Comprehensive Data Analysis

Job Market Data Processing, Cleaning, and Visualization

Saurabh Sharma

2025-09-27

Table of contents

## Overview

This analysis provides comprehensive data processing and visualization of the Lightcast job market dataset using PySpark 4.0.1 for big data processing and Plotly 6.3.0 with Kaleido 1.1.0 for interactive and static visualizations.

**📊 Data Source**: Uses real processed job market data from our PySpark pipeline (996 records) instead of synthetic sample data, ensuring authentic insights and analysis.

## Data Processing Pipeline

### Environment Setup and Data Loading

import os  
import sys  
from pathlib import Path  
import warnings  
warnings.filterwarnings('ignore')  
  
# Import our custom analysis classes  
sys.path.append('src')  
from data.spark\_analyzer import SparkJobAnalyzer, create\_spark\_analyzer  
from data.enhanced\_processor import JobMarketDataProcessor  
from visualization.simple\_plots import SalaryVisualizer  
  
# Visualization libraries  
import plotly.express as px  
import plotly.graph\_objects as go  
from plotly.subplots import make\_subplots  
import plotly.io as pio  
import pandas as pd  
import numpy as np  
  
# Configure Plotly for different output formats  
import kaleido  
  
# Set rendering mode based on output format  
def configure\_plotly\_renderer():  
 """Configure Plotly renderer based on output format"""  
 try:  
 # For HTML output (interactive)  
 pio.renderers.default = "plotly\_mimetype+notebook"  
 return "interactive"  
 except:  
 # For static output (PDF/DOCX)  
 pio.renderers.default = "png"  
 pio.kaleido.scope.plotlyjs = "/usr/local/lib/python3.12/site-packages/plotly/package\_data/plotly.min.js"  
 return "static"  
  
render\_mode = configure\_plotly\_renderer()  
print(f"Plotly renderer configured for: {render\_mode} mode")  
  
# Initialize our analyzer using the existing class  
print("🚀 Initializing SparkJobAnalyzer...")  
analyzer = create\_spark\_analyzer()  
spark = analyzer.spark  
  
print(f"✅ Spark Version: {spark.version}")  
print(f"✅ Analyzer ready with {analyzer.job\_data.count():,} records")

### Schema Definition and Data Loading

```{python}  
#| label: data-loading  
#| code-summary: "Load data using our SparkJobAnalyzer class"  
  
# Data is already loaded by SparkJobAnalyzer in the setup step  
df\_raw = analyzer.job\_data  
  
print(f"✅ Data loaded via SparkJobAnalyzer: {df\_raw.count():,} records")  
print("\n📋 Data Schema:")  
df\_raw.printSchema()  
  
print("\n📂 Sample Records:")  
df\_raw.show(5, truncate=False)  
  
# Get overall statistics using our analyzer  
stats = analyzer.get\_overall\_statistics()  
print("\n📈 Dataset Statistics:")  
for key, value in stats.items():  
 print(f" {key}: {value:,}")

def load\_job\_data(file\_path: str = None): ““” Load processed job market data using our existing infrastructure

Args:  
 file\_path: Optional path override (uses processed data by default)  
"""  
  
# Use our existing processed dataset  
processed\_path = "data/processed/job\_market\_processed.parquet"  
  
if Path(processed\_path).exists():  
 print(f"Loading processed data from: {processed\_path}")  
 df = spark.read.parquet(processed\_path)  
 print(f"✅ Real processed data loaded: {df.count():,} records")  
 return df  
  
elif file\_path and Path(file\_path).exists():  
 print(f"Loading raw data from: {file\_path}")  
 df = spark.read \  
 .option("header", "true") \  
 .option("inferSchema", "false") \  
 .schema(lightcast\_schema) \  
 .csv(file\_path)  
 print(f"Raw data loaded: {df.count():,} records")  
 return df  
  
else:  
 print("❌ No processed or raw data found!")  
 print("💡 Run the data processing pipeline first to create processed data")  
 return None

# Using real processed data - no need for synthetic data generation!

# Load the real processed data

df\_raw = load\_job\_data()

if df\_raw is not None: print(f”✅ Successfully loaded {df\_raw.count():,} records”) print(“📋 Data Schema:”) df\_raw.printSchema()

print("\n📊 Sample Records:")  
df\_raw.show(5, truncate=False)

else: print(“❌ Failed to load data - check data processing pipeline”)

## Data Quality Assessment  
  
::: {#data-quality-assessment .cell execution\_count=3}  
``` {.python .cell-code code-summary="Assess data quality using our class methods"}  
# Use JobMarketDataProcessor for comprehensive data quality assessment  
processor = JobMarketDataProcessor("ComprehensiveAnalysis")  
processor.df\_raw = df\_raw # Use the data from our analyzer  
  
