DEMETRA SYSTEMS

Задача: Единый дашборд аналитики рекламных каналов
Цель: Разработать дашборд для объединения аналитики всех рекламных каналов в одном месте.
Интегрируемые платформы: Instagram, Google Ads, TikTok, Telegram и другие
Отображаемые метрики:
Графики и динамика роста
Стоимость заявки и конверсии
Охваты и оценка эффективности каналов
AI-функционал:
Сравнение эффективности каналов
Прогнозирование окупаемости
Рекомендации по распределению бюджета
Результат: Быстрое принятие решений на основе данных с четкими рекомендациями по оптимизации рекламных кампаний.

give me the architectural solution for this one!

Для задачи создания единого дашборда аналитики рекламных каналов DEMETRA SYSTEMS с интеграцией Instagram, Google Ads, TikTok, Telegram и других платформ, включая AI-функционал, можно предложить следующую архитектурную схему.

Архитектурное решение единого дашборда аналитики рекламных каналов

Источники данных и интеграции

- Использование API интеграции с рекламными платформами Instagram, Google Ads, TikTok, Telegram для сбора данных о рекламных кампаниях, охватах, кликах, конверсиях.
- Поддержка дополнительных платформ через расширяемые адаптеры.
- ETL-процессы (Extract, Transform, Load) для очистки, нормализации и объединения данных в единую базу аналитики.

Хранилище и обработка данных

- Централизованное хранилище данных (Data Warehouse или Data Lake) для хранения объединённой информации.
- Использование потоковых технологий для обеспечения актуальности данных.
- Колонки и таблицы для хранения метрик: стоимость заявки, конверсии, охваты, эффективность.

Визуализация в дашборде

- Разработка фронтенда с графиками и визуализациями (линейные графики динамики роста, диаграммы распределения затрат, сравнительные графики эффективности каналов).
- Возможность drill-down для детального анализа по рекламным кампаниям и каналам.
- KPI-панели с ключевыми метриками и предупреждениями.

AI-функционал

- Модели машинного обучения для сравнения эффективности каналов по заданным метрикам.
- Прогнозирование окупаемости рекламных кампаний на основе исторических данных и тенденций.
- Рекомендательные системы для оптимального распределения бюджета по каналам с учётом прогноза и заданных бизнес-целей.
- Автоматизированные отчёты и подсказки для быстрого принятия решений.

Архитектурный стек (пример)

- Интеграция: RESTful API, Webhooks, SDK.
- Хранилище: облачные OLAP-базы данных (например, BigQuery, Snowflake).
- Обработка данных: Apache Kafka, Apache Spark.
- Модели AI: Python ML-библиотеки (scikit-learn, TensorFlow).
- Визуализация: React, D3.js, Grafana или специализированные BI-инструменты.
- Оркестрация: Apache Airflow для управления ETL и ML-пайплайнами.

Итог

Данное решение позволяет агрегировать данные со всех рекламных каналов в едином месте, обеспечивать прозрачную аналитику, применять искусственный интеллект для оценки и прогнозирования, а также формировать рекомендации по бюджету. Это повышает скорость и качество принятия решений по оптимизации рекламных кампаний. [1] [2] [3] [4]

⁂

DEMETRA SYSTEMS

Задача: Единый дашборд аналитики рекламных каналов

Цель: Разработать дашборд для объединения аналитики всех рекламных каналов в одном месте.

Интегрируемые платформы: Instagram, Google Ads, TikTok, Telegram и другие

Отображаемые метрики:

- Графики и динамика роста
- Стоимость заявки и конверсии
- Охваты и оценка эффективности каналов

AI-функционал:

- Сравнение эффективности каналов
- Прогнозирование окупаемости
- Рекомендации по распределению бюджета

Результат: Быстрое принятие решений на основе данных с четкими рекомендациями по оптимизации рекламных кампаний.

Data Visualization

API Integration

AI Analytics

Dashboard

DEMETRA SYSTEMS

Задача: Разработка AI-дашборда для анализа рекламных кампаний

□ Кейсовая задача:

Компания DEMETRA SYSTEMS ведет активные рекламные кампании в различных социальных сетях и digital-каналах. В настоящее время отсутствует единая система анализа эффективности рекламных активностей, что затрудняет оптимизацию маркетингового бюджета и принятие стратегических решений.

□ Задача:

Разработать интеллектуальный дашборд для комплексного анализа рекламных кампаний компании с использованием искусственного интеллекта.

□ Основные функции:

- Мультиканальный анализ: Интеграция данных из всех рекламных каналов (Instagram, Facebook, TikTok, Google Ads, Яндекс.Директ и др.)
- AI-аналитика: Автоматический анализ эффективности рекламных кампаний в реальном времени
- Прогнозные рекомендации: Интеллектуальные предложения по оптимизации рекламного бюджета
- Сравнительная аналитика: Сравнение эффективности разных рекламных каналов и стратегий
- Автоматические отчеты: Генерация детализированных отчетов с ключевыми метриками

▯ Ключевые возможности:

- Визуализация ROI по каналам
- AI-рекомендации по распределению бюджета
- Анализ целевой аудитории
- Прогнозирование эффективности кампаний
- Оптимизация стоимости привлечения клиента
- Анализ лучшего времени для публикаций

▯ Ожидаемый результат:

Интерактивный веб-дашборд с AI-аналитикой, предоставляющий маркетинговой команде инструменты для принятия data-driven решений и увеличения эффективности рекламных кампаний на 25-40%.

▯ Технические требования:

- Поддержка интеграции с API популярных рекламных платформ
- Реализация machine learning моделей для прогнозирования
- Адаптивный дизайн для работы на разных устройствах
- Система оповещений о критических изменениях показателей

Data Analytics AI/ML Dashboard Digital Marketing API Integration Data Visualization

это сверху контекст самого проекта,
помоги мне выстроить Архитектуру!

у нас уже есть:

- backend (fastapi, postgres) initial setup
- frontend (react) initial setup
- docker compose for front/back/db
- ML model
- LLM model

image.jpg - это csv файл датасета на котором мы обучаем ML model в виде (Social Media Advertising Dataset | Kaggle)
ml.ipynb - это ml model и дашборды на используя Saeborn (используя датасет для обучение и Booster)
recommendations.py - это LLM который на основе данных должен будет в текстовом виде рекомендовать что то!

это все вещи должны типа обновляться один раз в час!

так же какие улучшение ты можешь добавить по типу алертов?

ЦЕЛЬ: реализовать MVP MVP проект который можно будет показать как то что мы решили этот кейс в самом хакатоне!
и цель это создать и попробовать создать этот проект!!!
помоги мне с этим, у меня краткие сроки и мне нужно что то, с командой да сделать!!!

Отлично! Понял ситуацию. У вас краткие сроки и нужно быстро собрать MVP для хакатона. Давайте построим **практическую архитектуру**, которую можно реализовать за короткое время.

Архитектура MVP для DEMETRA SYSTEMS

Общая схема системы

```
┌─────────────────────────────────────────────────────────────────┐
│                         FRONTEND (React)                         │
│  ┌────────────────┐  ┌─────────────────┐  ┌─────────────────────────┐  │
│  │ Dashboard    │  │  Analytics     │  │  AI Recommendations    │  │
│  │   View       │  │  Charts        │  │       Panel            │  │
│  └────────────────┘  └─────────────────┘  └─────────────────────────┘  │
└─────────────────────────────────────┬───────────────────────────┘
                                 │ REST API
                                 ▼
┌─────────────────────────────────────────────────────────────────┐
│                         BACKEND (FastAPI)                        │
│  ┌─────────────────────────────────────────────────────────┐    │
│  │                   API Endpoints                          │   │
│  ├─────────────────────────────────────────────────────────┤    │
│  │  /api/campaigns  │  /api/analytics  │  /api/predictions │   │
│  │  /api/recommendations  │  /api/alerts  │  /api/refresh  │   │
│  └─────────────────────┬───────────────────────────────────┘    │
│                       │                                   │     │
│  ┌─────────────────────┼───────────────────────────────────┐    │
│  │    BUSINESS LOGIC LAYER                                  │   │
│  ├─────────────────────────────────────────────────────────┤    │
│  │  • Campaign Service  • Analytics Service                 │   │
│  │  • ML Prediction Service  • LLM Recommendation Service   │   │
│  │  • Alert Service  • Data Aggregation Service             │   │
│  └─────────────────────┬───────────────────────────────────┘    │
└─────────────────────────────┬───────────────────────────────────┘
                         │
           ┌─────────────────────┼─────────────────────┐
```

```
          ▼                 ▼                 ▼
  ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐
  │   PostgreSQL │   │  ML Model     │   │  LLM Model    │
  │   Database   │   │ (Pickle)      │   │ (HuggingFace)│
  │              │   │               │   │              │
  │ • Campaigns  │   │ HistGradient  │   │ Qwen2.5-7B   │
  │ • Metrics    │   │  Boosting     │   │  Instruct    │
  │ • Predictions│   │               │   │              │
  │ • Alerts     │   │               │   │              │
  └─────────────────┘ └─────────────────┘ └─────────────────┘
          │
          ▼
  ┌──────────────────────────────────────────────────────────────────────┐
  │              BACKGROUND SCHEDULER                          │
  │  ┌────────────────────────────────────────────────────────────────┐  │
  │  │   APScheduler / Celery + Redis (optional)          │ │
  │  ├────────────────────────────────────────────────────────────────┤  │
  │  │  • Hourly data refresh                             │ │
  │  │  • ML model predictions update                     │ │
  │  │  • LLM recommendations generation                  │ │
  │  │  • Alert checks and notifications                  │ │
  │  └────────────────────────────────────────────────────────────────┘  │
  └──────────────────────────────────────────────────────────────────────┘
```

Структура проекта (рекомендуемая)

```
demetra-dashboard/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py                    # FastAPI app entry point
│   │   ├── config.py                  # Settings, env variables
│   │   ├── database.py                # DB connection
│   │   ├── models/
│   │   │   ├── __init__.py
│   │   │   ├── campaign.py            # SQLAlchemy models
│   │   │   ├── metric.py
│   │   │   ├── prediction.py
│   │   │   └── alert.py
│   │   ├── schemas/
│   │   │   ├── __init__.py
│   │   │   ├── campaign.py            # Pydantic schemas
│   │   │   ├── analytics.py
│   │   │   └── recommendation.py
│   │   ├── api/
│   │   │   ├── __init__.py
│   │   │   ├── campaigns.py           # Campaign CRUD endpoints
│   │   │   ├── analytics.py           # Analytics endpoints
│   │   │   ├── predictions.py         # ML predictions endpoints
│   │   │   ├── recommendations.py     # LLM recommendations
│   │   │   └── alerts.py              # Alert endpoints
│   │   ├── services/
│   │   │   ├── __init__.py
│   │   │   ├── campaign_service.py
│   │   │   ├── analytics_service.py
│   │   │   ├── ml_service.py          # ML model integration
│   │   │   ├── llm_service.py         # LLM integration (recommendations.py)
│   │   │   └── alert_service.py
│   │   ├── ml/
│   │   │   ├── __init__.py
│   │   │   ├── model.pkl              # Trained ML model
│   │   │   ├── predictor.py           # ML prediction logic
│   │   │   └── features.py            # Feature engineering
│   │   ├── scheduler/
│   │   │   ├── __init__.py
│   │   │   └── tasks.py               # Scheduled tasks (hourly refresh)
│   │   └── utils/
│   │       ├── __init__.py
│   │       └── helpers.py
│   ├── requirements.txt
│   └── Dockerfile
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   │   ├── Dashboard.jsx
│   │   │   ├── ChannelComparison.jsx
│   │   │   ├── ROIChart.jsx
│   │   │   ├── MetricsCards.jsx
│   │   │   ├── AIRecommendations.jsx
│   │   │   └── AlertPanel.jsx
│   │   ├── services/
│   │   │   └── api.js                 # API client
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── package.json
│   └── Dockerfile
├── docker-compose.yml
├── .env
└── README.md
```

