
            query += " AND review_type = ?"
            params.append(review_type)
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)

        query += " ORDER BY scheduled_date DESC"

        cursor.execute(query, params)
        rows = cursor.fetchall()

        results = []
        for row in rows:
            results.append({
                "board_id": row["board_id"],
                "name": row["name"],
                "project_id": row["project_id"],
                "event_id": row["event_id"],
                "review_type": row["review_type"],
                "status": row["status"],
                "scheduled_date": row["scheduled_date"],
                "completed_date": row["completed_date"],
                "facilitator": row["facilitator"],
                "attendees": json.loads(row["attendees"]) if row["attendees"] else [],
                "rsid": row["rsid"],
                "brigade": row["brigade"],
                "battalion": row["battalion"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/twg/analysis")
async def get_twg_analysis(board_id: Optional[str] = None, status: Optional[str] = None):
    """Get TWG analysis items"""
    try:
        conn = _get_conn_with_twg()
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        query = "SELECT * FROM twg_analysis_items WHERE 1=1"
        params = []

        if board_id:
            query += " AND board_id = ?"
            params.append(board_id)
        if status:
            query += " AND status = ?"
            params.append(status)

        query += " ORDER BY priority DESC, created_at DESC"

        cursor.execute(query, params)
        rows = cursor.fetchall()

        results = []
        for row in rows:
            results.append({
                "analysis_id": row["analysis_id"],
                "board_id": row["board_id"],
                "category": row["category"],
                "title": row["title"],
                "description": row["description"],
                "findings": row["findings"],
                "recommendations": row["recommendations"],
                "priority": row["priority"],
                "status": row["status"],
                "assigned_to": row["assigned_to"],
                "due_date": row["due_date"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/twg/decisions")
async def get_twg_decisions(board_id: Optional[str] = None, decision_type: Optional[str] = None):
    """Get TWG decisions"""
    try:
        conn = _get_conn_with_twg()
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        query = "SELECT * FROM twg_decisions WHERE 1=1"
        params = []

        if board_id:
            query += " AND board_id = ?"
            params.append(board_id)
        if decision_type:
            query += " AND decision_type = ?"
            params.append(decision_type)

        query += " ORDER BY decision_date DESC"

        cursor.execute(query, params)
        rows = cursor.fetchall()

        results = []
        for row in rows:
            results.append({
                "decision_id": row["decision_id"],
                "board_id": row["board_id"],
                "analysis_id": row["analysis_id"],
                "decision_text": row["decision_text"],
                "decision_type": row["decision_type"],
                "rationale": row["rationale"],
                "impact": row["impact"],
                "decided_by": row["decided_by"],
                "decision_date": row["decision_date"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/twg/actions")
async def get_twg_actions(board_id: Optional[str] = None, status: Optional[str] = None):
    """Get TWG action items"""
    try:
        conn = _get_conn_with_twg()
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        query = "SELECT * FROM twg_action_items WHERE 1=1"
        params = []

        if board_id:
            query += " AND board_id = ?"
            params.append(board_id)
        if status:
            query += " AND status = ?"
            params.append(status)

        query += " ORDER BY priority DESC, due_date ASC"

        cursor.execute(query, params)
        rows = cursor.fetchall()

        results = []
        for row in rows:
            results.append({
                "action_id": row["action_id"],
                "board_id": row["board_id"],
                "decision_id": row["decision_id"],
                "action_text": row["action_text"],
                "assigned_to": row["assigned_to"],
                "due_date": row["due_date"],
                "status": row["status"],
                "priority": row["priority"],
                "completion_notes": row["completion_notes"],
                "completed_date": row["completed_date"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )

@app.post("/api/v2/twg/events")
async def create_or_update_twg_event(payload: Dict[str, Any]):
    try:
        conn = _get_conn_with_twg()
        cur = conn.cursor()
        cur.execute(
            """
            INSERT INTO twg_events (event_id, name, date, location, type, target_audience, expected_leads, budget, status, priority)
            VALUES (:event_id, :name, :date, :location, :type, :target_audience, :expected_leads, :budget, :status, :priority)
            ON CONFLICT(event_id) DO UPDATE SET
              name=excluded.name,
              date=excluded.date,
              location=excluded.location,
              type=excluded.type,
                target_audience=excluded.target_audience,
                expected_leads=excluded.expected_leads,
                budget=excluded.budget,
                status=excluded.status,
                priority=excluded.priority
              """,
              payload,
        )
        conn.commit()
        return JSONResponse(content={"status": "ok", "event_id": payload.get("event_id")})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.post("/api/v2/twg/aar")
async def submit_twg_aar(payload: Dict[str, Any]):
    try:
        conn = _get_conn_with_twg()
        cur = conn.cursor()
        cur.execute(
            """
            INSERT INTO twg_aar_reports (event_id, event_name, date, due_date, hours_since_event, status, submitted_by, content)
            VALUES (:event_id, :event_name, :date, :due_date, :hours_since_event, :status, :submitted_by, :content)
            ON CONFLICT(event_id) DO UPDATE SET
              event_name=excluded.event_name,
              date=excluded.date,
              due_date=excluded.due_date,
              hours_since_event=excluded.hours_since_event,
              status=excluded.status,
              submitted_by=excluded.submitted_by,
              content=excluded.content
            """,
            payload,
        )
        conn.commit()
        return JSONResponse(content={"status": "ok", "event_id": payload.get("event_id")})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.post("/api/v2/twg/agenda")
async def save_twg_agenda_item(item: Dict[str, Any]):
    try:
        conn = _get_conn_with_twg()
        cur = conn.cursor()
        cur.execute(
            """
            INSERT INTO twg_agenda_items (id, meeting_id, section, presenter, status, notes, order_index)
            VALUES (:id, :meeting_id, :section, :presenter, :status, :notes, :order_index)
            ON CONFLICT(id) DO UPDATE SET
              meeting_id=excluded.meeting_id,
              section=excluded.section,
              presenter=excluded.presenter,
              status=excluded.status,
              notes=excluded.notes,
              order_index=excluded.order_index
            """,
            item,
        )
        conn.commit()
        return JSONResponse(content={"status": "ok", "id": item.get("id")})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.get("/api/v2/twg/agenda")
async def get_twg_agenda(meeting_id: Optional[str] = None):
    try:
        conn = _get_conn_with_twg()
        cur = conn.cursor()
        if meeting_id:
            cur.execute(
                "SELECT id, meeting_id, section, presenter, status, notes, order_index FROM twg_agenda_items WHERE meeting_id=? ORDER BY order_index ASC",
                (meeting_id,)
            )
        else:
            cur.execute(
                "SELECT id, meeting_id, section, presenter, status, notes, order_index FROM twg_agenda_items ORDER BY meeting_id, order_index ASC"
            )
        rows = cur.fetchall()
        items = [
            {
                "id": r[0],
                "meeting_id": r[1],
                "section": r[2],
                "presenter": r[3],
                "status": r[4],
                "notes": r[5],
                "order_index": r[6],
            }
            for r in rows
        ]
        return JSONResponse(content={"status": "ok", "items": items, "count": len(items)})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.post("/api/v2/twg/budget")
async def update_twg_budget(budget: Dict[str, Any]):
    try:
        conn = _get_conn_with_twg()
        cur = conn.cursor()
        cur.execute(
            """
            INSERT INTO twg_budget (fy, total_budget, allocated, spent, remaining, q1, q2, q3, q4)
            VALUES (:fy, :total_budget, :allocated, :spent, :remaining, :q1, :q2, :q3, :q4)
            ON CONFLICT(fy) DO UPDATE SET
              total_budget=excluded.total_budget,
              allocated=excluded.allocated,
              spent=excluded.spent,
              remaining=excluded.remaining,
              q1=excluded.q1,
              q2=excluded.q2,
              q3=excluded.q3,
              q4=excluded.q4
            """,
            budget,
        )
        conn.commit()
        return JSONResponse(content={"status": "ok", "fy": budget.get("fy")})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})