print("=== DATA QUALITY ASSESSMENT (Class-Based) ===")  
  
# Get comprehensive data quality report from our processor  
try:  
 quality\_report = processor.assess\_data\_quality()  
   
 print(f"Total Records: {quality\_report.get('total\_records', 'N/A'):,}")  
 print(f"Total Columns: {quality\_report.get('total\_columns', 'N/A')}")  
   
 # Display missing value analysis  
 if 'missing\_analysis' in quality\_report:  
 print("\nMissing Value Analysis:")  
 print("-" \* 50)  
 for col\_info in quality\_report['missing\_analysis']:  
 col\_name = col\_info['column']  
 missing\_count = col\_info['missing\_count']  
 missing\_pct = col\_info['missing\_percentage']  
 print(f"{col\_name:20} | {missing\_count:>8,} ({missing\_pct:>5.1f}%)")  
   
 # Show recommendations  
 if 'recommendations' in quality\_report:  
 print(f"\nData Quality Recommendations:")  
 for rec in quality\_report['recommendations']:  
 print(f" - {rec}")  
   
except AttributeError:  
 # Fallback to basic analysis if method not implemented  
 print("Using basic data quality assessment...")  
   
 total\_rows = df\_raw.count()  
 total\_cols = len(df\_raw.columns)  
   
 print(f"Total Records: {total\_rows:,}")  
 print(f"Total Columns: {total\_cols}")  
   
 print("\nBasic Column Analysis:")  
 print("-" \* 30)  
 for col\_name in df\_raw.columns[:10]: # Show first 10 columns  
 non\_null = df\_raw.filter(df\_raw[col\_name].isNotNull()).count()  
 completeness = (non\_null / total\_rows) \* 100  
 print(f"{col\_name:20} | {completeness:>5.1f}% complete")

:::

## Data Cleaning and Preprocessing

# Use our JobMarketDataProcessor for comprehensive cleaning  
print("=== DATA CLEANING (Class-Based) ===")  
  
if df\_raw is not None:  
 # Check if data is already processed (has cleaned columns)  
 if "SALARY\_AVG\_IMPUTED" in df\_raw.columns:  
 print("✅ Data already processed - using existing cleaned data")  
 df\_processed = df\_raw  
 else:  
 print("🔧 Applying JobMarketDataProcessor cleaning pipeline...")  
   
 try:  
 # Use our comprehensive processor  
 df\_processed = processor.clean\_job\_data()  
   
 if df\_processed is None:  
 print("⚠️ Processor returned None, using analyzer data")  
 df\_processed = df\_raw  
 except AttributeError:  
 print("⚠️ Cleaning method not available, using processed data as-is")  
 df\_processed = df\_raw  
   
 # Show sample of processed data using our display format  
 print(f"\n📋 Processed Data Sample ({df\_processed.count():,} records):")  
   
 # Select appropriate columns based on what's available  
 available\_cols = df\_processed.columns  
 display\_cols = []  
   
 # Build display columns list based on available columns  
 if "title" in available\_cols or "TITLE" in available\_cols:  
 display\_cols.append("TITLE" if "TITLE" in available\_cols else "title")  
 if "company" in available\_cols or "COMPANY" in available\_cols:  
 display\_cols.append("COMPANY" if "COMPANY" in available\_cols else "company")  
 if "location" in available\_cols:  
 display\_cols.append("location")  
 if "salary\_avg\_imputed" in available\_cols:  
 display\_cols.append("salary\_avg\_imputed")  
 elif "SALARY\_AVG\_IMPUTED" in available\_cols:  
 display\_cols.append("SALARY\_AVG\_IMPUTED")  
   
