AI-Powered Data Visualization PoC - Complete Implementation Guide

Project Structure

```
ai-viz-poc/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py
│   │   ├── models/
│   │   ├── api/
│   │   ├── services/
│   │   └── tests/
│   ├── requirements.txt
│   ├── Dockerfile
│   └── docker-compose.yml
├── frontend/
│   ├── src/
│   ├── package.json
│   └── Dockerfile
├── database/
│   ├── init.sql
│   └── sample_data.sql
├── .github/
│   └── workflows/
│       ├── backend-ci.yml
│       └── frontend-ci.yml
├── tests/
├── docs/
└── README.md
```

Step 1: Environment Setup & Database

1.1 Initialize Project Structure

```
mkdir ai-viz-poc
cd ai-viz-poc
mkdir -p backend/app/{models,api,services,tests}
mkdir -p frontend/src
mkdir -p database
mkdir -p .github/workflows
mkdir tests docs
```

1.2 PostgreSQL Setup with Docker

Create `database/docker-compose.yml`:

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: vizpoc
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
      - ./sample_data.sql:/docker-entrypoint-initdb.d/sample_data.sql
volumes:
  postgres_data:
```

Create `database/init.sql`:

```
-- Create sample tables for PoC
CREATE TABLE sales_data (
    id SERIAL PRIMARY KEY,
    date DATE NOT NULL,
    region VARCHAR(50) NOT NULL,
    product VARCHAR(100) NOT NULL,
    sales_amount DECIMAL(10,2) NOT NULL,
    quantity INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE user_queries (
    id SERIAL PRIMARY KEY,
    query_text TEXT NOT NULL,
    chart_config JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_sales_date ON sales_data(date);
CREATE INDEX idx_sales_region ON sales_data(region);
```

Create `database/sample_data.sql`:

```
-- Insert sample data
INSERT INTO sales_data (date, region, product, sales_amount, quantity) VALUES
('2024-01-01', 'North', 'Product A', 1000.00, 10),
('2024-01-01', 'South', 'Product A', 1500.00, 15),
('2024-01-01', 'East', 'Product B', 2000.00, 20),
('2024-01-02', 'North', 'Product B', 1200.00, 12),
('2024-01-02', 'South', 'Product A', 1800.00, 18),
('2024-01-03', 'East', 'Product A', 2200.00, 22);
```

1.3 Testing Step 1

```
cd database
docker-compose up -d
# Test connection
docker exec -it database_postgres_1 psql -U admin -d vizpoc -c "SELECT COUNT(*) FROM
sales_data;"
```

Expected Output: Should return count of 6 records

Debug Commands:

```
# Check container status
docker ps

# View logs
docker logs database_postgres_1

# Connect to database manually
docker exec -it database_postgres_1 psql -U admin -d vizpoc
```

Step 2: Backend API Development

2.1 FastAPI Setup

Create `backend/requirements.txt`:

```
fastapi==0.104.1
uvicorn==0.24.0
psycopg2-binary==2.9.9
sqlalchemy==2.0.23
pydantic==2.5.0
python-multipart==0.0.6
openai==1.3.0
pytest==7.4.3
pytest-asyncio==0.21.1
httpx==0.25.2
python-dotenv==1.0.0
```

Create `backend/app/main.py`:

```
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import psycopg2
import json
import os
from typing import Dict, Any, List
from datetime import datetime

app = FastAPI(title="AI Viz PoC API", version="1.0.0")

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

# Database connection
def get_db_connection():
    return psycopg2.connect(
        host=os.getenv("DB_HOST", "localhost"),
        database=os.getenv("DB_NAME", "vizpoc"),
        user=os.getenv("DB_USER", "admin"),
        password=os.getenv("DB_PASSWORD", "password")
    )

# Pydantic models
class QueryRequest(BaseModel):
    query: str

class ChartConfig(BaseModel):
    chart_type: str
    data: List[Dict[str, Any]]
    config: Dict[str, Any]

@app.get("/")
async def root():
    return {"message": "AI Viz PoC API is running"}

@app.get("/health")
async def health_check():
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT 1")
        cursor.close()
        conn.close()
        return {"status": "healthy", "database": "connected"}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Database connection failed: {str(e)}")

@app.get("/data/sales")
async def get_sales_data():
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM sales_data ORDER BY date DESC LIMIT 100")
        columns = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()

        data = []
        for row in rows:
            data.append(dict(zip(columns, row)))

        cursor.close()
        conn.close()
        return {"data": data}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/query/text-to-viz")
async def text_to_visualization(request: QueryRequest):
    """Convert natural language query to visualization config"""
    try:
        # Simple rule-based approach for PoC
        query_lower = request.query.lower()

        # Get data from database
        conn = get_db_connection()
        cursor = conn.cursor()

        if "sales by region" in query_lower:
            cursor.execute("""
                SELECT region, SUM(sales_amount) as total_sales
                FROM sales_data
                GROUP BY region
            """)
            data = [{"region": row[0], "total_sales": float(row[1])} for row in cursor.fetchall()]
            chart_config = {
                "chart_type": "bar",
                "data": data,
                "config": {
                    "x": "region",
                    "y": "total_sales",
                    "title": "Sales by Region"
                }
            }
        elif "sales over time" in query_lower:
            cursor.execute("""
                SELECT date, SUM(sales_amount) as total_sales
                FROM sales_data
                GROUP BY date
                ORDER BY date
            """)
            data = [{"date": row[0].isoformat(), "total_sales": float(row[1])} for row in cursor.fetchall()]
            chart_config = {
                "chart_type": "line",
                "data": data,
                "config": {
                    "x": "date",
                    "y": "total_sales",
                    "title": "Sales Over Time"
                }
            }
        else:
            # Default: all sales data
            cursor.execute("SELECT region, product, sales_amount FROM sales_data")
            data = [{"region": row[0], "product": row[1], "sales_amount": float(row[2])} for row in
cursor.fetchall()]
            chart_config = {
                "chart_type": "table",
                "data": data,
                "config": {
                    "title": "Sales Data"
                }
            }

        cursor.close()
        conn.close()

        # Store query in database
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO user_queries (query_text, chart_config) VALUES (%s, %s)",
            (request.query, json.dumps(chart_config))
        )
        conn.commit()
        cursor.close()
        conn.close()

        return chart_config

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

2.2 Backend Tests

Create `backend/app/tests/test_main.py`:

```
import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "AI Viz PoC API is running"}

def test_health_check():
    response = client.get("/health")
    assert response.status_code == 200

def test_get_sales_data():
    response = client.get("/data/sales")
    assert response.status_code == 200
    assert "data" in response.json()

def test_text_to_viz():
    response = client.post("/query/text-to-viz", json={"query": "show me sales by region"})
    assert response.status_code == 200
    data = response.json()
    assert "chart_type" in data
    assert "data" in data
    assert "config" in data
```

2.3 Testing Step 2

```
cd backend
pip install -r requirements.txt
python -m pytest app/tests/ -v

# Run the server
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

Test endpoints:

```
# Health check
curl http://localhost:8000/health

# Get sales data
```

```
curl http://localhost:8000/data/sales

# Text to viz
curl -X POST "http://localhost:8000/query/text-to-viz" \
     -H "Content-Type: application/json" \
     -d '{"query": "show me sales by region"}'
```

Debug Commands:

```
# Check if server is running
netstat -tlnp | grep :8000

# View server logs
tail -f app.log

# Test database connection separately
python -c "import psycopg2; conn = psycopg2.connect(host='localhost', database='vizpoc', user='admin', password='password'); print('Connected successfully')"
```

Step 3: Frontend Development

3.1 React Setup

Create `frontend/package.json`:

