# Solutions Guide: Data Professional ChatGPT Tasks

## Beginner Level Solutions

### Task 1: Data Cleaning Assistant

```
import pandas as pd

def clean_dataset(df):
    # Fix date column
    df['date'] = pd.to_datetime(df['date'], errors='coerce')

    # Clean sales amount
    df['sales_amt'] = df['sales_amt'].str.replace('$',
'').str.replace(',', '').astype(float)

    # Standardize region
    df['region'] = df['region'].str.title()

    # Remove nulls in date (or could forward fill based on business rules)
    df = df.dropna(subset=['date'])

    return df

# Validation rules
validation_rules = {
    'date': {
        'check': lambda x: pd.notnull(x),
        'message': 'Date should not be null'
    },
    'sales_amt': {
        'check': lambda x: x > 0,
        'message': 'Sales amount should be positive'
    },
    'region': {
        'check': lambda x: x in ['North', 'South', 'East', 'West'],
        'message': 'Region should be one of North/South/East/West'
    }
}
```

### Task 2: SQL Query Optimization

```sql
-- Optimized query
SELECT
    c.id,
    c.name,
    c.email,
    o.order_date,
    o.amount
FROM customers c
LEFT JOIN orders o
    ON c.id = o.customer_id
    AND o.amount > 1000
ORDER BY o.order_date;

-- Index recommendations
CREATE INDEX idx_orders_amount_date ON orders(amount, order_date);
CREATE INDEX idx_orders_customer ON orders(customer_id);

-- Explanation:
-- 1. Selected only needed columns instead of customers.*
-- 2. Moved filter condition to JOIN clause for better optimization
-- 3. Added composite index for amount and order_date
-- 4. Added index for foreign key
```

Task 3: Data Pipeline Debug

```python
def transform_data(df):
    try:
        # Convert to datetime with error handling
        df['date'] = pd.to_datetime(df['date'], errors='coerce')

        # Add data validation
        if df['quantity'].isnull().any() or df['price'].isnull().any():
            raise ValueError("Missing values in quantity or price")

        # Calculate revenue
        df['revenue'] = df['quantity'].astype(float) *
df['price'].astype(float)

        # Store aggregation result
        revenue_by_category = df.groupby('category')
['revenue'].sum().reset_index()

        # Add logging
        logging.info(f"Processed {len(df)} rows with total revenue
```

```
{df['revenue'].sum()}")

        return df, revenue_by_category

    except Exception as e:
        logging.error(f"Error in transform_data: {str(e)}")
        raise
```

### Intermediate Level Solutions

#### Task 4: Feature Engineering

```
def engineer_features(df):
    # Time-based features
    df['days_since_last_purchase'] = (pd.Timestamp.now() -
df['last_purchase_date']).dt.days
    df['account_age_days'] = (pd.Timestamp.now() -
df['registration_date']).dt.days

    # Purchase behavior
    df['avg_purchase_frequency'] = df['total_purchases'] /
df['account_age_days']
    df['purchase_trend'] = df.groupby('customer_id')
['average_order_value'].pct_change()

    # Support engagement
    df['has_support_history'] = df['support_tickets'] > 0
    df['support_ticket_rate'] = df['support_tickets'] /
df['account_age_days']

    # Scaling
    scaler = StandardScaler()
    numeric_cols = ['days_since_last_purchase', 'avg_purchase_frequency',
'average_order_value']
    df[numeric_cols] = scaler.fit_transform(df[numeric_cols])

    return df
```

#### Task 5: Data Architecture Design

```
-- Fact Table
CREATE TABLE fact_sales (
```

```sql
    sale_id BIGINT PRIMARY KEY,
    date_key INT,
    product_key INT,
    customer_key INT,
    store_key INT,
    quantity INT,
    unit_price DECIMAL(10,2),
    total_amount DECIMAL(10,2),
    FOREIGN KEY (date_key) REFERENCES dim_date(date_key),
    FOREIGN KEY (product_key) REFERENCES dim_product(product_key),
    FOREIGN KEY (customer_key) REFERENCES dim_customer(customer_key),
    FOREIGN KEY (store_key) REFERENCES dim_store(store_key)
) PARTITION BY RANGE (date_key);

-- Type 2 SCD Dimension
CREATE TABLE dim_product (
    product_key INT PRIMARY KEY,
    product_id INT,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2),
    valid_from DATE,
    valid_to DATE,
    is_current BOOLEAN
);

-- Partitioning Strategy
CREATE TABLE fact_sales_2024_01 PARTITION OF fact_sales
    FOR VALUES FROM (20240101) TO (20240201);
```

