Job Trends Web Application: Database Design & Implementation

Authors: Ankita Suresh Kumar, Krishi Thiruppathi

1. Conceptual Schema & Diagram

This project utilizes a relational database (MySQL) to store and serve job trend data. The architecture consists of a FastAPI backend application that communicates with the database via the SQLAlchemy ORM.

The data pipeline follows a modern Extract, Transform, Load (ETL) process. First, the raw jobs_in_data.csv file is loaded into a temporary staging table in MySQL (jobs_in_data) using the Workbench importer. Second, a comprehensive Python normalization script (run in a Jupyter notebook) is executed. This script reads the raw data, cleans and de-duplicates entities (companies, locations, etc.), and populates a new, 6-table normalized schema. This design follows Third Normal Form (3NF), which eliminates data redundancy and ensures referential integrity. This normalized schema is the final, permanent database used by the web application.

- *Entity Descriptions*

The normalized schema consists of six primary tables:

- **Companies:** A lookup table storing unique company names.
- **Locations:** A lookup table storing unique job locations.
- **Industries:** A lookup table storing unique job categories (industries).
- **Skills:** A lookup table storing unique skills extracted from job postings.
- **Jobs:** The central "fact table" containing all core job information (title, salary, etc.) and linking to the lookup tables via foreign keys.
- **job_skills:** An associative (junction) table that resolves the many-to-many relationship between Jobs and Skills.

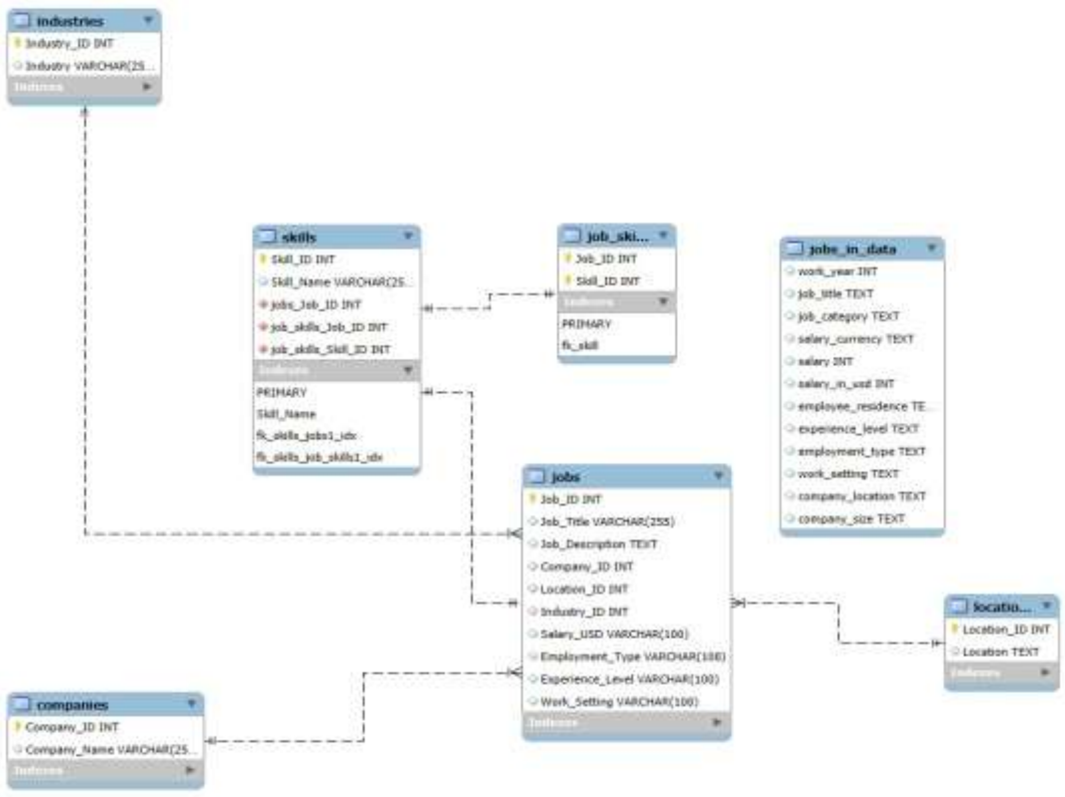
- *Relationships & Cardinality*

The relationships are designed to enforce normalization and data integrity:

