SkillMiner: An AI-Powered Career & Study Copilot

He Jiang HJ193@DUKE.EDU
Sung-Tse Wu (Jay) SW693@DUKE.EDU
Yiyun Yao YY508@DUKE.EDU
Isaac Vergara ISAAC.VERGARA@DUKE.EDU
Qingyu Yang QY59@DUKE.EDU

Final Project Report

1. Introduction

SkillMiner is an AI-powered career and study copilot designed to help job seekers navigate the competitive technology job market. The system analyzes resumes, extracts skills, identifies gaps between candidate qualifications and job requirements, retrieves relevant learning resources, and generates personalized study plans using a combination of Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), and a modern data engineering pipeline.

The project addresses a critical challenge in the job market: the disconnect between job seeker skills and employer requirements. By leveraging real-world job posting data from LinkedIn, SkillMiner provides data-driven insights into in-demand skills across four key technology roles: Data Analyst (DA), Data Scientist (DS), Data Engineer (DE), and Software Engineer (SWE).

The system architecture integrates multiple data sources, employs sophisticated data transformation pipelines, and delivers intelligent recommendations through a web-based interface. The backend utilizes FastAPI for RESTful API services, ChromaDB for vector storage and semantic search, and OpenAI's GPT models for natural language understanding and generation. The frontend is built with React and TypeScript, providing an intuitive user experience for resume upload, skill analysis, and study plan generation.

The data engineering component processes millions of job postings through a multi-layer pipeline (raw $\rightarrow$ bronze $\rightarrow$ silver $\rightarrow$ gold) using Polars for efficient data transformation. The processed data is stored in multiple formats including Parquet files, SQLite databases, and cloud storage (AWS S3 and RDS), enabling both analytical queries and real-time application access.

Key features of SkillMiner include:

- **Resume Analysis**: Automated PDF parsing and skill extraction from resumes
- **Skill Gap Identification**: Comparison of candidate skills against job requirements
- **RAG-Powered Chat**: Context-aware career advice using semantic search over job market data
- **Personalized Study Plans**: AI-generated learning roadmaps tailored to individual skill gaps
- **Data-Driven Insights**: Statistical analysis of job market trends and skill demand patterns

The project demonstrates modern data engineering principles including modularity, scalability, observability, and security. It incorporates containerization with Docker, continuous integration through GitHub Actions, comprehensive testing strategies, and cloud deployment on Vercel (frontend) and Railway (backend). The complete source code and documentation are available in the project repository Jiang et al. (2025).

2. Architecture

SkillMiner follows a microservices architecture with clear separation between data ingestion, processing, storage, and application layers. The system is organized into four main components: the data pipeline, backend API, frontend application, and supporting infrastructure.

2.1 System Overview

The architecture consists of three primary layers:

1. **Data Layer**: Handles data ingestion from Kaggle, transformation using Polars, and storage in multiple formats (Parquet, SQLite, AWS S3/RDS, Supabase PostgreSQL)
2. **Application Layer**: FastAPI backend providing RESTful APIs for resume analysis, skill matching, RAG-powered chat, and study plan generation
3. **Presentation Layer**: React-based frontend deployed on Vercel, providing user interface for all interactions

2.2 Data Pipeline Architecture

The data engineering pipeline implements a medallion architecture pattern with four distinct layers:

Raw Layer Contains original Kaggle datasets in Parquet format, including:

- `linkedin_job_postings.parquet`: Job posting metadata
- `job_summary.parquet`: Summarized job information
- `job_skills.parquet`: Skill-to-job mappings

Bronze Layer Cleaned and normalized data with consistent schema. The `CleanJobTransformer` standardizes column names, handles multiple schema variations, normalizes data types, and performs basic data quality checks.

Silver Layer Role-filtered and enriched data. This layer:

- Filters jobs matching target roles (DA, DS, DE, SWE) using regex pattern matching
- Joins text fields (title, description, skills) for NLP processing
- Derives missing work type (remote/hybrid/onsite) from text analysis
- Infers seniority levels (intern/junior/mid/senior/lead) from job titles

Gold Layer Aggregated analytical outputs:

- Top skills across all roles (ranked by frequency)
- Role-specific skill aggregations
- Normalized skill distributions for fair cross-role comparison

The pipeline is orchestrated through a Python-based `JobsPipeline` class that chains transformers and aggregators in sequence. While the README mentions Airflow for production orchestration, the current implementation uses a programmatic approach with Typer CLI for local execution.