Ɐ База данных PostgreSQL (схема)

```
-- Campaigns table
CREATE TABLE campaigns (
    id SERIAL PRIMARY KEY,
    campaign_id VARCHAR(50) UNIQUE NOT NULL,
    target_audience VARCHAR(100),
    campaign_goal VARCHAR(100),
    duration VARCHAR(50),
    channel_used VARCHAR(50),
    conversion_rate FLOAT,
    acquisition_cost FLOAT,
    roi FLOAT,
    location VARCHAR(100),
    language VARCHAR(50),
    clicks INTEGER,
    impressions INTEGER,
    engagement_score INTEGER,
    customer_segment VARCHAR(100),
    date DATE,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- ML Predictions table
CREATE TABLE predictions (
    id SERIAL PRIMARY KEY,
    campaign_id VARCHAR(50) REFERENCES campaigns(campaign_id),
    predicted_roi FLOAT,
    confidence_score FLOAT,
    risk_level VARCHAR(20),
    created_at TIMESTAMP DEFAULT NOW()
);

-- LLM Recommendations table
CREATE TABLE recommendations (
    id SERIAL PRIMARY KEY,
    recommendation_text TEXT,
    channel_comparison JSONB,
    budget_allocation JSONB,
    summary TEXT,
    created_at TIMESTAMP DEFAULT NOW()
);

-- Alerts table
CREATE TABLE alerts (
    id SERIAL PRIMARY KEY,
    alert_type VARCHAR(50),        -- 'roi_drop', 'cost_spike', 'low_conversion', 'anomal
    severity VARCHAR(20),           -- 'low', 'medium', 'high', 'critical'
    channel VARCHAR(50),
    message TEXT,
    metric_value FLOAT,
    threshold_value FLOAT,
    is_read BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT NOW()
);

-- Analytics Cache (hourly aggregations)
CREATE TABLE analytics_cache (
    id SERIAL PRIMARY KEY,
    channel VARCHAR(50),
    total_spend FLOAT,
    total_roi FLOAT,
    avg_conversion_rate FLOAT,
    total_clicks INTEGER,
    total_impressions INTEGER,
    period_start TIMESTAMP,
    period_end TIMESTAMP,
    created_at TIMESTAMP DEFAULT NOW()
);
```

🔧 Backend Services (ключевые компоненты)

1. ML Service (`services/ml_service.py`)

```
import pickle
import pandas as pd
import numpy as np
from typing import Dict, List

class MLService:
    def __init__(self, model_path: str = "app/ml/model.pkl"):
        with open(model_path, 'rb') as f:
            self.model = pickle.load(f)

    def prepare_features(self, campaign_data: Dict) -> pd.DataFrame:
        """Prepare features for ML model"""
        df = pd.DataFrame([campaign_data])

        # Feature engineering (based on your ml.ipynb)
        df['CTR'] = df['Clicks'] / df['Impressions']
        df['Engagement_Rate'] = df['Engagement_Score'] / df['Impressions']
        df['log_Impressions'] = np.log1p(df['Impressions'])
        df['log_Clicks'] = np.log1p(df['Clicks'])
        df['log_Engagement_Score'] = np.log1p(df['Engagement_Score'])

        # Extract date features
        df['Date'] = pd.to_datetime(df['Date'])
        df['Year'] = df['Date'].dt.year
        df['Month'] = df['Date'].dt.month
        df['DayOfWeek'] = df['Date'].dt.dayofweek

        return df

    def predict_roi(self, campaign_data: Dict) -> Dict:
        """Predict ROI for campaign"""
        features = self.prepare_features(campaign_data)
        prediction = self.model.predict(features)[^2_0]

        # Calculate confidence and risk level
        confidence = min(0.95, max(0.5, 1 - abs(prediction - campaign_data.get('roi', pre
        risk_level = self._calculate_risk_level(prediction, campaign_data)

        return {
            'predicted_roi': float(prediction),
            'confidence_score': float(confidence),
            'risk_level': risk_level,
            'current_roi': campaign_data.get('roi'),
            'roi_difference': float(prediction - campaign_data.get('roi', 0))
        }

    def _calculate_risk_level(self, predicted_roi: float, campaign_data: Dict) -> str:
        """Calculate risk level based on prediction"""
        if predicted_roi < 1.0:
            return 'high'
        elif predicted_roi < 2.0:
            return 'medium'
        else:
            return 'low'

    def batch_predict(self, campaigns: List[Dict]) -> List[Dict]:
        """Predict ROI for multiple campaigns"""
        return [self.predict_roi(campaign) for campaign in campaigns]
```

2. LLM Service (`services/llm_service.py`)

Используйте ваш `recommendations.py` как базу:

```
import os
import json
import requests
from typing import Dict, Any
from dotenv import load_dotenv

load_dotenv()

class LLMService:
    def __init__(self):
        self.hf_api_key = os.getenv("HF_API_KEY")
        self.hf_api_url = "https://api-inference.huggingface.co/models/Qwen/Qwen2.5-7B-Ir

        self.prompt_template = """
You are an analytics assistant for a marketing dashboard.
You ALWAYS return a strictly valid JSON object.

The user will provide:
- Aggregated analytics data from ad platforms
- ML model results (ROI drivers, forecasts, risk segments)

Your tasks:
1. Compare performance across channels.
2. Identify strengths, weaknesses, anomalies.
3. Analyze ML feature importance.
4. Give recommendations for:
   - budget allocation
   - CPC / CTR / CPM improvements
   - conversion optimization
   - ROI growth
5. Provide a short summary.

Your JSON response MUST follow this structure:
{
  "channel_comparison": {},
  "ml_predictions": {},
  "recommendations": {},
  "summary": ""
}

Rules:
- STRICTLY return valid JSON.
- NO extra text.
- Use ONLY metrics provided by the user.
- Respond in English.
"""

    def generate_recommendations(self, analytics_data: Dict, ml_predictions: Dict) -> Dic
        """Generate AI recommendations based on analytics and ML predictions"""

        user_data = {
            "analytics": analytics_data,
            "ml_predictions": ml_predictions
        }

        user_prompt = json.dumps(user_data, indent=2)

        headers = {
            "Authorization": f"Bearer {self.hf_api_key}",
            "Content-Type": "application/json"
        }

        payload = {
            "inputs": f"{self.prompt_template}\n\nUSER DATA:\n{user_prompt}",
            "parameters": {
                "temperature": 0.2,
                "max_new_tokens": 700
            }
        }

        try:
            response = requests.post(self.hf_api_url, headers=headers, json=payload, time
            response.raise_for_status()
            data = response.json()

            if isinstance(data, list) and "generated_text" in data[^2_0]:
                generated = data[^2_0]["generated_text"]
                # Extract JSON from response
                json_start = generated.find('{')
                json_end = generated.rfind('}') + 1
                if json_start != -1 and json_end > json_start:
                    return json.loads(generated[json_start:json_end])

            return {"error": "Failed to parse LLM response"}

        except Exception as e:
            return {"error": str(e)}
```

3. Alert Service (`services/alert_service.py`)

```
from typing import List, Dict
from datetime import datetime, timedelta
from sqlalchemy.orm import Session
from app.models.alert import Alert
from app.models.campaign import Campaign

class AlertService:
    THRESHOLDS = {
        'roi_drop': 1.5,           # ROI below 1.5
        'cost_spike': 600,         # Acquisition cost above $600
        'low_conversion': 0.05,    # Conversion rate below 5%
        'high_ctr_low_conversion': {'ctr': 0.15, 'conversion': 0.05},
        'engagement_drop': 3       # Engagement score below 3
    }

    def check_alerts(self, db: Session) -> List[Dict]:
        """Check for various alert conditions"""
        alerts = []

        # Get recent campaigns (last 24 hours)
        recent_campaigns = db.query(Campaign).filter(
            Campaign.updated_at >= datetime.now() - timedelta(hours=24)
        ).all()

        for campaign in recent_campaigns:
            # Check ROI drop
            if campaign.roi < self.THRESHOLDS['roi_drop']:
                alerts.append(self._create_alert(
                    alert_type='roi_drop',
                    severity='high',
                    channel=campaign.channel_used,
                    message=f"ROI dropped to {campaign.roi:.2f} for {campaign.channel_use
                    metric_value=campaign.roi,
                    threshold_value=self.THRESHOLDS['roi_drop']
                ))

            # Check cost spike
            if campaign.acquisition_cost > self.THRESHOLDS['cost_spike']:
                alerts.append(self._create_alert(
                    alert_type='cost_spike',
                    severity='medium',
                    channel=campaign.channel_used,
                    message=f"Acquisition cost spiked to ${campaign.acquisition_cost} for
                    metric_value=campaign.acquisition_cost,
                    threshold_value=self.THRESHOLDS['cost_spike']
                ))

            # Check low conversion
            if campaign.conversion_rate < self.THRESHOLDS['low_conversion']:
                alerts.append(self._create_alert(
                    alert_type='low_conversion',
                    severity='medium',
                    channel=campaign.channel_used,
                    message=f"Low conversion rate {campaign.conversion_rate:.2%} for {cam
                    metric_value=campaign.conversion_rate,
                    threshold_value=self.THRESHOLDS['low_conversion']
                ))

            # Check anomaly: high CTR but low conversion
            ctr = campaign.clicks / campaign.impressions if campaign.impressions > 0 else
            if (ctr > self.THRESHOLDS['high_ctr_low_conversion']['ctr'] and
                campaign.conversion_rate < self.THRESHOLDS['high_ctr_low_conversion']['co
                alerts.append(self._create_alert(
                    alert_type='anomaly',
                    severity='high',
                    channel=campaign.channel_used,
                    message=f"High CTR ({ctr:.2%}) but low conversion ({campaign.conversi
                    metric_value=campaign.conversion_rate,
                    threshold_value=self.THRESHOLDS['high_ctr_low_conversion']['conversic
                ))

            # Check engagement drop
            if campaign.engagement_score < self.THRESHOLDS['engagement_drop']:
                alerts.append(self._create_alert(
                    alert_type='engagement_drop',
                    severity='low',
                    channel=campaign.channel_used,
                    message=f"Low engagement score {campaign.engagement_score} for {campa
                    metric_value=campaign.engagement_score,
                    threshold_value=self.THRESHOLDS['engagement_drop']
                ))

        # Save alerts to database
        for alert_data in alerts:
            alert = Alert(**alert_data)
            db.add(alert)

        db.commit()
        return alerts

    def _create_alert(self, alert_type: str, severity: str, channel: str,
                     message: str, metric_value: float, threshold_value: float) -> Dict:
        """Create alert dictionary"""
        return {
            'alert_type': alert_type,
            'severity': severity,
            'channel': channel,
            'message': message,
            'metric_value': metric_value,
            'threshold_value': threshold_value,
            'is_read': False,
            'created_at': datetime.now()
        }

    def get_unread_alerts(self, db: Session) -> List[Alert]:
        """Get all unread alerts"""
        return db.query(Alert).filter(Alert.is_read == False).order_by(Alert.created_at.c

    def mark_alert_read(self, db: Session, alert_id: int):
        """Mark alert as read"""
        alert = db.query(Alert).filter(Alert.id == alert_id).first()
        if alert:
            alert.is_read = True
            db.commit()
```

