Backend Code

`requirements.txt`

```
Flask
Flask-CORS
pandas
numpy
scipy
```

`init_db.py`

```
import sqlite3
import pandas as pd
import os

def init_db():
    print("Initializing Database...")

    # Paths
    BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    USERS_FILE = os.path.join(BASE_DIR, 'users.csv')
    EVENTS_FILE = os.path.join(BASE_DIR, 'events.csv')
    DB_FILE = os.path.join(BASE_DIR, 'backend', 'database', 'vizsprints.db')

    # Connect to SQLite (creates file if not exists)
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    try:
        # Load Users
        if os.path.exists(USERS_FILE):
            print(f"Loading {USERS_FILE}...")
            users_df = pd.read_csv(USERS_FILE)
            users_df.to_sql('users', conn, if_exists='replace', index=False)
            print(f" - Inserted {len(users_df)} users")

            # Create index on user_id
            cursor.execute("CREATE INDEX IF NOT EXISTS idx_users_user_id ON users(user_id)")
        else:
            print(f"Warning: {USERS_FILE} not found!")

        # Load Events
        if os.path.exists(EVENTS_FILE):
            print(f"Loading {EVENTS_FILE}...")
            events_df = pd.read_csv(EVENTS_FILE)
            events_df.to_sql('events', conn, if_exists='replace', index=False)
            print(f" - Inserted {len(events_df)} events")

            # Create indices for performance
            cursor.execute("CREATE INDEX IF NOT EXISTS idx_events_user_id ON events(user_id)")
            cursor.execute("CREATE INDEX IF NOT EXISTS idx_events_event_name ON
events(event_name)")
```

```
            cursor.execute("CREATE INDEX IF NOT EXISTS idx_events_timestamp ON events(timestamp)")
        else:
            print(f"Warning: {EVENTS_FILE} not found!")

        print("Database initialization complete!")

    except Exception as e:
        print(f"Error initializing database: {e}")
    finally:
        conn.close()

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

`app.py`

```
from flask import Flask, jsonify, request
from flask_cors import CORS
import pandas as pd
import numpy as np
from scipy import stats
from datetime import datetime, timedelta
from collections import defaultdict
import json
import os

import sqlite3

app = Flask(__name__)
CORS(app)
```

Database path

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DB_FILE = os.path.join(BASE_DIR, 'backend', 'database', 'vizsprints.db')

users_df = None
events_df = None

def load_data():
    """Load data from SQLite database into pandas DataFrames"""
    global users_df, events_df

    try:
        if not os.path.exists(DB_FILE):
            print(f"Database not found at {DB_FILE}")
            return False

        conn = sqlite3.connect(DB_FILE)

        print("Loading data from database...")
        users_df = pd.read_sql_query("SELECT * FROM users", conn)
        events_df = pd.read_sql_query("SELECT * FROM events", conn)

```
        conn.close()

        # Parse timestamps
        users_df['joined_at'] = pd.to_datetime(users_df['joined_at'])
        events_df['timestamp'] = pd.to_datetime(events_df['timestamp'])

        print(f"Loaded {len(users_df)} users and {len(events_df)} events from database")
        return True
    except Exception as e:
        print(f"Error loading data: {e}")
        return False

@app.route('/api/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy',
        'users_loaded': users_df is not None,
        'events_loaded': events_df is not None
    })

@app.route('/api/users', methods=['GET'])
def get_users():
    """Get user data with optional filters"""
    try:
        # Optional filters
        country = request.args.get('country')
        device = request.args.get('device')
        subscription = request.args.get('subscription_status')

        df = users_df.copy()

        if country:
            df = df[df['country'] == country]
        if device:
            df = df[df['device'] == device]
        if subscription:
            df = df[df['subscription_status'] == subscription]

