Job Trends Web Application

Database Design & Architecture Report

Ankita Suresh Kumar
Frontend & API Lead

Krishi Thiruppathi
Backend & Database Lead

December 14, 2025

Abstract

Abstract — This report outlines the technical architecture of the Job Trends Web Application, a system designed to analyze global employment data. It details the transition from unstructured CSV data to a strict Third Normal Form (3NF) relational schema, the implementation of a robust Extract-Transform-Load (ETL) pipeline, and the development of a FastAPI backend. The resulting system ensures high data integrity through multi-layer constraints and delivers real-time analytical insights via optimized SQL queries.

1 Purpose

The primary purpose of the Job Trends Web Application is to provide a centralized, normalized platform for analyzing global employment statistics. By converting raw, unstructured CSV data into a structured relational database, the application allows users to query job market trends efficiently.

The target audience includes:

- **Data Analysts:** Who require clean, de-duplicated data for accurate reporting on salary and skill demand.
- **Job Seekers:** Who need to visualize which industries and locations offer the best opportunities.
- **Researchers:** Looking for aggregate trends in employment types and remote work settings.

2 App Functionalities

The application provides a comprehensive suite of features for managing and visualizing job data:

1. **Job Management (CRUD):** Users can Create, Read, Update, and Delete job listings via a user-friendly interface. This includes validation to ensure data integrity (e.g., preventing duplicate company names).

2. **Data Visualization:** The dashboard provides real-time analytics, including "Jobs per Skill" and "Average Salary by Job Role" using interactive charts.

3. **Search & Filter:** Users can filter the database by company, location, or specific technical skills to find relevant opportunities.

3 Conceptual Schema & Diagram

In the domain of data analytics, the structure of the underlying storage engine dictates the performance and reliability of the application. This project utilizes a relational database (MySQL) to store, serve, and analyze job trend data. The system architecture is built upon a **FastAPI** backend application that communicates with the persistent storage layer via the **SQLAlchemy** Object Relational Mapper (ORM).

3.1 Architectural Philosophy: The ETL Pipeline

The core data pipeline follows a modern **Extract, Transform, Load (ETL)** philosophy designed to handle data volatility. The process is bifurcated into two distinct stages:

1. **Ingestion (Staging):** The raw dataset, `jobs_in_data.csv`, is loaded into a temporary staging table (`jobs_in_data`) within MySQL. This preserves the original data state for auditing purposes.

2. **Normalization (Transformation):** A comprehensive Python script processes the staging table. It programmatically cleanses the data, identifies unique entities, and distributes them into a normalized schema.

3.2 Database Normalization (3NF)

The database design strictly adheres to the **Third Normal Form (3NF)** standard. Given the repetitive nature of job data (e.g., thousands of listings for "Data Scientist" or "New York"), a denormalized structure would result in significant data redundancy and update anomalies. By decoupling repeating groups into distinct lookup tables, we eliminate redundancy and ensure that an update to a company's details is instantly reflected across all associated job

postings.

3.3 Entity Descriptions

The normalized schema is composed of six primary entities:

- **Companies:** A dedicated lookup table storing unique company names.
- **Locations:** A lookup table standardizing geographical data.
- **Industries:** A categorization table allowing for efficient filtering of jobs by market sector.
- **Skills:** A lookup table storing unique technical competencies extracted from job descriptions.
- **Jobs:** The central "Fact Table" of the database. It contains transactional attributes specific to a job listing (e.g., Salary, Description) and links to the dimension tables via foreign keys.
- **job_skills:** An associative (junction) table designed to resolve the Many-to-Many relationship between Jobs and Skills.

3.4 Entity-Relationship (ER) Diagram

The ER diagram below was generated directly from our live MySQL database using the MySQL Workbench "Reverse Engineer" tool. It visually represents the final, normalized 6-table schema.

4 Data Integrity & Constraints

Data integrity is not merely a feature but a foundational requirement. To ensure reliability, we implemented a "Defense in Depth" strategy, applying constraints at both the database level and the application level.

4.1 Database-Level Constraints

These constraints are enforced by the MySQL engine, preventing invalid data states regardless of the source of the query.

- **Referential Integrity (Foreign Keys):** The `Jobs` table references `Companies.Company_ID`. This configuration strictly prohibits the creation of "orphan" job listings that belong to non-existent companies.