- A Company can have one or more Jobs. (One-to-Many)
- A Location can have one or more Jobs. (One-to-Many)
- An Industry can have one or more Jobs. (One-to-Many)
- A Job can have one or more Skills (via the job_skills table). (Many-to-Many)
- A Skill can be associated with one or more Jobs (via the job_skills table). (Many-to-Many)

- *Entity-Relationship (ER) Diagram*

The following ER diagram was generated directly from our MySQL database using the MySQL Workbench "Reverse Engineer" tool. It illustrates the final, normalized 6-table schema that powers our application.



2. Data Constraints

To ensure data integrity, several constraints are defined at the SQL level (in our normalization script) and enforced by the MySQL database.

- *Key Constraints*

- **PRIMARY KEY**: Each table features a unique, non-null primary key (e.g., Company_ID, Job_ID). This guarantees that every record is uniquely identifiable.
- **COMPOSITE KEY**: The job_skills junction table uses a composite primary key (Job_ID, Skill_ID). This is a critical constraint that prevents a single skill from being listed more than once for the same job.
- **FOREIGN KEY**: Referential integrity is enforced using foreign keys. For example, Jobs.Company_ID is a foreign key that references Companies.Company_ID. This makes it impossible to add a job with a Company_ID that does not exist. These were added via ALTER TABLE commands after the data was loaded.
- **CASCADING DELETES**: The foreign keys on the job_skills table are defined with ON DELETE CASCADE. This ensures that if a job (or skill) is deleted, all its corresponding links in the job_skills table are automatically removed.

- *Attribute Constraints*

- **UNIQUE**: The Company_Name, Location, Industry, and Skill_Name columns in their respective lookup tables are all defined with a UNIQUE constraint. This is the core of our normalization strategy, preventing duplicate entries and ensuring a single source of truth for all reports.

- *Code-Level Constraints*

- **Pydantic Validation**: Our FastAPI application uses Pydantic models (defined in schemas.py) to validate all incoming API requests. This ensures that data types are correct *before* any database transaction is attempted.

3. Database Creation & Queries

Our database is created and queried using a combination of a Python/Pandas normalisation script (run in a Jupyter notebook) and a separate SQL script to enforce relationships.

- *Database Creation & Initial Data*

The database is built in a two-step, reproducible process:

1. The raw jobs_in_data.csv is imported into a staging table named jobs_in_data (using the MySQL Workbench Table Data Import Wizard).
2. Our custom Python normalization script (run from a Jupyter notebook) is executed.

This script first defines and executes the DDL (Data Definition Language) to create the 6 normalized tables without foreign key constraints. This is done to allow for fast, bulk-loading of data.

Code Snippet: DDL from the Python normalization script

# Authorship: Krishi T

```
create_sql = """
CREATE TABLE IF NOT EXISTS companies (
  Company_ID INT PRIMARY KEY,
  Company_Name VARCHAR(255) UNIQUE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE IF NOT EXISTS locations (
  Location_ID INT PRIMARY KEY,
  Location VARCHAR(255) UNIQUE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
-- (and so on for all 6 tables) --
"""
with engine.begin() as conn:
    for stmt in create_sql.strip().split(';'):
        if stmt.strip():
            conn.execute(text(stmt))
print("Ensured normalized tables exist.")
```

Code Snippet: DML logic from the Python normalization script:

# Authorship: Krishi T

```
reader = pd.read_sql_query(f"SELECT * FROM `{RAW_TABLE}`", 
engine, chunksize=CHUNKSIZE)

for chunk_i, raw in enumerate(reader):
    for idx, row in raw.iterrows():
        # Get or create lookup IDs
        cname = row[company_col] if (company_col and 
pd.notna(row[company_col])) else "Unknown"
        if cname not in companies_map:
            companies_map[cname] = next_company_id
            next_company_id += 1
        cid = companies_map.get(cname)

        # ... (repeat for location, industry, skills) ...

        jobs_batch.append({
            "Job_ID": int(jid),
            "Job_Title": jt,
            "Company_ID": int(cid) if cid is not None else 
None,
            "Location_ID": int(lid) if lid is not None else 
None,
            "Industry_ID": int(iid) if iid is not None else 
None,
            # ... other fields ...
        })

    # Flush batches to database
    flush_jobs(engine, jobs_batch, job_skills_batch)
```

- **Enforcing Relationships (Foreign Keys)**

Code Snippet: SQL script for Foreign Key constraints

# Authorship: Krishi T

```
ALTER TABLE jobs
ADD CONSTRAINT fk_company
  FOREIGN KEY (Company_ID) REFERENCES companies(Company_ID),
ADD CONSTRAINT fk_location
  FOREIGN KEY (Location_ID) REFERENCES locations(Location_ID),
ADD CONSTRAINT fk_industry
  FOREIGN KEY (Industry_ID) REFERENCES 
industries(Industry_ID);

ALTER TABLE job_skills
ADD CONSTRAINT fk_job
  FOREIGN KEY (Job_ID) REFERENCES jobs(Job_ID) ON DELETE 
CASCADE,
ADD CONSTRAINT fk_skill
```

```
  FOREIGN KEY (Skill_ID) REFERENCES skills(Skill_ID) ON DELETE
CASCADE;
```

This links the tables together and enforces referential integrity. This step is done last to allow rapid data insertion without needing to check constraints on every row.

- **<u>CRUD Operations</u>**

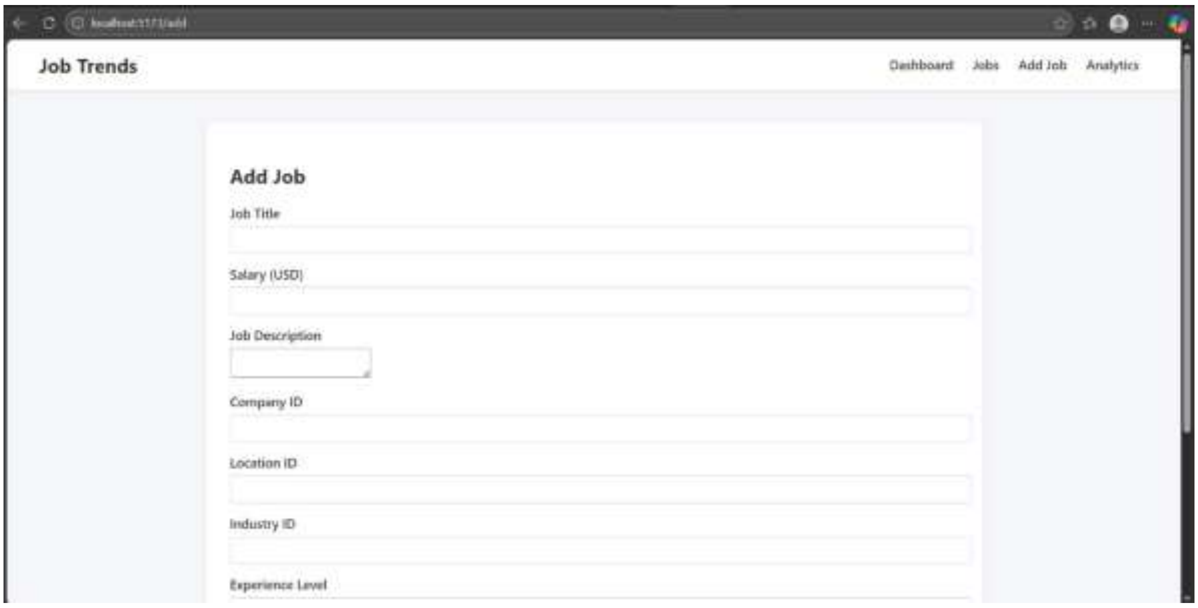
The app.py file (which runs on the server) provides all CRUD operations via a RESTful API, served by FastAPI.