        # Convert to JSON-friendly format
        df['joined_at'] = df['joined_at'].dt.strftime('%Y-%m-%dT%H:%M:%SZ')

        return jsonify({
            'users': df.to_dict('records'),
            'total': len(df)
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/events', methods=['GET'])
def get_events():
    """Get event data with optional filters"""
    try:
        # Optional filters
        user_id = request.args.get('user_id')
        event_name = request.args.get('event_name')
        start_date = request.args.get('start_date')
        end_date = request.args.get('end_date')

        df = events_df.copy()

        if user_id:
            df = df[df['user_id'] == user_id]
        if event_name:
            df = df[df['event_name'] == event_name]
        if start_date:
            df = df[df['timestamp'] >= pd.to_datetime(start_date)]
        if end_date:
            df = df[df['timestamp'] <= pd.to_datetime(end_date)]

        # Convert to JSON-friendly format
        df['timestamp'] = df['timestamp'].dt.strftime('%Y-%m-%dT%H:%M:%SZ')

        return jsonify({
            'events': df.to_dict('records')[:1000],  # Limit to 1000 events
            'total': len(df)
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/metrics', methods=['GET'])
def get_metrics():
    """Get overall engagement metrics"""
    try:
        # Total users
        total_users = len(users_df)

        # Active users (users with events in last 30 days)
        thirty_days_ago = events_df['timestamp'].max() - timedelta(days=30)
        active_user_ids = events_df[events_df['timestamp'] >= thirty_days_ago]['user_id'].unique()
        active_users = len(active_user_ids)

        # Conversion rate (users who completed at least one task)
        completed_users = events_df[events_df['event_name'] == 'complete_task']['user_id'].nunique()
        conversion_rate = (completed_users / total_users * 100) if total_users > 0 else 0

        # Revenue (estimate based on subscriptions)
        revenue_map = {'Free': 0, 'Premium': 29, 'Enterprise': 99}
        revenue = sum(users_df['subscription_status'].map(revenue_map))

        # Average events per user
        avg_events = len(events_df) / total_users if total_users > 0 else 0

        return jsonify({
            'total_users': int(total_users),
            'active_users': int(active_users),
            'conversion_rate': round(conversion_rate, 2),
            'revenue': int(revenue),
            'avg_events_per_user': round(avg_events, 2),
            'total_events': int(len(events_df))
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/cohorts', methods=['GET'])
def get_cohorts():
    """Calculate cohort retention analysis (Monthly)"""
    try:
        # Merge users with events
        df = events_df.merge(users_df[['user_id', 'joined_at']], on='user_id')

        # Calculate cohort month and event month
        df['cohort_month'] = df['joined_at'].dt.to_period('M')
        df['event_month'] = df['timestamp'].dt.to_period('M')

        # Calculate months since join
        df['months_since_join'] = (df['event_month'] - df['cohort_month']).apply(lambda x: x.n)

        # Group by cohort and month
        cohort_data = df.groupby(['cohort_month', 'months_since_join'])['user_id'].nunique().reset_index()
        cohort_data.columns = ['cohort_month', 'months_since_join', 'users']

        # Get cohort sizes
        cohort_sizes = df.groupby('cohort_month')['user_id'].nunique().reset_index()
        cohort_sizes.columns = ['cohort_month', 'cohort_size']

        # Merge and calculate retention percentage
        cohort_data = cohort_data.merge(cohort_sizes, on='cohort_month')
        cohort_data['retention'] = (cohort_data['users'] / cohort_data['cohort_size'] * 100).round(2)

        # Pivot for heatmap format
        cohort_pivot = cohort_data.pivot(
            index='cohort_month',
            columns='months_since_join',
            values='retention'
        ).fillna(0)

        # Format for frontend
        result = []
        max_months = 0
        if not cohort_pivot.empty:
            max_months = int(cohort_pivot.columns.max())

            # Reset index to make cohort_month a column
            cohort_pivot = cohort_pivot.reset_index()

            for _, row in cohort_pivot.iterrows():
                cohort_entry = {
                    'cohort': str(row['cohort_month']),
                    'size': int(cohort_sizes[cohort_sizes['cohort_month'] == row['cohort_month']]['cohort_size'].iloc[0])
                }
            # Add retention for each month
            for month_num in range(max_months + 1):
                if month_num in row:
                    cohort_entry[f'month_{month_num}'] = float(row[month_num])
                else:
                    cohort_entry[f'month_{month_num}'] = 0.0

            result.append(cohort_entry)

        return jsonify({
            'cohorts': result,
            'max_months': max_months
        })
    except Exception as e:
        print(f"Error calculating cohorts: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/ab-test', methods=['GET'])
def get_ab_test():
    """Get A/B test comparison statistics"""
    try:
        # Get optional limit parameters
        limit = request.args.get('limit', type=int)
        event_limit = request.args.get('event_limit', type=int)