# ====================
# LEAD STATUS REPORT ENDPOINTS
# ====================

@app.get("/api/v2/leads/status")
async def get_lead_status(
    days: Optional[int] = None,
    stage: Optional[str] = None,
    recruiter: Optional[str] = None,
    source: Optional[str] = None
):
    """Get detailed lead status information"""
    try:
        conn = sqlite3.connect(DB_FILE)
```

```
conn.row_factory = sqlite3.Row
cursor = conn.cursor()

# Build query with filters
query = """
    SELECT
        l.prid as lead_id,
        l.first_name,
        l.last_name,
        l.current_stage as stage,
        l.lead_source as source,
        l.recruiter_id as recruiter,
        l.lead_date as created_date,
        COALESCE(l.ship_date, l.enlistment_date, l.test_pass_date, l.test_date,
                 l.appointment_conducted_date, l.appointment_made_date,
                 l.prospect_date, l.lead_date) as last_activity_date,
        50 as propensity_score,
        CASE
            WHEN l.current_stage IN ('enlistment', 'ship') THEN 'converted'
            WHEN l.current_stage = 'loss' THEN 'lost'
            WHEN julianday('now') - julianday(COALESCE(l.ship_date, l.enlistment_date, l.test_pass_date, l.test_date,
                 l.appointment_conducted_date, l.appointment_made_date,
                 l.prospect_date, l.lead_date)) > 30 THEN 'unresponsive'
            WHEN COALESCE(l.ship_date, l.enlistment_date, l.test_pass_date, l.test_date,
                 l.appointment_conducted_date, l.appointment_made_date,
                 l.prospect_date) IS NOT NULL THEN 'contacted'
            ELSE 'active'
        END as status,
        CAST((julianday('now') - julianday(l.lead_date)) AS INTEGER) as days_in_stage,
        0 as contact_attempts
    FROM leads l
    WHERE 1=1
"""
params = []

if days:
    query += " AND julianday('now') - julianday(l.lead_date) <= ?"
    params.append(days)
if stage:
    query += " AND l.current_stage = ?"
    params.append(stage)
if recruiter:
    query += " AND l.recruiter_id = ?"
    params.append(recruiter)
if source:
    query += " AND l.lead_source = ?"
    params.append(source)

query += " ORDER BY l.lead_date DESC"

cursor.execute(query, params)
rows = cursor.fetchall()

results = []
for row in rows:
    results.append({
        "lead_id": row["lead_id"],
        "first_name": row["first_name"],
        "last_name": row["last_name"],
        "stage": row["stage"],
        "source": row["source"],
        "recruiter": row["recruiter"],
        "created_date": row["created_date"],
        "last_activity_date": row["last_activity_date"],
        "days_in_stage": row["days_in_stage"],
        "propensity_score": row["propensity_score"] or 0,
        "contact_attempts": row["contact_attempts"],
                "status": row["status"]
            })

        conn.close()

        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


@app.get("/api/v2/leads/metrics")
async def get_lead_metrics(
    days: Optional[int] = None,
    stage: Optional[str] = None,
    recruiter: Optional[str] = None
):
    """Get aggregated lead metrics"""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        base_filter = "WHERE 1=1"
        params = []

        if days:
            base_filter += " AND julianday('now') - julianday(lead_date) <= ?"
            params.append(days)

        # Metrics by stage
        cursor.execute(f"""
            SELECT
                current_stage as stage,
                COUNT(*) as count,
                CAST(AVG(julianday('now') - julianday(lead_date)) AS INTEGER) as avg_days,
                CAST(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM leads {base_filter}) AS REAL) as conversion_rate
            FROM leads
            {base_filter}
            GROUP BY current_stage
            ORDER BY
                CASE current_stage
                    WHEN 'lead' THEN 1
                    WHEN 'prospect' THEN 2
                    WHEN 'appointment_made' THEN 3
                    WHEN 'appointment_conducted' THEN 4
                    WHEN 'test' THEN 5
                    WHEN 'test_pass' THEN 6
                    WHEN 'enlistment' THEN 7
                    WHEN 'ship' THEN 8
                    ELSE 9
                END
        """, params)
        by_stage = []
        for row in cursor.fetchall():
            by_stage.append({
                "stage": row["stage"],
                "count": row["count"],
                "avg_days": round(row["avg_days"]) if row["avg_days"] else 0,
                "conversion_rate": round(row["conversion_rate"], 2) if row["conversion_rate"] else 0
            })