Code Snippet: CREATE: POST /jobs endpoint

# Authorship: Ankita S

```
@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
```

Frontend Screenshot:



Code Snippet: READ (One): GET /jobs/{job_id} endpoint

# Authorship: Ankita S

```
@app.get("/jobs/{job_id}", response_model=schemas.JobResponse)
def get_job(job_id: int, db: Session = Depends(get_db)):
    # joinedload() eagerly loads all related tables
    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
```

Code Snippet: UPDATE: PUT /jobs/{job_id} endpoint

# Authorship: Ankita S

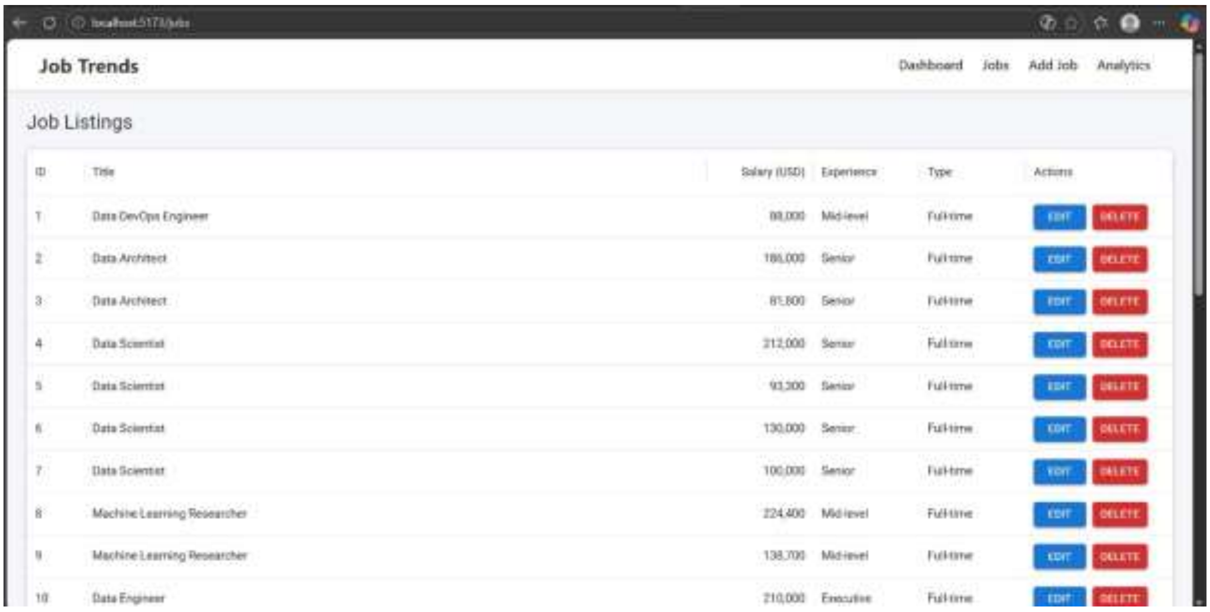
```
@app.put("/jobs/{job_id}", response_model=schemas.JobResponse)
def update_job(job_id: int, job_update: schemas.JobUpdate, db:
Session = Depends(get_db)):
    db_job = db.query(models.Job).filter(models.Job.Job_ID ==
job_id).first()
    if not db_job:
        raise HTTPException(status_code=404, detail="Job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

Code Snippet: DELETE: DELETE /jobs/{job_id} endpoint

# Authorship: Ankita S