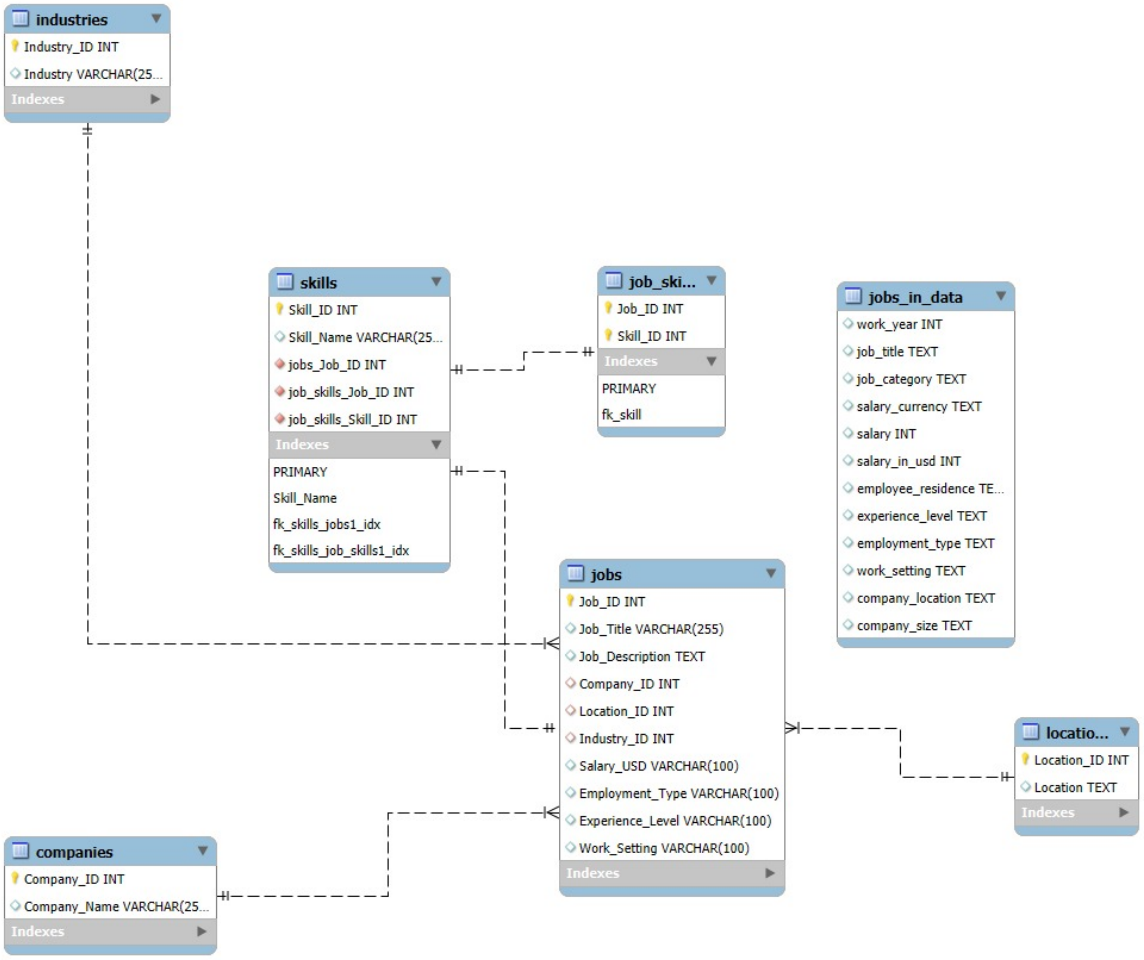

Figure 1: Entity-Relationship Diagram (Reverse Engineered)

- **Composite Primary Keys:** The `job_skills` junction table utilizes a Composite Key `(Job_ID, Skill_ID)`. This is critical for analytics, as it physically prevents a single skill from being counted twice for the same job.
- **Cascading Deletes:** Relationships linking to the `job_skills` table utilize `ON DELETE CASCADE`. If a job is removed, the database automatically cleans up associated skill mappings.
- **Uniqueness:** All lookup tables enforce `UNIQUE` constraints on their name columns, preventing duplicate entries for the same entity.

4.2 Schema

```python
from pydantic import BaseModel, ConfigDict
from typing import List, Optional

# Shared properties
class JobBase(BaseModel):
    job_title: str
    location: str
```

```
    min_salary: Optional[int] = None
    max_salary: Optional[int] = None

    # Additional fields
    company_name: Optional[str] = None  # For creating jobs
    experience_level: Optional[str] = None
    work_setting: Optional[str] = None
    work_year: Optional[int] = None
    job_category: Optional[str] = None
    company_size: Optional[str] = None

class JobCreate(JobBase):
    skills: List[str] = []

class Skill(BaseModel):
    skill_name: str

    model_config = ConfigDict(from_attributes=True)  # Updated for Pydantic
    v2

class Company(BaseModel):
    company_name: str

    model_config = ConfigDict(from_attributes=True)  # Updated for Pydantic
    v2

class JobResponse(JobBase):
    job_id: int
    company: Optional[Company] = None
    skills: List[Skill] = []

    model_config = ConfigDict(from_attributes=True)  # Updated for Pydantic
    v2
```

Listing 1: Schema.py

4.3 Application-Level Constraints

Beyond the database, the FastAPI application implements a validation layer using **Pydantic** models (defined in `schemas.py`). This ensures that incoming payloads are type-checked and sanitized before a database transaction is ever attempted.

5 Back-end Implementation: Database Creation & Queries

The system deployment utilizes a hybrid approach: a Python/Pandas script handles structural setup and bulk migration, while raw SQL handles constraint enforcement.

5.1 The Staging & Normalization Process

The database instantiation uses a "Constraint-Last" loading strategy for performance optimization:

1. **DDL Execution:** The Python script executes Data Definition Language commands to create the 6 target tables *without* foreign keys.

2. **Bulk Loading:** Data is processed and inserted. Inserting data without active foreign key checks is significantly faster for large datasets.

3. **Constraint Application:** Once data is loaded, `ALTER TABLE` commands are executed to enable Foreign Keys, retroactively verifying the integrity of the bulk load.