```
{
  "name": "ai-viz-frontend",
  "version": "1.0.0",
  "private": true,
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "axios": "^1.6.0",
    "recharts": "^2.8.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/jest-dom": "^5.16.5",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "devDependencies": {
    "react-scripts": "5.0.1"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Create `frontend/src/App.js`:

```
import React, { useState } from 'react';
import axios from 'axios';
import { BarChart, Bar, XAxis, YAxis, CartesianGrid, Tooltip, Legend, LineChart, Line } from
'recharts';
import './App.css';

const API_BASE_URL = process.env.REACT_APP_API_URL || 'http://localhost:8000';

function App() {
  const [query, setQuery] = useState('');
  const [chartData, setChartData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setLoading(true);
    setError(null);

    try {
      const response = await axios.post(`${API_BASE_URL}/query/text-to-viz`, {
        query: query
      });
      setChartData(response.data);
    } catch (err) {
      setError('Failed to generate visualization: ' + err.message);
    } finally {
      setLoading(false);
    }
  };

  const renderChart = () => {
    if (!chartData) return null;

    const { chart_type, data, config } = chartData;

    switch (chart_type) {
      case 'bar':
        return (
          <BarChart width={600} height={300} data={data}>
            <CartesianGrid strokeDasharray="3 3" />
            <XAxis dataKey={config.x} />
            <YAxis />
            <Tooltip />
            <Legend />
            <Bar dataKey={config.y} fill="#8884d8" />
          </BarChart>
        );
      case 'line':
        return (
          <LineChart width={600} height={300} data={data}>
            <CartesianGrid strokeDasharray="3 3" />
            <XAxis dataKey={config.x} />
            <YAxis />
            <Tooltip />
            <Legend />
            <Line type="monotone" dataKey={config.y} stroke="#8884d8" />
          </LineChart>
        );
      case 'table':
        return (
          <table className="data-table">
            <thead>
              <tr>
                {Object.keys(data[0] || {}).map(key => (
                  <th key={key}>{key}</th>
                ))}
              </tr>
            </thead>
            <tbody>
              {data.map((row, index) => (
                <tr key={index}>
                  {Object.values(row).map((value, idx) => (
                    <td key={idx}>{value}</td>
                  ))}
                </tr>
              ))}
            </tbody>
          </table>
        );
      default:
        return <p>Unsupported chart type: {chart_type}</p>;
    }
  };

  return (
    <div className="App">
      <header className="App-header">
        <h1>AI-Powered Data Visualization PoC</h1>

        <form onSubmit={handleSubmit} className="query-form">
          <input
            type="text"
            value={query}
            onChange={(e) => setQuery(e.target.value)}
            placeholder="Enter your query (e.g., 'show me sales by region')"
            className="query-input"
            disabled={loading}
          />
          <button type="submit" disabled={loading || !query.trim()}>
            {loading ? 'Generating...' : 'Generate Visualization'}
          </button>
        </form>

        {error && <div className="error-message">{error}</div>}

        <div className="chart-container">
          {chartData && (
            <div>
              <h2>{chartData.config.title}</h2>
```

```
              {renderChart()}
            </div>
          )}
        </div>

        <div className="sample-queries">
          <h3>Try these sample queries:</h3>
          <ul>
            <li>show me sales by region</li>
            <li>sales over time</li>
            <li>display all sales data</li>
          </ul>
        </div>
      </header>
    </div>
  );
}

export default App;
```

3.2 Frontend Tests

Create `frontend/src/App.test.js`:

```
import { render, screen, fireEvent, waitFor } from '@testing-library/react';
import axios from 'axios';
import App from './App';

jest.mock('axios');
const mockedAxios = axios;

test('renders app title', () => {
  render(<App />);
  const titleElement = screen.getByText(/AI-Powered Data Visualization PoC/i);
  expect(titleElement).toBeInTheDocument();
});

test('submits query and displays chart', async () => {
  const mockResponse = {
    data: {
      chart_type: 'bar',
      data: [{ region: 'North', total_sales: 1000 }],
      config: { x: 'region', y: 'total_sales', title: 'Sales by Region' }
    }
  };

  mockedAxios.post.mockResolvedValue(mockResponse);

  render(<App />);

  const input = screen.getByPlaceholderText(/Enter your query/i);
  const button = screen.getByText(/Generate Visualization/i);

  fireEvent.change(input, { target: { value: 'sales by region' } });
  fireEvent.click(button);

  await waitFor(() => {
    expect(screen.getByText(/Sales by Region/i)).toBeInTheDocument();
  });
});
```

3.3 Testing Step 3

```
cd frontend
npm install
npm test

# Run the frontend
npm start
```

Manual Testing:

1. Open http://localhost:3000
2. Try sample queries: "show me sales by region", "sales over time"
3. Verify charts render correctly

Debug Commands:

```
# Check if frontend is running
netstat -tlnp | grep :3000

# Build for production
npm run build

# Check console for errors
# Open browser dev tools (F12) and check console
```

Step 4: CI/CD Pipeline

4.1 Backend CI/CD

Create `.github/workflows/backend-ci.yml`:

```
name: Backend CI/CD

on:
  push:
    branches: [ main, develop ]
    paths: [ 'backend/**' ]
  pull_request:
    branches: [ main ]
    paths: [ 'backend/**' ]

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:15
        env:
          POSTGRES_DB: vizpoc
          POSTGRES_USER: admin
          POSTGRES_PASSWORD: password
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432

    steps:
    - uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
```

```
      python-version: '3.11'

  - name: Install dependencies
    run: |
      cd backend
      pip install -r requirements.txt

  - name: Set up database
    env:
      DB_HOST: localhost
      DB_NAME: vizpoc
      DB_USER: admin
      DB_PASSWORD: password
    run: |
      cd database
      PGPASSWORD=password psql -h localhost -U admin -d vizpoc -f init.sql
      PGPASSWORD=password psql -h localhost -U admin -d vizpoc -f sample_data.sql

  - name: Run tests
    env:
      DB_HOST: localhost
      DB_NAME: vizpoc
      DB_USER: admin
      DB_PASSWORD: password
    run: |
      cd backend
      python -m pytest app/tests/ -v --cov=app --cov-report=xml

  - name: Upload coverage to Codecov
    uses: codecov/codecov-action@v3
    with:
      file: ./backend/coverage.xml

deploy:
  needs: test
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'

  steps:
  - uses: actions/checkout@v4

  - name: Build Docker image
    run: |
      cd backend
```

```
        docker build -t ai-viz-backend:${{ github.sha }} .

    - name: Deploy to staging
      run: |
        echo "Deploy to staging environment"
        # Add your deployment commands here
```

4.2 Frontend CI/CD

Create `.github/workflows/frontend-ci.yml`:

```
name: Frontend CI/CD

on:
  push:
    branches: [ main, develop ]
    paths: [ 'frontend/**' ]
  pull_request:
    branches: [ main ]
    paths: [ 'frontend/**' ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'
        cache: 'npm'
        cache-dependency-path: frontend/package-lock.json

    - name: Install dependencies
      run: |
        cd frontend
        npm ci

    - name: Run tests
      run: |
        cd frontend
        npm test -- --coverage --watchAll=false

    - name: Build application
      run: |
        cd frontend
        npm run build

    - name: Run E2E tests
      run: |
        cd frontend
        # Add E2E tests here (Cypress, Playwright, etc.)
        echo "E2E tests would run here"

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'
        cache: 'npm'
        cache-dependency-path: frontend/package-lock.json

    - name: Build for production
      run: |
        cd frontend
        npm ci
        npm run build

    - name: Deploy to staging
      run: |
        echo "Deploy frontend to staging"
        # Add deployment commands (AWS S3, Netlify, etc.)
```

4.3 Docker Setup

Create `backend/Dockerfile`:

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app/ ./app/

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Create `frontend/Dockerfile`:

```
FROM node:18-alpine as build

WORKDIR /app
COPY package*.json ./
RUN npm ci

COPY src/ ./src/
COPY public/ ./public/
RUN npm run build

FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Create `docker-compose.yml` (root level):

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: vizpoc
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
      - ./database/sample_data.sql:/docker-entrypoint-initdb.d/sample_data.sql

  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DB_HOST=postgres
      - DB_NAME=vizpoc
      - DB_USER=admin
      - DB_PASSWORD=password
    depends_on:
      - postgres

  frontend:
    build: ./frontend
    ports:
      - "3000:80"
    environment:
      - REACT_APP_API_URL=http://localhost:8000
    depends_on:
      - backend

volumes:
  postgres_data:
```

4.4 Testing Step 4

```
# Test full stack with Docker
docker-compose up --build

# Test CI/CD locally (if using act)
act -j test

# Push to GitHub and check Actions
git add .
git commit -m "Add CI/CD pipeline"
git push origin main
```

Step 5: Integration Testing & Debugging

5.1 End-to-End Test Script

Create `tests/e2e_test.py`:

```
import requests
import time
import json

def test_full_pipeline():
    base_url = "http://localhost:8000"

    # Test 1: Health check
    print("Testing health check...")
    response = requests.get(f"{base_url}/health")
    assert response.status_code == 200
    print("✓ Health check passed")

    # Test 2: Get sales data
    print("Testing sales data retrieval...")
    response = requests.get(f"{base_url}/data/sales")
    assert response.status_code == 200
    data = response.json()
    assert "data" in data
    assert len(data["data"]) > 0
    print(f"✓ Retrieved {len(data['data'])} sales records")

    # Test 3: Text to visualization
    print("Testing text-to-viz conversion...")
    queries = [
        "show me sales by region",
        "sales over time",
        "display all sales data"
    ]

    for query in queries:
        response = requests.post(f"{base_url}/query/text-to-viz",
                               json={"query": query})
        assert response.status_code == 200
        chart_data = response.json()
        assert "chart_type" in chart_data
        assert "data" in chart_data
        assert "config" in chart_data
        print(f"✓ Query '{query}' generated {chart_data['chart_type']} chart")
    print("All tests passed! 🎉")

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

5.2 Performance Testing

Create `tests/performance_test.py`:

```
import requests
import time
import statistics
import concurrent.futures

def measure_response_time(url, payload=None):
    start_time = time.time()
    if payload:
        response = requests.post(url, json=payload)
    else:
        response = requests.get(url)
    end_time = time.time()
    return end_time - start_time, response.status_code

def load_test():
    base_url = "http://localhost:8000"

    # Test concurrent requests
    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
        futures = []
        for _ in range(50):
            future = executor.submit(measure_response_time,
                                     f"{base_url}/query/text-to-viz",
                                     {"query": "show me sales by region"})
            futures.append(future)

        results = [future.result() for future in futures]

    response_times = [result[0] for result in results]
    status_codes = [result[1] for result in results]

    print(f"Response times - Mean: {statistics.mean(response_times):.2f}s")
    print(f"Response times - Median: {statistics.median(response_times):.2f}s")
    print(f"Response times - Max: {max(response_times):.2f}s")
    print(f"Success rate: {status_codes.count(200)/len(status_codes)*100:.1f}%")

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

5.3 Debugging Tools & Commands

Database Debugging:

```
# Connect to PostgreSQL
docker exec -it ai-viz-poc_postgres_1 psql -U admin -d vizpoc

# Check query logs
docker logs ai-viz-poc_postgres_1

# Monitor database performance
docker exec -it ai-viz-poc_postgres_1 psql -U admin -d vizpoc -c "
SELECT query, mean_exec_time, calls
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;"
```

Backend Debugging:

```
# View backend logs
docker logs ai-viz-poc_backend_1 -f

# Check API performance
curl -w "@curl-format.txt" -s -o /dev/null http://localhost:8000/health

# Monitor resource usage
docker stats ai-viz-poc_backend_1
```

Frontend Debugging:

```
# Check frontend build
docker exec -it ai-viz-poc_frontend_1 ls -la /usr/share/nginx/html

# Monitor network requests
# Use browser dev tools Network tab

# Check console errors
# Use browser dev tools Console tab
```

Step 6: Monitoring & Production Readiness

6.1 Health Monitoring

Create `backend/app/monitoring.py`:

```
import psutil
import time
from datetime import datetime

def get_system_metrics():
    return {
        "cpu_percent": psutil.cpu_percent(),
        "memory_percent": psutil.virtual_memory().percent,
        "disk_usage": psutil.disk_usage('/').percent,
        "timestamp": datetime.utcnow().isoformat()
    }

def get_database_metrics():
    # Add database connection pool monitoring
    # Query performance metrics
    # Connection count
    pass
```

6.2 Logging Configuration

Create `backend/app/logging_config.py`:

```
import logging
import json
from datetime import datetime

class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "level": record.levelname,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        }
        if hasattr(record, 'user_id'):
            log_entry['user_id'] = record.user_id
        if hasattr(record, 'request_id'):
            log_entry['request_id'] = record.request_id
        return json.dumps(log_entry)

def setup_logging():
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Add JSON formatter for production
    json_handler = logging.StreamHandler()
    json_handler.setFormatter(JSONFormatter())

    logger = logging.getLogger("ai_viz_poc")
    logger.addHandler(json_handler)
    return logger
```

6.3 Error Tracking & Alerting

Create `backend/app/error_tracking.py`:

```
import logging
from datetime import datetime
from typing import Dict, Any
import smtplib
from email.mime.text import MIMEText

class ErrorTracker:
    def __init__(self):
        self.error_counts = {}
        self.alert_threshold = 10

    def log_error(self, error: Exception, context: Dict[str, Any] = None):
        error_key = f"{type(error).__name__}:{str(error)}"

        if error_key not in self.error_counts:
            self.error_counts[error_key] = {
                "count": 0,
                "first_seen": datetime.utcnow(),
                "last_seen": datetime.utcnow(),
                "context": context or {}
            }

        self.error_counts[error_key]["count"] += 1
        self.error_counts[error_key]["last_seen"] = datetime.utcnow()

        # Send alert if threshold exceeded
        if self.error_counts[error_key]["count"] >= self.alert_threshold:
            self.send_alert(error_key, self.error_counts[error_key])

    def send_alert(self, error_key: str, error_info: Dict):
        # Implement your alerting mechanism (email, Slack, PagerDuty, etc.)
        logging.critical(f"ALERT: Error threshold exceeded for {error_key}")
        logging.critical(f"Error info: {error_info}")

error_tracker = ErrorTracker()
```