 # Show sample with available columns  
 if display\_cols:  
 df\_processed.select(\*display\_cols[:5]).show(10, truncate=False)  
 else:  
 print("Showing first few columns:")  
 df\_processed.select(\*available\_cols[:5]).show(5, truncate=False)  
else:  
 print("❌ Cannot proceed without data - check data loading")

## Exploratory Data Analysis

### Summary Statistics

print("=== SUMMARY STATISTICS (Class-Based) ===")  
  
# Use our SparkJobAnalyzer for comprehensive statistics  
stats = analyzer.get\_overall\_statistics()  
  
print(f"Total Job Postings: {stats['total\_jobs']:,}")  
print(f"Median Salary: ${stats['median\_salary']:,}")  
print(f"Mean Salary: ${stats['mean\_salary']:,}")  
print(f"Salary Range: ${stats['min\_salary']:,} - ${stats['max\_salary']:,}")  
print(f"25th-75th Percentile: ${stats['salary\_25th']:,} - ${stats['salary\_75th']:,}")  
  
# Industry analysis using our analyzer  
print("\n🏢 Top Industries by Median Salary:")  
industry\_analysis = analyzer.get\_industry\_analysis(top\_n=8)  
print(industry\_analysis.to\_string(index=False))  
  
# Experience level analysis  
print("\n🎓 Experience Level Distribution:")  
experience\_analysis = analyzer.get\_experience\_analysis()  
print(experience\_analysis.to\_string(index=False))  
  
# Geographic analysis  
print("\n🌍 Top Geographic Markets:")  
geographic\_analysis = analyzer.get\_geographic\_analysis(top\_n=8)  
print(geographic\_analysis.to\_string(index=False))  
  
# Skills analysis  
print("\n💻 Skills Premium Analysis:")  
skills\_analysis = analyzer.get\_skills\_analysis(top\_n=6)  
print(skills\_analysis.to\_string(index=False))  
  
# Convert sample to pandas for plotting (smaller sample for performance)  
df\_sample = df\_processed.sample(fraction=0.05).toPandas()  
print(f"\n📋 Sample data prepared: {len(df\_sample):,} records for visualization")

## Interactive Visualizations

### Job Postings by Industry

def plot\_jobs\_by\_industry(analyzer\_results):  
 """Create interactive plot using SparkJobAnalyzer results"""  
   
 # Use analyzer results instead of raw data processing  
 industry\_data = analyzer\_results.copy()  
   
 # Create interactive bar plot  
 fig = px.bar(  
 industry\_data,  
 x="Job Count",  
 y="Industry",  
 orientation="h",  
 title="Job Postings by Industry (SparkJobAnalyzer Results)",  
 labels={"Job Count": "Number of Job Postings", "Industry": "Industry"},  
 color="Median Salary",  
 color\_continuous\_scale="viridis",  
 hover\_data=["AI Premium", "Remote %"]  
 )  
   
 fig.update\_layout(  
 height=600,  
 yaxis={"categoryorder": "total ascending"},  
 showlegend=False  
 )  
   
 # Add value labels  
 fig.update\_traces(texttemplate='%{x:,}', textposition='outside')  
   
 return fig  
  
# Create plot using our analyzer results (already computed above)  
fig\_industry = plot\_jobs\_by\_industry(industry\_analysis)  
fig\_industry.show()

### Salary Distribution by Industry

def plot\_salary\_by\_industry(analyzer\_results):  
 """Create salary analysis plot using SparkJobAnalyzer results"""  
   
 # Use pre-computed analyzer results  
 industry\_data = analyzer\_results.copy()  
   
 # Create enhanced bar plot showing median salaries  
 fig = px.bar(  
 industry\_data,  
 x="Industry",   
 y="Median Salary",  
 title="Median Salary by Industry (SparkJobAnalyzer Analysis)",  
 labels={  
 "Median Salary": "Median Salary (USD)",  
 "Industry": "Industry"  
 },  
 color="Median Salary",  
 color\_continuous\_scale="RdYlBu\_r",  
 hover\_data=["Job Count", "AI Premium", "Remote %"]  
 )  
   
 fig.update\_layout(  
 height=600,  
 xaxis\_tickangle=-45  
 )  
   