### Task 6: Performance Monitoring

```python
from prometheus_client import Counter, Histogram, start_http_server
import time

# Metrics
PIPELINE_RUNS = Counter('pipeline_runs_total', 'Total number of pipeline runs')
PROCESSING_TIME = Histogram('processing_duration_seconds', 'Time spent processing data')
RECORDS_PROCESSED = Counter('records_processed_total', 'Total records processed')
ERRORS = Counter('pipeline_errors_total', 'Total number of errors', ['error_type'])
```

```python
def monitor_pipeline():
    try:
        start_time = time.time()
        PIPELINE_RUNS.inc()

        # Process data with metrics
        with PROCESSING_TIME.time():
            df = process_data()
            RECORDS_PROCESSED.inc(len(df))

        # Success metrics
        duration = time.time() - start_time
        log.info(f"Pipeline completed in {duration:.2f}s")

        # Alert if duration too long
        if duration > 300:  # 5 minutes
            alert_team("Pipeline running slowly", duration)

    except Exception as e:
        ERRORS.labels(error_type=type(e).__name__).inc()
        log.error(f"Pipeline failed: {str(e)}")
        raise
```

## Advanced Level Solutions

### Task 7: Machine Learning Pipeline

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE

def create_ml_pipeline(X, y):
    # Create pipeline with SMOTE for imbalanced data
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('smote', SMOTE(random_state=42)),
        ('classifier', RandomForestClassifier(class_weight='balanced'))
    ])

    # Cross-validation with multiple metrics
    cv_scores = cross_validate(
        pipeline, X, y,
```

```python
        cv=StratifiedKFold(n_splits=5),
        scoring={
            'accuracy': 'accuracy',
            'precision': 'precision_weighted',
            'recall': 'recall_weighted',
            'f1': 'f1_weighted',
            'auc': 'roc_auc_ovr_weighted'
        }
    )

    return pipeline, cv_scores
```

### Task 8: Data Quality Framework

```python
class DataQualityChecker:
    def __init__(self):
        self.tests = {
            'completeness': self.check_completeness,
            'uniqueness': self.check_uniqueness,
            'validity': self.check_validity
        }

    def check_completeness(self, df, columns):
        results = {}
        for col in columns:
            null_pct = (df[col].isnull().sum() / len(df)) * 100
            results[col] = {
                'pass': null_pct <= 5,
                'null_percentage': null_pct
            }
        return results

    def check_uniqueness(self, df, columns):
        return {
            col: {
                'pass': df[col].nunique() == len(df),
                'duplicate_count': len(df) - df[col].nunique()
            }
            for col in columns
        }

    def generate_report(self, results):
        return pd.DataFrame(results).to_html()
```

### Task 9: Real-time Analytics

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import window, count, avg

def process_streaming_data(spark):
    # Create streaming DataFrame
    stream_df = (spark
        .readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "transactions")
        .load()
    )

    # Process with windowing
    query = (stream_df
        .withWatermark("timestamp", "10 minutes")
        .groupBy(
            window("timestamp", "5 minutes", "1 minute"),
            "merchant_id"
        )
        .agg(
            count("transaction_id").alias("tx_count"),
            avg("amount").alias("avg_amount")
        )
    )

    # Output to sink
    return query.writeStream
        .outputMode("append")
        .format("console")
        .start()
```

### Task 10: Integration Challenge

```python
# Architecture components
class DataPipeline:
    def __init__(self):
        self.quality_checker = DataQualityChecker()
        self.monitoring = MetricsCollector()

    def process_batch(self, data):
        with self.monitoring.measure_time():
```

```python
            # 1. Extract
            raw_data = self.extract_data(data)

            # 2. Quality Check
            quality_results = self.quality_checker.run_checks(raw_data)
            if not quality_results['pass']:
                raise DataQualityException(quality_results)

            # 3. Transform
            transformed_data = self.transform_data(raw_data)

            # 4. Load
            self.load_data(transformed_data)

            # 5. Monitor
            self.monitoring.record_metrics(transformed_data)

    def deploy(self):
        return {
            'docker_compose': self.generate_docker_compose(),
            'kubernetes': self.generate_k8s_manifests(),
            'monitoring': self.generate_monitoring_config()
        }
```

## Assessment Guidelines

### Key Points to Check:

1. Code Quality:
    - Proper error handling
    - Clear documentation
    - Efficient algorithms
    - Appropriate logging
2. Architecture:
    - Scalability considerations
    - Component isolation
    - Clear interfaces
    - Error recovery
3. Performance:
    - Efficient data structures
    - Optimized queries

- Proper indexing
- Resource usage

4. Best Practices:
    - Testing approaches
    - Monitoring implementation
    - Security considerations
    - Documentation standards