Step 7: Advanced Features & Optimization

7.1 LLM Integration for Better NLP

Create `backend/app/services/llm_service.py`:

```
import openai
import json
from typing import Dict, Any
import os

class LLMService:
    def __init__(self):
        # Initialize your preferred LLM service
        # For OpenAI (requires API key)
        self.openai_client = openai.OpenAI(
            api_key=os.getenv("OPENAI_API_KEY")
        ) if os.getenv("OPENAI_API_KEY") else None

    def parse_query_to_sql(self, natural_query: str, schema: Dict[str, Any]) -> str:
        """Convert natural language to SQL query"""
        system_prompt = f"""
        You are a SQL query generator. Convert natural language queries to SQL.
        Available tables and schema:
        {json.dumps(schema, indent=2)}

        Rules:
        1. Only use tables and columns from the provided schema
        2. Always use proper SQL syntax
        3. Include appropriate WHERE clauses for filtering
        4. Use GROUP BY for aggregations
        5. Return only the SQL query, no explanations
        """

        if not self.openai_client:
            # Fallback to rule-based approach
            return self._rule_based_sql_generation(natural_query)

        try:
            response = self.openai_client.chat.completions.create(
                model="gpt-3.5-turbo",
                messages=[
                    {"role": "system", "content": system_prompt},
                    {"role": "user", "content": natural_query}
                ],
                max_tokens=200,
                temperature=0.1
            )
            return response.choices[0].message.content.strip()
        except Exception as e:
            logging.error(f"LLM query generation failed: {e}")
            return self._rule_based_sql_generation(natural_query)

    def _rule_based_sql_generation(self, query: str) -> str:
        """Fallback rule-based SQL generation"""
        query_lower = query.lower()

        if "sales by region" in query_lower:
            return "SELECT region, SUM(sales_amount) as total_sales FROM sales_data GROUP
BY region"
        elif "sales over time" in query_lower:
            return "SELECT date, SUM(sales_amount) as total_sales FROM sales_data GROUP
BY date ORDER BY date"
        elif "top products" in query_lower:
            return "SELECT product, SUM(sales_amount) as total_sales FROM sales_data GROUP
BY product ORDER BY total_sales DESC LIMIT 10"
```

```
        else:
            return "SELECT * FROM sales_data LIMIT 100"

    def suggest_chart_type(self, query: str, data_structure: Dict) -> str:
        """Suggest appropriate chart type based on query and data"""
        query_lower = query.lower()

        # Rule-based chart type suggestion
        if any(word in query_lower for word in ["over time", "trend", "timeline"]):
            return "line"
        elif any(word in query_lower for word in ["compare", "by region", "by product"]):
            return "bar"
        elif any(word in query_lower for word in ["distribution", "proportion", "percentage"]):
            return "pie"
        elif any(word in query_lower for word in ["correlation", "relationship"]):
            return "scatter"
        else:
            # Default based on data structure
            if len(data_structure.get("columns", [])) <= 3:
                return "bar"
            else:
                return "table"

llm_service = LLMService()
```

7.2 Caching Layer

Create `backend/app/services/cache_service.py`:

```
import redis
import json
import hashlib
from typing import Any, Optional
import os

class CacheService:
    def __init__(self):
        self.redis_client = None
        try:
            self.redis_client = redis.Redis(
                host=os.getenv("REDIS_HOST", "localhost"),
                port=int(os.getenv("REDIS_PORT", 6379)),
                decode_responses=True
            )
            # Test connection
            self.redis_client.ping()
        except Exception as e:
            print(f"Redis connection failed: {e}")
            self.redis_client = None

    def _generate_key(self, query: str, params: dict = None) -> str:
        """Generate cache key from query and parameters"""
        cache_input = f"{query}:{json.dumps(params or {}, sort_keys=True)}"
        return hashlib.md5(cache_input.encode()).hexdigest()

    def get(self, query: str, params: dict = None) -> Optional[Any]:
        """Get cached result"""
        if not self.redis_client:
            return None

        try:
            key = self._generate_key(query, params)
            cached_result = self.redis_client.get(key)
            if cached_result:
                return json.loads(cached_result)
        except Exception as e:
            print(f"Cache get error: {e}")
        return None

    def set(self, query: str, result: Any, params: dict = None, ttl: int = 3600):
        """Cache result with TTL (default 1 hour)"""
        if not self.redis_client:
            return

        try:
            key = self._generate_key(query, params)
            self.redis_client.setex(key, ttl, json.dumps(result, default=str))
        except Exception as e:
            print(f"Cache set error: {e}")

    def invalidate_pattern(self, pattern: str):
        """Invalidate all keys matching pattern"""
        if not self.redis_client:
            return

        try:
            keys = self.redis_client.keys(pattern)
            if keys:
                self.redis_client.delete(*keys)
        except Exception as e:
            print(f"Cache invalidation error: {e}")

cache_service = CacheService()
```

7.3 Enhanced API with Caching and LLM

Update `backend/app/main.py` to include advanced features:

```
# Add these imports to the existing main.py
from app.services.llm_service import llm_service
from app.services.cache_service import cache_service
from app.error_tracking import error_tracker
import logging

# Add this enhanced endpoint
@app.post("/query/advanced-text-to-viz")
async def advanced_text_to_visualization(request: QueryRequest):
    """Enhanced text-to-viz with LLM and caching"""
    try:
        # Check cache first
        cached_result = cache_service.get(request.query)
        if cached_result:
            return cached_result

        # Get database schema
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("""
            SELECT column_name, data_type
            FROM information_schema.columns
            WHERE table_name = 'sales_data'
        """)
        schema_info = {
            "sales_data": {
                "columns": {row[0]: row[1] for row in cursor.fetchall()}
            }
        }

        # Generate SQL using LLM
        sql_query = llm_service.parse_query_to_sql(request.query, schema_info)
        logging.info(f"Generated SQL: {sql_query}")

        # Execute query
        cursor.execute(sql_query)
        columns = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()

        data = []
        for row in rows:
            data.append(dict(zip(columns, [float(val) if isinstance(val, (int, float)) else str(val) for val
in row])))

        cursor.close()
        conn.close()

        # Suggest chart type
        chart_type = llm_service.suggest_chart_type(request.query, {"columns": columns})

        # Build chart configuration
        chart_config = {
            "chart_type": chart_type,
            "data": data,
            "config": {
                "title": f"Analysis: {request.query}",
                "sql_query": sql_query
            }
        }

        # Determine axes for charts
        if chart_type in ["bar", "line"] and len(columns) >= 2:
            chart_config["config"]["x"] = columns[0]
            chart_config["config"]["y"] = columns[1]

        # Cache the result
        cache_service.set(request.query, chart_config, ttl=1800)  # 30 minutes

        # Store in database
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO user_queries (query_text, chart_config) VALUES (%s, %s)",
            (request.query, json.dumps(chart_config))
        )
        conn.commit()
        cursor.close()
        conn.close()

        return chart_config

    except Exception as e:
        error_tracker.log_error(e, {"query": request.query})
        raise HTTPException(status_code=500, detail=str(e))

# Add analytics endpoint
@app.get("/analytics/query-stats")
async def get_query_analytics():
    """Get analytics on user queries"""
    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Most popular queries
        cursor.execute("""
            SELECT query_text, COUNT(*) as frequency
            FROM user_queries
            GROUP BY query_text
            ORDER BY frequency DESC
            LIMIT 10
        """)
        popular_queries = [{"query": row[0], "count": row[1]} for row in cursor.fetchall()]

        # Chart type distribution
        cursor.execute("""
            SELECT chart_config->>'chart_type' as chart_type, COUNT(*) as count
            FROM user_queries
            WHERE chart_config IS NOT NULL
            GROUP BY chart_config->>'chart_type'
        """)
        chart_types = [{"type": row[0], "count": row[1]} for row in cursor.fetchall()]

        cursor.close()
        conn.close()

        return {
            "popular_queries": popular_queries,
            "chart_type_distribution": chart_types
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Step 8: Testing & Validation

8.1 Comprehensive Test Suite

Create `tests/test_integration.py`:

```
import pytest
import requests
import json
import time
from concurrent.futures import ThreadPoolExecutor

BASE_URL = "http://localhost:8000"

class TestIntegration:
    def setup_method(self):
        """Setup before each test"""
        # Ensure database is ready
        response = requests.get(f"{BASE_URL}/health")
        assert response.status_code == 200

    def test_basic_functionality(self):
        """Test basic API functionality"""
        # Test health check
        response = requests.get(f"{BASE_URL}/health")
        assert response.status_code == 200
        assert response.json()["status"] == "healthy"