2.3 Backend Architecture

The backend follows a modular design with clear separation of concerns:

- **API Layer** (`src/api/`): FastAPI routers handling HTTP requests for health checks, file uploads, chat interactions, skill analysis, and study plan generation
- **Service Layer** (`src/services/`): Business logic including skill matching algorithms that compare resumes against job descriptions
- **RAG System** (`src/rag/`):
  - `retriever.py`: ChromaDB-based semantic search over job role and skill embeddings
  - `parser.py`: PDF resume parsing and text extraction
- **Database Clients** (`src/db/`): Interfaces to Supabase (PostgreSQL) for user data and AWS RDS for job market data
- **LLM Integration** (`src/llm/`): OpenAI client wrapper for GPT model interactions

The backend uses ChromaDB as a persistent vector store, seeded from the gold layer Parquet files. When a user queries the chat system, the retriever performs semantic search over role-skill embeddings, retrieves relevant context, and augments LLM prompts with this information.

2.4 Storage Architecture

SkillMiner employs a multi-storage strategy optimized for different use cases:

- **Parquet Files**: Efficient columnar storage for analytical workloads, organized in bronze/silver/gold directories
- **SQLite**: Local relational database for querying processed job data, exported from Parquet via `SQLiteMaterializer`
- **Supabase PostgreSQL**: Cloud-hosted database for:
  - User authentication and profiles
  - Chat message history
  - Generated study plans
  - Long-term memory for personalized recommendations
- **AWS Services**:
  - S3: Object storage for raw data and intermediate artifacts
  - RDS: Relational database for production job market data queries
- **ChromaDB**: Vector database for semantic search, stored locally in `backend/chroma/` directory

2.5 Deployment Architecture

The system is deployed across multiple cloud platforms:

- **Frontend**: Vercel (serverless React application)
- **Backend API**: Railway (containerized FastAPI service)
- **Databases**: Supabase (managed PostgreSQL) and AWS RDS
- **Data Processing**: AWS EC2 for ETL pipeline execution (mentioned in README)

All components communicate via REST APIs, with CORS middleware configured to allow cross-origin requests between frontend and backend domains.

3. Key Components

3.1 Data Ingestion

SkillMiner ingests real-world job posting data from Kaggle, specifically the LinkedIn Job Postings dataset. The ingestion process is handled by `scripts/download_kaggle.py`, which:

- Authenticates with Kaggle API using credentials
- Downloads dataset files in their original format
- Converts data to Parquet format for efficient storage and processing
- Validates data integrity and schema consistency

The dataset contains millions of job postings with fields including job titles, company names, locations, work types, seniority levels, job descriptions, and associated skills. This multi-source data (job postings, summaries, and skill mappings) provides a comprehensive view of the technology job market.

3.2 Data Transformation Pipeline

The transformation pipeline is built using Polars, a high-performance DataFrame library written in Rust. The pipeline consists of several key components:

3.2.1 Transformers

- **CleanJobTransformer**: Normalizes raw data by:
  - Mapping variant column names to standard schema (e.g., `job_title`, `title`, `position` → `title`)
  - Converting data types (strings to datetime, normalizing text)
  - Parsing and cleaning skill lists (handling JSON arrays, comma-separated strings)
  - Filtering out invalid records (missing title or company)
- **RoleFilterTransformer**: Filters jobs matching target roles using regex patterns. For example, the pattern `\b(data scientist|scientist)\b` matches job titles containing "data scientist" or "scientist" as whole words.

- **TextJoinTransformer**: Concatenates title, description, and skills into a single text field for NLP processing, enabling semantic search and text analysis.
- **DeriveWorkTypeTransformer**: Infers work type (remote/hybrid/onsite) from job descriptions using keyword matching when the field is missing.
- **DeriveSeniorityTransformer**: Extracts seniority levels from job titles using pattern matching (e.g., "senior", "jr.", "lead", "principal").

3.2.2 Aggregators

- **TopSkillsAggregator**: Counts skill frequencies across all jobs, ranks them, and returns the top N skills (default: 40).
- **RoleSkillsAggregator**: Groups skills by job role, creating unique sorted lists of skills required for each role (DA, DS, DE, SWE).
- **RoleSkillCountsAggregator**: Produces long-format data with skill counts per role, enabling statistical analysis and visualization.

The pipeline uses Polars' lazy evaluation framework, which allows for query optimization and efficient memory usage. Transformations are chained together, and execution is deferred until data is written to disk.