        # Metrics by recruiter
        cursor.execute(f"""
            SELECT
                recruiter_id as recruiter,
                COUNT(*) as total_leads,
                SUM(CASE WHEN current_stage NOT IN ('loss', 'ship') THEN 1 ELSE 0 END) as active_leads,
                SUM(CASE WHEN current_stage IN ('enlistment', 'ship') THEN 1 ELSE 0 END) as converted,
                CAST(SUM(CASE WHEN current_stage IN ('enlistment', 'ship') THEN 1 ELSE 0 END) * 100.0 / COUNT(*) AS REAL) as conversion_rate
            FROM leads
            {base_filter}
            GROUP BY recruiter_id
            ORDER BY total_leads DESC
        """, params)
        by_recruiter = []
        for row in cursor.fetchall():
            by_recruiter.append({
                "recruiter": row["recruiter"],
                "total_leads": row["total_leads"],
                "active_leads": row["active_leads"],
                "converted": row["converted"],
                "conversion_rate": round(row["conversion_rate"], 2) if row["conversion_rate"] else 0
            })

        # Metrics by source
        cursor.execute(f"""
            SELECT
                lead_source as source,
                COUNT(*) as leads,
                CAST(SUM(CASE WHEN current_stage IN ('enlistment', 'ship') THEN 1 ELSE 0 END) * 100.0 / COUNT(*) AS REAL) as conversion_rate,
                50 as avg_propensity
            FROM leads
            {base_filter}
            GROUP BY lead_source
            ORDER BY leads DESC
        """, params)
        by_source = []
        for row in cursor.fetchall():
            by_source.append({
                "source": row["source"],
                "leads": row["leads"],
                "conversion_rate": round(row["conversion_rate"], 2) if row["conversion_rate"] else 0,
                "avg_propensity": round(row["avg_propensity"]) if row["avg_propensity"] else 0
            })

        conn.close()

        return JSONResponse(content={
            "status": "ok",
            "data": {
                "by_stage": by_stage,
                "by_recruiter": by_recruiter,
                "by_source": by_source
            }
        })

    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )


# --- 420T Talent Acquisition Technician Endpoints ---
# Import and include 420T router
from backend.routers.talent_acquisition_420t import router as talent_420t_router
app.include_router(talent_420t_router, prefix="/api/v2/420t", tags=["420T Talent Acquisition"])

# --- Company Standings & Helpdesk Endpoints ---
from backend.routers.standings_helpdesk import router as standings_helpdesk_router
```

```
app.include_router(standings_helpdesk_router, prefix="/api/v2", tags=["Standings & Helpdesk"])

# --- Army System Integrations ---
from backend.routers.integrations import router as integrations_router
app.include_router(integrations_router, prefix="/api/v2/integrations", tags=["Army Systems Integration"])

# --- Budget Tracking ---
from backend.routers.budget import router as budget_router
app.include_router(budget_router, prefix="/api/v2", tags=["Budget Management"])

# --- Data Import (Bulk CSV/Excel Upload) ---
from backend.routers.data_upload import router as data_upload_router
app.include_router(data_upload_router, prefix="/api/v2", tags=["Data Upload"])


if __name__ == "__main__":
    uvicorn.run("taaip_service:app", host="0.0.0.0", port=8000, reload=False)

# ====================
# ML PREDICTIONS & EVENT PERFORMANCE APIs
# ====================

@app.post("/api/v2/events/{event_id}/predict")
async def predict_event_performance(event_id: str):
    """Generate ML prediction for event performance"""
    try:
        from ml_prediction_engine import generate_event_prediction
        prediction = generate_event_prediction(event_id)
        if 'error' in prediction:
            return JSONResponse(status_code=404, content={"status": "error", "message": prediction['error']})
        return JSONResponse(content={"status": "ok", "prediction": prediction})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.get("/api/v2/events/performance")
async def get_events_performance(event_type: Optional[str] = None, rsid: Optional[str] = None):
    """Get events with predicted vs actual performance comparison"""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        query = "SELECT event_id, name, event_type_category, location, start_date, budget, status, predicted_leads, predicted_conversions, predicted_roi, predicted_cost_per
        params = []
        if event_type:
            query += " AND event_type_category = ?"
            params.append(event_type)
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)
        query += " ORDER BY start_date DESC LIMIT 100"
        cursor.execute(query, params)
        rows = cursor.fetchall()
        results = []
        for row in rows:
            results.append({"event_id": row["event_id"], "name": row["name"], "event_type_category": row["event_type_category"], "location": row["location"], "start_date": r
        conn.close()
        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.get("/api/v2/nominations")
async def get_marketing_nominations(status: Optional[str] = None, nomination_type: Optional[str] = None, rsid: Optional[str] = None):
    """Get marketing nominations with predictions"""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        query = "SELECT * FROM marketing_nominations WHERE 1=1"
        params = []
        if status:
            query += " AND status = ?"
            params.append(status)
        if nomination_type:
            query += " AND nomination_type = ?"
            params.append(nomination_type)
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)
        query += " ORDER BY nomination_date DESC LIMIT 100"
        cursor.execute(query, params)
        rows = cursor.fetchall()
        results = []
        for row in rows:
            results.append({"nomination_id": row["nomination_id"], "nomination_type": row["nomination_type"], "description": row["description"], "nomination_date": row["n
        conn.close()
        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.get("/api/v2/g2-zones")
async def get_g2_zone_performance(rsid: Optional[str] = None, trend: Optional[str] = None):
    """Get G2 Zone lead performance data"""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        query = "SELECT * FROM g2_zone_performance WHERE 1=1"
        params = []
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)
        if trend:
            query += " AND trend_direction = ?"
            params.append(trend)
        query += " ORDER BY lead_count DESC"
        cursor.execute(query, params)
        rows = cursor.fetchall()
        results = []
        for row in rows:
            results.append({"zone_id": row["zone_id"], "zone_name": row["zone_name"], "geographic_area": row["geographic_area"], "population": row["population"], "milita
        conn.close()
        return JSONResponse(content={"status": "ok", "data": results, "count": len(results)})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

@app.get("/api/v2/g2-zones/summary")
async def get_g2_zones_summary(rsid: Optional[str] = None):
    """Get aggregated G2 Zone performance summary"""
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        query = "SELECT COUNT(*) as total_zones, SUM(lead_count) as total_leads, SUM(qualified_leads) as total_qualified, SUM(enlistment_count) as total_enlistments,
        params = []
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)
        cursor.execute(query, params)
        summary = dict(cursor.fetchone())
        cursor.execute("SELECT zone_name, lead_count, qualification_rate, conversion_rate FROM g2_zone_performance WHERE 1=1 {} ORDER BY lead_count DESC L
        top_zones = [dict(row) for row in cursor.fetchall()]
        conn.close()
        return JSONResponse(content={"status": "ok", "summary": summary, "top_zones": top_zones})
    except Exception as e:
        return JSONResponse(status_code=500, content={"status": "error", "message": str(e)})