        # Test data retrieval
        response = requests.get(f"{BASE_URL}/data/sales")
        assert response.status_code == 200
        data = response.json()
        assert "data" in data
        assert len(data["data"]) > 0

    def test_text_to_viz_queries(self):
        """Test various text-to-visualization queries"""
        test_queries = [
            {"query": "show me sales by region", "expected_chart": "bar"},
            {"query": "sales over time", "expected_chart": "line"},
            {"query": "display all sales data", "expected_chart": "table"}
        ]

        for test_case in test_queries:
            response = requests.post(f"{BASE_URL}/query/text-to-viz",
                                   json={"query": test_case["query"]})
            assert response.status_code == 200

            chart_data = response.json()
            assert "chart_type" in chart_data
            assert "data" in chart_data
            assert "config" in chart_data
            assert chart_data["chart_type"] == test_case["expected_chart"]

    def test_advanced_text_to_viz(self):
        """Test advanced LLM-powered text-to-viz"""
        response = requests.post(f"{BASE_URL}/query/advanced-text-to-viz",
                               json={"query": "show me top 5 products by sales"})

        if response.status_code == 200:  # Only if LLM service is available
            chart_data = response.json()
            assert "chart_type" in chart_data
            assert "data" in chart_data
            assert "sql_query" in chart_data["config"]

    def test_caching_behavior(self):
        """Test that caching works correctly"""
        query = "show me sales by region for caching test"

        # First request
        start_time = time.time()
        response1 = requests.post(f"{BASE_URL}/query/text-to-viz",
                                json={"query": query})
        first_request_time = time.time() - start_time

        # Second request (should be cached)
        start_time = time.time()
        response2 = requests.post(f"{BASE_URL}/query/text-to-viz",
                                json={"query": query})
        second_request_time = time.time() - start_time

        assert response1.status_code == 200
        assert response2.status_code == 200
        assert response1.json() == response2.json()
```

```
        # Second request should be faster (if caching is working)
        # Note: This assertion might be flaky in some environments
        # assert second_request_time < first_request_time

    def test_concurrent_requests(self):
        """Test system under concurrent load"""
        def make_request():
            response = requests.post(f"{BASE_URL}/query/text-to-viz",
                                   json={"query": "show me sales by region"})
            return response.status_code == 200

        with ThreadPoolExecutor(max_workers=5) as executor:
            futures = [executor.submit(make_request) for _ in range(10)]
            results = [future.result() for future in futures]

        # All requests should succeed
        assert all(results)

    def test_error_handling(self):
        """Test error handling for invalid requests"""
        # Empty query
        response = requests.post(f"{BASE_URL}/query/text-to-viz",
                               json={"query": ""})
        # Should handle gracefully

        # Invalid JSON
        response = requests.post(f"{BASE_URL}/query/text-to-viz",
                               data="invalid json",
                               headers={"Content-Type": "application/json"})
        assert response.status_code == 422  # Unprocessable Entity

    def test_analytics_endpoint(self):
        """Test analytics functionality"""
        # Make some queries first
        queries = ["sales by region", "sales over time", "sales by region"]
        for query in queries:
            requests.post(f"{BASE_URL}/query/text-to-viz",
                        json={"query": query})

        # Get analytics
        response = requests.get(f"{BASE_URL}/analytics/query-stats")
        if response.status_code == 200:  # Only if endpoint exists
            analytics = response.json()
            assert "popular_queries" in analytics
        assert "chart_type_distribution" in analytics

if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

8.2 Performance Benchmarking

Create `tests/benchmark.py`:

```
import time
import requests
import statistics
import matplotlib.pyplot as plt
from concurrent.futures import ThreadPoolExecutor
import json

class PerformanceBenchmark:
    def __init__(self, base_url="http://localhost:8000"):
        self.base_url = base_url
        self.results = {}

    def benchmark_endpoint(self, endpoint, method="GET", payload=None, iterations=100):
        """Benchmark a specific endpoint"""
        response_times = []
        errors = 0

        for _ in range(iterations):
            start_time = time.time()
            try:
                if method.upper() == "POST":
                    response = requests.post(f"{self.base_url}{endpoint}",
                                           json=payload, timeout=30)
                else:
                    response = requests.get(f"{self.base_url}{endpoint}", timeout=30)

                if response.status_code != 200:
                    errors += 1
            except requests.exceptions.RequestException:
                errors += 1

            end_time = time.time()
            response_times.append(end_time - start_time)
        return {
            "mean": statistics.mean(response_times),
            "median": statistics.median(response_times),
            "std_dev": statistics.stdev(response_times) if len(response_times) > 1 else 0,
            "min": min(response_times),
            "max": max(response_times),
            "error_rate": errors / iterations * 100,
            "throughput": iterations / sum(response_times)  # requests per second
        }

    def concurrent_load_test(self, endpoint, method="GET", payload=None,
                             concurrent_users=10, requests_per_user=20):
        """Test concurrent load"""
        def user_session():
            session_times = []
            for _ in range(requests_per_user):
                start_time = time.time()
                try:
                    if method.upper() == "POST":
                        response = requests.post(f"{self.base_url}{endpoint}",
                                                 json=payload, timeout=30)
                    else:
                        response = requests.get(f"{self.base_url}{endpoint}", timeout=30)
                    return response.status_code == 200
                except:
                    return False
                finally:
                    session_times.append(time.time() - start_time)
            return session_times

        with ThreadPoolExecutor(max_workers=concurrent_users) as executor:
            futures = [executor.submit(user_session) for _ in range(concurrent_users)]
            results = [future.result() for future in futures]

        # Flatten results and calculate metrics
        all_times = []
        success_count = 0
        for result in results:
            if isinstance(result, list):
                all_times.extend(result)
                success_count += len(result)
            elif result:
                success_count += 1
        total_requests = concurrent_users * requests_per_user
        return {
            "total_requests": total_requests,
            "successful_requests": success_count,
            "success_rate": success_count / total_requests * 100,
            "mean_response_time": statistics.mean(all_times) if all_times else 0,
            "concurrent_users": concurrent_users
        }

    def run_full_benchmark(self):
        """Run comprehensive benchmark suite"""
        print("🚀 Starting Performance Benchmark Suite...")

        # Endpoint benchmarks
        endpoints = [
            ("/health", "GET", None),
            ("/data/sales", "GET", None),
            ("/query/text-to-viz", "POST", {"query": "show me sales by region"}),
        ]

        for endpoint, method, payload in endpoints:
            print(f"\n📊 Benchmarking {method} {endpoint}...")
            self.results[endpoint] = self.benchmark_endpoint(endpoint, method, payload)
            self.print_benchmark_results(endpoint, self.results[endpoint])

        # Load testing
        print(f"\n🔥 Running concurrent load test...")
        load_result = self.concurrent_load_test(
            "/query/text-to-viz", "POST",
            {"query": "show me sales by region"},
            concurrent_users=5, requests_per_user=10
        )
        self.print_load_test_results(load_result)

        # Generate report
        self.generate_report()

    def print_benchmark_results(self, endpoint, results):
        """Print formatted benchmark results"""
        print(f"  Mean Response Time: {results['mean']:.3f}s")
        print(f"  Median Response Time: {results['median']:.3f}s")
        print(f"  Min/Max: {results['min']:.3f}s / {results['max']:.3f}s")
        print(f"  Standard Deviation: {results['std_dev']:.3f}s")
        print(f"  Error Rate: {results['error_rate']:.1f}%")
        print(f"  Throughput: {results['throughput']:.1f} req/s")

    def print_load_test_results(self, results):
        """Print formatted load test results"""
        print(f"  Total Requests: {results['total_requests']}")
        print(f"  Successful Requests: {results['successful_requests']}")
        print(f"  Success Rate: {results['success_rate']:.1f}%")
        print(f"  Mean Response Time: {results['mean_response_time']:.3f}s")
        print(f"  Concurrent Users: {results['concurrent_users']}")

    def generate_report(self):
        """Generate detailed performance report"""
        report = {
            "benchmark_timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
            "endpoint_benchmarks": self.results,
            "summary": {
                "fastest_endpoint": min(self.results.keys(),
                                      key=lambda k: self.results[k]['mean']),
                "slowest_endpoint": max(self.results.keys(),
                                      key=lambda k: self.results[k]['mean'])
            }
        }

        with open("performance_report.json", "w") as f:
            json.dump(report, f, indent=2)

        print(f"\n📈 Performance report saved to 'performance_report.json'")
        print(f"   Fastest endpoint: {report['summary']['fastest_endpoint']}")
        print(f"   Slowest endpoint: {report['summary']['slowest_endpoint']}")

if __name__ == "__main__":
    benchmark = PerformanceBenchmark()
    benchmark.run_full_benchmark()
```