4. Scheduler (`scheduler/tasks.py`)

Используйте APScheduler для простоты:

```
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.triggers.interval import IntervalTrigger
from sqlalchemy.orm import Session
from app.database import SessionLocal
from app.services.ml_service import MLService
from app.services.llm_service import LLMService
from app.services.alert_service import AlertService
from app.services.analytics_service import AnalyticsService
import logging

logger = logging.getLogger(__name__)

def hourly_refresh_task():
    """Task that runs every hour to refresh predictions and recommendations"""
    logger.info("Starting hourly refresh task...")

    db: Session = SessionLocal()

    try:
        # 1. Refresh ML predictions
        ml_service = MLService()
        analytics_service = AnalyticsService()

        campaigns = analytics_service.get_recent_campaigns(db)
        predictions = ml_service.batch_predict(campaigns)
        analytics_service.save_predictions(db, predictions)

        logger.info(f"Updated {len(predictions)} ML predictions")

        # 2. Generate LLM recommendations
        llm_service = LLMService()
        analytics_data = analytics_service.get_aggregated_analytics(db)
        ml_predictions_data = analytics_service.get_latest_predictions(db)

        recommendations = llm_service.generate_recommendations(analytics_data, ml_predict
        analytics_service.save_recommendations(db, recommendations)

        logger.info("Generated new LLM recommendations")

        # 3. Check for alerts
        alert_service = AlertService()
        alerts = alert_service.check_alerts(db)

        logger.info(f"Generated {len(alerts)} new alerts")

    except Exception as e:
        logger.error(f"Error in hourly refresh task: {str(e)}")

    finally:
        db.close()

def start_scheduler():
    """Start the background scheduler"""
    scheduler = BackgroundScheduler()

    # Run every hour
    scheduler.add_job(
        func=hourly_refresh_task,
        trigger=IntervalTrigger(hours=1),
        id='hourly_refresh',
        name='Hourly data refresh and predictions',
        replace_existing=True
    )

    # Run immediately on startup
    scheduler.add_job(
        func=hourly_refresh_task,
        id='startup_refresh',
        name='Startup refresh',
        replace_existing=True
    )

    scheduler.start()
    logger.info("Scheduler started successfully")
```

Frontend Components (React)

Dashboard Layout

```
// src/components/Dashboard.jsx
import React, { useEffect, useState } from 'react';
import { api } from '../services/api';
import MetricsCards from './MetricsCards';
import ChannelComparison from './ChannelComparison';
import ROIChart from './ROIChart';
import AIRecommendations from './AIRecommendations';
import AlertPanel from './AlertPanel';

function Dashboard() {
  const [analytics, setAnalytics] = useState(null);
  const [recommendations, setRecommendations] = useState(null);
  const [alerts, setAlerts] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchData();
    // Refresh every 5 minutes
    const interval = setInterval(fetchData, 5 * 60 * 1000);
    return () => clearInterval(interval);
  }, []);

  const fetchData = async () => {
    try {
      const [analyticsRes, recommendationsRes, alertsRes] = await Promise.all([
        api.getAnalytics(),
        api.getRecommendations(),
        api.getAlerts()
      ]);

      setAnalytics(analyticsRes.data);
      setRecommendations(recommendationsRes.data);
      setAlerts(alertsRes.data);
    } catch (error) {
      console.error('Error fetching data:', error);
    } finally {
      setLoading(false);
    }
  };

  if (loading) return <div>Loading...</div>;

  return (
    <div className="dashboard">
      <h1>DEMETRA SYSTEMS - AI Analytics Dashboard</h1>

      <AlertPanel alerts={alerts} />

      <MetricsCards analytics={analytics} />

      <div className="charts-row">
        <ChannelComparison data={analytics.channels} />
        <ROIChart data={analytics.roi_trend} />
      </div>

      <AIRecommendations recommendations={recommendations} />
    </div>
  );
}

export default Dashboard;
```

Дополнительные улучшения для MVP

1. Система алертов (расширенная)

Типы алертов:

Тип алерта	Условие	Severity	Действие
`roi_drop`	ROI < 1.5	High	Немедленное уведомление
`cost_spike`	Cost > $600	Medium	Проверить бюджет
`low_conversion`	CR < 5%	Medium	Оптимизировать кампанию
`anomaly_high_ctr_low_conv`	CTR > 15% AND CR < 5%	High	Проверить landing page
`engagement_drop`	Score < 3	Low	Улучшить креативы

Тип алерта	Условие	Severity	Действие
`budget_overrun`	Spend > 90% of budget	Critical	Остановить кампанию
`channel_underperform`	Channel ROI < avg ROI * 0.7	Medium	Перераспределить бюджет

2. Real-time уведомления (Websockets - опционально)

Для MVP можно использовать простой polling (каждые 30 сек), но для продакшна - WebSockets:

```
# backend/app/api/websockets.py
from fastapi import WebSocket, WebSocketDisconnect
from typing import List

class ConnectionManager:
    def __init__(self):
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def broadcast_alert(self, alert: dict):
        for connection in self.active_connections:
            await connection.send_json(alert)

manager = ConnectionManager()

@app.websocket("/ws/alerts")
async def websocket_endpoint(websocket: WebSocket):
    await manager.connect(websocket)
    try:
        while True:
            await websocket.receive_text()
    except WebSocketDisconnect:
        manager.disconnect(websocket)
```

3. Экспорт отчетов (для демо)

```
# backend/app/api/reports.py
from fastapi import APIRouter
from fastapi.responses import FileResponse
import pandas as pd
from datetime import datetime

router = APIRouter()

@router.get("/export/pdf")
async def export_pdf_report(db: Session = Depends(get_db)):
    """Generate PDF report (use reportlab or weasyprint)"""
    # Implementation...
    pass

@router.get("/export/excel")
async def export_excel_report(db: Session = Depends(get_db)):
    """Generate Excel report"""
    analytics = get_analytics_data(db)
    df = pd.DataFrame(analytics)

    filename = f"demetra_report_{datetime.now().strftime('%Y%m%d')}.xlsx"
    filepath = f"/tmp/{filename}"

    df.to_excel(filepath, index=False)
    return FileResponse(filepath, filename=filename)
```

4. Дашборд метрики (KPI cards)

```
// src/components/MetricsCards.jsx
function MetricsCards({ analytics }) {
  const metrics = [
    {
      title: 'Total ROI',
      value: analytics.total_roi.toFixed(2),
      change: analytics.roi_change,
      trend: analytics.roi_change > 0 ? 'up' : 'down'
    },
    {
      title: 'Avg Conversion Rate',
      value: `${(analytics.avg_conversion_rate * 100).toFixed(2)}%`,
      change: analytics.conversion_change,
      trend: analytics.conversion_change > 0 ? 'up' : 'down'
    },
    {
      title: 'Total Clicks',
      value: analytics.total_clicks.toLocaleString(),
      change: analytics.clicks_change,
      trend: analytics.clicks_change > 0 ? 'up' : 'down'
    },
    {
      title: 'Avg Cost per Acquisition',
      value: `$${analytics.avg_acquisition_cost.toFixed(2)}`,
      change: analytics.cost_change,
      trend: analytics.cost_change < 0 ? 'up' : 'down' // lower is better
    }
  ];

  return (
    <div className="metrics-grid">
      {metrics.map((metric, index) => (
        <div key={index} className="metric-card">
          <h3>{metric.title}</h3>
          <div className="metric-value">{metric.value}</div>
          <div className={`metric-change ${metric.trend}`}>
            {metric.trend === 'up' ? '↑' : '↓'} {Math.abs(metric.change).toFixed(1)}%
          </div>
        </div>
      ))}
    </div>
  );
}
```

⚡ План реализации для хакатона (приоритеты)

День 1: Core Backend + DB

1. ✅ Setup FastAPI structure
2. ✅ Create database models and migrations
3. ✅ Implement CRUD endpoints for campaigns
4. ✅ Integrate ML model (pickle load + predict endpoint)
5. ✅ Basic analytics aggregation endpoint

День 2: AI Integration + Frontend

1. ✅ Integrate LLM service (recommendations.py)
2. ✅ Implement alert service with basic rules
3. ✅ Setup React dashboard layout
4. ✅ Create chart components (use recharts or chart.js)
5. ✅ Connect frontend to backend API

День 3: Scheduler + Polish

1. ✅ Implement APScheduler for hourly tasks
2. ✅ Add alert panel in frontend
3. ✅ Add AI recommendations panel
4. ✅ Styling and UX improvements
5. ✅ Testing and bug fixes
6. ✅ Prepare demo data and presentation

MVP Features Checklist

Must Have (P0):

- [x] Display campaign data from CSV/DB
- [x] ML predictions for ROI
- [x] LLM recommendations

- [x] Basic charts (ROI by channel, trends)
- [x] Alert system (3-5 alert types)
- [x] Hourly refresh scheduler

Should Have (P1):

- [ ] Channel comparison view
- [ ] Budget allocation suggestions
- [ ] Export to Excel
- [ ] Alert notifications in UI

Nice to Have (P2):

- [ ] WebSocket real-time updates
- [ ] Advanced filtering
- [ ] Custom date ranges
- [ ] PDF reports

Нужна помощь с конкретной частью реализации? Могу сгенерировать полный код для любого компонента!

⁂

Backend MVP Requirements - DEMETRA SYSTEMS

Исходя из вашей текущей структуры проекта, вот **минимальные требования для MVP** с фокусом на быструю реализацию для хакатона.

▯ Core MVP Backend Features (Must Have)

1. Database Models (`app/models/`)

Создайте 4 основные модели (замените текущую `book.py`):

`campaign.py`

```
from sqlalchemy import Column, Integer, String, Float, Date, DateTime
from sqlalchemy.sql import func
from app.configs.database import Base

class Campaign(Base):
    __tablename__ = "campaigns"

    id = Column(Integer, primary_key=True, index=True)
    campaign_id = Column(String(50), unique=True, index=True)
    target_audience = Column(String(100))
    campaign_goal = Column(String(100))
    duration = Column(String(50))
    channel_used = Column(String(50), index=True)
    conversion_rate = Column(Float)
    acquisition_cost = Column(Float)
    roi = Column(Float)
    location = Column(String(100))
    language = Column(String(50))
    clicks = Column(Integer)
    impressions = Column(Integer)
    engagement_score = Column(Integer)
    customer_segment = Column(String(100))
    date = Column(Date)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())
```

`prediction.py`

```
from sqlalchemy import Column, Integer, String, Float, ForeignKey, DateTime
from sqlalchemy.sql import func
from app.configs.database import Base

class Prediction(Base):
    __tablename__ = "predictions"

    id = Column(Integer, primary_key=True, index=True)
    campaign_id = Column(String(50), ForeignKey("campaigns.campaign_id"))
    predicted_roi = Column(Float)
    confidence_score = Column(Float)
    risk_level = Column(String(20))  # 'low', 'medium', 'high'
    feature_importance = Column(String(500))  # JSON string
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

`recommendation.py`

```
from sqlalchemy import Column, Integer, Text, DateTime
from sqlalchemy.sql import func
from sqlalchemy.dialects.postgresql import JSONB
from app.configs.database import Base

class Recommendation(Base):
    __tablename__ = "recommendations"

    id = Column(Integer, primary_key=True, index=True)
    channel_comparison = Column(JSONB)
    ml_predictions = Column(JSONB)
    recommendations = Column(JSONB)
    summary = Column(Text)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

`alert.py`

```
from sqlalchemy import Column, Integer, String, Float, Boolean, Text, DateTime
from sqlalchemy.sql import func
from app.configs.database import Base

class Alert(Base):
    __tablename__ = "alerts"

    id = Column(Integer, primary_key=True, index=True)
    alert_type = Column(String(50))  # 'roi_drop', 'cost_spike', etc.
    severity = Column(String(20))    # 'low', 'medium', 'high', 'critical'
    channel = Column(String(50))
    message = Column(Text)
    metric_value = Column(Float)
    threshold_value = Column(Float)
    is_read = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

2. Pydantic Schemas (`app/schemas/`)

`campaign.py`

```
from pydantic import BaseModel
from datetime import date, datetime
from typing import Optional

class CampaignBase(BaseModel):
    campaign_id: str
    target_audience: Optional[str] = None
    campaign_goal: Optional[str] = None
    duration: Optional[str] = None
    channel_used: str
    conversion_rate: float
    acquisition_cost: float
    roi: float
    location: Optional[str] = None
    language: Optional[str] = None
    clicks: int
    impressions: int
    engagement_score: int
    customer_segment: Optional[str] = None
    date: date

class CampaignCreate(CampaignBase):
    pass

class CampaignResponse(CampaignBase):
    id: int
    created_at: datetime
    updated_at: Optional[datetime] = None

    class Config:
        from_attributes = True

class ChannelStats(BaseModel):
    channel: str
    total_campaigns: int
    avg_roi: float
    avg_conversion_rate: float
    total_clicks: int
    total_impressions: int
    avg_acquisition_cost: float
```

`analytics.py`