        # Get optional parameters for simulation
        confidence_level = request.args.get('confidence_level', 0.95, type=float)
        manual_n_a = request.args.get('manual_n_a', type=int)
        manual_conv_a = request.args.get('manual_conv_a', type=float) # %
        manual_n_b = request.args.get('manual_n_b', type=int)
        manual_conv_b = request.args.get('manual_conv_b', type=float) # %

        # Check if we are in simulation mode
        is_simulation = all(v is not None for v in [manual_n_a, manual_conv_a, manual_n_b, manual_conv_b])

        results = {}

        if is_simulation:
            # Simulation Mode
            n_a = manual_n_a
            n_b = manual_n_b
            conv_a = manual_conv_a / 100
            conv_b = manual_conv_b / 100

            # Create dummy funnel data for visualization
            results['variant_A'] = {
                'total_users': n_a,
                'total_events': 0,
                'avg_events_per_user': 0,
                'funnel': [{'stage': 'Conversion', 'users': int(n_a * conv_a), 'conversion_rate': manual_conv_a}]
            }
            results['variant_B'] = {
                'total_users': n_b,
                'total_events': 0,
                'avg_events_per_user': 0,
                'funnel': [{'stage': 'Conversion', 'users': int(n_b * conv_b), 'conversion_rate': manual_conv_b}]
            }

            # Simple lift based on conversion rates
            if conv_a > 0:
                results['lift'] = round(((conv_b - conv_a) / conv_a) * 100, 2)
            else:
                results['lift'] = 0

        else:
            # Live Data Mode
            # Use a subset of users if limit is provided
            current_users_df = users_df
            if limit and limit > 0:
                current_users_df = users_df.head(limit)

            # Use a subset of events if event_limit is provided
            current_events_df = events_df
            if event_limit and event_limit > 0:
                current_events_df = events_df.head(event_limit)

            # Merge users with events
            df = current_events_df.merge(current_users_df[['user_id', 'ab_variant']], on='user_id')

            # Define funnel stages
            funnel_stages = ['signup_success', 'view_dashboard', 'start_project', 'complete_task', 'invite_user']

            # Base data calculation for A and B
            for variant in ['A', 'B']:
                variant_users = current_users_df[current_users_df['ab_variant'] == variant]
                variant_events = df[df['ab_variant'] == variant]

                # Calculate metrics for each funnel stage
                funnel_metrics = []
                total_users = len(variant_users)

                for stage in funnel_stages:
                users_at_stage = variant_events[variant_events['event_name'] == stage]['user_id'].nunique()
                conversion_rate = (users_at_stage / total_users * 100) if total_users > 0 else 0
                funnel_metrics.append({
                    'stage': stage,
                    'users': int(users_at_stage),
                    'conversion_rate': round(conversion_rate, 2)
                })

                # Overall metrics
                avg_events = len(variant_events) / total_users if total_users > 0 else 0

                results[f'variant_{variant}'] = {
                'total_users': int(total_users),
                'total_events': int(len(variant_events)),
                'avg_events_per_user': round(avg_events, 2),
                'funnel': funnel_metrics
                }

            # Calculate stats input from live data (using last stage conversion for simplification)
            n_a = results['variant_A']['total_users']
            n_b = results['variant_B']['total_users']

            # Use the last funnel stage as the 'conversion' event for stats
            conv_users_a = results['variant_A']['funnel'][-1]['users']
            conv_users_b = results['variant_B']['funnel'][-1]['users']

            conv_a = conv_users_a / n_a if n_a > 0 else 0
            conv_b = conv_users_b / n_b if n_b > 0 else 0