# ============================================================================
# CALENDAR / SCHEDULER / STATUS REPORTS API ENDPOINTS
# ============================================================================

@app.get("/api/v2/calendar/events")
async def get_calendar_events(
    start_date: str = None,
    end_date: str = None,
    event_type: str = None,
    priority: str = None,
    status: str = None,
    rsid: str = None
):
    """Get calendar events with optional filters"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM calendar_events WHERE 1=1"
        params = []

        if start_date:
            query += " AND start_datetime >= ?"
            params.append(start_date)
        if end_date:
            query += " AND end_datetime <= ?"
            params.append(end_date)
        if event_type:
            query += " AND event_type = ?"
            params.append(event_type)
        if priority:
            query += " AND priority = ?"
            params.append(priority)
        if status:
            query += " AND status = ?"
            params.append(status)
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)

        query += " ORDER BY start_datetime ASC"

        cursor.execute(query, params)
        events = [dict(row) for row in cursor.fetchall()]

        # Calculate summary statistics
        cursor.execute("""
            SELECT
                COUNT(*) as total_events,
                SUM(CASE WHEN status = 'scheduled' AND start_datetime > datetime('now') THEN 1 ELSE 0 END) as upcoming_events,
                SUM(CASE WHEN status IN ('scheduled', 'in_progress') AND start_datetime < datetime('now') THEN 1 ELSE 0 END) as overdue_events,
                SUM(CASE WHEN status = 'completed' THEN 1 ELSE 0 END) as completed_events,
                SUM(CASE WHEN start_datetime BETWEEN datetime('now') AND datetime('now', '+7 days') THEN 1 ELSE 0 END) as next_7_days_count,
                SUM(CASE WHEN start_datetime BETWEEN datetime('now') AND datetime('now', '+30 days') THEN 1 ELSE 0 END) as next_30_days_count
            FROM calendar_events
        """)
        summary_row = cursor.fetchone()
        summary = dict(summary_row) if summary_row else {}

        # Events by type
        cursor.execute("""
            SELECT event_type, COUNT(*) as count
            FROM calendar_events
            GROUP BY event_type
        """)
        events_by_type = {row['event_type']: row['count'] for row in cursor.fetchall()}

        # Events by priority
        cursor.execute("""
            SELECT priority, COUNT(*) as count
            FROM calendar_events
            GROUP BY priority
        """)
        events_by_priority = {row['priority']: row['count'] for row in cursor.fetchall()}

        # Events by status
        cursor.execute("""
            SELECT status, COUNT(*) as count
            FROM calendar_events
            GROUP BY status
        """)
        events_by_status = {row['status']: row['count'] for row in cursor.fetchall()}

        summary['events_by_type'] = events_by_type
        summary['events_by_priority'] = events_by_priority
        summary['events_by_status'] = events_by_status

        conn.close()

        return JSONResponse({
            "status": "ok",
            "events": events,
            "summary": summary
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/calendar/events")
async def create_calendar_event(request: Request):
    """Create a new calendar event"""
    try:
        data = await request.json()
        conn = get_db_conn()
        cursor = conn.cursor()

        event_id = f"cal_{secrets.token_hex(6)}"

        cursor.execute("""
            INSERT INTO calendar_events (
                event_id, title, description, event_type, category,
                start_datetime, end_datetime, all_day, location, attendees,
                status, priority, recurrence_rule, recurrence_end_date,
                reminder_minutes, linked_entity_type, linked_entity_id,
                created_by, assigned_to, notes, rsid, brigade, battalion
            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            event_id,
            data.get('title'),
            data.get('description'),
            data.get('event_type', 'other'),
            data.get('category'),
            data.get('start_datetime'),
            data.get('end_datetime'),
            data.get('all_day', 0),
            data.get('location'),
            data.get('attendees'),
            data.get('status', 'scheduled'),
            data.get('priority', 'medium'),
            data.get('recurrence_rule'),
            data.get('recurrence_end_date'),
            data.get('reminder_minutes', 60),
            data.get('linked_entity_type'),
            data.get('linked_entity_id'),
            data.get('created_by'),
            data.get('assigned_to'),
            data.get('notes'),
            data.get('rsid'),
            data.get('brigade'),
            data.get('battalion')
        ))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "event_id": event_id,
            "message": "Calendar event created successfully"
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/calendar/events/{calendar_event_id}/create-project")
async def create_project_from_calendar_event(calendar_event_id: str):
    """Automatically create a project from a calendar event (EMM integration)"""
    try:
        import uuid
        conn = get_db_conn()
        cursor = conn.cursor()

        # Fetch calendar event
        cursor.execute("SELECT * FROM calendar_events WHERE event_id = ?", (calendar_event_id,))
        row = cursor.fetchone()

        if not row:
            return JSONResponse({"status": "error", "message": "Calendar event not found"}, status_code=404)

        calendar_event = dict(row)

        # Check if project already exists for this calendar event
        cursor.execute("""
            SELECT project_id FROM projects
            WHERE name = ? AND event_id IS NULL
            LIMIT 1
        """, (f"{calendar_event['title']} - Planning",))

        existing = cursor.fetchone()
        if existing:
            return JSONResponse({
                "status": "ok",
                "project_id": existing['project_id'],
                "message": "Project already exists for this event"
            })

        # Create a recruiting event entry first (optional, for tracking)
        recruiting_event_id = None
        if calendar_event.get('event_type') in ['event', 'marketing']:
            recruiting_event_id = f"evt_{uuid.uuid4().hex[:12]}"
            now = datetime.utcnow().isoformat()

            cursor.execute("""
                INSERT INTO events (
                    event_id, name, type, location, start_date, end_date,
                    status, created_at, updated_at, rsid, brigade, battalion
                ) VALUES (?, ?, ?, ?, ?, ?, 'planned', ?, ?, ?, ?, ?)
            """, (
                recruiting_event_id,
                calendar_event['title'],
                calendar_event.get('event_type', 'In-Person-Meeting'),
                calendar_event.get('location', ''),
                calendar_event.get('start_datetime', '')[:10],  # Extract date
                calendar_event.get('end_datetime', '')[:10],
                now, now,
                calendar_event.get('rsid'),
                calendar_event.get('brigade'),
                calendar_event.get('battalion')
            ))