```
from pydantic import BaseModel
from typing import List, Dict

class AnalyticsSummary(BaseModel):
    total_campaigns: int
    total_roi: float
    avg_roi: float
    avg_conversion_rate: float
    total_clicks: int
    total_impressions: int
    avg_acquisition_cost: float
    roi_change: float  # % change
    conversion_change: float
    clicks_change: float
    cost_change: float
    channels: List[Dict]
    top_performing_channel: str
    worst_performing_channel: str

class ROITrend(BaseModel):
    date: str
    roi: float
    channel: str
```

`prediction.py`

```
from pydantic import BaseModel
from datetime import datetime
from typing import Optional, Dict

class PredictionBase(BaseModel):
    campaign_id: str
    predicted_roi: float
    confidence_score: float
    risk_level: str
    feature_importance: Optional[Dict] = None

class PredictionResponse(PredictionBase):
    id: int
    created_at: datetime
    class Config:
        from_attributes = True
```

`recommendation.py`

```
from pydantic import BaseModel
from datetime import datetime
from typing import Dict, Optional

class RecommendationResponse(BaseModel):
    id: int
    channel_comparison: Dict
    ml_predictions: Dict
    recommendations: Dict
    summary: str
    created_at: datetime

    class Config:
        from_attributes = True
```

`alert.py`

```
from pydantic import BaseModel
from datetime import datetime

class AlertResponse(BaseModel):
    id: int
    alert_type: str
    severity: str
    channel: str
    message: str
    metric_value: float
    threshold_value: float
    is_read: bool
    created_at: datetime

    class Config:
        from_attributes = True
```

3. API Routers (`app/routers/`)

Замените `book.py` на следующие роутеры:

`campaigns.py`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List
from app.configs.database import get_db
from app.models.campaign import Campaign
```

```python
from app.schemas.campaign import CampaignCreate, CampaignResponse

router = APIRouter(prefix="/api/campaigns", tags=["campaigns"])

@router.get("/", response_model=List[CampaignResponse])
def get_campaigns(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    campaigns = db.query(Campaign).offset(skip).limit(limit).all()
    return campaigns

@router.get("/{campaign_id}", response_model=CampaignResponse)
def get_campaign(campaign_id: str, db: Session = Depends(get_db)):
    campaign = db.query(Campaign).filter(Campaign.campaign_id == campaign_id).first()
    if not campaign:
        raise HTTPException(status_code=404, detail="Campaign not found")
    return campaign

@router.post("/", response_model=CampaignResponse, status_code=201)
def create_campaign(campaign: CampaignCreate, db: Session = Depends(get_db)):
    db_campaign = Campaign(**campaign.dict())
    db.add(db_campaign)
    db.commit()
    db.refresh(db_campaign)
    return db_campaign

@router.get("/channel/{channel_name}", response_model=List[CampaignResponse])
def get_campaigns_by_channel(channel_name: str, db: Session = Depends(get_db)):
    campaigns = db.query(Campaign).filter(Campaign.channel_used == channel_name).all()
    return campaigns
```

`analytics.py`

```python
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from sqlalchemy import func
from app.configs.database import get_db
from app.models.campaign import Campaign
from app.schemas.analytics import AnalyticsSummary, ROITrend
from app.schemas.campaign import ChannelStats
from typing import List
from datetime import datetime, timedelta

router = APIRouter(prefix="/api/analytics", tags=["analytics"])

@router.get("/summary", response_model=AnalyticsSummary)
def get_analytics_summary(db: Session = Depends(get_db)):
    """Get overall analytics summary"""

    # Current period stats
    total_campaigns = db.query(Campaign).count()
    current_stats = db.query(
        func.avg(Campaign.roi).label('avg_roi'),
        func.sum(Campaign.roi).label('total_roi'),
        func.avg(Campaign.conversion_rate).label('avg_conversion'),
        func.sum(Campaign.clicks).label('total_clicks'),
        func.sum(Campaign.impressions).label('total_impressions'),
        func.avg(Campaign.acquisition_cost).label('avg_cost')
    ).first()

    # Previous period stats (for % change calculation)
    cutoff_date = datetime.now() - timedelta(days=30)
    previous_stats = db.query(
        func.avg(Campaign.roi).label('avg_roi'),
        func.avg(Campaign.conversion_rate).label('avg_conversion'),
        func.sum(Campaign.clicks).label('total_clicks'),
        func.avg(Campaign.acquisition_cost).label('avg_cost')
    ).filter(Campaign.date < cutoff_date.date()).first()

    # Calculate % changes
    roi_change = ((current_stats.avg_roi - previous_stats.avg_roi) / previous_stats.avg_r
    conversion_change = ((current_stats.avg_conversion - previous_stats.avg_conversion) /
    clicks_change = ((current_stats.total_clicks - previous_stats.total_clicks) / previou
    cost_change = ((current_stats.avg_cost - previous_stats.avg_cost) / previous_stats.av

    # Channel stats
    channel_stats = db.query(
        Campaign.channel_used,
        func.avg(Campaign.roi).label('avg_roi')
    ).group_by(Campaign.channel_used).all()

    channels_data = [{"channel": ch, "avg_roi": roi} for ch, roi in channel_stats]
    top_channel = max(channel_stats, key=lambda x: x[1])[0] if channel_stats else "N/A"
    worst_channel = min(channel_stats, key=lambda x: x[1])[0] if channel_stats else "N/A'

    return AnalyticsSummary(
        total_campaigns=total_campaigns,
        total_roi=float(current_stats.total_roi or 0),
        avg_roi=float(current_stats.avg_roi or 0),
        avg_conversion_rate=float(current_stats.avg_conversion or 0),
        total_clicks=int(current_stats.total_clicks or 0),
        total_impressions=int(current_stats.total_impressions or 0),
        avg_acquisition_cost=float(current_stats.avg_cost or 0),
        roi_change=float(roi_change),
        conversion_change=float(conversion_change),
        clicks_change=float(clicks_change),
        cost_change=float(cost_change),
        channels=channels_data,
        top_performing_channel=top_channel,
        worst_performing_channel=worst_channel
    )

@router.get("/channels", response_model=List[ChannelStats])
def get_channel_stats(db: Session = Depends(get_db)):
    """Get statistics grouped by channel"""

    stats = db.query(
        Campaign.channel_used,
        func.count(Campaign.id).label('total_campaigns'),
        func.avg(Campaign.roi).label('avg_roi'),
        func.avg(Campaign.conversion_rate).label('avg_conversion'),
        func.sum(Campaign.clicks).label('total_clicks'),
        func.sum(Campaign.impressions).label('total_impressions'),
        func.avg(Campaign.acquisition_cost).label('avg_cost')
    ).group_by(Campaign.channel_used).all()

    return [
        ChannelStats(
            channel=stat.channel_used,
            total_campaigns=stat.total_campaigns,
            avg_roi=float(stat.avg_roi),
            avg_conversion_rate=float(stat.avg_conversion),
            total_clicks=int(stat.total_clicks),
            total_impressions=int(stat.total_impressions),
            avg_acquisition_cost=float(stat.avg_cost)
        )
        for stat in stats
    ]

@router.get("/roi-trend", response_model=List[ROITrend])
def get_roi_trend(db: Session = Depends(get_db)):
    """Get ROI trend over time"""

    trends = db.query(
        Campaign.date,
        Campaign.channel_used,
        func.avg(Campaign.roi).label('avg_roi')
    ).group_by(Campaign.date, Campaign.channel_used).order_by(Campaign.date).all()

    return [
        ROITrend(
            date=str(trend.date),
            roi=float(trend.avg_roi),
            channel=trend.channel_used
        )
        for trend in trends
    ]
```

predictions.py

```
from fastapi import APIRouter, Depends, BackgroundTasks
from sqlalchemy.orm import Session
from typing import List
from app.configs.database import get_db
from app.models.prediction import Prediction
from app.schemas.prediction import PredictionResponse
from app.services.ml_service import MLService

router = APIRouter(prefix="/api/predictions", tags=["predictions"])

@router.get("/", response_model=List[PredictionResponse])
def get_predictions(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    predictions = db.query(Prediction).order_by(Prediction.created_at.desc()).offset(skip
    return predictions

@router.get("/latest", response_model=List[PredictionResponse])
def get_latest_predictions(db: Session = Depends(get_db)):
    """Get latest predictions for each campaign"""
    # Simplified: just get most recent 50
    predictions = db.query(Prediction).order_by(Prediction.created_at.desc()).limit(50).a
    return predictions

@router.post("/generate")
def generate_predictions(background_tasks: BackgroundTasks, db: Session = Depends(get_db)
    """Trigger ML prediction generation for all campaigns"""
    ml_service = MLService()

    # Run in background
    from app.services.analytics_service import AnalyticsService
    analytics_service = AnalyticsService()
    background_tasks.add_task(analytics_service.generate_all_predictions, db)

    return {"message": "Prediction generation started"}
```

`recommendations.py`

```
from fastapi import APIRouter, Depends, BackgroundTasks
from sqlalchemy.orm import Session
from app.configs.database import get_db
from app.models.recommendation import Recommendation
from app.schemas.recommendation import RecommendationResponse

router = APIRouter(prefix="/api/recommendations", tags=["recommendations"])

@router.get("/latest", response_model=RecommendationResponse)
def get_latest_recommendation(db: Session = Depends(get_db)):
    """Get the most recent AI recommendation"""
    recommendation = db.query(Recommendation).order_by(Recommendation.created_at.desc()).
    if not recommendation:
        return {"message": "No recommendations yet"}
    return recommendation

@router.post("/generate")
def generate_recommendation(background_tasks: BackgroundTasks, db: Session = Depends(get_
    """Trigger LLM recommendation generation"""
    from app.services.analytics_service import AnalyticsService
    analytics_service = AnalyticsService()
    background_tasks.add_task(analytics_service.generate_recommendations, db)

    return {"message": "Recommendation generation started"}
```

`alerts.py`

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from typing import List
from app.configs.database import get_db
from app.models.alert import Alert
from app.schemas.alert import AlertResponse

router = APIRouter(prefix="/api/alerts", tags=["alerts"])
```

```
@router.get("/", response_model=List[AlertResponse])
def get_all_alerts(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    alerts = db.query(Alert).order_by(Alert.created_at.desc()).offset(skip).limit(limit).
    return alerts

@router.get("/unread", response_model=List[AlertResponse])
def get_unread_alerts(db: Session = Depends(get_db)):
    """Get all unread alerts"""
    alerts = db.query(Alert).filter(Alert.is_read == False).order_by(Alert.created_at.des
    return alerts

@router.patch("/{alert_id}/read")
def mark_alert_as_read(alert_id: int, db: Session = Depends(get_db)):
    """Mark an alert as read"""
    alert = db.query(Alert).filter(Alert.id == alert_id).first()
    if alert:
        alert.is_read = True
        db.commit()
        return {"message": "Alert marked as read"}
    return {"error": "Alert not found"}

@router.post("/check")
def check_for_alerts(db: Session = Depends(get_db)):
    """Manually trigger alert checking"""
    from app.services.alert_service import AlertService
    alert_service = AlertService()
    alerts = alert_service.check_alerts(db)
    return {"message": f"Generated {len(alerts)} new alerts"}
```

4. Services (`app/services/`)

`ml_service.py`

```
import pickle
import pandas as pd
import numpy as np
from typing import Dict, List
import os

class MLService:
    def __init__(self):
        model_path = os.path.join(os.path.dirname(__file__), "../../MML/model.pkl")
        if os.path.exists(model_path):
            with open(model_path, 'rb') as f:
                self.model = pickle.load(f)
        else:
            self.model = None
            print("Warning: ML model not found. Predictions will be mock.")

