# Manike B2B AI Engine

## Project Workflow Documentation

*Multi-Tenant AI Scene Orchestrator & Itinerary Engine*

Version: 2.0.0

Architecture: FastAPI + PostgreSQL + Milvus

Date: February 2026

# 1. Architecture Overview

The Manike B2B AI Engine is a multi-tenant backend system designed for travel companies. It uses a hybrid database architecture combining PostgreSQL for structured data and Milvus for vector-based semantic search. The system orchestrates multiple AI services to generate cinematic travel content including images, videos, and itineraries.

### Technology Stack

- Framework: FastAPI (Python)
- Relational DB: PostgreSQL (via SQLModel/SQLAlchemy)
- Vector DB: Milvus (for AI-powered semantic search)
- Authentication: JWT (JSON Web Tokens)
- Media Processing: FFmpeg
- Storage: S3-compatible object storage
- AI Models: LLM (GPT-4/Claude), Image Gen, Video Gen (pluggable)

### Project Structure

```
menike_backend_poc/
|-- app/
|   |-- api/              # API endpoint routers
|   |   |-- auth.py       # Login & JWT token generation
|   |   |-- scenes.py     # Scene orchestration endpoints
|   |   |-- itinerary.py  # Itinerary generation endpoints
|   |   |-- experiences.py# Experience CRUD (Milvus)
|   |   |-- tenants.py    # Tenant management (Milvus)
|   |-- core/             # Core infrastructure
|   |   |-- auth.py       # JWT logic & password hashing
|   |   |-- database.py   # PostgreSQL engine & sessions
|   |   |-- milvus_client.py # Milvus connection & queries
|   |-- models/           # Data models
|   |   |-- sql_models.py # Tenant, User, Scene (PostgreSQL)
|   |   |-- milvus_schema.py # Vector collection schemas
|   |-- services/         # Business logic
|   |   |-- orchestrator.py  # AI scene pipeline
|   |   |-- generators.py    # LLM, Image, Video generators
|   |   |-- media_processor.py # FFmpeg wrapper
|   |   |-- storage.py       # S3 file upload
|   |-- main.py           # FastAPI app entry point
|-- seed_db.py            # Database seeding script
|-- requirements.txt      # Python dependencies
|-- .env                  # Environment variables
```

## 2. Complete Working Flow

The following describes the end-to-end flow of how a request travels through the system, from user authentication to AI content generation and storage.

### Phase A: Authentication Flow

#### 1 User Login

The client sends a POST request to /auth/login with email and password. The system verifies the credentials against the PostgreSQL 'user' table using PBKDF2_SHA256 password hashing.

#### 2 JWT Token Generation

On successful authentication, the server generates a JWT token containing the user's ID and tenant_id. This token is valid for 24 hours and must be included in the Authorization header of all subsequent requests.

#### 3 Tenant Extraction

Every protected endpoint uses the get_current_tenant_id dependency, which decodes the JWT token, retrieves the user from PostgreSQL, and returns their tenant_id. This ensures complete data isolation between tenants.

### Phase B: Scene Orchestration Flow (Core AI Pipeline)

#### 4 Scene Request

The authenticated client sends a POST request to /scenes/ with a scene name and description (e.g., 'Galle Fort Sunset' with a cinematic description).

#### 5 Database Record Creation

The SceneOrchestrator creates a new Scene record in PostgreSQL with status='pending', linked to the authenticated tenant_id. It then updates the status to 'processing'.

#### 6 LLM Prompt Engineering

The LLMPromptEngine analyzes the scene description and generates structured prompts:

- image_prompt: Optimized for photorealistic image generation
- video_prompt: Optimized for cinematic video generation
- negative_prompt: Quality guard rails for AI models

#### 7 AI Image Generation

The ImageGenerator receives the image_prompt and produces a high-quality travel image. Currently uses a mock implementation; in production, this connects to Stable Diffusion, DALL-E, or Leonardo AI.