        # Create project
        project_id = f"prj_{uuid.uuid4().hex[:12]}"
        now = datetime.utcnow().isoformat()

        # Calculate dates (start 2 weeks before event, target on event date)
        from datetime import datetime as dt, timedelta
        event_start = dt.fromisoformat(calendar_event['start_datetime'].replace('Z', '+00:00'))
        project_start = (event_start - timedelta(days=14)).isoformat()
        project_target = event_start.isoformat()

        cursor.execute("""
            INSERT INTO projects (
                project_id, name, event_id, start_date, target_date,
                owner_id, objectives, success_criteria, status,
                created_at, updated_at, rsid, brigade, battalion
            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, 'planning', ?, ?, ?, ?, ?)
        """, (
            project_id,
            f"{calendar_event['title']} - Planning",
            recruiting_event_id,
            project_start[:10],
            project_target[:10],
            calendar_event.get('created_by', 'system'),
            f"Plan and execute {calendar_event['title']}. {calendar_event.get('description', '')}",
            "Successfully execute event and achieve target metrics",
            now, now,
            calendar_event.get('rsid'),
            calendar_event.get('brigade'),
            calendar_event.get('battalion')
        ))

        # Create default tasks for event planning
        default_tasks = [
            {
                "title": "Finalize Event Logistics",
                "description": "Confirm venue, setup, and equipment",
                "priority": "high",
                "days_before": 7
            },
            {
                "title": "Prepare Marketing Materials",
                "description": "Design and print promotional materials",
                "priority": "high",
                "days_before": 10
            },
            {
                "title": "Coordinate Team Assignments",
                "description": "Assign roles and responsibilities to team members",
                "priority": "medium",
                "days_before": 5
            },
            {
                "title": "Conduct Pre-Event Briefing",
                "description": "Brief team on objectives and procedures",
                "priority": "high",
                "days_before": 1
            }
        ]
```

```
        for task_template in default_tasks:
            task_id = f"tsk_{uuid.uuid4().hex[:12]}"
            task_due = (event_start - timedelta(days=task_template['days_before'])).isoformat()[:10]

            cursor.execute("""
                INSERT INTO tasks (
                    task_id, project_id, title, description,
                    assigned_to, due_date, status, priority, created_at
                ) VALUES (?, ?, ?, ?, ?, ?, 'open', ?, ?)
            """, (
                task_id, project_id, task_template['title'], task_template['description'],
                calendar_event.get('assigned_to', 'team'), task_due, task_template['priority'], now
            ))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "project_id": project_id,
            "event_id": recruiting_event_id,
            "message": "Project created successfully from calendar event",
            "tasks_created": len(default_tasks)
        })

    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/calendar/reports")
async def get_status_reports(
    report_type: str = None,
    report_category: str = None,
    status: str = None,
    rsid: str = None,
    limit: int = 50
):
    """Get status reports with optional filters"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM status_reports WHERE 1=1"
        params = []

        if report_type:
            query += " AND report_type = ?"
            params.append(report_type)
        if report_category:
            query += " AND report_category = ?"
            params.append(report_category)
        if status:
            query += " AND status = ?"
            params.append(status)
        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)

        query += " ORDER BY generated_date DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        reports = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "reports": reports
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/calendar/reports/generate")
async def generate_status_report(request: Request):
    """Generate a status report based on type and category"""
    try:
        data = await request.json()
        report_type = data.get('report_type', 'monthly')  # daily, weekly, monthly, quarterly, annual
        report_category = data.get('report_category', 'overall')  # events, marketing, recruiting, etc.

        conn = get_db_conn()
        cursor = conn.cursor()

        # Calculate period based on report type
        from datetime import datetime, timedelta
        end_date = datetime.now()

        if report_type == 'daily':
            start_date = end_date - timedelta(days=1)
        elif report_type == 'weekly':
            start_date = end_date - timedelta(weeks=1)
        elif report_type == 'monthly':
            start_date = end_date - timedelta(days=30)
        elif report_type == 'quarterly':
            start_date = end_date - timedelta(days=90)
        elif report_type == 'annual':
            start_date = end_date - timedelta(days=365)
        else:
            start_date = end_date - timedelta(days=30)

        # Generate report based on category
        summary = f"{report_type.upper()} {report_category.upper()} Report"
        key_metrics = {}

        if report_category in ['events', 'overall']:
            # Event metrics
            cursor.execute("""
                SELECT
                    COUNT(*) as total_events,
                    SUM(CASE WHEN status = 'completed' THEN 1 ELSE 0 END) as completed_events,
                    SUM(CASE WHEN status = 'cancelled' THEN 1 ELSE 0 END) as cancelled_events
                FROM calendar_events
                WHERE start_datetime BETWEEN ? AND ?
            """, (start_date.isoformat(), end_date.isoformat()))
            event_metrics = dict(cursor.fetchone() or {})
            key_metrics['events'] = event_metrics

        if report_category in ['marketing', 'overall']:
            # Marketing metrics
            cursor.execute("""
                SELECT
                    COUNT(*) as total_nominations,
                    SUM(CASE WHEN status = 'approved' THEN 1 ELSE 0 END) as approved,
                    AVG(predicted_roi) as avg_predicted_roi
                FROM marketing_nominations
                WHERE nomination_date BETWEEN ? AND ?
            """, (start_date.isoformat(), end_date.isoformat()))
            marketing_metrics = dict(cursor.fetchone() or {})
            key_metrics['marketing'] = marketing_metrics

        if report_category in ['recruiting', 'overall']:
            # Recruiting metrics
            cursor.execute("""
                SELECT
                COUNT(*) as total_leads,
                SUM(CASE WHEN current_stage = 'enlistment' THEN 1 ELSE 0 END) as enlistments,
                SUM(CASE WHEN current_stage = 'ship' THEN 1 ELSE 0 END) as ships
            FROM leads
            WHERE created_at BETWEEN ? AND ?
        """, (start_date.isoformat(), end_date.isoformat()))
        recruiting_metrics = dict(cursor.fetchone() or {})
        key_metrics['recruiting'] = recruiting_metrics