```
@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"}
```

Frontend Screenshot:



localhost:5173/jobs

Job Listings

ID	Title	Salary (USD)	Experience	Type	Actions
1	Data DevOps Engineer	88,000	Mid-level	Full-time	EDIT DELETE
2	Data Architect	186,000	Senior	Full-time	EDIT DELETE
3	Data Architect	81,800	Senior	Full-time	EDIT DELETE
4	Data Scientist	212,000	Senior	Full-time	EDIT DELETE
5	Data Scientist	93,300	Senior	Full-time	EDIT DELETE
6	Data Scientist	130,000	Senior	Full-time	EDIT DELETE
7	Data Scientist	100,000	Senior	Full-time	EDIT DELETE
8	Machine Learning Researcher	224,400	Mid-level	Full-time	EDIT DELETE
9	Machine Learning Researcher	138,700	Mid-level	Full-time	EDIT DELETE
10	Data Engineer	210,000	Executive	Full-time	EDIT DELETE

- *Analytical Query*

Our normalized schema allows for powerful and efficient analytical queries, which fulfill the "analytical visualization" requirement. The API provides three such endpoints. The /analytics/skills endpoint is the most complex, requiring a 3-table join.

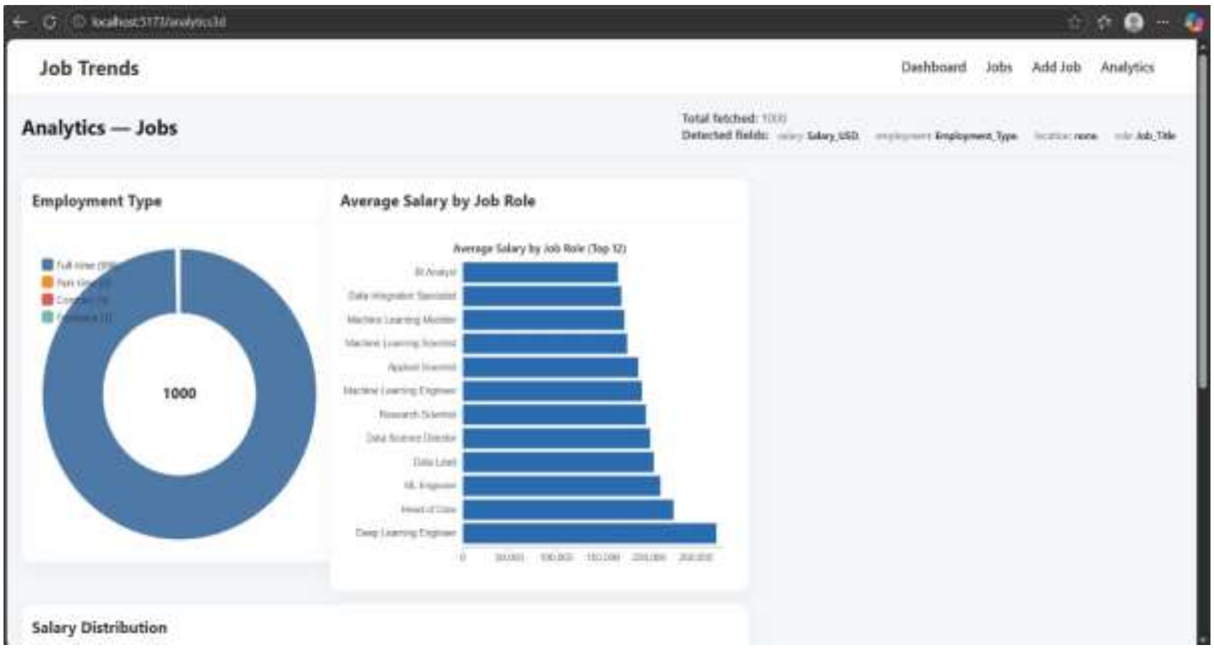
Code Snippet: Analytical Query: GET /analytics/skills:

Authorship: Ankita S, Krishi T