            # Calculate lift (B vs A)
            if results['variant_A']['total_users'] > 0:
                lift = ((results['variant_B']['avg_events_per_user'] - results['variant_A']['avg_events_per_user']) /
                    results['variant_A']['avg_events_per_user'] * 100)
                results['lift'] = round(lift, 2)
            else:
                results['lift'] = 0

        # --- Statistical Calculations (Z-Test & Power) ---
        stats_result = {
            'p_value': 1.0,
            'significant': False,
            'power': 0.0,
            'z_score': 0.0,
            'confidence_level': confidence_level
        }

        if n_a > 0 and n_b > 0:
            # Pooled probability
            p_pool = (n_a * conv_a + n_b * conv_b) / (n_a + n_b)

            # Standard Error
            se = np.sqrt(p_pool * (1 - p_pool) * (1/n_a + 1/n_b))

            if se > 0:
                # Z-Score
                z_score = (conv_b - conv_a) / se
                stats_result['z_score'] = float(round(z_score, 4))

                # P-Value (Two-tailed)
                p_value = 2 * (1 - stats.norm.cdf(abs(z_score)))
                stats_result['p_value'] = float(round(p_value, 4))

                # Significance
                alpha = 1 - confidence_level
                stats_result['significant'] = bool(p_value < alpha)

                # Power Calculation
                # Effect size
                h = 2 * (np.arcsin(np.sqrt(conv_b)) - np.arcsin(np.sqrt(conv_a)))
```

```
            # Sample size for power (harmonic mean approx)
            n_harm = 2 * n_a * n_b / (n_a + n_b)

            # Power (1 - beta)
            # alpha/2 for two-tailed
            z_alpha = stats.norm.ppf(1 - alpha/2)
            z_beta = abs(h) * np.sqrt(n_harm/2) - z_alpha
            power = stats.norm.cdf(z_beta)
            stats_result['power'] = float(round(power, 4))

        results['stats'] = stats_result

        return jsonify(results)
    except Exception as e:
        print(f"Error in ab-test: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/funnel', methods=['GET'])
def get_funnel():
    """Get funnel conversion metrics"""
    try:
        funnel_stages = ['signup_success', 'view_dashboard', 'start_project', 'complete_task', 'invite_user']

        total_users = len(users_df)
        funnel_data = []

        for i, stage in enumerate(funnel_stages):
            users_at_stage = events_df[events_df['event_name'] == stage]['user_id'].nunique()
            conversion_from_total = (users_at_stage / total_users * 100) if total_users > 0 else 0

            # Conversion from previous stage
            if i > 0:
                prev_stage_users = funnel_data[i-1]['users']
                conversion_from_prev = (users_at_stage / prev_stage_users * 100) if prev_stage_users > 0 else 0
            else:
                conversion_from_prev = 100.0

            funnel_data.append({
                'stage': stage.replace('_', ' ').title(),
                'users': int(users_at_stage),
                'conversion_from_total': round(conversion_from_total, 2),
                'conversion_from_previous': round(conversion_from_prev, 2),
                'drop_off': round(100 - conversion_from_prev, 2)
            })