3.3 Data Storage and Querying

3.3.1 Storage Systems

The system uses multiple storage backends optimized for different access patterns:

- **Parquet Files**: Primary storage format for processed data, providing:
  - Columnar compression for efficient storage
  - Schema evolution support
  - Fast analytical queries with Polars
- **SQLite Database**: Exported from Parquet for SQL-based querying. The `SQLiteMaterializer` handles:
  - Type conversion (Polars types → SQLite types)
  - List/array serialization to JSON strings
  - Datetime formatting to ISO-8601 strings
  - Automatic index creation on common query columns
- **Supabase PostgreSQL**: Stores:
  - User accounts and authentication data
  - Chat conversation history (`chat_messages` table)
  - Generated study plans with metadata
  - Long-term memory for personalized recommendations (`ltm_memory` table)
- **AWS RDS**: Production database for job market data, enabling scalable queries for the backend API.

3.3.2 Querying Capabilities

SQL queries are used extensively for data analysis. The `notebooks/insights.sql` file contains complex analytical queries including:

- Role distribution calculations with percentage breakdowns
- Top skills aggregation with normalization across roles
- Geographic distribution analysis (US states and countries)
- Skill frequency analysis by role

These queries support the exploratory data analysis notebooks and provide insights visualized in the project's documentation.

3.4 Data Analysis and Insights

The analysis component performs statistical analysis using both Polars (for pipeline operations) and Pandas (for exploratory analysis in Jupyter notebooks). Key analyses include:

- **Exploratory Data Analysis** (`01_eda.ipynb`):
  - Missing value analysis and data quality checks
  - Job distribution by work type (remote/hybrid/onsite)
  - Top hiring companies and locations
  - Monthly posting trends
- **Skill Insights** (`02_insights.ipynb`):
  - Overall top 20 skills across all roles
  - Normalized skill distributions (accounting for role size differences)
  - Role-specific top 10 skills for DA, DS, DE, and SWE
  - Geographic distribution of opportunities by role

Key findings from the analysis:

- Python and SQL are the two most critical skills across all technology roles
- Software engineering positions significantly outnumber DA, DS, and DE roles combined
- Each role has distinct skill requirements, making personalized learning paths essential
- Geographic concentration varies by role, with certain regions showing higher demand

3.5 Backend API Services

3.5.1 REST API Endpoints

The FastAPI backend provides the following endpoints:

- `GET /health`: Health check endpoint for monitoring
- `POST /upload`: Resume PDF upload and parsing
- `POST /chat`: RAG-powered chat interface for career advice
- `POST /analysis`: Skill gap analysis comparing resume to job description
- `POST /study-plan`: Generate personalized study plan
- `GET /study-plan/{plan_id}`: Retrieve saved study plan

3.5.2 RAG System

The Retrieval-Augmented Generation system enhances LLM responses with relevant context from the job market data:

1. **Vector Store Initialization**: ChromaDB is seeded from the gold layer Parquet file (`role_skills_by_title.parquet`), creating embeddings for each role-skill combination using OpenAI's `text-embedding-3-large` model.
2. **Query Processing**: When a user submits a chat message:
   - The system combines resume text (if available) with the user's message
   - Performs semantic search over the vector store
   - Retrieves top K most relevant role-skill contexts
   - Builds a context block with retrieved information
3. **Response Generation**: The context, resume, and user message are sent to OpenAI GPT models, which generate contextually relevant career advice.

3.5.3 Skill Matching Service

The `SkillMatcher` service provides intelligent skill gap analysis:

1. Extracts skills from both resume and job description using LLM
2. Validates extracted skills against the database
3. Categorizes skills as technical or soft skills
4. Identifies matched and missing skills
5. Calculates a match score (0-100) using LLM-based evaluation

The service uses RAG to retrieve relevant role context, improving the accuracy of skill matching by understanding industry standards.

3.6 Frontend Application

The frontend is built with React, TypeScript, and Vite, providing a modern single-page application experience. Key pages include:

- **Login Page**: Supabase authentication integration
- **Upload Page**: Resume PDF upload with drag-and-drop interface
- **Chatbot Page**: Interactive chat interface with persistent conversation history
- **Study Plan Page**: Display and management of generated study plans
- **Skill Report Page**: Visualization of skill analysis results
- **Dashboard**: Overview of user progress and recommendations

The frontend communicates with the backend via REST APIs and manages state using React hooks. Supabase client is used for authentication and real-time data synchronization.