5.2 CRUD API Implementation

The backend server (`app.py`) provides a comprehensive RESTful API. Below is the implementation logic for the four primary CRUD operations.

5.2.1 Create: POST /jobs

The creation endpoint accepts a validated JSON payload via the `JobCreate` Pydantic schema. It initializes a new SQLAlchemy model instance and commits it to the database.

```
@app.post("/jobs", response_model=schemas.JobResponse, status_code=201)
def create_job(job: schemas.JobCreate, db: Session = Depends(get_db)):
    db_job = models.Job(**job.model_dump())
    db.add(db_job)
    db.commit()
    db.refresh(db_job)
    return db_job
```

Listing 2: Create Job Endpoint

5.2.2 Read: GET /jobs/{id}

To retrieve a single job, the API employs SQLAlchemy's `joinedload` strategy. This performs an "Eager Load," fetching the Job alongside its related Company, Location, Industry, and Skills in a single SQL query to avoid performance bottlenecks.

```
@app.get("/jobs/{job_id}", response_model=schemas.JobResponse)
def get_job(job_id: int, db: Session = Depends(get_db)):
    job = (
        db.query(models.Job)
        .options(
            joinedload(models.Job.company),
            joinedload(models.Job.location),
            joinedload(models.Job.industry),
```

```
            joinedload(models.Job.skills)
        )
        .filter(models.Job.Job_ID == job_id)
        .first()
    )
    if not job:
        raise HTTPException(status_code=404, detail="Job not found")
    return job
```

Listing 3: Read Job (Eager Loading)

5.2.3 Update: PUT /jobs/{id}

The update endpoint utilizes the `JobUpdate` schema. It checks for the existence of the target job and iteratively updates only the fields provided in the request payload.

```
@app.put("/jobs/{job_id}", response_model=schemas.JobResponse)
def update_job(job_id: int, job_update: schemas.JobUpdate, db: Session =
    Depends(get_db)):
    db_job = db.query(models.Job).filter(models.Job.Job_ID == job_id).first
    ()
    if not db_job:
        raise HTTPException(status_code=404, detail="Job not found")

    update_data = job_update.model_dump(exclude_unset=True)
    for key, value in update_data.items():
        setattr(db_job, key, value)

    db.commit()
    db.refresh(db_job)
    return db_job
```

Listing 4: Update Job Endpoint

5.2.4 Delete: DELETE /jobs/{id}

The delete operation removes the job record. Crucially, because of the `ON DELETE CASCADE` constraint defined in our database schema, this action automatically cleans up all associated entries in the `job_skills` table.

```
@app.delete("/jobs/{job_id}", response_model=schemas.MessageResponse)
def delete_job(job_id: int, db: Session = Depends(get_db)):
    db_job = db.query(models.Job).filter(models.Job.Job_ID == job_id).first
    ()
    if not db_job:
        raise HTTPException(status_code=404, detail="Job not found")

    db.delete(db_job)
    db.commit()
```

```
    return {"message": "Job deleted successfully"}
```

Listing 5: Delete Job Endpoint

5.3 Analytical Aggregation

To power the visualization dashboard, the API provides specialized endpoints. The `/analytics/skills` endpoint demonstrates the power of the normalized schema, executing a 3-table join to aggregate skill demand across the entire dataset efficiently.

```
@app.get("/analytics/skills", response_model=List[schemas.AggResponse])
def agg_by_skill(db: Session = Depends(get_db)):
    # Analytical query: Counts jobs per skill.
    # This joins Jobs, job_skills, and Skills.
    query = (
        db.query(
            models.Skill.Skill_Name,
            func.count(models.Job.Job_ID).label("count")
        )
        .join(models.job_skills_table, models.Skill.Skill_ID == models.
    job_skills_table.c.Skill_ID)
        .join(models.Job, models.Job.Job_ID == models.job_skills_table.c.
    Job_ID)
        .group_by(models.Skill.Skill_Name)
        .order_by(func.count(models.Job.Job_ID).desc())
        .limit(20)
        .all()
    )
    return [{"label": r[0], "count": r[1]} for r in query]
```

Listing 6: Analytical Aggregation Query

6 Frontend Application Interface

The user interface is built with React (Vite) and serves as the presentation layer for the FastAPI backend. The application is divided into three primary functional modules: the Listing Dashboard, the Creation Interface, and the Analytical Suite.

6.1 Module 1: Job Dashboard

The Job Listings page acts as the central hub for browsing employment records. It features a responsive data grid with integrated search, sorting, and pagination capabilities.