        # Create report
        report_id = f"rpt_{secrets.token_hex(6)}"

        cursor.execute("""
            INSERT INTO status_reports (
                report_id, report_type, report_category,
                report_period_start, report_period_end, generated_date,
                status, summary, key_metrics
            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            report_id,
            report_type,
            report_category,
            start_date.isoformat(),
            end_date.isoformat(),
            datetime.now().isoformat(),
            'completed',
            summary,
            json.dumps(key_metrics)
        ))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "report_id": report_id,
            "summary": summary,
            "key_metrics": key_metrics,
            "message": "Status report generated successfully"
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/calendar/upcoming")
async def get_upcoming_events(days: int = 7, rsid: str = None):
    """Get upcoming events for the next N days"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = """
            SELECT * FROM calendar_events
            WHERE start_datetime BETWEEN datetime('now') AND datetime('now', '+' || ? || ' days')
            AND status IN ('scheduled', 'in_progress')
        """
        params = [days]

        if rsid:
            query += " AND rsid = ?"
            params.append(rsid)

        query += " ORDER BY start_datetime ASC"

        cursor.execute(query, params)
        events = [dict(row) for row in cursor.fetchall()]

        conn.close()
        return JSONResponse({
            "status": "ok",
            "events": events,
            "count": len(events)
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/calendar/notifications")
async def get_notifications(status: str = None, limit: int = 50):
    """Get notifications for the user"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM notifications WHERE 1=1"
        params = []

        if status:
            query += " AND status = ?"
            params.append(status)

        query += " ORDER BY created_at DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        notifications = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "notifications": notifications
        })
    except Exception as e:
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)


# ============================================================================
# User Management Endpoints
# ============================================================================

class CreateUserRequest(BaseModel):
    username: str
    email: str
    password: str
    first_name: str
    last_name: str
    rank: str
    role: str = "analyst"
    tier: int = 3
    start_date: str
    end_date: str
    permissions: list[str] = []

class UpdateUserRequest(BaseModel):
    email: Optional[str] = None
    rank: Optional[str] = None
    role: Optional[str] = None
    tier: Optional[int] = None
    is_active: Optional[bool] = None

class PermissionRequest(BaseModel):
    permissions: list[str]
    action: str  # "grant" or "revoke"

@app.get("/api/v2/users")
```

```
async def get_users(is_active: Optional[bool] = None):
    """Get all users with their permissions"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT id, username, email, rank, role, tier, is_active, created_at, updated_at, last_login FROM users"
        params = []

        if is_active is not None:
            query += " WHERE is_active = ?"
            params.append(1 if is_active else 0)

        cursor.execute(query, params)
        users = [dict(row) for row in cursor.fetchall()]

        # Get permissions for each user
        for user in users:
            cursor.execute("""
                SELECT permission FROM user_permissions
                WHERE user_id = ?
            """, (user['id'],))
            user['permissions'] = [row['permission'] for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "users": users
        })
    except Exception as e:
        logging.error(f"Error fetching users: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/users/{user_id}")
async def get_user(user_id: int):
    """Get a single user by ID"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        cursor.execute("""
            SELECT id, username, email, rank, role, tier, is_active, created_at, updated_at, last_login
            FROM users WHERE id = ?
        """, (user_id,))

        user = cursor.fetchone()
        if not user:
            conn.close()
            return JSONResponse({"status": "error", "message": "User not found"}, status_code=404)

        user = dict(user)

        # Get permissions
        cursor.execute("""
            SELECT permission, granted_by, granted_at
            FROM user_permissions
            WHERE user_id = ?
        """, (user_id,))
        user['permissions'] = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "user": user
        })
    except Exception as e:
        logging.error(f"Error fetching user: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/users")
async def create_user(request: CreateUserRequest):
    """Create a new user"""
    try:
        import hashlib
        import secrets

        conn = get_db_conn()
        cursor = conn.cursor()

        # Check if username or email already exists
        cursor.execute("SELECT id FROM users WHERE username = ? OR email = ?",
                      (request.username, request.email))
        if cursor.fetchone():
            conn.close()
            return JSONResponse(
                {"status": "error", "message": "Username or email already exists"},
                status_code=400
            )

        # Hash password
        salt = secrets.token_hex(16)
        password_hash = hashlib.sha256(f"{request.password}{salt}".encode()).hexdigest()

        now = datetime.now().isoformat()

        # Create user
        cursor.execute("""
            INSERT INTO users
            (username, email, password_hash, password_salt, first_name, last_name, rank, role, tier, start_date, end_date, is_active, created_at, updated_at)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            request.username,
            request.email,
            password_hash,
            salt,
            request.first_name,
            request.last_name,
            request.rank,
            request.role,
            request.tier,
            request.start_date,
            request.end_date,
            1,
            now,
            now
        ))

        user_id = cursor.lastrowid

        # Grant permissions
        for perm in request.permissions:
            cursor.execute("""
                INSERT INTO user_permissions (user_id, permission, granted_by, granted_at)
                VALUES (?, ?, ?, ?)
            """, (user_id, perm, 1, now))  # granted_by = 1 (admin)

        # Log action
        cursor.execute("""
            INSERT INTO user_audit_log (action, user_id, performed_by, details, timestamp)
            VALUES (?, ?, ?, ?, ?)
        """, ("create_user", user_id, 1, json.dumps({"username": request.username}), now))
        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "message": "User created successfully",
            "user_id": user_id
        })
    except Exception as e:
        logging.error(f"Error creating user: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.put("/api/v2/users/{user_id}")
async def update_user(user_id: int, request: UpdateUserRequest):
    """Update user details"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        # Check if user exists
        cursor.execute("SELECT id FROM users WHERE id = ?", (user_id,))
        if not cursor.fetchone():
            conn.close()
            return JSONResponse({"status": "error", "message": "User not found"}, status_code=404)

        # Build update query
        updates = []
        params = []

        if request.email is not None:
            updates.append("email = ?")
            params.append(request.email)
        if request.rank is not None:
            updates.append("rank = ?")
            params.append(request.rank)
        if request.role is not None:
            updates.append("role = ?")
            params.append(request.role)
        if request.tier is not None:
            updates.append("tier = ?")
            params.append(request.tier)
        if request.is_active is not None:
            updates.append("is_active = ?")
            params.append(1 if request.is_active else 0)

        updates.append("updated_at = ?")
        params.append(datetime.now().isoformat())
        params.append(user_id)

        cursor.execute(f"""
            UPDATE users SET {', '.join(updates)}
            WHERE id = ?
        """, params)