3.7 Containerization and CI/CD

3.7.1 Docker

The project includes Docker configurations for reproducible environments:

- `database/Dockerfile`: Containerizes the data pipeline with all dependencies
- `database/docker-compose.yml`: Orchestrates build and test execution
- Backend and frontend can be containerized for deployment (Railway uses containerization)

3.7.2 Continuous Integration

GitHub Actions workflows provide automated testing and quality checks:

- `database-ci.yml`:
  - Runs on pushes/PRs affecting `database/` directory
  - Installs Python 3.11 and dependencies
  - Executes flake8 linting
  - Runs unit and integration tests with coverage
- `backend-ci.yml`:
  - Similar structure for backend code
  - Includes optional environment variables for API keys (for integration tests)
  - Tests API endpoints and RAG functionality

Both workflows ensure code quality and prevent regressions before merging to main branch.

4. Implementation

4.1 Pipeline Execution Flow

The data pipeline execution follows a sequential workflow orchestrated by the `JobsPipeline` class:

1. **Data Loading**: The pipeline scans the `data/raw/` directory for Parquet files, excluding the skills link table which is handled separately.
2. **Skills Attachment**: Before cleaning, skills are attached to job records. The system handles two schema variations:
   - Link table format: `(job_id, skill_name)` rows are aggregated per job
   - Pre-aggregated format: `(job_link, job_skills)` strings are joined directly
3. **Bronze Layer Processing**: The `CleanJobTransformer` normalizes the schema, handles missing values, parses dates, and standardizes skill lists. Output is written to `data/bronze/jobs.parquet`.
4. **Silver Layer Processing**: Multiple transformations are applied:
   - Role filtering to keep only DA/DS/DE/SWE jobs
   - Text joining for NLP-ready content
   - Work type and seniority derivation from text patterns

   Output is written to `data/silver/jobs_text.parquet`.
5. **Gold Layer Processing**: Two aggregations are performed:
   - Top skills across all roles → `data/gold/top_skills.parquet`
   - Role-specific skill aggregations → `data/gold/role_skills_by_title.parquet`

The pipeline uses Polars' lazy evaluation, meaning transformations are not executed until `collect()` or `save_lazy()` is called. This allows Polars to optimize the query plan and minimize memory usage.

4.2 Data Quality and Schema Management

The pipeline implements robust schema handling to accommodate variations in source data:

- **Column Name Normalization**: The `CleanJobTransformer` uses a candidate-based approach, checking multiple possible column names (e.g., `job_title`, `title`, `position`) and mapping to a standard schema.
- **Type Coercion**: Data types are explicitly cast (e.g., strings to datetime, categorical to UTF-8) to ensure consistency.
- **Missing Value Handling**: The pipeline filters out records missing critical fields (title, company) and uses default values or derived values for optional fields (work type, seniority).
- **Skill List Parsing**: Handles multiple skill formats:
  - JSON arrays: `["Python", "SQL"]`

- Comma-separated strings: `"Python, SQL"`
- Pre-aggregated strings from link tables

All are normalized to Polars list types for consistent processing.

4.3 Vector Store Implementation

The RAG system's vector store is implemented using ChromaDB with the following architecture:

- **Collection Structure**: A single collection named `"roles_skills"` stores embeddings of role-skill combinations.
- **Embedding Model**: OpenAI's `text-embedding-3-large` generates 3072-dimensional vectors for each document.
- **Document Format**: Each document combines:
  - Role title (lowercase): e.g., "data scientist"
  - Skills list: comma-separated skills for that role

  Example: `"data scientist | python, sql, machine learning, statistics"`
- **Seeding Process**: The `_seed_roles_from_parquet()` function:
  1. Reads the gold layer Parquet file
  2. Constructs text fields from `title_lc` and `skills_for_role` columns
  3. Truncates text to 30,000 characters (conservative limit for embedding model)
  4. Converts metadata (lists/arrays) to ChromaDB-compatible strings
  5. Upserts documents in batches of 500 to avoid payload limits
- **Retrieval Process**: When querying:
  1. User message and resume text are combined into a query string
  2. ChromaDB performs cosine similarity search
  3. Top K results (default: 5) are retrieved with metadata
  4. Results are formatted into context blocks for LLM prompts

The vector store is persistent, stored in `backend/chroma/`, and automatically seeded on first use if empty.