- **Data Fetching:** Consumes the `GET /jobs` endpoint to retrieve the latest market data.
- **Client-Side Logic:** Implements real-time filtering that searches across job titles, companies, and locations simultaneously.

```
import React, { useState, useEffect } from "react";

export default function JobList() {
  const [jobs, setJobs] = useState([]);
  const [searchTerm, setSearchTerm] = useState("");
  const [filteredJobs, setFilteredJobs] = useState([]);

  // Fetching Data
  useEffect(() => {
    fetch("http://127.0.0.1:8000/jobs?limit=5000")
      .then((res) => res.json())
      .then((data) => {
        setJobs(data);
        setFilteredJobs(data);
      });
  }, []);

  // Real-time Search Filter
  useEffect(() => {
    if (jobs.length === 0) return;
    const filtered = jobs.filter(job => {
      const searchLower = searchTerm.toLowerCase();
      return (
        job.job_title?.toLowerCase().includes(searchLower) ||
        job.company?.company_name?.toLowerCase().includes(searchLower) ||
        job.location?.toLowerCase().includes(searchLower)
      );
    });
    setFilteredJobs(filtered);
  }, [searchTerm, jobs]);

  return (
```

```
    <div className="joblist-enhanced">
      <div className="controls-bar">
        <input
          type="text"
          placeholder="Search by title, location, company..."
          onChange={(e) => setSearchTerm(e.target.value)}
        />
      </div>
      {/* Table rendering logic... */}
    </div>
  );
}
```

Listing 7: JobList.jsx (Search & Filter Logic)

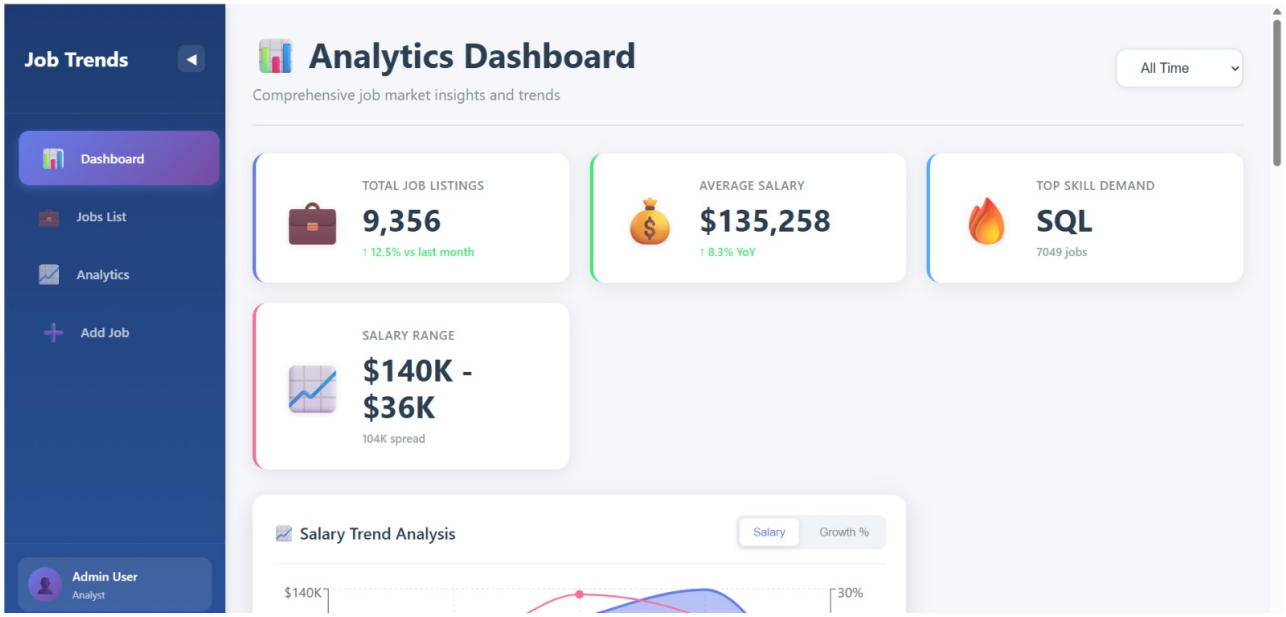

Figure 2: Frontend Interface: Job Listings Dashboard

6.2 Module 2: Job Creation Interface

The "Add Job" module allows users to input new market data into the system. It matches the backend Pydantic model structure, ensuring data validation (such as numeric salary fields) before submission.

```
import React, { useState } from "react";
import { useNavigate } from "react-router-dom";

export default function JobForm() {
  const navigate = useNavigate();
  const [form, setForm] = useState({
    job_title: "",
    min_salary: "",
    company_name: "",
    location: "",
    job_category: ""
  });

  const handleSubmit = (e) => {
    e.preventDefault();
    // Parse numeric inputs before sending to API
    const payload = {
        ...form,
        min_salary: parseInt(form.min_salary) || 0,
        max_salary: parseInt(form.max_salary) || 0,
    };

    fetch("http://127.0.0.1:8000/jobs", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload),
    })
    .then(res => {
        if(res.ok) navigate("/jobs");
    });
  };

  return (
    <div className="container">
      <form onSubmit={handleSubmit}>
        <input placeholder="Job Title" onChange={e => setForm({...form,
   job_title: e.target.value})} />
        <input placeholder="Min Salary" type="number" onChange={e => setForm
   ({...form, min_salary: e.target.value})} />
        <button type="submit">Create Job</button>
      </form>  </div>  );}
```

Listing 8: JobForm.jsx (Submission Logic)