    def prepare_features(self, campaign: Dict) -> pd.DataFrame:
        """Prepare features for ML model (based on your ml.ipynb)"""
        df = pd.DataFrame([campaign])
        # Feature engineering
        df['CTR'] = df['clicks'] / df['impressions'] if df['impressions'].iloc[0] > 0 els
        df['Engagement_Rate'] = df['engagement_score'] / df['impressions'] if df['impress
        df['log_Impressions'] = np.log1p(df['impressions'])
        df['log_Clicks'] = np.log1p(df['clicks'])
        df['log_Engagement_Score'] = np.log1p(df['engagement_score'])

        # Date features
        if 'date' in df.columns:
            df['date'] = pd.to_datetime(df['date'])
            df['Year'] = df['date'].dt.year
            df['Month'] = df['date'].dt.month
            df['DayOfWeek'] = df['date'].dt.dayofweek

        return df

    def predict_roi(self, campaign_data: Dict) -> Dict:
        """Predict ROI for a campaign"""
        if self.model is None:
            # Mock prediction for MVP
            current_roi = campaign_data.get('roi', 2.0)
            predicted_roi = current_roi * (1 + np.random.uniform(-0.1, 0.2))
            confidence = np.random.uniform(0.7, 0.95)
        else:
            features = self.prepare_features(campaign_data)
            predicted_roi = self.model.predict(features)[0]
            confidence = 0.85  # Simplified

        risk_level = 'low' if predicted_roi > 2.0 else ('medium' if predicted_roi > 1.0 e

        return {
            'campaign_id': campaign_data.get('campaign_id'),
            'predicted_roi': float(predicted_roi),
            'confidence_score': float(confidence),
            'risk_level': risk_level,
            'feature_importance': {}  # Add if needed
        }
```

`llm_service.py`

```
import os
import json
import requests
from typing import Dict
from dotenv import load_dotenv

load_dotenv()

class LLMService:
    def __init__(self):
        self.hf_api_key = os.getenv("HF_API_KEY")
        self.hf_api_url = "https://api-inference.huggingface.co/models/Qwen/Qwen2.5-7B-In

    def generate_recommendations(self, analytics_data: Dict, ml_predictions: Dict) -> Dic
```

```
        """Generate AI recommendations"""

        prompt = f"""
You are a marketing analytics AI assistant. Analyze the following data and provide recomm

Analytics Data:
{json.dumps(analytics_data, indent=2)}

ML Predictions:
{json.dumps(ml_predictions, indent=2)}

Provide a JSON response with:
1. channel_comparison: Compare channel performance
2. ml_predictions: Interpret ML predictions
3. recommendations: Actionable recommendations for budget allocation and optimization
4. summary: Brief summary

Respond ONLY with valid JSON.
"""

        if not self.hf_api_key:
            # Mock response for MVP without API key
            return self._mock_recommendation(analytics_data)

        headers = {
            "Authorization": f"Bearer {self.hf_api_key}",
            "Content-Type": "application/json"
        }

        payload = {
            "inputs": prompt,
            "parameters": {"temperature": 0.3, "max_new_tokens": 600}
        }

        try:
            response = requests.post(self.hf_api_url, headers=headers, json=payload, time
            response.raise_for_status()
            data = response.json()

            if isinstance(data, list) and "generated_text" in data[0]:
                text = data[0]["generated_text"]
                # Extract JSON
                json_start = text.find('{')
                json_end = text.rfind('}') + 1
                if json_start != -1 and json_end > json_start:
                    return json.loads(text[json_start:json_end])

            return self._mock_recommendation(analytics_data)

        except Exception as e:
            print(f"LLM Error: {e}")
            return self._mock_recommendation(analytics_data)

    def _mock_recommendation(self, analytics_data: Dict) -> Dict:
        """Fallback mock recommendation for MVP"""
        return {
            "channel_comparison": {
                "top_channel": analytics_data.get('top_performing_channel', 'Instagram'),
                "analysis": "Instagram shows highest ROI. Facebook needs optimization."
            },
            "ml_predictions": {
                "trend": "ROI expected to increase by 15% with current strategy"
            },
            "recommendations": {
                "budget_allocation": f"Increase budget for {analytics_data.get('top_perfo
                "optimization": "Reduce spend on underperforming channels"
            },
            "summary": "Overall performance is positive. Focus on top channels and optimi
        }
```

alert_service.py

```
from typing import List, Dict
from datetime import datetime, timedelta
from sqlalchemy.orm import Session
from app.models.alert import Alert
from app.models.campaign import Campaign

class AlertService:
    THRESHOLDS = {
        'roi_drop': 1.5,
        'cost_spike': 600,
        'low_conversion': 0.05,
        'engagement_drop': 3
    }

    def check_alerts(self, db: Session) -> List[Dict]:
        """Check for alert conditions"""
        alerts = []

        # Get recent campaigns
        recent = db.query(Campaign).filter(
            Campaign.updated_at >= datetime.now() - timedelta(hours=24)
        ).all()

        for campaign in recent:
            # ROI drop
            if campaign.roi < self.THRESHOLDS['roi_drop']:
                alerts.append(self._create_alert(
                    'roi_drop', 'high', campaign.channel_used,
                    f"ROI dropped to {campaign.roi:.2f} for {campaign.channel_used}",
                    campaign.roi, self.THRESHOLDS['roi_drop']
                ))

            # Cost spike
            if campaign.acquisition_cost > self.THRESHOLDS['cost_spike']:
                alerts.append(self._create_alert(
                    'cost_spike', 'medium', campaign.channel_used,
                    f"Cost spiked to ${campaign.acquisition_cost} for {campaign.channel_u
                    campaign.acquisition_cost, self.THRESHOLDS['cost_spike']
                ))
            # Low conversion
            if campaign.conversion_rate < self.THRESHOLDS['low_conversion']:
                alerts.append(self._create_alert(
                    'low_conversion', 'medium', campaign.channel_used,
                    f"Low conversion {campaign.conversion_rate:.2%} for {campaign.channel
                    campaign.conversion_rate, self.THRESHOLDS['low_conversion']
                ))

        # Save to DB
        for alert_data in alerts:
            alert = Alert(**alert_data)
            db.add(alert)
        db.commit()

        return alerts

    def _create_alert(self, alert_type, severity, channel, message, value, threshold):
        return {
            'alert_type': alert_type,
            'severity': severity,
            'channel': channel,
            'message': message,
            'metric_value': value,
            'threshold_value': threshold,
            'is_read': False
        }
```

`analytics_service.py`

```
from sqlalchemy.orm import Session
from app.models.campaign import Campaign
from app.models.prediction import Prediction
from app.models.recommendation import Recommendation
from app.services.ml_service import MLService
from app.services.llm_service import LLMService
from typing import List, Dict

class AnalyticsService:
    def get_recent_campaigns(self, db: Session) -> List[Dict]:
        """Get recent campaigns for prediction"""
        campaigns = db.query(Campaign).limit(100).all()
        return [
            {
                'campaign_id': c.campaign_id,
                'clicks': c.clicks,
                'impressions': c.impressions,
                'engagement_score': c.engagement_score,
                'conversion_rate': c.conversion_rate,
                'roi': c.roi,
                'date': c.date
            }
            for c in campaigns
        ]
    def generate_all_predictions(self, db: Session):
        """Generate predictions for all campaigns"""
        ml_service = MLService()
        campaigns = self.get_recent_campaigns(db)

        for campaign_data in campaigns:
            prediction = ml_service.predict_roi(campaign_data)
            db_prediction = Prediction(**prediction)
            db.add(db_prediction)

        db.commit()

    def generate_recommendations(self, db: Session):
        """Generate LLM recommendations"""
        llm_service = LLMService()

        # Get analytics data
        from app.routers.analytics import get_analytics_summary
        analytics = get_analytics_summary(db)

        # Get recent predictions
        predictions = db.query(Prediction).order_by(Prediction.created_at.desc()).limit(1
        ml_data = {
            'recent_predictions': [
                {'campaign_id': p.campaign_id, 'predicted_roi': p.predicted_roi, 'risk':
                for p in predictions
            ]
        }

        # Generate recommendations
        result = llm_service.generate_recommendations(analytics.dict(), ml_data)

        # Save to DB
        rec = Recommendation(
            channel_comparison=result.get('channel_comparison', {}),
            ml_predictions=result.get('ml_predictions', {}),
            recommendations=result.get('recommendations', {}),
            summary=result.get('summary', '')
        )
        db.add(rec)
        db.commit()
```

5. Scheduler (`app/scheduler/`)

Create `tasks.py`:

```
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.triggers.interval import IntervalTrigger
from app.configs.database import SessionLocal
from app.services.analytics_service import AnalyticsService
from app.services.alert_service import AlertService
import logging

logger = logging.getLogger(__name__)
```

```python
def hourly_refresh():
    """Hourly refresh task"""
    logger.info("Starting hourly refresh...")
    db = SessionLocal()

    try:
        analytics_service = AnalyticsService()
        alert_service = AlertService()

        # Generate predictions
        analytics_service.generate_all_predictions(db)
        logger.info("Predictions updated")

        # Generate recommendations
        analytics_service.generate_recommendations(db)
        logger.info("Recommendations updated")

        # Check alerts
        alerts = alert_service.check_alerts(db)
        logger.info(f"Generated {len(alerts)} alerts")

    except Exception as e:
        logger.error(f"Error in hourly refresh: {e}")
    finally:
        db.close()

def start_scheduler():
    """Start background scheduler"""
    scheduler = BackgroundScheduler()

    # Every hour
    scheduler.add_job(
        hourly_refresh,
        trigger=IntervalTrigger(hours=1),
        id='hourly_refresh',
        replace_existing=True
    )

    # Run on startup
    scheduler.add_job(hourly_refresh, id='startup_refresh')

    scheduler.start()
    logger.info("Scheduler started")
```

6. Update `main.py`

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.configs.database import engine, Base
from app.routers import campaigns, analytics, predictions, recommendations, alerts
from app.scheduler.tasks import start_scheduler
import logging
```

```
# Create tables
Base.metadata.create_all(bind=engine)

app = FastAPI(title="DEMETRA SYSTEMS - AI Analytics API", version="1.0.0")

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include routers
app.include_router(campaigns.router)
app.include_router(analytics.router)
app.include_router(predictions.router)
app.include_router(recommendations.router)
app.include_router(alerts.router)

# Start scheduler
@app.on_event("startup")
def startup_event():
    logging.basicConfig(level=logging.INFO)
    start_scheduler()
    logging.info("Application started")

@app.get("/")
def read_root():
    return {"message": "DEMETRA SYSTEMS AI Analytics API", "version": "1.0.0"}

@app.get("/health")
def health_check():
    return {"status": "healthy"}
```

7. Data Seeder (`app/utils/seeder.py`)

```
import pandas as pd
from sqlalchemy.orm import Session
from app.models.campaign import Campaign
from app.configs.database import SessionLocal
import os

def seed_campaigns_from_csv():
    """Load CSV data into database"""
    db = SessionLocal()

    csv_path = os.path.join(os.path.dirname(__file__), "../../MML/data/Social_Media_Adver

    if not os.path.exists(csv_path):
        print("CSV file not found!")
        return
    df = pd.read_csv(csv_path)
    df['Date'] = pd.to_datetime(df['Date'])

    for _, row in df.iterrows():
        campaign = Campaign(
            campaign_id=str(row['Campaign_ID']),
            target_audience=row['Target_Audience'],
            campaign_goal=row['Campaign_Goal'],
            duration=row['Duration'],
            channel_used=row['Channel_Used'],
            conversion_rate=float(row['Conversion_Rate']),
            acquisition_cost=float(row['Acquisition_Cost'].replace('$', '').replace(',',
            roi=float(row['ROI']),
            location=row['Location'],
            language=row['Language'],
            clicks=int(row['Clicks']),
            impressions=int(row['Impressions']),
            engagement_score=int(row['Engagement_Score']),
            customer_segment=row['Customer_Segment'],
            date=row['Date'].date()
        )
        db.add(campaign)

    db.commit()
    db.close()
    print(f"Seeded {len(df)} campaigns!")

if __name__ == "__main__":
    seed_campaigns_from_csv()
```

pyproject.toml Dependencies

Add these to your `pyproject.toml`:

```
[tool.poetry.dependencies]
python = "^3.11"
fastapi = "^0.104.0"
uvicorn = {extras = ["standard"], version = "^0.24.0"}
sqlalchemy = "^2.0.0"
psycopg2-binary = "^2.9.9"
pydantic = "^2.4.0"
python-dotenv = "^1.0.0"
pandas = "^2.1.0"
numpy = "^1.26.0"
scikit-learn = "^1.3.0"
requests = "^2.31.0"
apscheduler = "^3.10.4"
```