Step 9: Deployment & Production Setup

9.1 Production Docker Configuration

Create `docker-compose.prod.yml`:

```
version: '3.8'
services:
```

```
postgres:
  image: postgres:15
  environment:
    POSTGRES_DB: vizpoc
    POSTGRES_USER: ${DB_USER}
    POSTGRES_PASSWORD: ${DB_PASSWORD}
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
  networks:
    - app-network
  restart: unless-stopped

redis:
  image: redis:7-alpine
  networks:
    - app-network
  restart: unless-stopped
  command: redis-server --appendonly yes
  volumes:
    - redis_data:/data

backend:
  build:
    context: ./backend
    dockerfile: Dockerfile.prod
  environment:
    - DB_HOST=postgres
    - DB_NAME=vizpoc
    - DB_USER=${DB_USER}
    - DB_PASSWORD=${DB_PASSWORD}
    - REDIS_HOST=redis
    - OPENAI_API_KEY=${OPENAI_API_KEY}
    - ENVIRONMENT=production
  depends_on:
    - postgres
    - redis
  networks:
    - app-network
  restart: unless-stopped
  deploy:
    replicas: 2
    resources:
      limits:
          memory: 512M
        reservations:
          memory: 256M

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile.prod
    environment:
      - REACT_APP_API_URL=${API_URL}
    depends_on:
      - backend
    networks:
      - app-network
    restart: unless-stopped

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    depends_on:
      - frontend
      - backend
    networks:
      - app-network
    restart: unless-stopped

networks:
  app-network:
    driver: bridge

volumes:
  postgres_data:
  redis_data:
```

9.2 Production Environment Variables

Create `.env.prod`:

```
# Database
DB_USER=prod_user
DB_PASSWORD=secure_password_here
DB_NAME=vizpoc_prod

# API Configuration
API_URL=https://your-domain.com/api
OPENAI_API_KEY=your_openai_key_here

# Security
JWT_SECRET=your_jwt_secret_here
CORS_ORIGINS=https://your-domain.com

# Monitoring
SENTRY_DSN=your_sentry_dsn_here
LOG_LEVEL=INFO

# Performance
REDIS_MAX_CONNECTIONS=20
DB_POOL_SIZE=20
```

9.3 Nginx Configuration

Create `nginx/nginx.conf`:

```
upstream backend {
    server backend:8000;
}

upstream frontend {
    server frontend:80;
}

server {
    listen 80;
    server_name your-domain.com;

    # Redirect HTTP to HTTPS
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
```

```
server_name your-domain.com;

ssl_certificate /etc/nginx/ssl/cert.pem;
ssl_certificate_key /etc/nginx/ssl/key.pem;

# Security headers
add_header X-Frame-Options DENY;
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains";

# API routes
location /api/ {
    proxy_pass http://backend/;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    # Timeouts
    proxy_read_timeout 300s;
    proxy_connect_timeout 75s;

    # Rate limiting
    limit_req zone=api burst=100 nodelay;
}

# Frontend routes
location / {
    proxy_pass http://frontend;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    # Enable gzip compression
    gzip on;
    gzip_types text/plain text/css application/json application/javascript;
}

# Health check endpoint
location /health {
    access_log off;
    return 200 "healthy\n";
```

```
        add_header Content-Type text/plain;
    }
}

# Rate limiting zones
http {
    limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;

    # Logging
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}
```

Step 10: Final Testing & Go-Live Checklist

10.1 Pre-Production Checklist

Create `deployment/go-live-checklist.md`:

```
# Go-Live Checklist

## Infrastructure ✅
- [ ] Production database configured with proper backup strategy
- [ ] Redis cache configured and tested
- [ ] SSL certificates installed and tested
- [ ] Domain DNS configured correctly
- [ ] Load balancer/reverse proxy configured
- [ ] Monitoring and alerting setup

## Security ✅
- [ ] Environment variables secured (no secrets in code)
- [ ] API rate limiting configured
- [ ] CORS policies configured correctly
- [ ] SQL injection testing completed
- [ ] Authentication/authorization implemented (if required)
- [ ] Security headers configured in Nginx
- [ ] Input validation implemented for all endpoints
- [ ] Error messages don't expose sensitive information

## Performance ✅
- [ ] Database queries optimized with proper indexes
- [ ] Caching strategy implemented and tested
```

- [ ] Static assets optimized and compressed
- [ ] CDN configured for static content (if applicable)
- [ ] Load testing completed with acceptable results
- [ ] Memory and CPU usage monitored under load

## Functionality ✅
- [ ] All core features working in production environment
- [ ] Text-to-visualization queries working correctly
- [ ] Database connections stable
- [ ] Error handling working properly
- [ ] Logging capturing necessary information
- [ ] Health check endpoints responding correctly

## Monitoring & Maintenance ✅
- [ ] Application logs centralized and searchable
- [ ] Performance metrics being collected
- [ ] Database backup and restore procedures tested
- [ ] Rollback plan documented and tested
- [ ] On-call procedures established
- [ ] Documentation updated and accessible

## Business Requirements ✅
- [ ] PowerBI integration tested (if applicable)
- [ ] User acceptance testing completed
- [ ] Performance requirements met
- [ ] Scalability requirements validated
- [ ] Data accuracy verified
- [ ] User training materials prepared

