DataOps Implementation Strategy for LLM Data Loading

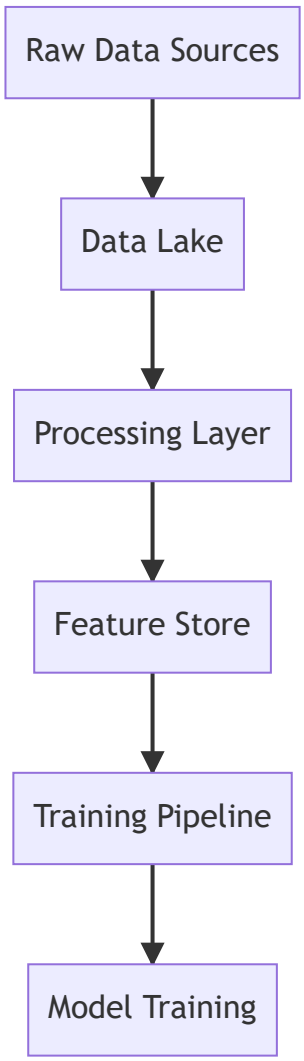
(bonus for deep understanding)

Author : Badr TAJINI - Large Language model (LLMs) - ESIEE 2024-2025

This is only an effort on the part of the research lecturer to show you how a pipeline in dataops can be tailored to a dataloader that is being used in this course to handle raw data.

1. Data Infrastructure Architecture

1.1 Storage Layer



- **Raw Data Storage**:
  - Object storage (S3/GCS) for raw text data
  - Parquet/Arrow formats for structured metadata
  - Version control for datasets using DVC or Delta Lake
- **Feature Store**:
  - Redis/MongoDB for caching frequently accessed sequences
  - Vector storage for embeddings (FAISS/Milvus)
  - Distributed cache for hot data

1.2 Processing Infrastructure

```
from typing import Optional, Dict, Any
from dataclasses import dataclass
from abc import ABC, abstractmethod

@dataclass
class DataConfig:
    max_length: int
    stride: int
    batch_size: int
    shuffle_buffer: int
    num_workers: int
    cache_size: int

class BaseDataLoader(ABC):
    @abstractmethod
    def load_data(self):
        pass

    @abstractmethod
    def preprocess(self):
        pass

    @abstractmethod
    def validate(self):
        pass

class DistributedGPTDataLoader(BaseDataLoader):
    def __init__(
        self,
        config: DataConfig,
        storage_client: Any,
        cache_client: Any,
        monitoring_client: Any
    ):
        self.config = config
        self.storage = storage_client
        self.cache = cache_client
        self.monitoring = monitoring_client

    def load_data(self):
        # Implement distributed data loading
        pass

    def preprocess(self):
        # Implement preprocessing with monitoring
        pass

    def validate(self):
        # Implement data validation
        pass
```

2. Data Quality and Validation

2.1 Validation Pipeline

```
class DataValidator:
    def __init__(self, schema: Dict):
        self.schema = schema

    def validate_sequence(self, sequence: torch.Tensor) -> bool:
        # Length validation
        if sequence.size(1) != self.schema['sequence_length']:
            return False

        # Value range validation
        if torch.any(sequence < 0) or torch.any(sequence >=
self.schema['vocab_size']):
            return False

        return True

class DataQualityMonitor:
    def __init__(self, metrics_client):
        self.metrics = metrics_client

    def log_sequence_stats(self, batch):
        self.metrics.gauge('sequence_length', batch.size(1))
        self.metrics.gauge('batch_size', batch.size(0))
        self.metrics.histogram('token_distribution', batch.flatten())
```

3. Observability and Monitoring

3.1 Metrics Collection

- Processing throughput
- Data quality metrics
- Resource utilization
- Cache hit rates
- Error rates

3.2 Monitoring Dashboard

```python
class DataLoaderMetrics:
    def __init__(self, prometheus_client):
        self.throughput = prometheus_client.Counter(
            'data_loader_throughput_sequences_total',
            'Number of sequences processed'
        )
        self.latency = prometheus_client.Histogram(
            'data_loader_latency_seconds',
            'Time to load and process sequences'
        )
        self.error_rate = prometheus_client.Counter(
            'data_loader_errors_total',
            'Number of data loading errors'
        )
```

4. Scaling Strategy

4.1 Horizontal Scaling

```python
class DistributedDataManager:
    def __init__(self, num_workers: int, shard_size: int):
        self.num_workers = num_workers
        self.shard_size = shard_size

    def shard_dataset(self, dataset_path: str):
        # Implement dataset sharding logic
        pass

    def merge_shards(self, shard_paths: List[str]):
        # Implement shard merging logic
        pass
```

4.2 Caching Strategy

```python
class CacheManager:
    def __init__(self, redis_client, cache_size: int):
        self.redis = redis_client
        self.cache_size = cache_size

    def cache_sequence(self, key: str, sequence: torch.Tensor):
        # Implement LRU caching logic
        pass

    def get_cached_sequence(self, key: str) -> Optional[torch.Tensor]:
        # Implement cache retrieval logic
        pass
```

5. Error Handling and Recovery

5.1 Resilient Data Loading

```python
class ResilientDataLoader:
    def __init__(self, max_retries: int = 3):
        self.max_retries = max_retries

    @retry(max_attempts=3, backoff=exponential_backoff)
    def load_batch(self):
        try:
            # Implement resilient batch loading
            pass
        except Exception as e:
            self.handle_error(e)
            raise
```

6. CI/CD Pipeline Integration

6.1 Data Pipeline Testing

```yaml
# .github/workflows/data-pipeline.yml
name: Data Pipeline Tests
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Data Quality Tests
        run: python -m pytest tests/data/
      - name: Run Performance Tests
        run: python -m pytest tests/performance/
```