▯ MVP Implementation Checklist

Phase 1 (Day 1):

- [ ] Create all 4 models (campaign, prediction, recommendation, alert)
- [ ] Create all schemas
- [ ] Implement campaigns router (CRUD)
- [ ] Implement analytics router (summary, channels, roi-trend)
- [ ] Run seeder to load CSV data

Phase 2 (Day 2):

- [ ] Implement ML service (use your ml.ipynb model)
- [ ] Implement LLM service (use your recommendations.py)
- [ ] Implement predictions router
- [ ] Implement recommendations router
- [ ] Test ML + LLM integration

Phase 3 (Day 3):

- [ ] Implement alert service
- [ ] Implement alerts router
- [ ] Implement scheduler (APScheduler)
- [ ] Test hourly refresh
- [ ] Final testing + bug fixes

Это минимально необходимый backend для MVP. Все готово к интеграции с вашими ML и LLM моделями! Нужна помощь с конкретной частью?

▯ MINIMAL MVP BACKEND - 6 HOURS LEFT

✅ Postgres is PERFECT for MVP

- **YES, use Postgres** - structured data, ACID compliance, easy queries
- NoSQL adds complexity you DON'T need for hackathon
- Your data is relational (campaigns → predictions) = Postgres wins

✅ APScheduler is PERFECT

- **NO Celery needed** - APScheduler is simpler, faster setup
- Celery = overkill for hourly tasks
- APScheduler = 5 lines of code, works instantly

EXACT IMPLEMENTATION STEPS (6 HOURS)

HOUR 1: Database Setup

Step 1: Create Models (`app/models/`)

`campaign.py`:

```
from sqlalchemy import Column, Integer, String, Float, DateTime, Date
from sqlalchemy.sql import func
from app.configs.database import Base

class Campaign(Base):
    __tablename__ = "campaigns"
    id = Column(Integer, primary_key=True)
    campaign_id = Column(String(50), unique=True, index=True)
    channel = Column(String(50), index=True)  # facebook, instagram, google_ads, tiktok
    conversion_rate = Column(Float)
    cost = Column(Float)
    roi = Column(Float)
    clicks = Column(Integer)
    impressions = Column(Integer)
    engagement = Column(Integer)
    date = Column(Date)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

`prediction.py`:

```
from sqlalchemy import Column, Integer, String, Float, DateTime
from sqlalchemy.sql import func
from app.configs.database import Base

class Prediction(Base):
    __tablename__ = "predictions"
    id = Column(Integer, primary_key=True)
    campaign_id = Column(String(50))
    predicted_roi = Column(Float)
    risk_level = Column(String(20))
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

`recommendation.py`:

```
from sqlalchemy import Column, Integer, Text, DateTime
from sqlalchemy.sql import func
from sqlalchemy.dialects.postgresql import JSONB
from app.configs.database import Base

class Recommendation(Base):
    __tablename__ = "recommendations"
    id = Column(Integer, primary_key=True)
    recommendations = Column(JSONB)
    summary = Column(Text)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

`ml_dashboard.py`:

```
from sqlalchemy import Column, Integer, String, DateTime, LargeBinary
from sqlalchemy.sql import func
from app.configs.database import Base

class MLDashboard(Base):
    __tablename__ = "ml_dashboards"
    id = Column(Integer, primary_key=True)
    dashboard_type = Column(String(50))  # 'roi_analysis', 'channel_comparison'
    image_data = Column(LargeBinary)  # PNG bytes
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

Step 2: Alembic Migration

```
# In backend folder
poetry add alembic

# Initialize alembic
alembic init alembic

# Edit alembic.ini - set sqlalchemy.url to your postgres connection
# sqlalchemy.url = postgresql://user:password@localhost:5432/demetra_db

# Edit alembic/env.py - add these lines
from app.configs.database import Base
from app.models.campaign import Campaign
from app.models.prediction import Prediction
from app.models.recommendation import Recommendation
from app.models.ml_dashboard import MLDashboard

target_metadata = Base.metadata

# Create migration
alembic revision --autogenerate -m "initial tables"

# Apply migration
alembic upgrade head
```

HOUR 2: Mock API Endpoints (Faker)

Step 3: Create Mock API Service (`app/services/mock_api.py`)

```
from faker import Faker
import random
from datetime import datetime, timedelta
from typing import List, Dict

fake = Faker()

class MockAPIService:
    """Simulates real API responses from ad platforms"""

    CHANNELS = ['facebook', 'instagram', 'google_ads', 'tiktok']

    @staticmethod
    def fetch_facebook_ads() -> List[Dict]:
        """Mock Facebook Ads API response"""
        campaigns = []
        for _ in range(random.randint(3, 8)):
            campaigns.append({
                'campaign_id': f'FB_{fake.uuid4()[:8]}',
                'channel': 'facebook',
                'conversion_rate': round(random.uniform(0.01, 0.20), 3),
                'cost': round(random.uniform(100, 800), 2),
                'roi': round(random.uniform(0.5, 8.0), 2),
                'clicks': random.randint(100, 5000),
                'impressions': random.randint(1000, 50000),
                'engagement': random.randint(50, 1000),
                'date': (datetime.now() - timedelta(days=random.randint(0, 7))).date()
            })
        return campaigns

    @staticmethod
    def fetch_instagram_ads() -> List[Dict]:
        """Mock Instagram Ads API response"""
        campaigns = []
        for _ in range(random.randint(3, 8)):
            campaigns.append({
                'campaign_id': f'IG_{fake.uuid4()[:8]}',
                'channel': 'instagram',
                'conversion_rate': round(random.uniform(0.01, 0.20), 3),
                'cost': round(random.uniform(100, 800), 2),
                'roi': round(random.uniform(0.5, 8.0), 2),
                'clicks': random.randint(100, 5000),
                'impressions': random.randint(1000, 50000),
                'engagement': random.randint(50, 1000),
                'date': (datetime.now() - timedelta(days=random.randint(0, 7))).date()
            })
        return campaigns

    @staticmethod
    def fetch_google_ads() -> List[Dict]:
        """Mock Google Ads API response"""
        campaigns = []
        for _ in range(random.randint(3, 8)):
            campaigns.append({
                'campaign_id': f'GA_{fake.uuid4()[:8]}',
                'channel': 'google_ads',
                'conversion_rate': round(random.uniform(0.01, 0.20), 3),
                'cost': round(random.uniform(100, 800), 2),
                'roi': round(random.uniform(0.5, 8.0), 2),
                'clicks': random.randint(100, 5000),
                'impressions': random.randint(1000, 50000),
                'engagement': random.randint(50, 1000),
                'date': (datetime.now() - timedelta(days=random.randint(0, 7))).date()
            })
        return campaigns

    @staticmethod
    def fetch_tiktok_ads() -> List[Dict]:
        """Mock TikTok Ads API response"""
        campaigns = []
        for _ in range(random.randint(3, 8)):
            campaigns.append({
                'campaign_id': f'TT_{fake.uuid4()[:8]}',
                'channel': 'tiktok',
                'conversion_rate': round(random.uniform(0.01, 0.20), 3),
                'cost': round(random.uniform(100, 800), 2),
                'roi': round(random.uniform(0.5, 8.0), 2),
                'clicks': random.randint(100, 5000),
                'impressions': random.randint(1000, 50000),
                'engagement': random.randint(50, 1000),
                'date': (datetime.now() - timedelta(days=random.randint(0, 7))).date()
            })
        return campaigns

    @classmethod
    def fetch_all_channels(cls) -> List[Dict]:
        """Fetch data from all channels"""
        all_campaigns = []
        all_campaigns.extend(cls.fetch_facebook_ads())
        all_campaigns.extend(cls.fetch_instagram_ads())
        all_campaigns.extend(cls.fetch_google_ads())
        all_campaigns.extend(cls.fetch_tiktok_ads())
        return all_campaigns
```

Step 4: Data Sync Service (`app/services/sync_service.py`)

```
from sqlalchemy.orm import Session
from app.models.campaign import Campaign
from app.services.mock_api import MockAPIService
from typing import List, Dict

class SyncService:
    """Syncs data from mock APIs to database"""

    @staticmethod
    def sync_campaigns(db: Session) -> int:
        """Fetch from mock APIs and save to DB"""
        # Fetch from all channels
        campaigns_data = MockAPIService.fetch_all_channels()
        count = 0
        for data in campaigns_data:
            # Check if campaign already exists
            existing = db.query(Campaign).filter(
                Campaign.campaign_id == data['campaign_id']
            ).first()

            if not existing:
                # Create new campaign
                campaign = Campaign(**data)
                db.add(campaign)
                count += 1

        db.commit()
        return count
```

HOUR 3: ML Dashboard Generation

Step 5: ML Dashboard Service (`app/services/ml_dashboard_service.py`)

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from io import BytesIO
from sqlalchemy.orm import Session
from app.models.campaign import Campaign
from app.models.ml_dashboard import MLDashboard

class MLDashboardService:

    @staticmethod
    def generate_roi_chart(db: Session) -> bytes:
        """Generate ROI comparison chart"""
        campaigns = db.query(Campaign).all()
        df = pd.DataFrame([{
            'channel': c.channel,
            'roi': c.roi,
            'clicks': c.clicks,
            'cost': c.cost
        } for c in campaigns])

        plt.figure(figsize=(10, 6))
        sns.barplot(data=df, x='channel', y='roi', estimator='mean')
        plt.title('Average ROI by Channel')
        plt.xlabel('Channel')
        plt.ylabel('ROI')

        # Save to bytes
        buf = BytesIO()
        plt.savefig(buf, format='png', dpi=150, bbox_inches='tight')
        plt.close()
        buf.seek(0)
        return buf.read()

    @staticmethod
    def generate_conversion_chart(db: Session) -> bytes:
        """Generate conversion rate chart"""
        campaigns = db.query(Campaign).all()
        df = pd.DataFrame([{
            'channel': c.channel,
            'conversion_rate': c.conversion_rate
        } for c in campaigns])

        plt.figure(figsize=(10, 6))
        sns.boxplot(data=df, x='channel', y='conversion_rate')
        plt.title('Conversion Rate by Channel')
        plt.xlabel('Channel')
        plt.ylabel('Conversion Rate')

        buf = BytesIO()
        plt.savefig(buf, format='png', dpi=150, bbox_inches='tight')
        plt.close()
        buf.seek(0)
        return buf.read()

    @staticmethod
    def save_dashboards(db: Session):
        """Generate and save all dashboard charts"""
        # ROI Chart
        roi_img = MLDashboardService.generate_roi_chart(db)
        roi_dashboard = MLDashboard(
            dashboard_type='roi_analysis',
            image_data=roi_img
        )
        db.add(roi_dashboard)

        # Conversion Chart
        conv_img = MLDashboardService.generate_conversion_chart(db)
        conv_dashboard = MLDashboard(
            dashboard_type='conversion_analysis',
            image_data=conv_img
        )
        db.add(conv_dashboard)

        db.commit()
```

HOUR 4: Scheduler + Routers

Step 6: Scheduler (`app/scheduler/tasks.py`)