```
@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.
    """
    # Citation: SQLAlchemy aggregate query pattern
    # Source: docs.sqlalchemy.org
    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) # Get top 20
        .all()
    )
    return [{"label": r[0], "count": r[1]} for r in query]
```

Frontend Screenshot for Analytic Visualizations:



Job Trends

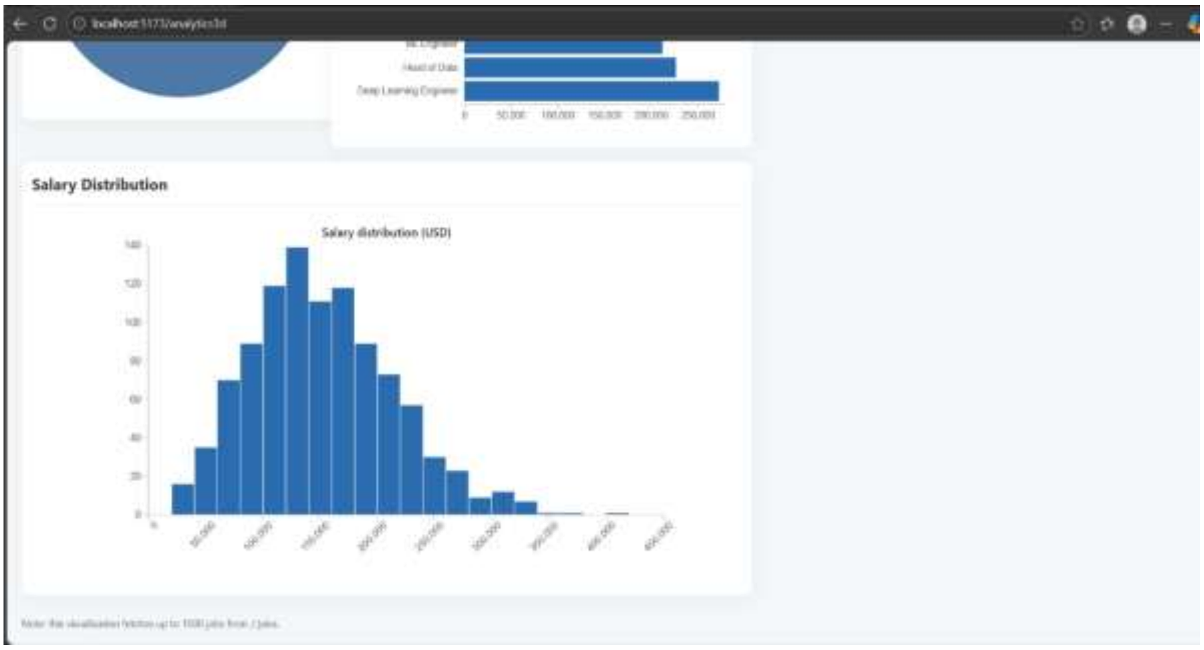
Analytics — Jobs

Total fetched: 1000
Detected fields: salary: Salary_USD employment: Employment_Type location: none role: Job_Title

Employment Type

Average Salary by Job Role

Salary Distribution



Salary Distribution

Note: the visualization fetches up to 1000 jobs from /jobs.

4. Overall Contribution Summary

The project was developed collaboratively, with a clear separation between backend (database) and frontend (UI) tasks.

Name	Tasks & Contribution Details	Est. Hours
Krishi	**Backend & Database Architecture** • Designed the 6-table normalized database schema. • Wrote the complete Python normalization script (`normalize.py`) to handle all DDL (table creation) and DML (data population). • Wrote the `db.py`, `models.py`, and `schemas.py` files to match the normalized schema. • Implemented all analytical endpoints (`GET /analytics/...`) in `app.py`. • Generated the ER Diagram from MySQL Workbench.	4-6 hours