10.2 Smoke Tests for Production

Create `tests/smoke_tests.py`:

```
#!/usr/bin/env python3
"""
Production smoke tests - run these after deployment to verify basic functionality
"""
import requests
import sys
import time
from typing import Dict, Any

class ProductionSmokeTests:
```

```
def __init__(self, base_url: str):
    self.base_url = base_url.rstrip('/')
    self.session = requests.Session()
    self.session.timeout = 30

def test_health_endpoints(self) -> bool:
    """Test all health check endpoints"""
    endpoints = ['/health', '/api/health']

    for endpoint in endpoints:
        try:
            response = self.session.get(f"{self.base_url}{endpoint}")
            if response.status_code != 200:
                print(f"❌ Health check failed: {endpoint} returned {response.status_code}")
                return False
            print(f"✅ Health check passed: {endpoint}")
        except Exception as e:
            print(f"❌ Health check error: {endpoint} - {e}")
            return False
    return True

def test_database_connectivity(self) -> bool:
    """Test database connectivity through API"""
    try:
        response = self.session.get(f"{self.base_url}/api/data/sales")
        if response.status_code != 200:
            print(f"❌ Database connectivity test failed: {response.status_code}")
            return False

        data = response.json()
        if 'data' not in data or len(data['data']) == 0:
            print("❌ Database connectivity test failed: No data returned")
            return False

        print("✅ Database connectivity test passed")
        return True
    except Exception as e:
        print(f"❌ Database connectivity test error: {e}")
        return False

def test_core_functionality(self) -> bool:
    """Test core text-to-viz functionality"""
    test_queries = [
        "show me sales by region",
        "sales over time",
        "display sales data"
    ]

    for query in test_queries:
        try:
            response = self.session.post(
                f"{self.base_url}/api/query/text-to-viz",
                json={"query": query}
            )

            if response.status_code != 200:
                print(f"❌ Core functionality test failed for '{query}': {response.status_code}")
                return False

            result = response.json()
            required_fields = ['chart_type', 'data', 'config']
            if not all(field in result for field in required_fields):
                print(f"❌ Core functionality test failed for '{query}': Missing required fields")
                return False

            print(f"✅ Core functionality test passed for '{query}'")
        except Exception as e:
            print(f"❌ Core functionality test error for '{query}': {e}")
            return False

    return True

def test_frontend_accessibility(self) -> bool:
    """Test that frontend is accessible"""
    try:
        response = self.session.get(self.base_url)
        if response.status_code != 200:
            print(f"❌ Frontend accessibility test failed: {response.status_code}")
            return False

        # Check if it contains expected content
        if "AI-Powered Data Visualization" not in response.text:
            print("❌ Frontend accessibility test failed: Expected content not found")
            return False

        print("✅ Frontend accessibility test passed")
        return True
    except Exception as e:
        print(f"❌ Frontend accessibility test error: {e}")
        return False

def test_performance_baseline(self) -> bool:
    """Test basic performance requirements"""
    endpoint = f"{self.base_url}/api/query/text-to-viz"
    payload = {"query": "show me sales by region"}

    response_times = []
    for _ in range(5):
        start_time = time.time()
        try:
            response = self.session.post(endpoint, json=payload)
            if response.status_code != 200:
                print(f"❌ Performance test failed: {response.status_code}")
                return False
        except Exception as e:
            print(f"❌ Performance test error: {e}")
            return False

        response_time = time.time() - start_time
        response_times.append(response_time)

    avg_response_time = sum(response_times) / len(response_times)
    max_response_time = max(response_times)

    # Performance thresholds
    if avg_response_time > 5.0:  # 5 seconds average
        print(f"❌ Performance test failed: Average response time {avg_response_time:.2f}s >
5.0s")
        return False

    if max_response_time > 10.0:  # 10 seconds max
        print(f"❌ Performance test failed: Max response time {max_response_time:.2f}s >
10.0s")
        return False

    print(f"✅ Performance test passed: Avg {avg_response_time:.2f}s, Max
{max_response_time:.2f}s")
    return True

def run_all_tests(self) -> bool:
    """Run all smoke tests"""
    print(f"🚀 Running production smoke tests for {self.base_url}")
    print("=" * 60)

    tests = [
        ("Health Endpoints", self.test_health_endpoints),
        ("Database Connectivity", self.test_database_connectivity),
        ("Core Functionality", self.test_core_functionality),
        ("Frontend Accessibility", self.test_frontend_accessibility),
        ("Performance Baseline", self.test_performance_baseline)
    ]

    all_passed = True
    for test_name, test_func in tests:
        print(f"\n🧪 Running {test_name} test...")
        try:
            if not test_func():
                all_passed = False
        except Exception as e:
            print(f"❌ {test_name} test crashed: {e}")
            all_passed = False

    print("\n" + "=" * 60)
    if all_passed:
        print("🎉 All smoke tests passed! System is ready for production.")
        return True
    else:
        print("💥 Some smoke tests failed! Please fix issues before going live.")
        return False

def main():
if len(sys.argv) != 2:
    print("Usage: python smoke_tests.py <base_url>")
    print("Example: python smoke_tests.py https://your-domain.com")
    sys.exit(1)

base_url = sys.argv[1]
smoke_tests = ProductionSmokeTests(base_url)

success = smoke_tests.run_all_tests()
sys.exit(0 if success else 1)

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

10.3 PowerBI Integration Module

Create `powerbi/custom_visual.ts`:

```
/*
 * PowerBI Custom Visual for AI-Powered Data Visualization
 */

module powerbi.extensibility.visual {
    interface ViewModel {
        query: string;
        chartData: any;
        isLoading: boolean;
    }

    export class AiVizCustomVisual implements IVisual {
        private target: HTMLElement;
        private host: IVisualHost;
        private apiEndpoint: string = "https://your-api-domain.com/api";

        constructor(options: VisualConstructorOptions) {
            this.target = options.element;
            this.host = options.host;
            this.initializeUI();
        }

        private initializeUI(): void {
            // Create container
            const container = document.createElement('div');
            container.className = 'ai-viz-container';
            container.innerHTML = `
                <div class="query-section">
                    <input type="text" id="queryInput" placeholder="Enter your data visualization
query..." />
                    <button id="generateBtn">Generate Visualization</button>
                </div>
                <div id="loadingSpinner" class="loading hidden">Generating visualization...</div>
                <div id="chartContainer" class="chart-container"></div>
                <div id="errorMessage" class="error hidden"></div>
            `;

            this.target.appendChild(container);
            this.attachEventListeners();
        }
```

```
private attachEventListeners(): void {
    const generateBtn = this.target.querySelector('#generateBtn') as HTMLButtonElement;
    const queryInput = this.target.querySelector('#queryInput') as HTMLInputElement;

    generateBtn.addEventListener('click', () => this.generateVisualization());
    queryInput.addEventListener('keypress', (e) => {
        if (e.key === 'Enter') {
            this.generateVisualization();
        }
    });
}

private async generateVisualization(): Promise<void> {
    const queryInput = this.target.querySelector('#queryInput') as HTMLInputElement;
    const query = queryInput.value.trim();

    if (!query) return;

    this.showLoading(true);
    this.hideError();

    try {
        const response = await fetch(`${this.apiEndpoint}/query/text-to-viz`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ query })
        });

        if (!response.ok) {
            throw new Error(`API request failed: ${response.status}`);
        }

        const chartConfig = await response.json();
        this.renderChart(chartConfig);
    } catch (error) {
        this.showError(`Failed to generate visualization: ${error.message}`);
    } finally {
        this.showLoading(false);
    }
}
```

```
private renderChart(chartConfig: any): void {
    const chartContainer = this.target.querySelector('#chartContainer') as HTMLElement;
    chartContainer.innerHTML = '';

    // Convert our chart config to PowerBI visual format
    switch (chartConfig.chart_type) {
        case 'bar':
            this.renderBarChart(chartContainer, chartConfig);
            break;
        case 'line':
            this.renderLineChart(chartContainer, chartConfig);
            break;
        case 'pie':
            this.renderPieChart(chartContainer, chartConfig);
            break;
        default:
            this.renderTable(chartContainer, chartConfig);
    }
}

private renderBarChart(container: HTMLElement, config: any): void {
    // Use D3.js or your preferred charting library
    const chart = document.createElement('div');
    chart.innerHTML = `<h3>${config.config.title}</h3>`;

    // Create simple bar chart representation
    const data = config.data;
    const maxValue = Math.max(...data.map(d => d[config.config.y] || 0));

    data.forEach(item => {
        const bar = document.createElement('div');
        bar.className = 'bar-item';
        const percentage = (item[config.config.y] / maxValue) * 100;
        bar.innerHTML = `
            <div class="bar-label">${item[config.config.x]}</div>
            <div class="bar" style="width: ${percentage}%; background: #0078d4;">
                <span class="bar-value">${item[config.config.y]}</span>
            </div>
        `;
        chart.appendChild(bar);
    });

    container.appendChild(chart);
}
```

```
private renderLineChart(container: HTMLElement, config: any): void {
    // Implement line chart rendering
    const chart = document.createElement('div');
    chart.innerHTML = `<h3>${config.config.title}</h3><p>Line chart visualization</p>`;
    container.appendChild(chart);
}

private renderPieChart(container: HTMLElement, config: any): void {
    // Implement pie chart rendering
    const chart = document.createElement('div');
    chart.innerHTML = `<h3>${config.config.title}</h3><p>Pie chart visualization</p>`;
    container.appendChild(chart);
}

private renderTable(container: HTMLElement, config: any): void {
    const table = document.createElement('table');
    table.className = 'data-table';

    if (config.data && config.data.length > 0) {
        // Create header
        const header = table.createTHead();
        const headerRow = header.insertRow();
        Object.keys(config.data[0]).forEach(key => {
            const th = document.createElement('th');
            th.textContent = key;
            headerRow.appendChild(th);
        });