```
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.triggers.interval import IntervalTrigger
from app.configs.database import SessionLocal
from app.services.sync_service import SyncService
from app.services.ml_dashboard_service import MLDashboardService
from app.services.ml_service import MLService
from app.services.llm_service import LLMService
import logging

logger = logging.getLogger(__name__)

def hourly_sync():
    """Runs every hour"""
    logger.info("Starting hourly sync...")
    db = SessionLocal()

    try:
        # 1. Sync campaigns from mock APIs
        count = SyncService.sync_campaigns(db)
        logger.info(f"Synced {count} new campaigns")

        # 2. Generate ML predictions
        ml_service = MLService()
        campaigns = db.query(Campaign).limit(50).all()
        for campaign in campaigns:
            prediction = ml_service.predict_roi({
                'campaign_id': campaign.campaign_id,
                'clicks': campaign.clicks,
                'impressions': campaign.impressions,
                'engagement': campaign.engagement,
                'roi': campaign.roi
            })
            db.add(Prediction(**prediction))
        db.commit()
        logger.info("ML predictions generated")

        # 3. Generate LLM recommendations
        llm_service = LLMService()
        analytics = get_analytics_summary(db)
        recommendations = llm_service.generate_recommendations(analytics)
        db.add(Recommendation(**recommendations))
        db.commit()
        logger.info("LLM recommendations generated")

        # 4. Generate ML dashboards (Seaborn charts)
        MLDashboardService.save_dashboards(db)
        logger.info("ML dashboards generated")

    except Exception as e:
        logger.error(f"Error: {e}")
    finally:
        db.close()

def start_scheduler():
    scheduler = BackgroundScheduler()
    scheduler.add_job(
        hourly_sync,
        trigger=IntervalTrigger(hours=1),
        id='hourly_sync',
        replace_existing=True
    )
    scheduler.add_job(hourly_sync, id='startup')  # Run immediately
    scheduler.start()
    logger.info("Scheduler started")
```

Step 7: API Routers

`app/routers/campaigns.py`:

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.configs.database import get_db
from app.models.campaign import Campaign

router = APIRouter(prefix="/api/campaigns", tags=["campaigns"])

@router.get("/")
def get_campaigns(db: Session = Depends(get_db)):
    return db.query(Campaign).order_by(Campaign.created_at.desc()).limit(100).all()

@router.get("/channels")
def get_by_channel(db: Session = Depends(get_db)):
    from sqlalchemy import func
    stats = db.query(
        Campaign.channel,
        func.count(Campaign.id).label('count'),
        func.avg(Campaign.roi).label('avg_roi'),
        func.avg(Campaign.conversion_rate).label('avg_conv')
    ).group_by(Campaign.channel).all()

    return [{'channel': s[0], 'count': s[1], 'avg_roi': float(s[2]), 'avg_conversion': fl
```

`app/routers/dashboards.py`:

```
from fastapi import APIRouter, Depends, Response
from sqlalchemy.orm import Session
from app.configs.database import get_db
from app.models.ml_dashboard import MLDashboard

router = APIRouter(prefix="/api/dashboards", tags=["dashboards"])

@router.get("/roi-chart")
def get_roi_chart(db: Session = Depends(get_db)):
    """Returns PNG image of ROI chart"""
    dashboard = db.query(MLDashboard).filter(
        MLDashboard.dashboard_type == 'roi_analysis'
    ).order_by(MLDashboard.created_at.desc()).first()

    if dashboard:
        return Response(content=dashboard.image_data, media_type="image/png")
    return {"error": "No dashboard found"}

@router.get("/conversion-chart")
def get_conversion_chart(db: Session = Depends(get_db)):
    """Returns PNG image of conversion chart"""
    dashboard = db.query(MLDashboard).filter(
        MLDashboard.dashboard_type == 'conversion_analysis'
    ).order_by(MLDashboard.created_at.desc()).first()

    if dashboard:
        return Response(content=dashboard.image_data, media_type="image/png")
    return {"error": "No dashboard found"}
```

`app/routers/recommendations.py`:

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.configs.database import get_db
from app.models.recommendation import Recommendation

router = APIRouter(prefix="/api/recommendations", tags=["recommendations"])

@router.get("/latest")
def get_latest(db: Session = Depends(get_db)):
    rec = db.query(Recommendation).order_by(Recommendation.created_at.desc()).first()
    return rec if rec else {"message": "No recommendations yet"}
```

Step 8: Update `main.py`

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.configs.database import engine, Base
from app.routers import campaigns, dashboards, recommendations
from app.scheduler.tasks import start_scheduler

Base.metadata.create_all(bind=engine)

app = FastAPI(title="DEMETRA AI Analytics API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(campaigns.router)
app.include_router(dashboards.router)
app.include_router(recommendations.router)

@app.on_event("startup")
def startup():
    start_scheduler()

@app.get("/")
```

```
def root():
    return {"message": "DEMETRA AI Analytics API", "status": "running"}
```

HOUR 5-6: Frontend (SIMPLEST)

Simple Frontend with Polling

`frontend/src/App.js`:

```
import React, { useEffect, useState } from 'react';
import './App.css';

const API_URL = 'http://localhost:8000/api';

function App() {
  const [channels, setChannels] = useState([]);
  const [roiChart, setRoiChart] = useState('');
  const [convChart, setConvChart] = useState('');
  const [recommendations, setRecommendations] = useState(null);

  const fetchData = async () => {
    try {
      // Fetch channel stats
      const channelsRes = await fetch(`${API_URL}/campaigns/channels`);
      setChannels(await channelsRes.json());

      // Fetch dashboard images
      setRoiChart(`${API_URL}/dashboards/roi-chart?t=${Date.now()}`);
      setConvChart(`${API_URL}/dashboards/conversion-chart?t=${Date.now()}`);

      // Fetch recommendations
      const recRes = await fetch(`${API_URL}/recommendations/latest`);
      setRecommendations(await recRes.json());
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  };

  useEffect(() => {
    fetchData();
    // Poll every 30 seconds
    const interval = setInterval(fetchData, 30000);
    return () => clearInterval(interval);
  }, []);

  return (
    <div className="App">
      <header>
        <h1>DEMETRA SYSTEMS - AI Analytics Dashboard</h1>
      </header>

      <section className="stats">
        <h2>Channel Performance</h2>
        <div className="channel-grid">
          {channels.map(ch => (
            <div key={ch.channel} className="card">
              <h3>{ch.channel}</h3>
              <p>Campaigns: {ch.count}</p>
              <p>Avg ROI: {ch.avg_roi.toFixed(2)}</p>
              <p>Avg Conversion: {(ch.avg_conversion * 100).toFixed(2)}%</p>
            </div>
          ))}
        </div>
      </section>

      <section className="charts">
        <h2>ML Analytics</h2>
        <div className="chart-grid">
          <div>
            <h3>ROI by Channel</h3>
            {roiChart && <img src={roiChart} alt="ROI Chart" />}
          </div>
          <div>
            <h3>Conversion Rates</h3>
            {convChart && <img src={convChart} alt="Conversion Chart" />}
          </div>
        </div>
      </section>

      <section className="recommendations">
        <h2>AI Recommendations</h2>
        {recommendations && (
          <div className="rec-box">
            <p><strong>Summary:</strong> {recommendations.summary}</p>
            <pre>{JSON.stringify(recommendations.recommendations, null, 2)}</pre>
          </div>
        )}
      </section>
    </div>
  );
}

export default App;
```

`frontend/src/App.css`:

```
.App {
  max-width: 1400px;
  margin: 0 auto;
  padding: 20px;
  font-family: Arial, sans-serif;
}

header {
  background: #1a73e8;
  color: white;
  padding: 20px;
  border-radius: 8px;
  margin-bottom: 20px;
}

.channel-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  gap: 20px;
  margin: 20px 0;
}

.card {
  background: #f5f5f5;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.chart-grid {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 20px;
  margin: 20px 0;
}

.chart-grid img {
  width: 100%;
  border-radius: 8px;
  box-shadow: 0 2px 8px rgba(0,0,0,0.1);
}

.rec-box {
  background: #e8f5e9;
  padding: 20px;
  border-radius: 8px;
  border-left: 4px solid #4caf50;
}

pre {
  background: #f5f5f5;
  padding: 10px;
  border-radius: 4px;
  overflow-x: auto;
}
```

Dependencies (`pyproject.toml`)

```
[tool.poetry.dependencies]
python = "^3.11"
fastapi = "^0.104.0"
uvicorn = {extras = ["standard"], version = "^0.24.0"}
sqlalchemy = "^2.0.0"
psycopg2-binary = "^2.9.9"
alembic = "^1.12.0"
faker = "^20.0.0"
```

```
pandas = "^2.1.0"
numpy = "^1.26.0"
seaborn = "^0.13.0"
matplotlib = "^3.8.0"
requests = "^2.31.0"
apscheduler = "^3.10.4"
python-dotenv = "^1.0.0"
```

EXECUTION ORDER (6 HOURS)

```
# HOUR 1
cd backend
poetry install
alembic revision --autogenerate -m "initial"
alembic upgrade head

# HOUR 2-3
# Create all services (mock_api.py, sync_service.py, ml_dashboard_service.py)
# Create all routers
# Update main.py

# HOUR 4
# Test scheduler
python -m uvicorn app.main:app --reload

# HOUR 5-6
cd ../frontend
npm install
npm start
```

✅ MINIMAL CHECKLIST

- [x] Postgres with 4 tables
- [x] Mock API (Faker) for 4 channels
- [x] Hourly sync (APScheduler)
- [x] ML dashboard generation (Seaborn PNG)
- [x] LLM recommendations
- [x] Simple frontend with polling
- [x] NO ALERTS
- [x] NO PDF
- [x] NO Celery

THIS IS THE ABSOLUTE MINIMUM FOR 6 HOURS. GOOD LUCK!

🚀 ENHANCED SYSTEM: ML Dashboards & Prediction Tables Integration

✨ Your goals and requirements:

1. **Backend endpoint:**
   - Takes ML predictions (tables/charts from ml_presentable.ipynb).
   - Outputs Seaborn chart PNGs for frontend (by dashboard: ROI, conversion/time, reach/time, etc.)
2. **Database:**
   - 4 tables: One for each ML dashboard prediction output (store forecasts/statistics for each type).
3. **Scheduler:**
   - Polling/sync **every 30 min** (APScheduler).
   - Ability to **run sync instantly ("hands-on")** via API endpoint.
4. **Frontend:**
   - Polls every 30 min.
   - Requests dashboard PNGs and corresponding predication tables.

🗄 ENHANCED DATABASE TABLES (with SQLAlchemy definitions):

1. ROI Forecasts by Channel

```
class ML_ROI_Forecast(Base):
    __tablename__ = "ml_roi_forecast"
    id = Column(Integer, primary_key=True)
    channel = Column(String(50), index=True)
    roi_forecast = Column(Float)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

2. Conversion & ROI Over Time

```
class ML_ROI_TimeSeries(Base):
    __tablename__ = "ml_roi_timeseries"
    id = Column(Integer, primary_key=True)
    date = Column(Date)
    channel = Column(String(50), index=True)
    roi_actual = Column(Float)
    conversion_actual = Column(Float)
    roi_pred = Column(Float)
    conversion_pred = Column(Float)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

3. Reach & CTR Over Time

```
class ML_Reach_CTR_TimeSeries(Base):
    __tablename__ = "ml_reach_ctr_timeseries"
    id = Column(Integer, primary_key=True)
    date = Column(Date)
    channel = Column(String(50), index=True)
    reach_actual = Column(Integer)
    ctr_actual = Column(Float)
    reach_pred = Column(Integer)
    ctr_pred = Column(Float)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

4. Budget Recommendations by Channel

```
class ML_Budget_Rec(Base):
    __tablename__ = "ml_budget_rec"
    id = Column(Integer, primary_key=True)
    channel = Column(String(50), index=True)
    roi_pred = Column(Float)
    conversion_pred = Column(Float)
    acquisition_cost = Column(Float)
    recommended_budget = Column(Float)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

▯ ENDPOINTS FOR FRONTEND

- `/api/ml-dashboard/roi-forecast` → PNG chart + prediction table (`ml_roi_forecast`)
- `/api/ml-dashboard/roi-timeseries` → PNG chart + table (`ml_roi_timeseries`)
- `/api/ml-dashboard/reach-ctr-timeseries` → PNG chart + table (`ml_reach_ctr_timeseries`)
- `/api/ml-dashboard/budget-recommendation` → PNG chart + table (`ml_budget_rec`)

Each endpoint returns:

- PNG chart (in-memory from Seaborn/Matplotlib)
- Table: latest prediction rows

▯ SCHEDULER & "HANDS-ON" SYNC API

- **APScheduler**: Runs every 30 min (`ml_sync_service.run_all()`):
  - Re-runs your ML pipeline, updates 4 tables, and generates latest PNGs.
- **Manual sync API endpoint**: `/api/sync/run-now`
  - Runs same `ml_sync_service.run_all()` function instantly (for demo or manual refresh).