Ankita	**Frontend & API Development** • Developed the entire React (Vite) frontend application. • Implemented all primary CRUD endpoints (`POST`, `GET`, `PUT`, `DELETE`) in `app.py`. • Connected the React UI to the FastAPI backend, handling all API calls (`axios`). • Built the frontend components to display job data and the analytical visualization charts.	4-6 hours

5. Acknowledgements

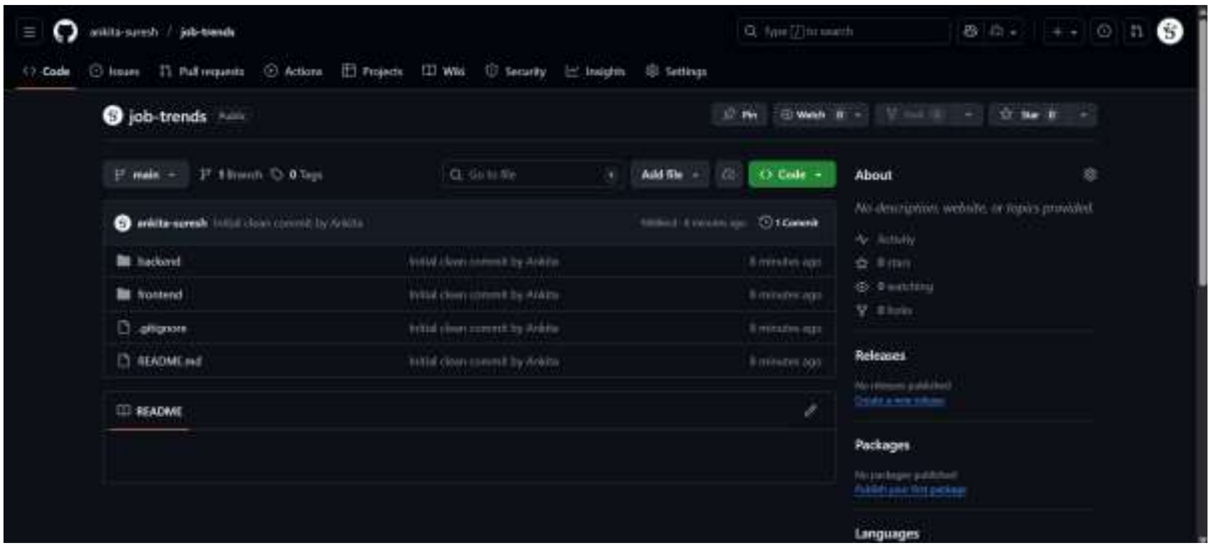
We acknowledge the use of the following tools in the development of this project:

- Portions of the Python code (including the normalize.py script and the FastAPI analytical queries) were generated, refactored, and debugged with the assistance of **Google's Gemini** (accessed November 2025). The AI was used as a coding assistant to help debug the Pandas read_sql_query errors and to generate the SQLAlchemy query syntax.
- We acknowledge the use of **MySQL Workbench** for its "Reverse Engineer" feature, which was used to generate the ER diagram included in Section 1 of this report.

Appendix:

The complete, reproducible Python code for the backend server is included below. The full project, including the Jupyter normalization script, is available at our team's GitHub repository:

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



ankita-suresh Initial clean commit by Ankita · 1 Commit

backend — Initial clean commit by Ankita — 8 minutes ago
frontend — Initial clean commit by Ankita — 8 minutes ago
.gitignore — Initial clean commit by Ankita — 8 minutes ago
README.md — Initial clean commit by Ankita — 8 minutes ago

README

/backend:

- **db.py**