4.4 API Request Processing

The backend processes requests through the following flow:

1. **Request Reception**: FastAPI receives HTTP request and validates input using Pydantic schemas.
2. **Authentication** (where applicable): Supabase client verifies user session tokens.
3. **Business Logic Execution**:

- For `/chat`: RAG retrieval → context building → LLM generation
- For `/analysis`: PDF parsing → skill extraction → database lookup → matching algorithm
- For `/study-plan`: Analysis retrieval → LLM plan generation → database storage

4. **Response Formatting**: Results are serialized to JSON using Pydantic models, ensuring type safety and API contract compliance.

5. **Error Handling**: Global exception handlers catch and log errors, returning user-friendly error messages while preserving detailed logs for debugging.

4.5 Testing Strategy

The project implements comprehensive testing at multiple levels:

4.5.1 Unit Tests

- **Database Module**: Tests for schema validation, data type conversions, and transformer logic
- **Backend Module**: Tests for API schemas, configuration loading, and utility functions
- **RAG Module**: Tests for retrieval logic, context building, and embedding handling

Unit tests use mocking to avoid external API calls and database dependencies, ensuring fast execution and reliability.

4.5.2 Integration Tests

- **Pipeline Smoke Tests**: End-to-end pipeline execution with test data, verifying all transformation stages
- **API Integration Tests**: Full request/response cycles for critical endpoints, using test fixtures

Integration tests use small test datasets generated by `scripts/make_test_data.py`, which creates stratified samples preserving data distribution characteristics.

4.5.3 Test Data Management

The project includes a test data generation script that:

- Reads full datasets
- Stratifies by role to maintain representation
- Generates tiny Parquet files (`data/test/tiny_*.parquet`)
- Preserves schema consistency for realistic testing

4.6 Deployment Process

4.6.1 Backend Deployment (Railway)

1. Code is pushed to GitHub main branch
2. Railway detects changes and triggers build
3. Docker container is built with backend dependencies
4. Environment variables are injected (API keys, database URLs)
5. Application starts with `uvicorn src.api.main:app --host 0.0.0.0 --port $PORT`
6. Health checks verify service availability

4.6.2 Frontend Deployment (Vercel)

1. Code push triggers Vercel build
2. Dependencies are installed via `npm install`
3. Production build is created with `npm run build`
4. Static assets are deployed to CDN
5. Environment variables configure API endpoints

4.6.3 Database Migrations

Supabase migrations are managed through SQL files in `backend/supabase/migrations/`:

- `create_ltm_memory_table.sql`: Creates long-term memory table for personalized recommendations
- `create_chat_messages_table.sql`: Creates chat history table

Migrations are applied manually or through Supabase CLI, ensuring schema consistency across environments.

4.7 Performance Optimizations

Several optimizations ensure efficient data processing:

- **Lazy Evaluation**: Polars defers computation until necessary, allowing query optimization
- **Columnar Storage**: Parquet format enables column pruning and efficient compression
- **Batch Processing**: Vector store upserts and database inserts use batching to reduce I/O overhead
- **Caching**: ChromaDB collection is reused across requests, avoiding re-initialization
- **Connection Pooling**: Database clients use connection pools for efficient resource management

4.8 Security Implementation

Security measures are implemented at multiple layers:

- **Authentication**: Supabase handles user authentication with JWT tokens
- **API Security**: CORS middleware restricts origins to known frontend domains
- **Environment Variables**: Sensitive credentials (API keys, database URLs) are stored as environment variables, never committed to version control
- **Input Validation**: Pydantic schemas validate all API inputs, preventing injection attacks
- **File Upload Limits**: Resume uploads are restricted to PDF format and reasonable size limits
- **Error Handling**: Detailed error messages are logged server-side but sanitized for client responses

4.9 Observability and Monitoring

The system includes logging and monitoring capabilities:

- **Structured Logging**: Python's logging module with configurable levels (INFO, WARNING, ERROR)
- **Health Checks**: `/health` endpoint enables uptime monitoring
- **Error Tracking**: Global exception handlers log stack traces for debugging
- **CI/CD Badges**: GitHub Actions provide build status indicators in README files

References

He Jiang, Sung-Tse (Jay) Wu, Yiyun Yao, Isaac Vergara, and Qingyu Yang. Skillminer: An ai-powered career & study copilot, 2025. URL `https://github.com/JayWu0512/SkillMiner`. GitHub repository containing the complete SkillMiner project implementation, including data pipeline, backend API, frontend application, and documentation.