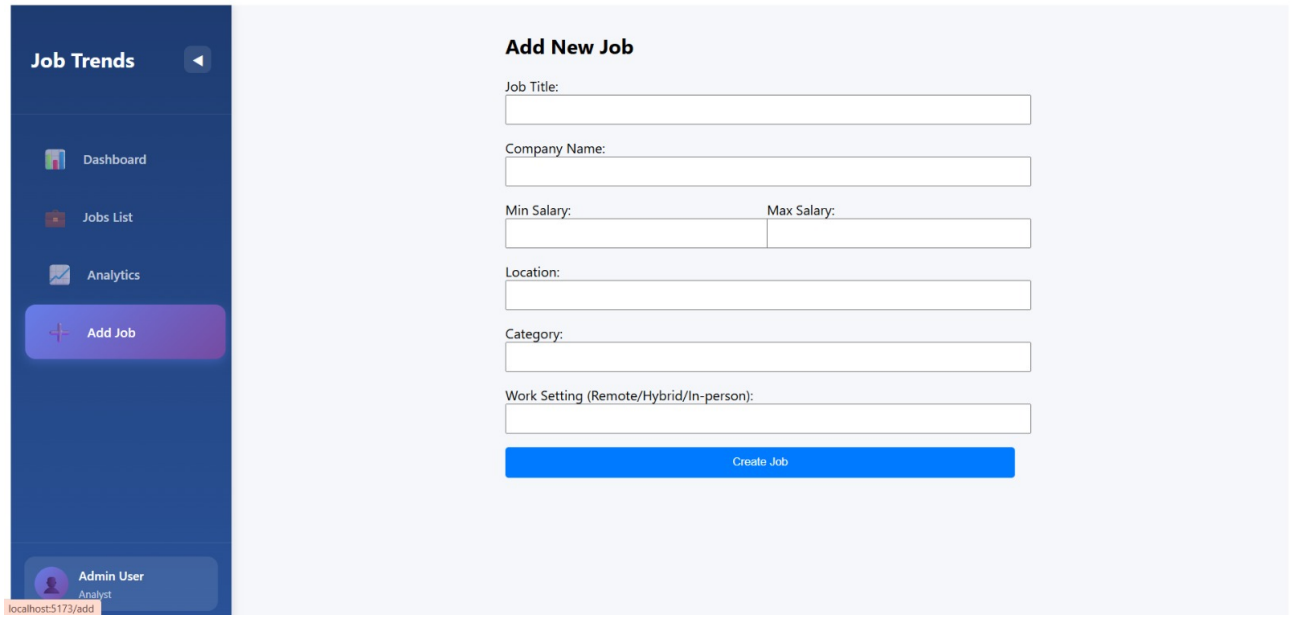

Figure 3: Frontend Interface: Add Job Form

6.3 Module 3: Job Listings Dashboard

The Job Listings page acts as the central hub for browsing employment records. It features a responsive data grid with integrated search, sorting, and pagination capabilities.

```
import React, { useState, useEffect } from "react";

export default function JobList() {
  const [jobs, setJobs] = useState([]);
  const [searchTerm, setSearchTerm] = useState("");
  const [filteredJobs, setFilteredJobs] = useState([]);

  // Fetching Data & Real-time Search Filter Logic
  useEffect(() => {
    fetch("http://127.0.0.1:8000/jobs?limit=5000")
      .then((res) => res.json())
      .then((data) => {
        setJobs(data);
        setFilteredJobs(data);
      });
  }, []);

  return (
    <div className="joblist-enhanced">
      <div className="controls-bar">
        <input
          type="text"
          placeholder="Search by title, location, company..."
          onChange={(e) => setSearchTerm(e.target.value)}
```

```
        />
      </div>
      {/* Table rendering logic... */}
    </div>
  );
}
```

Listing 9: JobList.jsx (Search & Filter Logic)