#### 8 AI Video Generation

The VideoGenerator receives the video_prompt and produces a cinematic video clip. Currently uses a mock; in production, connects to Veo 3 or Runway ML.

### 9 Media Processing (FFmpeg)

The MediaProcessor uses FFmpeg to optimize the generated video:

- Compression for web streaming
- Format standardization (MP4/H.264)
- Resolution and bitrate optimization

### 10 Storage Upload

The optimized media file is uploaded to S3-compatible storage. A public URL is generated and stored back in the Scene record in PostgreSQL.

### 11 Completion

The Scene record is updated with status='completed', the media_url, and a timestamp. The complete scene object is returned to the client.

## Phase C: Itinerary Generation Flow

### 12 Itinerary Request

The client sends a POST to /itinerary/generate with destination, number of days, and interests. The system generates a structured multi-day travel plan.

### 13 AI Planning

The LLM creates a logical itinerary with activities, travel times, and distances. It also generates image prompts for each activity for visual enrichment.

## Phase D: Experience Vector Search Flow

### 14 Experience Storage

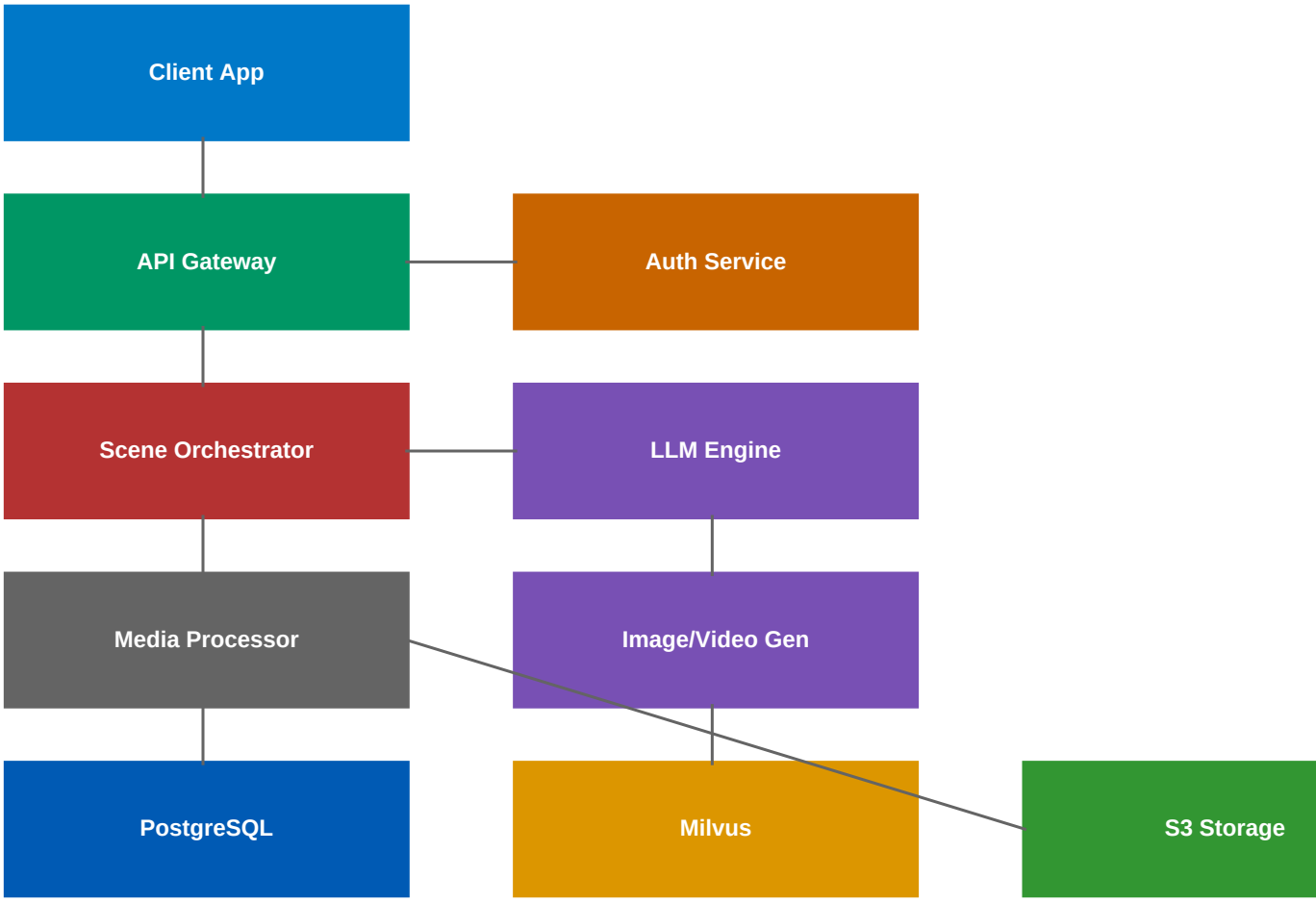
Travel experiences are stored in Milvus as vector embeddings (768-dimensional). Each experience is tagged with a tenant_id for data isolation.