 # Add value labels  
 fig.update\_traces(texttemplate='$%{y:,.0f}', textposition='outside')  
   
 return fig  
  
# Create plot using analyzer results  
fig\_salary\_industry = plot\_salary\_by\_industry(industry\_analysis)  
fig\_salary\_industry.show()

### Employment Type Analysis

def plot\_employment\_type\_analysis(df):  
 """Analyze salary by employment type"""  
   
 # Calculate statistics by employment type  
 emp\_stats = df.filter(col("REMOTE\_ALLOWED\_CLEAN").isin(["Remote", "Hybrid", "On-site"])) \  
 .groupBy("REMOTE\_ALLOWED\_CLEAN") \  
 .agg(  
 count("\*").alias("job\_count"),  
 avg("SALARY\_AVG\_IMPUTED").alias("avg\_salary"),  
 median("SALARY\_AVG\_IMPUTED").alias("median\_salary")  
 ).toPandas()  
   
 # Create subplot with multiple metrics  
 fig = make\_subplots(  
 rows=1, cols=2,  
 subplot\_titles=("Job Count by Employment Type", "Average Salary by Employment Type"),  
 specs=[[{"secondary\_y": False}, {"secondary\_y": False}]]  
 )  
   
 # Job count bar chart  
 fig.add\_trace(  
 go.Bar(  
 x=emp\_stats["REMOTE\_ALLOWED\_CLEAN"],  
 y=emp\_stats["job\_count"],  
 name="Job Count",  
 marker\_color="lightblue",  
 text=emp\_stats["job\_count"],  
 texttemplate='%{text:,}',  
 textposition='outside'  
 ),  
 row=1, col=1  
 )  
   
 # Salary bar chart  
 fig.add\_trace(  
 go.Bar(  
 x=emp\_stats["REMOTE\_ALLOWED\_CLEAN"],  
 y=emp\_stats["avg\_salary"],  
 name="Average Salary",  
 marker\_color="lightcoral",  
 text=emp\_stats["avg\_salary"],  
 texttemplate='$%{text:,.0f}',  
 textposition='outside'  
 ),  
 row=1, col=2  
 )  
   
 fig.update\_layout(  
 height=500,  
 title\_text="Employment Type Analysis: Remote vs Hybrid vs On-site",  
 showlegend=False  
 )  
   
 fig.update\_xaxes(title\_text="Employment Type", row=1, col=1)  
 fig.update\_xaxes(title\_text="Employment Type", row=1, col=2)  
 fig.update\_yaxes(title\_text="Number of Jobs", row=1, col=1)  
 fig.update\_yaxes(title\_text="Average Salary (USD)", row=1, col=2)  
   
 return fig  
  
# Create and display the plot  
fig\_employment = plot\_employment\_type\_analysis(df\_processed)  
fig\_employment.show()

### AI vs Traditional Roles Analysis

def plot\_ai\_vs\_traditional\_analysis(df):  
 """Comprehensive AI vs traditional roles analysis"""  
   
 # Get comparison data  
 ai\_comparison = df.groupBy("IS\_AI\_ROLE") \  
 .agg(  
 count("\*").alias("job\_count"),  
 avg("SALARY\_AVG\_IMPUTED").alias("avg\_salary"),  
 median("SALARY\_AVG\_IMPUTED").alias("median\_salary")  
 ).toPandas()  
   
 ai\_comparison["role\_type"] = ai\_comparison["IS\_AI\_ROLE"].map({True: "AI/ML Roles", False: "Traditional Roles"})  
   
 # Salary distribution data for violin plot  
 salary\_dist = df.select("IS\_AI\_ROLE", "SALARY\_AVG\_IMPUTED") \  
 .filter(col("SALARY\_AVG\_IMPUTED").isNotNull()) \  
 .sample(fraction=0.1) \  
 .toPandas()  
   
 salary\_dist["role\_type"] = salary\_dist["IS\_AI\_ROLE"].map({True: "AI/ML Roles", False: "Traditional Roles"})  
   