        # Log action
        cursor.execute("""
            INSERT INTO user_audit_log (action, user_id, performed_by, details, timestamp)
            VALUES (?, ?, ?, ?, ?)
        """, ("update_user", user_id, 1, json.dumps(request.dict(exclude_none=True)), datetime.now().isoformat()))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "message": "User updated successfully"
        })
    except Exception as e:
        logging.error(f"Error updating user: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/users/{user_id}/permissions")
async def manage_permissions(user_id: int, request: PermissionRequest):
    """Grant or revoke permissions for a user"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        # Check if user exists
        cursor.execute("SELECT id FROM users WHERE id = ?", (user_id,))
        if not cursor.fetchone():
            conn.close()
            return JSONResponse({"status": "error", "message": "User not found"}, status_code=404)

        now = datetime.now().isoformat()

        if request.action == "grant":
            for perm in request.permissions:
                cursor.execute("""
                    INSERT OR REPLACE INTO user_permissions (user_id, permission, granted_by, granted_at)
                    VALUES (?, ?, ?, ?)
                """, (user_id, perm, 1, now))

            action_log = "grant_permissions"
        elif request.action == "revoke":
            for perm in request.permissions:
                cursor.execute("""
                    DELETE FROM user_permissions
                    WHERE user_id = ? AND permission = ?
                """, (user_id, perm))

            action_log = "revoke_permissions"
        else:
            conn.close()
            return JSONResponse({"status": "error", "message": "Invalid action"}, status_code=400)

        # Log action
        cursor.execute("""
            INSERT INTO user_audit_log (action, user_id, performed_by, details, timestamp)
            VALUES (?, ?, ?, ?, ?)
        """, (action_log, user_id, 1, json.dumps({"permissions": request.permissions}), now))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "message": f"Permissions {request.action}ed successfully"
        })
    except Exception as e:
        logging.error(f"Error managing permissions: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.post("/api/v2/users/{user_id}/deactivate")
async def deactivate_user(user_id: int):
    """Deactivate a user account"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        # Check if user exists
        cursor.execute("SELECT id, is_active FROM users WHERE id = ?", (user_id,))
        user = cursor.fetchone()
        if not user:
            conn.close()
            return JSONResponse({"status": "error", "message": "User not found"}, status_code=404)

        # Toggle active status
        new_status = 0 if user['is_active'] else 1

        cursor.execute("""
            UPDATE users SET is_active = ?, updated_at = ?
            WHERE id = ?
        """, (new_status, datetime.now().isoformat(), user_id))

        # Log action
        cursor.execute("""
            INSERT INTO user_audit_log (action, user_id, performed_by, details, timestamp)
            VALUES (?, ?, ?, ?, ?)
        """, ("deactivate_user" if new_status == 0 else "activate_user", user_id, 1, json.dumps({}), datetime.now().isoformat()))

        conn.commit()
        conn.close()

        return JSONResponse({
            "status": "ok",
            "message": f"User {'deactivated' if new_status == 0 else 'activated'} successfully"
        })
    except Exception as e:
        logging.error(f"Error deactivating user: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)


# ============================================================================
# Marketing Engagement Performance Endpoints
# ============================================================================

@app.get("/api/v2/marketing/campaigns")
async def get_marketing_campaigns(
    status: Optional[str] = None,
    platform: Optional[str] = None,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None
):
    """Get marketing campaigns with optional filters"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM marketing_campaigns WHERE 1=1"
        params = []

        if status:
            query += " AND status = ?"
            params.append(status)
        if platform:
            query += " AND platform = ?"
            params.append(platform)
        if start_date:
            query += " AND start_date >= ?"
            params.append(start_date)
        if end_date:
            query += " AND end_date <= ?"
            params.append(end_date)

        query += " ORDER BY start_date DESC"

        cursor.execute(query, params)
        campaigns = [dict(row) for row in cursor.fetchall()]

        conn.close()
        return JSONResponse({
            "status": "ok",
            "campaigns": campaigns,
            "count": len(campaigns)
        })
    except Exception as e:
        logging.error(f"Error fetching campaigns: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/engagement-metrics")
async def get_engagement_metrics(
    campaign_id: Optional[str] = None,
    platform: Optional[str] = None,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None,
    limit: int = 100
):
    """Get marketing engagement metrics"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM marketing_engagement_metrics WHERE 1=1"
        params = []

        if campaign_id:
            query += " AND campaign_id = ?"
            params.append(campaign_id)
        if platform:
            query += " AND platform = ?"
            params.append(platform)
        if start_date:
            query += " AND metric_date >= ?"
            params.append(start_date)
        if end_date:
            query += " AND metric_date <= ?"
            params.append(end_date)

        query += " ORDER BY metric_date DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        metrics = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "metrics": metrics,
            "count": len(metrics)
        })
    except Exception as e:
        logging.error(f"Error fetching engagement metrics: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/social-media-posts")
async def get_social_media_posts(
    platform: Optional[str] = None,
    campaign_id: Optional[str] = None,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None,
    limit: int = 50
):
    """Get social media posts with engagement data"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()
        query = "SELECT * FROM social_media_posts WHERE 1=1"
        params = []

        if platform:
            query += " AND platform = ?"
            params.append(platform)
        if campaign_id:
            query += " AND campaign_id = ?"
            params.append(campaign_id)
        if start_date:
            query += " AND posted_date >= ?"
            params.append(start_date)
        if end_date:
            query += " AND posted_date <= ?"
            params.append(end_date)

        query += " ORDER BY posted_date DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        posts = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "posts": posts,
            "count": len(posts)
        })
    except Exception as e:
        logging.error(f"Error fetching social media posts: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/platforms")
async def get_marketing_platforms():
    """Get all marketing platform integrations"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        cursor.execute("SELECT * FROM marketing_platform_integrations ORDER BY platform_name")
        platforms = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "platforms": platforms
        })
    except Exception as e:
        logging.error(f"Error fetching platforms: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/overview")
async def get_marketing_overview(days: int = 30):
    """Get marketing performance overview"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        # Calculate date range
        from datetime import timedelta
        end_date = datetime.now()
        start_date = end_date - timedelta(days=days)
        start_str = start_date.strftime('%Y-%m-%d')

        # Total campaigns
        cursor.execute("""
            SELECT COUNT(*) as total,
                   SUM(CASE WHEN status = 'active' THEN 1 ELSE 0 END) as active
            FROM marketing_campaigns
        """)
        campaigns_data = dict(cursor.fetchone())