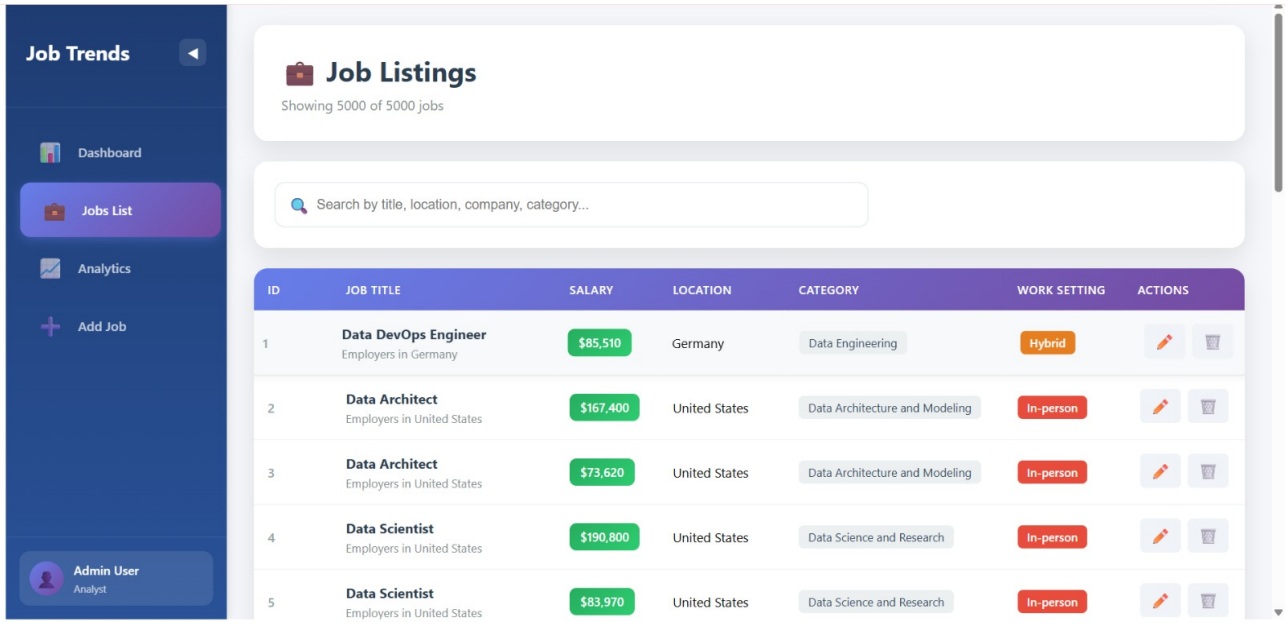

Figure 4: Frontend Interface: Job Listing

6.4 Module 4: Analytical Dashboard

The dashboard provides comprehensive market insights using the **Recharts** library. It fetches a pre-aggregated summary from the backend and visualizes key metrics such as salary trends, skill demand, and work setting distributions.

```
import React, { useEffect, useState } from "react";
import { ComposedChart, BarChart, Line, Bar, XAxis, YAxis, Tooltip } from "
    recharts";
import axios from "axios";

const Dashboard = () => {
  const [data, setData] = useState(null);

  // Fetch Analytics Summary
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get("http://127.0.0.1:8000/analytics/
    summary");
```

```
        setData(response.data);
      } catch (err) {
        console.error("Backend Connection Failed", err);
      }
    };
    fetchData();
  }, []);

  if (!data) return <div className="dashboard-loading">Loading...</div>;

  // Render Charts
  return (
    <div className="dashboard-enhanced">
      {/* KPI Cards */}
      <div className="kpi-grid">
        <div className="kpi-card">
            <h2>{data.total_jobs}</h2>
            <p>Total Job Listings</p>
        </div>
        <div className="kpi-card success">
            <h2>${data.avg_salary.toLocaleString()}</h2>
            <p>Average Salary</p>
        </div>
      </div>

      {/* Visualization Charts */}
      <div className="charts-container">
        <div className="chart-card">
            <h3>Top 10 In-Demand Skills</h3>
            <BarChart data={data.top_skills} layout="vertical">
                <XAxis type="number" />
                <YAxis type="category" dataKey="name" width={100} />
                <Tooltip />
                <Bar dataKey="count" fill="#667eea" />
            </BarChart>
        </div>
      </div>
    </div>
  );
};

export default Dashboard;
```

Listing 10: Dashboard.jsx (Visualization Logic)

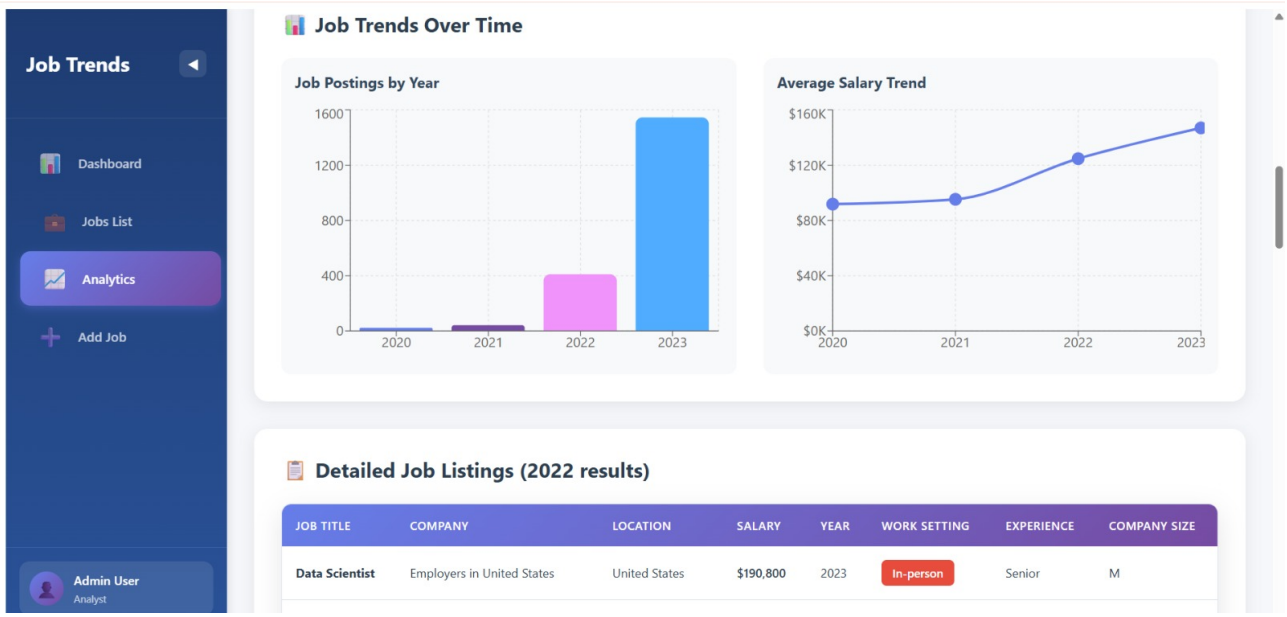

Figure 5: Frontend Interface: Analytical Dashboard

7 Contribution Summary

The project followed a strict separation of concerns, allowing for parallel development of the persistence layer and the presentation layer.