 # Create comprehensive subplot  
 fig = make\_subplots(  
 rows=2, cols=2,  
 subplot\_titles=(  
 "Job Count Comparison",   
 "Average Salary Comparison",  
 "Salary Distribution",   
 "AI Premium by Experience Level"  
 ),  
 specs=[  
 [{"type": "bar"}, {"type": "bar"}],  
 [{"type": "violin"}, {"type": "scatter"}]  
 ]  
 )  
   
 # Job count comparison  
 fig.add\_trace(  
 go.Bar(  
 x=ai\_comparison["role\_type"],  
 y=ai\_comparison["job\_count"],  
 name="Job Count",  
 marker\_color=["#FF6B6B", "#4ECDC4"],  
 text=ai\_comparison["job\_count"],  
 texttemplate='%{text:,}',  
 textposition='outside'  
 ),  
 row=1, col=1  
 )  
   
 # Salary comparison  
 fig.add\_trace(  
 go.Bar(  
 x=ai\_comparison["role\_type"],  
 y=ai\_comparison["avg\_salary"],  
 name="Average Salary",  
 marker\_color=["#FF6B6B", "#4ECDC4"],  
 text=ai\_comparison["avg\_salary"],  
 texttemplate='$%{text:,.0f}',  
 textposition='outside'  
 ),  
 row=1, col=2  
 )  
   
 # Salary distribution violin plot  
 for i, role\_type in enumerate(["Traditional Roles", "AI/ML Roles"]):  
 subset = salary\_dist[salary\_dist["role\_type"] == role\_type]  
 fig.add\_trace(  
 go.Violin(  
 y=subset["SALARY\_AVG\_IMPUTED"],  
 name=role\_type,  
 box\_visible=True,  
 meanline\_visible=True,  
 fillcolor=["#4ECDC4", "#FF6B6B"][i],  
 opacity=0.7  
 ),  
 row=2, col=1  
 )  
   
 # AI premium by experience level  
 exp\_analysis = df.groupBy("EXPERIENCE\_LEVEL\_CLEAN", "IS\_AI\_ROLE") \  
 .agg(avg("SALARY\_AVG\_IMPUTED").alias("avg\_salary")) \  
 .toPandas()  
   
 exp\_pivot = exp\_analysis.pivot(index="EXPERIENCE\_LEVEL\_CLEAN", columns="IS\_AI\_ROLE", values="avg\_salary").fillna(0)  
   
 if True in exp\_pivot.columns and False in exp\_pivot.columns:  
 exp\_pivot["premium\_pct"] = ((exp\_pivot[True] - exp\_pivot[False]) / exp\_pivot[False] \* 100).fillna(0)  
   
 fig.add\_trace(  
 go.Scatter(  
 x=exp\_pivot.index,  
 y=exp\_pivot["premium\_pct"],  
 mode='lines+markers',  
 name="AI Premium %",  
 line=dict(color="#FF6B6B", width=3),  
 marker=dict(size=8)  
 ),  
 row=2, col=2  
 )  
   
 fig.update\_layout(  
 height=800,  
 title\_text="Comprehensive AI vs Traditional Roles Analysis",  
 showlegend=False  
 )  
   
 # Update axes labels  
 fig.update\_xaxes(title\_text="Role Type", row=1, col=1)  
 fig.update\_xaxes(title\_text="Role Type", row=1, col=2)  
 fig.update\_xaxes(title\_text="Experience Level", row=2, col=2)  
   
 fig.update\_yaxes(title\_text="Number of Jobs", row=1, col=1)  
 fig.update\_yaxes(title\_text="Average Salary (USD)", row=1, col=2)  
 fig.update\_yaxes(title\_text="Salary (USD)", row=2, col=1)  
 fig.update\_yaxes(title\_text="Premium %", row=2, col=2)  
   
 return fig  
  
# Create and display the comprehensive analysis  
fig\_ai\_analysis = plot\_ai\_vs\_traditional\_analysis(df\_processed)  
fig\_ai\_analysis.show()

### Geographic Salary Heatmap

def plot\_geographic\_salary\_heatmap(df):  
 """Create geographic salary heatmap"""  
   