□ BACKEND COMPONENTS & HIGH-LEVEL FLOW

FastAPI routers example:

```
@app.get("/api/ml-dashboard/roi-forecast")
def get_roi_forecast(db: Session = Depends(get_db)):
    # Query ML_ROI_Forecast table for recent data
    pred = db.query(ML_ROI_Forecast).order_by(ML_ROI_Forecast.created_at.desc()).limit(1
    # Read PNG (save in folder or in table as binary)
    with open("/tmp/roi_forecast_by_channel.png", "rb") as f:
        chart_data = f.read()
    # Send chart as PNG + predictions as JSON
    return Response(content=chart_data, media_type="image/png"), [p.as_dict() for p in p
```

Or, for strictly separated endpoints, PNG at `/image`, table at `/data`.

Sync service example:

```
class ML_Sync_Service:
    def run_all(self):
        # 1. Run notebook code (or import functions)
        # 2. Save latest Seaborn PNGs
        # 3. Overwrite the four prediction tables with newest rows/results
        pass

# In scheduler:
scheduler.add_job(
    ML_Sync_Service().run_all,
    'interval',
    minutes=30,
    id='ml_dashboard_refresh'
)

# In manual API:
@app.post("/api/sync/run-now")
def sync_now():
    ML_Sync_Service().run_all()
    return {"status": "success"}
```

□ PNG CHART GENERATION STRATEGY

- Your notebook (`ml_presentable.ipynb`) already outputs all needed tables and Seaborn charts as PNG (use `plt.savefig(...)`)
- Save each dashboard image to temp/static folder (or in DB as bytes if simple)
- Use FastAPI to stream PNGs to frontend using `Response(content=..., media_type='image/png')`
- For tables, send as JSON from SQLAlchemy query

□ FRONTEND POLLING REQUIREMENTS

- **Polling every 30 min**: Use `setInterval` in React (or fetch-on-load + refresh button for demo)
- Multiple endpoints: Each dashboard's PNG & prediction table fetched via `/api/ml-dashboard/*`
- Show loading state if fetching PNG

□ HANDS-ON SETUP: BACKEND REQUIREMENTS

0) Add in `requirements.txt` / `pyproject.toml`:

- `fastapi`
- `sqlalchemy`
- `psycopg2-binary`
- `pandas`
- `seaborn`
- `matplotlib`
- `apscheduler`
- `faker` (for fake data/mocking, if needed)
- `python-dotenv`

1) Define 4 ML prediction tables (see above) and run Alembic migration.

2) ML sync service:

- At refresh, parse notebook or run your ML prediction code for each dashboard:
  - Extract tables as pandas DataFrames.
  - Save/replace predictions in respective DB tables.
  - Generate PNGs and save to disk/DB.

3) APScheduler setup:

- Every 30 min call the above sync code.

4) API endpoints:

- `/api/ml-dashboard/<dashboard>`: Returns PNG image and JSON table rows for each type.

5) Manual sync endpoint:

- `/api/sync/run-now`: Calls the ML sync service on demand.

▯ SHORT CHECKLIST FOR 6 HOURS

- [x] Define 4 ML prediction tables + Alembic migration
- [x] ML sync service pulls data, predicts, writes to DB, saves PNG charts
- [x] Endpoints for PNG chart + table per dashboard
- [x] Scheduler every 30 min
- [x] Manual sync endpoint for demo
- [x] Frontend polls endpoints every 30 min and displays chart/table

This version fits your hackathon MVP and keeps it agentic/minimal, with everything ready for backend and frontend handoff.

Want ready Python code for the ML sync service or FastAPI endpoint? Just specify the dashboard, and I'll generate it!

⁂

can you please give to me the Architecture Diagrams using plantuml! to explain overall system!

```
@startuml
title DEMETRA SYSTEMS - AI Advertising Analytics MVP Architecture

'===== Clients (Frontend) =====
actor "Marketing Team" as User
rectangle "Frontend (React)" as Frontend {
  [Main Dashboard]
  [Charts & Tables]
  [PNG Dashboard Viewer]
  [Polling Service]
}

User --> Frontend : Uses web dashboard

'===== API Layer =====
rectangle "Backend (FastAPI)" as Backend {
  [Campaign API]
  [ML Dashboard API (/api/ml-dashboard/*)]
  [Manual Sync API]
}

Frontend --> [ML Dashboard API (/api/ml-dashboard/*)] : GET prediction tables & PNG chart
Frontend --> [Manual Sync API] : POST request to manual sync
```

```
'===== Scheduler =====
component "APScheduler" as Scheduler

Scheduler ..> Backend : Triggers ML sync every 30 min

'===== ML Sync Service =====
rectangle "ML Sync Service" as MLSync {
  [Run ML notebook/model]
  [Generate/Save Seaborn PNGs]
  [Write prediction tables to DB]
}

Scheduler --> MLSync : Calls run_all()
[Manual Sync API] --> MLSync : Calls run_all()

'===== Database =====
database "PostgreSQL" as DB {
  [campaigns]
  [ml_roi_forecast]
  [ml_roi_timeseries]
  [ml_reach_ctr_timeseries]
  [ml_budget_rec]
}

Backend --> DB : Read/write tables (\nCRUD, prediction output)
MLSync --> DB : Writes new prediction tables

'===== ML Model Layer =====
rectangle "ML Model/Notebook (ml_presentable.ipynb)" as MLModel {
  [ROI Model]
  [Budget Recommendation Code]
  [Seaborn Chart Generation]
  [Prediction Table Generation]
}

MLSync --> MLModel : Runs inference, chart, table code

'===== File Storage (optional) =====
folder "PNG Chart Storage" as PNGStore

MLSync --> PNGStore : Save PNG chart files
Backend --> PNGStore : Streams PNG to frontend

@enduml
```

How to read:

- **Frontend (React)** continuously polls the backend for updated ML dashboards (PNG charts + table data).
- **Backend (FastAPI)** exposes APIs for prediction tables, serving PNG dashboards, and lets you trigger a manual sync.

- **Scheduler (APScheduler)** triggers `ML Sync Service` every 30 min to run ML pipeline: model inference, prediction table write, Seaborn chart generation.
- **ML Sync Service** runs your notebook/model code, writes tables to Postgres, and saves PNGs.
- **Database (Postgres)** stores raw campaign data and ML prediction outputs (4+ tables, one for each dashboard/prediction).
- **ML Model Layer** is called (imported/runnable functions, or notebook parsing)—outputs tables/charts into system.
- **Optional PNG file storage**: PNG chart images saved by the sync, streamed to frontend via API.

Copy-paste to PlantUML online editor or IntelliJ/VSCode PlantUML plugin. Would you like a component breakdown for each API?

please can you give other type of diagrams,

and also give the ERD like diagram for all tables that we have in DB

Here are two types of diagrams in PlantUML:

1. System Component Diagram (PlantUML)

```
@startuml
title DEMETRA SYSTEMS - Component Overview

package "Frontend" {
  [React Dashboard]
  [Polling Service]
}

package "Backend (FastAPI)" {
  [Campaigns API]
  [ML Dashboard API]
  [Manual Sync API]
  [Scheduler (APScheduler)]
  [ML Sync Service]
}

package "ML Layer" {
  [ML Model/Notebook\n(ml_presentable.ipynb)]
}

database "PostgreSQL DB" {
  [campaigns]
  [ml_roi_forecast]
  [ml_roi_timeseries]
  [ml_reach_ctr_timeseries]
  [ml_budget_rec]
}
```

```
folder "PNG Chart Folder" as PNGStore

[React Dashboard] --> [ML Dashboard API] : GET charts/tables
[Polling Service] --> [ML Dashboard API] : Periodic polling
[React Dashboard] --> [Manual Sync API] : Run sync

[Scheduler (APScheduler)] --> [ML Sync Service] : every 30 min
[Manual Sync API] --> [ML Sync Service] : on request

[ML Sync Service] --> [ML Model/Notebook\n(ml_presentable.ipynb)] : run inference
[ML Sync Service] --> [PostgreSQL DB] : write predictions/tables
[ML Sync Service] --> PNGStore : save PNG charts

[ML Dashboard API] --> [PostgreSQL DB]
[ML Dashboard API] --> PNGStore
[Campaigns API] --> [PostgreSQL DB]
@enduml
```

2. ERD (Entity-Relationship Diagram) of Database

```
@startuml
title DEMETRA SYSTEMS - DB ERD

entity "campaigns" as campaigns {
  *id : int
  campaign_id : varchar
  channel : varchar
  conversion_rate : float
  cost : float
  roi : float
  clicks : int
  impressions : int
  engagement : int
  date : date
  created_at : datetime
}

entity "ml_roi_forecast" {
  *id : int
  channel : varchar
  roi_forecast : float
  created_at : datetime
}

entity "ml_roi_timeseries" {
  *id : int
  date : date
  channel : varchar
  roi_actual : float
  conversion_actual : float
  roi_pred : float
  conversion_pred : float
  created_at : datetime
}

entity "ml_reach_ctr_timeseries" {
  *id : int
  date : date
  channel : varchar
  reach_actual : int
  ctr_actual : float
  reach_pred : int
  ctr_pred : float
  created_at : datetime
}

entity "ml_budget_rec" {
  *id : int
  channel : varchar
  roi_pred : float
  conversion_pred : float
  acquisition_cost : float
  recommended_budget : float
  created_at : datetime
}

campaigns ||..|| ml_roi_timeseries : uses channel/date
campaigns ||..|| ml_reach_ctr_timeseries : uses channel/date

ml_roi_forecast ||..|| campaigns : channel

ml_budget_rec ||..|| campaigns : channel

@enduml
```

Instructions:

- Paste these diagrams into plantuml.com or your IDE's PlantUML viewer for rendering.
- You can further fine-tune relationships or add more attributes as needed.

Let me know if you need sequence diagrams or API callflows!

1. https://libril.com/blog/unified-analytics-dashboard-setup
2. https://analyticsliv.com/blogs/track-conversions-ga4-meta-tiktok-google-ads
3. https://www.womenconquerbiz.com/ai-simplifies-marketing-budget-planning/
4. https://madgicx.com/blog/budget-forecasting-with-ai
5. https://blog.mvpproject.io/guides/integration-of-tiktok-ads-and-telegram-how-to-attract-audience-and-convert-into-clients/?lang=en
6. https://hightime.media/dashbordi-po-marketingu/
7. https://pizza-padre.com/5-fun-pizza-options-inspired-by-the-caped-crusader/
8. https://www.sprinklr.com/blog/unified-analytics-next-era-of-high-performance-marketing/
9. https://docs.cloudera.com/data-warehouse/cloud/unified-analytics-overview/topics/dw-ua-architecture.html

10. https://www.decisionfoundry.com/marketing-data/articles/master-marketing-dashboard-mastery-examples-and-templates/
11. image.jpg
12. recommendations.py
13. ml.ipynb
14. Screenshot_22-11-2025_25043_github.com.jpg