Name	Contribution Details	Effort
Krishi	**Backend & Database Architecture** • Architected the normalized 3NF database schema. • Developed the `normalize.py` ETL script for DDL/DML operations. • Implemented SQLAlchemy models and Pydantic schemas. • Engineered the complex analytical API endpoints.	9-12 hours
Ankita	**Frontend & API Integration** • Developed the React (Vite) frontend application. • Implemented CRUD endpoints in `app.py`. • Integrated the frontend with the backend via Axios. • Built UI components for data visualization and charts.	9-12 hours

Table 1: Team Contributions

8 Acknowledgements

We acknowledge the use of the following tools and resources:

- **AI Assistance:** Portions of the Python code, specifically the normalization logic and

complex SQLAlchemy query syntax, were generated, refactored, and debugged with the assistance of Google's Gemini (accessed November 2025).

- **Database Tooling:** MySQL Workbench (Reverse Engineer feature) was used to generate the ER diagrams.

9 Reflections

9.1 Learning Outcomes

Through this project, we gained a deep understanding of the practical application of database normalization. We learned that while 3NF introduces complexity in querying (requiring multiple joins), it vastly improves data integrity compared to flat files.

9.2 Challenges Faced & Deployment Issues

The most significant challenge was handling the deployment of the application.

- **Deployment Failures:** We attempted to deploy the application to a cloud provider, but encountered persistent errors related to the database connection string and environment variable injection in the production build.
- **Local Hosting Only:** Due to these unresolved deployment errors, the application is currently submitted as a locally hosted project. We have provided full screenshots to demonstrate functionality.
- **CORS Issues:** We also faced CORS (Cross-Origin Resource Sharing) errors when connecting the React frontend to the FastAPI backend locally, which were resolved by configuring `CORSMiddleware` in `app.py`.

9.3 Future Improvements

If we had more time, we would implement:

- **User Authentication:** Adding JWT-based login so only authorized users can Delete or Update jobs.
- **Advanced Filtering:** Allowing users to filter by salary ranges (e.g., > $100k) rather than just exact matches.

A Backend Source Code

The complete, reproducible source code is provided below. The full repository is available at:

https://github.com/ankita-suresh/job-trends.

https://github.com/krishi18/job-trends.

A.1 File: db.py (Configuration)

```python
# db.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base
from urllib.parse import quote_plus

DB_USER = "root"
DB_PASS = "pass123$" # Your actual password
DB_HOST = "127.0.0.1"
DB_PORT = 3306
DB_NAME = "job_trends"

# Use quote_plus for the password
DATABASE_URL = f"mysql+pymysql://{DB_USER}:{quote_plus(DB_PASS)}@{DB_HOST}:{DB_PORT}/{DB_NAME}?charset=utf8mb4"

engine = create_engine(DATABASE_URL, pool_pre_ping=True)
SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False)
Base = declarative_base()
```

A.2 File: models.py (ORM Models)

```python
# models.py
from sqlalchemy import Column, Integer, String, Text, ForeignKey, Table
from sqlalchemy.orm import relationship
from db import Base

# Junction Table (Many-to-Many)
job_skills_table = Table('job_skills', Base.metadata,
    Column('Job_ID', Integer, ForeignKey('jobs.Job_ID'), primary_key=True),
    Column('Skill_ID', Integer, ForeignKey('skills.Skill_ID'), primary_key=True)
)

# --- Lookup Tables ---
class Company(Base):
    __tablename__ = "companies"
    Company_ID = Column(Integer, primary_key=True, index=True)
    Company_Name = Column(String(255), unique=True)
    jobs = relationship("Job", back_populates="company")
```

```python

class Location(Base):
    __tablename__ = "locations"
    Location_ID = Column(Integer, primary_key=True, index=True)
    Location = Column(String(255), unique=True)
    jobs = relationship("Job", back_populates="location")

class Industry(Base):
    __tablename__ = "industries"
    Industry_ID = Column(Integer, primary_key=True, index=True)
    Industry = Column(String(255), unique=True)
    jobs = relationship("Job", back_populates="industry")

class Skill(Base):
    __tablename__ = "skills"
    Skill_ID = Column(Integer, primary_key=True, index=True)
    Skill_Name = Column(String(255), unique=True)
    jobs = relationship("Job", secondary=job_skills_table, back_populates="
   skills")

# --- Main Job Table ---
class Job(Base):
    __tablename__ = "jobs"
    Job_ID = Column(Integer, primary_key=True, index=True)
    Job_Title = Column(String(255))
    Job_Description = Column(Text, nullable=True)
    Salary_USD = Column(String(100), nullable=True)
    Employment_Type = Column(String(100), nullable=True)
    Experience_Level = Column(String(100), nullable=True)
    Work_Setting = Column(String(100), nullable=True)

    # Foreign Keys
    Company_ID = Column(Integer, ForeignKey('companies.Company_ID'))
    Location_ID = Column(Integer, ForeignKey('locations.Location_ID'))
    Industry_ID = Column(Integer, ForeignKey('industries.Industry_ID'))

    # Relationships
    company = relationship("Company", back_populates="jobs")
    location = relationship("Location", back_populates="jobs")
    industry = relationship("Industry", back_populates="jobs")
    skills = relationship("Skill", secondary=job_skills_table,
   back_populates="jobs")
```