 # Calculate average salary by state  
 state\_salaries = df.groupBy("STATE\_CLEAN") \  
 .agg(  
 avg("SALARY\_AVG\_IMPUTED").alias("avg\_salary"),  
 count("\*").alias("job\_count")  
 ) \  
 .filter(col("job\_count") >= 10) \  
 .toPandas()  
   
 # Create choropleth map  
 fig = px.choropleth(  
 state\_salaries,  
 locations="STATE\_CLEAN",  
 color="avg\_salary",  
 locationmode="USA-states",  
 scope="usa",  
 title="Average Salary by State",  
 labels={"avg\_salary": "Average Salary (USD)"},  
 color\_continuous\_scale="viridis",  
 hover\_data=["job\_count"]  
 )  
   
 fig.update\_layout(height=600)  
   
 return fig  
  
# Create and display geographic heatmap  
fig\_geographic = plot\_geographic\_salary\_heatmap(df\_processed)  
fig\_geographic.show()

## Advanced Analytics

### Correlation Analysis

def plot\_correlation\_analysis(df):  
 """Perform correlation analysis on key variables"""  
   
 # Prepare data for correlation analysis  
 corr\_data = df.select(  
 "SALARY\_AVG\_IMPUTED",  
 "EXPERIENCE\_YEARS",   
 "IS\_AI\_ROLE",  
 when(col("REMOTE\_ALLOWED\_CLEAN") == "Remote", 1)  
 .when(col("REMOTE\_ALLOWED\_CLEAN") == "Hybrid", 0.5)  
 .otherwise(0).alias("REMOTE\_SCORE")  
 ).toPandas()  
   
 # Convert boolean to numeric  
 corr\_data["IS\_AI\_ROLE"] = corr\_data["IS\_AI\_ROLE"].astype(int)  
   
 # Calculate correlation matrix  
 corr\_matrix = corr\_data.corr()  
   
 # Create heatmap  
 fig = px.imshow(  
 corr\_matrix,  
 title="Correlation Matrix: Job Market Variables",  
 color\_continuous\_scale="RdBu\_r",  
 aspect="auto",  
 text\_auto=True  
 )  
   
 fig.update\_layout(height=500)  
   
 return fig, corr\_matrix  
  
# Create correlation analysis  
fig\_corr, corr\_matrix = plot\_correlation\_analysis(df\_processed)  
fig\_corr.show()  
  
print("\nCorrelation Matrix:")  
print(corr\_matrix.round(3))

## Data Export and Caching

print("=== EXPORTING ANALYSIS RESULTS (Class-Based) ===")  
  
# Create output directory  
output\_path = "data/processed/"  
Path(output\_path).mkdir(parents=True, exist\_ok=True)  
  
# Save class-based analysis results  
analysis\_results = {  
 "industry\_analysis": industry\_analysis,  
 "experience\_analysis": experience\_analysis,  
 "geographic\_analysis": geographic\_analysis,  
 "skills\_analysis": skills\_analysis,  
 "overall\_statistics": stats  
}  
  
# Export analysis results to CSV files  
for analysis\_name, data in analysis\_results.items():  
 if isinstance(data, pd.DataFrame):  
 csv\_path = f"{output\_path}/{analysis\_name}\_results.csv"  
 data.to\_csv(csv\_path, index=False)  
 print(f"Saved {analysis\_name}: {csv\_path}")  
  
# Save overall statistics as JSON  
import json  
stats\_path = f"{output\_path}/overall\_statistics.json"  
with open(stats\_path, "w") as f:  
 json.dump(stats, f, indent=2)  
print(f"Saved statistics: {stats\_path}")  
  
# Use processor to save processed data if available  
try:  
 if hasattr(processor, 'export\_processed\_data'):  
 processor.export\_processed\_data(output\_path)  
 else:  
 # Fallback data export  
 if df\_processed is not None:  
 parquet\_path = f"{output\_path}/job\_market\_processed.parquet"  
 df\_processed.write.mode("overwrite").parquet(parquet\_path)  
 print(f"Saved processed data: {parquet\_path}")  
   