        // Create body
        const tbody = table.createTBody();
        config.data.forEach(row => {
            const tr = tbody.insertRow();
            Object.values(row).forEach(value => {
                const td = tr.insertCell();
                td.textContent = String(value);
            });
        });
    }

    container.appendChild(table);
}

private showLoading(show: boolean): void {
```

```
            const spinner = this.target.querySelector('#loadingSpinner') as HTMLElement;
            if (show) {
                spinner.classList.remove('hidden');
            } else {
                spinner.classList.add('hidden');
            }
        }

        private showError(message: string): void {
            const errorDiv = this.target.querySelector('#errorMessage') as HTMLElement;
            errorDiv.textContent = message;
            errorDiv.classList.remove('hidden');
        }

        private hideError(): void {
            const errorDiv = this.target.querySelector('#errorMessage') as HTMLElement;
            errorDiv.classList.add('hidden');
        }

        public update(options: VisualUpdateOptions) {
            // Handle PowerBI data updates
            const dataView = options.dataViews[0];
            if (dataView) {
                // Process PowerBI data if needed
                console.log('PowerBI data updated:', dataView);
            }
        }

        public destroy(): void {
            // Cleanup when visual is removed
        }
    }
}
```

10.4 Deployment Scripts

Create `deployment/deploy.sh`:

```
#!/bin/bash

set -e  # Exit on any error

echo "🚀 Starting deployment process..."
```

```
# Configuration
ENVIRONMENT=${1:-staging}
PROJECT_NAME="ai-viz-poc"
BACKUP_DIR="/backups/$(date +%Y%m%d_%H%M%S)"

echo "Deploying to: $ENVIRONMENT"

# Pre-deployment checks
echo "📋 Running pre-deployment checks..."

# Check if docker-compose is available
if ! command -v docker-compose &> /dev/null; then
   echo "❌ docker-compose is not installed"
   exit 1
fi

# Check if required files exist
REQUIRED_FILES=("docker-compose.yml" ".env.$ENVIRONMENT")
for file in "${REQUIRED_FILES[@]}"; do
   if [ ! -f "$file" ]; then
      echo "❌ Required file missing: $file"
      exit 1
   fi
done

echo "✅ Pre-deployment checks passed"

# Backup current deployment (if exists)
if [ "$ENVIRONMENT" = "production" ]; then
   echo "💾 Creating backup..."
   mkdir -p "$BACKUP_DIR"

   # Backup database
   docker-compose exec -T postgres pg_dump -U admin vizpoc >
"$BACKUP_DIR/database_backup.sql" || echo "⚠️  Database backup failed (container might
not be running)"

   # Backup environment file
   cp ".env.$ENVIRONMENT" "$BACKUP_DIR/" || echo "⚠️  Environment file backup failed"

   echo "✅ Backup created at $BACKUP_DIR"
fi
```

```
# Build and deploy
echo "🔨 Building and deploying..."

# Stop existing containers
docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT down

# Pull latest images and build
docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT pull
docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT build --no-cache

# Start services
docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT up -d

# Wait for services to be ready
echo "⏳ Waiting for services to be ready..."
sleep 30

# Run smoke tests
echo "🧪 Running post-deployment smoke tests..."
if [ "$ENVIRONMENT" = "production" ]; then
  python3 tests/smoke_tests.py "https://your-production-domain.com"
else
  python3 tests/smoke_tests.py "http://localhost"
fi

if [ $? -eq 0 ]; then
  echo "🎉 Deployment successful!"

  # Clean up old images
  echo "🧹 Cleaning up old Docker images..."
  docker image prune -f

  echo "✅ Deployment completed successfully for $ENVIRONMENT"
else
  echo "💥 Smoke tests failed! Rolling back..."

  # Rollback
  docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT down

  if [ "$ENVIRONMENT" = "production" ] && [ -d "$BACKUP_DIR" ]; then
    echo "🔄 Restoring from backup..."
    # Restore database if backup exists
    if [ -f "$BACKUP_DIR/database_backup.sql" ]; then
      docker-compose -f docker-compose.yml --env-file .env.$ENVIRONMENT up -d postgres
```

```
            sleep 10
            docker-compose exec -T postgres psql -U admin -d vizpoc <
"$BACKUP_DIR/database_backup.sql"
        fi
    fi

    exit 1
fi
```

10.5 Monitoring and Alerting Setup

Create `monitoring/prometheus.yml`:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "alert_rules.yml"

scrape_configs:
  - job_name: 'ai-viz-backend'
    static_configs:
      - targets: ['backend:8000']
    metrics_path: '/metrics'
    scrape_interval: 5s

  - job_name: 'postgres'
    static_configs:
      - targets: ['postgres:5432']

  - job_name: 'redis'
    static_configs:
      - targets: ['redis:6379']

alerting:
  alertmanagers:
    - static_configs:
        - targets:
          - alertmanager:9093
```

Create `monitoring/alert_rules.yml`:

```
groups:
  - name: ai-viz-alerts
    rules:
      - alert: HighResponseTime
        expr: histogram_quantile(0.95, rate(http_request_duration_seconds_bucket[5m])) > 5
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High response time detected"
          description: "95th percentile response time is {{ $value }}s"

      - alert: HighErrorRate
        expr: rate(http_requests_total{status=~"5.."}[5m]) / rate(http_requests_total[5m]) > 0.1
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value | humanizePercentage }}"

      - alert: DatabaseDown
        expr: up{job="postgres"} == 0
        for: 30s
        labels:
          severity: critical
        annotations:
          summary: "Database is down"
          description: "PostgreSQL database is not responding"

      - alert: RedisDown
        expr: up{job="redis"} == 0
        for: 30s
        labels:
          severity: warning
        annotations:
          summary: "Redis cache is down"
          description: "Redis cache is not responding"
```

Final Testing Commands

Complete System Test

```
# 1. Start the complete system
docker-compose up -d

# 2. Wait for services to be ready
sleep 30

# 3. Run comprehensive tests
python tests/e2e_test.py
python tests/benchmark.py
python tests/smoke_tests.py http://localhost

# 4. Test frontend manually
open http://localhost:3000

# 5. Test PowerBI integration (if available)
# Deploy custom visual and test with sample data

# 6. Performance validation
curl -w "@curl-format.txt" -s -o /dev/null http://localhost:8000/health
```

Debug Commands Reference

```
# View all container logs
docker-compose logs -f

# Check individual service logs
docker-compose logs backend
docker-compose logs frontend
docker-compose logs postgres

# Monitor resource usage
docker stats

# Connect to database
docker-compose exec postgres psql -U admin -d vizpoc

# Check Redis cache
docker-compose exec redis redis-cli

# View application metrics
curl http://localhost:8000/metrics

# Test API endpoints
curl -X POST "http://localhost:8000/query/text-to-viz" \
  -H "Content-Type: application/json" \
  -d '{"query": "show me sales by region"}'
```

Success Metrics & KPIs

Track these metrics to measure PoC success:

Technical Metrics

- **Response Time**: < 3 seconds average for text-to-viz queries
- **Uptime**: > 99.5% availability
- **Error Rate**: < 1% of requests
- **Concurrent Users**: Support 100+ concurrent users
- **Data Accuracy**: 100% accurate SQL generation and chart rendering

Business Metrics

- **Query Success Rate**: > 95% of natural language queries produce useful visualizations
- **User Satisfaction**: Measured through feedback forms
- **PowerBI Integration**: Seamless integration with existing BI workflows
- **Time to Insight**: Reduce time from question to visualization by 80%

This completes your comprehensive step-by-step implementation guide for the AI-powered data visualization PoC. Each step includes detailed testing procedures, debugging commands, and validation criteria to ensure a successful deployment.