### 15 Semantic Search

The client can search for similar experiences using POST /experiences/search/ with a query vector. Milvus performs HNSW-based approximate nearest neighbor search filtered by tenant_id, returning the most relevant results ranked by cosine similarity.

## 3. Data Flow Diagram

The following illustrates how data moves through the system layers:



## 4. API Endpoints Summary

| Method | Endpoint | Description |
|---|---|---|
| POST | `/auth/login` | Authenticate user, returns JWT token |
| POST | `/scenes/` | Create a new AI-orchestrated scene |
| GET | `/scenes/` | List all scenes for the authenticated tenant |
| POST | `/itinerary/generate` | Generate a multi-day travel itinerary |
| GET | `/experiences/` | List all experiences from Milvus |
| POST | `/experiences/` | Create a new experience in Milvus |
| POST | `/experiences/search/` | Semantic vector search for experiences |
| POST | `/tenants/` | Create a new tenant in Milvus |
| GET | `/tenants/` | List all tenants |
| GET | `/tenants/{id}` | Get a specific tenant by ID |
| PUT | `/tenants/{id}` | Update a tenant |
| DELETE | `/tenants/{id}` | Delete a tenant |

## 5. Database Models

### PostgreSQL Tables

- Tenant: id, name, api_key, config, created_at
- User: id, tenant_id (FK), email, hashed_password, role, created_at
- Scene: id, tenant_id, name, description, status, media_url, created_at, updated_at

### Milvus Collections

- experiences: id, tenant_id, embedding (768-dim), metadata (JSON), slug
- tenants: id, name, apikey, metadata (JSON), embedding (2-dim placeholder)

# 6. How to Run the Project

### Prerequisites

- Python 3.10+
- Docker Desktop (for PostgreSQL and Milvus)
- pip (Python package manager)

### Step 1: Start Docker Services

```
# Start PostgreSQL
docker run -d --name postgres_local \
  -e POSTGRES_USER=admin \
  -e POSTGRES_PASSWORD=admin \
  -e POSTGRES_DB=menike \
  -p 5432:5432 postgres:latest

# Start Milvus (using docker-compose)
# Follow: https://milvus.io/docs/install_standalone-docker.md
```

### Step 2: Configure Environment

```
# .env file contents:
DATABASE_URL=postgresql://admin:admin@localhost:5432/menike
```

### Step 3: Install Dependencies & Seed

```
pip install -r requirements.txt
python seed_db.py
```

### Step 4: Run the Server

```
export PYTHONPATH=$PYTHONPATH:$(pwd)
python app/main.py

# Server starts at http://localhost:8000
# Swagger UI at http://localhost:8000/docs
```

### Step 5: Test the Flow via Swagger

1. Open http://localhost:8000/docs
2. POST /auth/login with: username=admin@manike.ai, password=admin123
3. Copy the access_token from the response
4. Click 'Authorize' button, paste the token
5. POST /scenes/ to create a scene
6. GET /scenes/ to verify the scene was created