 # Save sample as CSV  
 csv\_path = f"{output\_path}/job\_market\_sample.csv"  
 df\_processed.sample(fraction=0.01).toPandas().to\_csv(csv\_path, index=False)  
 print(f"Saved CSV sample: {csv\_path}")  
except Exception as e:  
 print(f"Export using fallback method: {e}")  
  
# Generate comprehensive analysis report  
report\_path = f"{output\_path}/comprehensive\_analysis\_report.md"  
with open(report\_path, "w") as f:  
 f.write("# Comprehensive Job Market Analysis Report\n\n")  
 f.write(f"\*\*Generated:\*\* {pd.Timestamp.now()}\n\n")  
 f.write(f"\*\*Analysis Framework:\*\* Class-based SparkJobAnalyzer + JobMarketDataProcessor\n\n")  
   
 f.write("## Overall Statistics\n\n")  
 for key, value in stats.items():  
 f.write(f"- \*\*{key.replace('\_', ' ').title()}:\*\* {value:,}\n")  
   
 f.write("\n## Top Industries by Median Salary\n\n")  
 f.write(industry\_analysis.to\_markdown(index=False))  
   
 f.write("\n\n## Skills Premium Analysis\n\n")  
 f.write(skills\_analysis.to\_markdown(index=False))  
   
 f.write("\n\n## Analysis Architecture\n\n")  
 f.write("This analysis was generated using our custom class-based architecture:\n")  
 f.write("- \*\*SparkJobAnalyzer\*\*: SQL-based analysis engine\n")  
 f.write("- \*\*JobMarketDataProcessor\*\*: Comprehensive data processing\n")  
 f.write("- \*\*SalaryVisualizer\*\*: Visualization utilities\n\n")  
 f.write("All results are dynamically generated from real data using PySpark.\n")  
  
print(f"Comprehensive report saved: {report\_path}")  
  
# Cache the processed DataFrame  
if df\_processed is not None:  
 df\_processed.cache()  
 print(f"\n✅ Analysis complete with {df\_processed.count():,} records cached")

## Summary and Architecture Benefits

### 🏧 Class-Based Architecture Success

This analysis demonstrates our **“Don’t Reinvent the Wheel”** principle using a robust class-based architecture:

**🔄 Key Classes Utilized:** 1. **SparkJobAnalyzer**: SQL-based analysis engine replacing manual DataFrame operations 2. **JobMarketDataProcessor**: Comprehensive data processing and quality assessment  
3. **SalaryVisualizer**: Consistent visualization utilities with graceful fallbacks

**🎨 UML Architecture Benefits:** - **Single Responsibility**: Each class handles specific domain concerns - **Dependency Inversion**: Analysis depends on abstractions, not concrete implementations - **Interface Segregation**: Clean method interfaces for specific analysis needs - **Reusability**: Same classes used across Quarto documents and notebooks

### 📊 Analysis Results

**Key Findings from Real Data Analysis:** - **Industry Leaders**: {industry\_analysis.iloc[0][‘Industry’]} leads with ${industry\_analysis.iloc[0][‘Median Salary’]:,} median salary - **Skills Premium**: {skills\_analysis.iloc[0][‘Skill Category’]} shows {skills\_analysis.iloc[0][‘Premium %’]:+}% premium - **Geographic Distribution**: {geographic\_analysis.iloc[0][‘Location’]} offers ${geographic\_analysis.iloc[0][‘Median Salary’]:,} median - **Experience Progression**: {len(experience\_analysis)} distinct experience levels analyzed

### 🚀 Next Steps

1. **🔄 Extend Class Methods**: Add new analysis methods to existing classes
2. **📈 Dashboard Integration**: Use class results in interactive dashboards
3. **📋 Automated Reporting**: Schedule class-based analysis pipelines
4. **🔍 Advanced Modeling**: Implement ML models using our data infrastructure

### 🎨 Design Pattern Success

**Before (Manual Functions):** 300+ lines of duplicated data processing code  
**After (Class-Based):** Reusable methods with consistent interfaces and error handling

**Architecture Documentation:** See docs/class\_architecture.md for complete UML diagram

*This analysis showcases how proper software engineering principles create maintainable, scalable, and reusable data science workflows.*