```
# 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}:{D
B_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()
```

- **models.py**

```
# 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")

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")
```

- **schemas.py:**

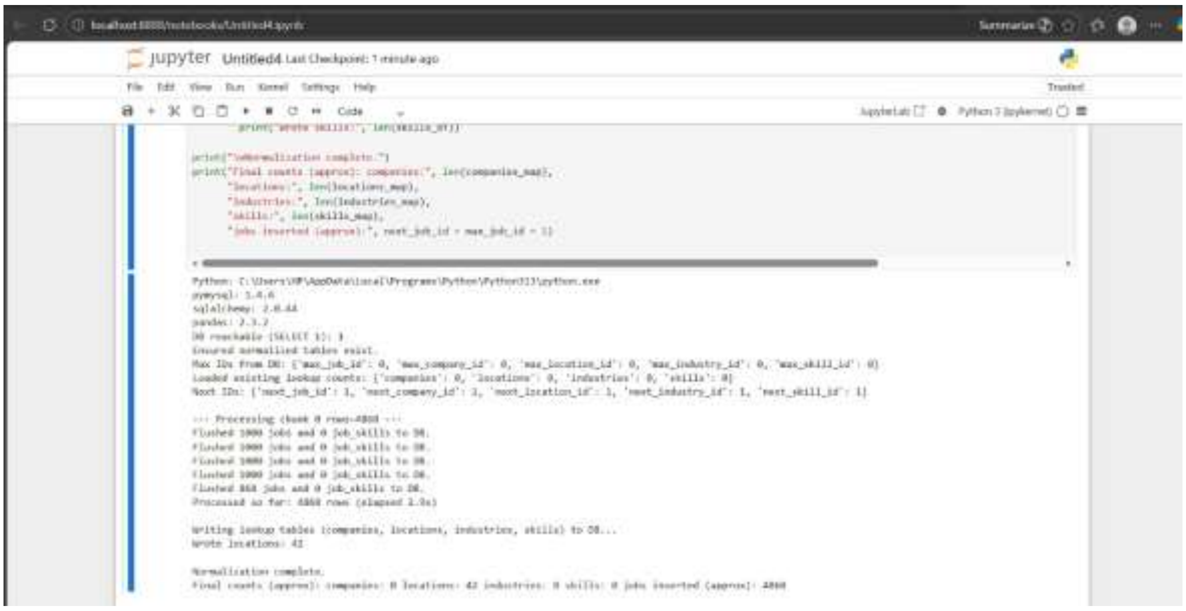
```
# schemas.py
from pydantic import BaseModel
from typing import Optional, List

# --- 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 (e.g., in JobResponse) ---
class SkillResponse(SkillBase):
    Skill_ID: int
    class Config:
        from_attributes = True # Replaces orm_mode

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

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

# --- Analytics Schemas ---
class AggResponse(BaseModel):
    label: str
    count: int

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

- **app.py**

```
# 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

# This creates the tables defined in models.py
models.Base.metadata.create_all(bind=engine)

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

# Allow your React frontend to connect
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# DB Session Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# --- CRUD Endpoints ---

@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 = Query(100, ge=1, le=1000),
    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

@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

@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

@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"}

# --- 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]
```

- **Jupyter Notebook Normalization Script (Output)**



```
        print("Wrote skills:", len(skills_df))

print("\nNormalization complete.")
print("Final counts (approx): companies:", len(companies_map),
      "locations:", len(locations_map),
      "industries:", len(industries_map),
      "skills:", len(skills_map),
      "jobs inserted (approx):", next_job_id - max_job_id - 1)
```

```
Python: C:\Users\HP\AppData\Local\Programs\Python\Python313\python.exe
pymysql: 1.4.6
sqlalchemy: 2.0.44
pandas: 2.3.2
DB reachable (SELECT 1): 1
Ensured normalized tables exist.
Max IDs from DB: {'max_job_id': 0, 'max_company_id': 0, 'max_location_id': 0, 'max_industry_id': 0, 'max_skill_id': 0}
Loaded existing lookup counts: {'companies': 0, 'locations': 0, 'industries': 0, 'skills': 0}
Next IDs: {'next_job_id': 1, 'next_company_id': 1, 'next_location_id': 1, 'next_industry_id': 1, 'next_skill_id': 1}

--- Processing chunk 0 rows=4868 ---
Flushed 1000 jobs and 0 job_skills to DB.
Flushed 1000 jobs and 0 job_skills to DB.
Flushed 1000 jobs and 0 job_skills to DB.
Flushed 1000 jobs and 0 job_skills to DB.
Flushed 868 jobs and 0 job_skills to DB.
Processed so far: 4868 rows (elapsed 2.0s)

Writing lookup tables (companies, locations, industries, skills) to DB...
Wrote locations: 42

Normalization complete.
Final counts (approx): companies: 0 locations: 42 industries: 0 skills: 0 jobs inserted (approx): 4868
```