        # Aggregate metrics for the period
        cursor.execute("""
            SELECT
                SUM(impressions) as total_impressions,
                SUM(views) as total_views,
                SUM(engagements) as total_engagements,
                SUM(clicks) as total_clicks,
                SUM(conversions) as total_conversions,
                AVG(engagement_rate) as avg_engagement_rate,
                AVG(click_through_rate) as avg_ctr,
                AVG(conversion_rate) as avg_conversion_rate
            FROM marketing_engagement_metrics
            WHERE metric_date >= ?
        """, (start_str,))
        metrics_data = dict(cursor.fetchone())

        # Top performing platforms
        cursor.execute("""
            SELECT platform,
                   SUM(engagements) as total_engagements,
                   SUM(impressions) as total_impressions,
                   AVG(engagement_rate) as avg_engagement_rate
            FROM marketing_engagement_metrics
            WHERE metric_date >= ?
            GROUP BY platform
            ORDER BY total_engagements DESC
            LIMIT 5
        """, (start_str,))
        top_platforms = [dict(row) for row in cursor.fetchall()]

        # Recent social media performance
        cursor.execute("""
            SELECT platform,
                   COUNT(*) as post_count,
                   SUM(engagements) as total_engagements,
                   SUM(impressions) as total_impressions
            FROM social_media_posts
            WHERE posted_date >= ?
            GROUP BY platform
        """, (start_str,))
        social_performance = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "overview": {
                "total_campaigns": campaigns_data.get('total', 0) or 0,
                "active_campaigns": campaigns_data.get('active', 0) or 0,
                "total_impressions": metrics_data.get('total_impressions', 0) or 0,
                "total_views": metrics_data.get('total_views', 0) or 0,
                "total_engagements": metrics_data.get('total_engagements', 0) or 0,
                "total_clicks": metrics_data.get('total_clicks', 0) or 0,
                "total_conversions": metrics_data.get('total_conversions', 0) or 0,
                "avg_engagement_rate": round(metrics_data.get('avg_engagement_rate', 0) or 0, 2),
                "avg_ctr": round(metrics_data.get('avg_ctr', 0) or 0, 2),
                "avg_conversion_rate": round(metrics_data.get('avg_conversion_rate', 0) or 0, 2),
                "top_platforms": top_platforms,
                "social_performance": social_performance,
                "period_days": days
            }
```

```
        })
    except Exception as e:
        logging.error(f"Error fetching marketing overview: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/email-metrics")
async def get_email_metrics(
    campaign_id: Optional[str] = None,
    platform: Optional[str] = None,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None,
    limit: int = 50
):
    """Get email marketing metrics"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM email_marketing_metrics WHERE 1=1"
        params = []

        if campaign_id:
            query += " AND campaign_id = ?"
            params.append(campaign_id)
        if platform:
            query += " AND platform = ?"
            params.append(platform)
        if start_date:
            query += " AND send_date >= ?"
            params.append(start_date)
        if end_date:
            query += " AND send_date <= ?"
            params.append(end_date)

        query += " ORDER BY send_date DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        emails = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "emails": emails,
            "count": len(emails)
        })
    except Exception as e:
        logging.error(f"Error fetching email metrics: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

@app.get("/api/v2/marketing/digital-ads")
async def get_digital_ads(
    campaign_id: Optional[str] = None,
    platform: Optional[str] = None,
    start_date: Optional[str] = None,
    end_date: Optional[str] = None,
    limit: int = 50
):
    """Get digital advertising performance"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        query = "SELECT * FROM digital_advertising WHERE 1=1"
        params = []
        if campaign_id:
            query += " AND campaign_id = ?"
            params.append(campaign_id)
        if platform:
            query += " AND platform = ?"
            params.append(platform)
        if start_date:
            query += " AND start_date >= ?"
            params.append(start_date)
        if end_date:
            query += " AND end_date <= ?"
            params.append(end_date)

        query += " ORDER BY start_date DESC LIMIT ?"
        params.append(limit)

        cursor.execute(query, params)
        ads = [dict(row) for row in cursor.fetchall()]

        conn.close()

        return JSONResponse({
            "status": "ok",
            "ads": ads,
            "count": len(ads)
        })
    except Exception as e:
        logging.error(f"Error fetching digital ads: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)

# ============================================================================
# Universal Data Upload Endpoints
# ============================================================================

class UniversalUploadRequest(BaseModel):
    data: list[dict]
    category: str

@app.post("/api/v2/upload/{category}")
async def upload_data(category: str, request: UniversalUploadRequest):
    """Universal data upload endpoint that routes to appropriate tables"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        rows_inserted = 0
        now = datetime.now().isoformat()

        # Store in generic import table for all categories
        cursor.execute("""
        CREATE TABLE IF NOT EXISTS data_imports (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            category TEXT,
            data TEXT,
            rows_count INTEGER,
            imported_at TEXT
        )
        """)

        cursor.execute("""
        INSERT INTO data_imports (category, data, rows_count, imported_at)
        VALUES (?, ?, ?, ?)
        """, (category, json.dumps(request.data), len(request.data), now))

        rows_inserted = len(request.data)

        conn.commit()
        conn.close()
        return JSONResponse({
            "status": "ok",
            "message": f"Successfully imported {rows_inserted} rows for {category}",
            "rows_processed": rows_inserted,
            "category": category
        })
    except Exception as e:
        logging.error(f"Error uploading data: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)


@app.get("/api/v2/upload/history")
async def get_upload_history():
    """Retrieve upload history from data_imports table"""
    try:
        conn = get_db_conn()
        cursor = conn.cursor()

        cursor.execute("""
        SELECT id, category, data, rows_count, imported_at
        FROM data_imports
        ORDER BY imported_at DESC
        LIMIT 100
        """)

        rows = cursor.fetchall()
        conn.close()

        history = []
        for row in rows:
            try:
                data = json.loads(row[2]) if row[2] else []
            except:
                data = []

            history.append({
                "id": row[0],
                "category": row[1],
                "data": data,
                "rows_count": row[3],
                "imported_at": row[4]
            })

        return JSONResponse({
            "status": "ok",
            "history": history
        })
    except Exception as e:
        logging.error(f"Error fetching upload history: {e}")
        return JSONResponse({"status": "error", "message": str(e)}, status_code=500)
```