A.3 File: schemas.py (Validation)

```python
# schemas.py
from pydantic import BaseModel
from typing import Optional, List

```

```python
# Base Schemas
class SkillBase(BaseModel):
    Skill_Name: str

class CompanyBase(BaseModel):
    Company_Name: str

class LocationBase(BaseModel):
    Location: str

class IndustryBase(BaseModel):
    Industry: str

# Schemas for Linking
class SkillResponse(SkillBase):
    Skill_ID: int
    class Config:
        from_attributes = True

class CompanyResponse(CompanyBase):
    Company_ID: int
    class Config:
        from_attributes = True

class LocationResponse(LocationBase):
    Location_ID: int
    class Config:
        from_attributes = True

class IndustryResponse(IndustryBase):
    Industry_ID: int
    class Config:
        from_attributes = True

# --- Job Schemas ---
class JobBase(BaseModel):
    Job_Title: Optional[str] = None
    Job_Description: Optional[str] = None
    Salary_USD: Optional[str] = None
    Employment_Type: Optional[str] = None
    Experience_Level: Optional[str] = None
    Work_Setting: Optional[str] = None
    Company_ID: Optional[int] = None
    Location_ID: Optional[int] = None
    Industry_ID: Optional[int] = None

class JobCreate(JobBase):
    pass

```

```python
class JobUpdate(JobBase):
    pass

class JobResponse(JobBase):
    Job_ID: int
    company: Optional[CompanyResponse] = None
    location: Optional[LocationResponse] = None
    industry: Optional[IndustryResponse] = None
    skills: List[SkillResponse] = []

    class Config:
        from_attributes = True

class AggResponse(BaseModel):
    label: str
    count: int

class MessageResponse(BaseModel):
    message: str
```

A.4 File: app.py (API Routes)

```python
# app.py
from typing import List
from fastapi import FastAPI, Depends, HTTPException, Query
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session, joinedload
from sqlalchemy import func
import models, schemas
from db import SessionLocal, engine

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

app = FastAPI(title="Job Trends API")

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Dependency
def get_db():
    db = SessionLocal()
    try:
```

```
        yield db
    finally:
        db.close()

# --- CRUD Endpoints ---
# (Note: CRUD Logic is identical to Section 3 of this report, repeated here for completeness)

@app.post("/jobs", response_model=schemas.JobResponse, status_code=201)
def create_job(job: schemas.JobCreate, db: Session = Depends(get_db)):
    db_job = models.Job(**job.model_dump())
    db.add(db_job)
    db.commit()
    db.refresh(db_job)
    return db_job

@app.get("/jobs", response_model=List[schemas.JobResponse])
def get_jobs(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    jobs = (
        db.query(models.Job)
        .options(
            joinedload(models.Job.company),
            joinedload(models.Job.location),
            joinedload(models.Job.industry),
            joinedload(models.Job.skills)
        )
        .offset(skip)
        .limit(limit)
        .all()
    )
    return jobs

# ... (Additional CRUD operations get_job, update_job, delete_job omitted for brevity as they match Section 3)

# --- Analytical Endpoints ---
@app.get("/analytics/locations", response_model=List[schemas.AggResponse])
def agg_by_location(db: Session = Depends(get_db)):
    query = (
        db.query(
            models.Location.Location,
            func.count(models.Job.Job_ID).label("count")
        )
        .join(models.Job, models.Job.Location_ID == models.Location.Location_ID)
        .group_by(models.Location.Location)
        .order_by(func.count(models.Job.Job_ID).desc())
        .limit(20)
```

```
        .all()
    )
    return [{"label": r[0] or "Unknown", "count": r[1]} for r in query]

@app.get("/analytics/industries", response_model=List[schemas.AggResponse])
def agg_by_industry(db: Session = Depends(get_db)):
    query = (
        db.query(
            models.Industry.Industry,
            func.count(models.Job.Job_ID).label("count")
        )
        .join(models.Job, models.Job.Industry_ID == models.Industry.
    Industry_ID)
        .group_by(models.Industry.Industry)
        .order_by(func.count(models.Job.Job_ID).desc())
        .limit(20)
        .all()
    )
    return [{"label": r[0] or "Unknown", "count": r[1]} for r in query]

@app.get("/analytics/skills", response_model=List[schemas.AggResponse])
def agg_by_skill(db: Session = Depends(get_db)):
    query = (
        db.query(
            models.Skill.Skill_Name,
            func.count(models.Job.Job_ID).label("count")
        )
        .join(models.job_skills_table, models.Skill.Skill_ID == models.
    job_skills_table.c.Skill_ID)
        .join(models.Job, models.Job.Job_ID == models.job_skills_table.c.
    Job_ID)
        .group_by(models.Skill.Skill_Name)
        .order_by(func.count(models.Job.Job_ID).desc())
        .limit(20)
        .all()
    )
    return [{"label": r[0] or "Unknown", "count": r[1]} for r in query]
```