        return jsonify({
            'funnel': funnel_data,
            'total_users': int(total_users)
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

```
@app.route('/api/user-sessions', methods=['GET'])
def get_user_sessions():
    """Calculate user session times and total hours spent"""
    try:
        # Get optional parameters
        limit = request.args.get('limit', 100, type=int)
        sort_by = request.args.get('sort_by', 'total_hours')  # total_hours, total_sessions, last_activity

        # Sort events by user and timestamp
        df = events_df.sort_values(['user_id', 'timestamp']).copy()

        # Calculate time difference between consecutive events for each user
        df['time_diff'] = df.groupby('user_id')['timestamp'].diff()

        # Session timeout: 30 minutes
        session_timeout = timedelta(minutes=30)

        # Mark new sessions (time_diff > 30 minutes or first event)
        df['new_session'] = (df['time_diff'].isna()) | (df['time_diff'] > session_timeout)

        # Assign session IDs
        df['session_id'] = df.groupby('user_id')['new_session'].cumsum()

        # Calculate session durations
        session_stats = df.groupby(['user_id', 'session_id']).agg({
            'timestamp': ['min', 'max', 'count']
        }).reset_index()

        session_stats.columns = ['user_id', 'session_id', 'session_start', 'session_end', 'event_count']

        # Calculate session duration in hours
         session_stats['duration_hours'] = (session_stats['session_end'] - session_stats['session_start']).dt.total_seconds() /
3600

        # For single-event sessions, assign minimum duration of 1 minute
        session_stats.loc[session_stats['event_count'] == 1, 'duration_hours'] = 1/60

        # Aggregate by user
        user_stats = session_stats.groupby('user_id').agg({
            'session_id': 'count',
            'duration_hours': 'sum',
            'session_start': 'min',
            'session_end': 'max'
        }).reset_index()

        user_stats.columns = ['user_id', 'total_sessions', 'total_hours', 'first_activity', 'last_activity']

        # Calculate average session duration
        user_stats['avg_session_duration'] = user_stats['total_hours'] / user_stats['total_sessions']

        # Determine if user is active (activity in last 7 days)
        max_timestamp = events_df['timestamp'].max()
        seven_days_ago = max_timestamp - timedelta(days=7)
        user_stats['status'] = user_stats['last_activity'].apply(
            lambda x: 'active' if x >= seven_days_ago else 'inactive'
        )

        # Sort by requested field
        if sort_by == 'total_hours':
            user_stats = user_stats.sort_values('total_hours', ascending=False)
        elif sort_by == 'total_sessions':
            user_stats = user_stats.sort_values('total_sessions', ascending=False)
        elif sort_by == 'last_activity':
            user_stats = user_stats.sort_values('last_activity', ascending=False)

        # Limit results
        user_stats = user_stats.head(limit)

        # Convert timestamps to strings
        user_stats['first_activity'] = user_stats['first_activity'].dt.strftime('%Y-%m-%dT%H:%M:%SZ')
        user_stats['last_activity'] = user_stats['last_activity'].dt.strftime('%Y-%m-%dT%H:%M:%SZ')

        # Round numeric values
        user_stats['total_hours'] = user_stats['total_hours'].round(2)
        user_stats['avg_session_duration'] = user_stats['avg_session_duration'].round(2)

        return jsonify({
            'user_sessions': user_stats.to_dict('records'),
            'total_users': len(user_stats)
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500


@app.route('/api/kpi-time-series', methods=['GET'])
def get_kpi_time_series():
    """Get KPI time series data (DAU, Signups)"""
    try:
        # Calculate Daily Active Users (DAU)
        # Group events by date and count unique users
        events_df['date'] = events_df['timestamp'].dt.date
        dau_series = events_df.groupby('date')['user_id'].nunique()
        df_dau = pd.DataFrame({'date': dau_series.index, 'dau': dau_series.values})
        df_dau['date'] = pd.to_datetime(df_dau['date'])

        # Calculate Signups per Day
        users_df['date'] = users_df['joined_at'].dt.date
        signups_series = users_df.groupby('date').size()
        df_signups = pd.DataFrame({'date': signups_series.index, 'signups': signups_series.values})
        df_signups['date'] = pd.to_datetime(df_signups['date'])

        # Merge on date
        # Use outer join to ensure we have all dates from both series
        df_merged = pd.merge(df_dau, df_signups, on='date', how='outer').fillna(0)

        # Sort by date
        df_merged = df_merged.sort_values('date')
        result = []
        for _, row in df_merged.iterrows():
            result.append({
                'date': row['date'].strftime('%Y-%m-%d'),
                'dau': int(row['dau']),
                'signups': int(row['signups'])
            })

        return jsonify(result)
    except Exception as e:
        print(f"Error calculating KPI time series: {e}")
        return jsonify({'error': str(e)}), 500


if __name__ == '__main__':
    print("Loading data...")
    if load_data():
        print("Starting Flask server on http://localhost:5000")
        app.run(debug=True, port=5000)
    else:
        print("Failed to load data. Please check Database file.")
```